Automatic Resource-Constrained Static Task Parallelization : A Generic Approach

Dounia Khaldi

▶ **To cite this version:**

Dounia Khaldi. Automatic Resource-Constrained Static Task Parallelization : A Generic Approach. Other [cs.OH]. Ecole Nationale Supérieure des Mines de Paris, 2013. English. NNT : 2013ENMP0031 . pastel-00935483

HAL Id: pastel-00935483
https://pastel.hal.science/pastel-00935483

Submitted on 23 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°432 :
Sciences des Métiers de l'Ingénieur

Doctorat ParisTech

THÈSE

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité « Informatique temps-réel, robotique et automatique »

présentée et soutenue publiquement par

Dounia KHALDI

le 27 novembre 2013

Parallélisation automatique et statique de tâches sous contraintes de ressources

– une approche générique –

~ ~ ~

Automatic Resource-Constrained Static Task Parallelization

– A Generic Approach –

Directeur de thèse : **François IRIGOIN**
Encadrement de la thèse : **Corinne ANCOURT**

Jury

Corinne ANCOURT, Chargée de recherche, CRI, MINES ParisTech	Examinateur
Cédric BASTOUL, Professeur, Université de Strasbourg	Président & Examinateur
Henri-Pierre CHARLES, Ingénieur chercheur en informatique, CEA	Rapporteur
François IRIGOIN, Directeur de recherche, CRI, MINES ParisTech	Directeur de thèse
Paul H J KELLY, Professeur, SPO, Imperial College London	Rapporteur

T H È S E

MINES ParisTech
Centre de recherche en informatique
35, rue Saint-Honoré, 77305 Fontainebleau Cedex, France

A la mémoire de ma nièce Nada Khaldi

Remerciements

Soyons reconnaissants aux personnes qui nous donnent du bonheur ; elles sont les charmants jardiniers par qui nos âmes sont fleuries. **Marcel Proust**

La réalisation d'une thèse est naturellement un travail personnel, mais en aucun cas un travail solitaire. Je tiens à remercier tout d'abord mon directeur de thèse, François Irigoin, et mon encadrante, Corinne Ancourt, pour m'avoir fait confiance malgré les connaissances plutôt légères que j'avais, en octobre 2010, sur les systèmes de parallélisation automatique. J'adresse également, et tout particulièrement, mes vifs remerciements et témoigne toute ma reconnaissance à Pierre Jouvelot pour l'expérience enrichissante et pleine d'intérêt qu'il m'a fait vivre durant ces trois années au sein du CRI.

Je remercie également tous ceux sans lesquels cette thèse ne serait pas ce qu'elle est, aussi bien par les discussions que j'ai eu la chance d'avoir avec eux, leurs suggestions ou leurs contributions. Je pense ici en particulier à Cédric Bastoul, qui de plus m'a fait l'honneur de présider le jury de cette thèse, et Paul Kelly et Henri-Pierre Charles, qui ont accepté d'être les rapporteurs de cette thèse, et je les en remercie, de même que pour leur participation au jury. Ils ont également contribué par leurs nombreuses remarques et suggestions à améliorer la qualité de ce mémoire, et je leur en suis très reconnaissante.

Je passe ensuite une dédicace spéciale à toutes les personnes que j'ai eu le plaisir de côtoyer au CRI, à savoir Jacqueline Altimira, Amira Mensi, Mehdi Amini, Vivien Maisonneuve, Olivier Hermant, Claude Tadonki, Antoniu Pop, Nelson Lossing, Pierre Guillou, Pierre Beauguitte, Benoît Pin, Laurent Daverio et, en particulier, Claire Médrala, avec qui j'ai partagé le même bureau. Je tiens aussi à remercier François Willot, pour son aide pour les expérimentations sur une machine à mémoire distribuée.

Je tiens aussi à remercier Sarah, Ghania et Halim, Razika et Rochdi pour leurs esprits taquins, ainsi que Khaled, Soumia, Molka, Naira, Nilsa, Arnaud et Karter pour leurs nombreux coups de main, leur sens pratique, leur efficacité et leur sourire. J'adresse enfin une pensée particulière à mes parents.

Abstract

This thesis intends to show how to efficiently exploit the parallelism present in applications in order to enjoy the performance benefits that multiprocessors can provide, using a new automatic task parallelization methodology for compilers. The key characteristics we focus on are resource constraints and static scheduling. This methodology includes the techniques required to decompose applications into tasks and generate equivalent parallel code, using a generic approach that targets both different parallel languages and architectures. We apply this methodology in the existing tool PIPS, a comprehensive source-to-source compilation platform.

This thesis mainly focuses on three issues. First, since extracting task parallelism from sequential codes is a scheduling problem, we design and implement an efficient, automatic scheduling algorithm called BDSC for parallelism detection; the result is a scheduled SDG, a new task graph data structure. In a second step, we design a new generic parallel intermediate representation extension called SPIRE, in which parallelized code may be expressed. Finally, we wrap up our goal of automatic parallelization in a new BDSC- and SPIRE-based parallel code generator, which is integrated within the PIPS compiler framework. It targets both shared and distributed memory systems using automatically generated OpenMP and MPI code.

Résumé

Le but de cette thèse est d'exploiter efficacement le parallélisme présent dans les applications informatiques séquentielles afin de bénéficier des performances fournies par les multiprocesseurs, en utilisant une nouvelle méthodologie pour la parallélisation automatique des tâches au sein des compilateurs. Les caractéristiques clés de notre approche sont la prise en compte des contraintes de ressources et le caractère statique de l'ordonnancement des tâches. Notre méthodologie contient les techniques nécessaires pour la décomposition des applications en tâches et la génération de code parallèle équivalent, en utilisant une approche générique qui vise différents langages et architectures parallèles. Nous implémentons cette méthodologie dans le compilateur source-à-source PIPS. Cette thèse répond principalement à trois questions. Primo, comme l'extraction du parallélisme de tâches des codes séquentiels est un problème d'ordonnancement, nous concevons et implémentons un algorithme d'ordonnancement efficace, que nous nommons BDSC, pour la détection du parallélisme ; le résultat est un SDG ordonnancé, qui est une nouvelle structure de données de graphe de tâches. Secondo, nous proposons une nouvelle extension générique des représentations intermédiaires séquentielles en des représentations intermédiaires parallèles que nous nommons SPIRE, pour la représentation des codes parallèles. Enfin, nous développons, en utilisant BDSC et SPIRE, un générateur de code que nous intégrons dans PIPS. Ce générateur de code cible les systèmes à mémoire partagée et à mémoire distribuée via des codes OpenMP et MPI générés automatiquement.

Contents

Remerciements **iv**

Abstract **vi**

Résumé **viii**

Introduction (*en français*) **8**

1 Introduction **10**
1.1 Context 10
1.2 Motivation 11
1.3 Contributions 13
1.4 Thesis Outline 14

2 Perspectives: Bridging Parallel Architectures and Software Parallelism **18**
2.1 Parallel Architectures 19
2.1.1 Multiprocessors 20
2.1.2 Memory Models 22
2.2 Parallelism Paradigms 23
2.3 Mapping Paradigms to Architectures 25
2.4 Program Dependence Graph (PDG) 26
2.4.1 Control Dependence Graph (CDG) 26
2.4.2 Data Dependence Graph (DDG) 27
2.5 List Scheduling 28
2.5.1 Background 28
2.5.2 Algorithm 29
2.6 PIPS: Automatic Parallelizer and Code Transformation Framework 32
2.6.1 Transformer Analysis 33
2.6.2 Precondition Analysis 33
2.6.3 Array Region Analysis 34
2.6.4 "Complexity" Analysis 36
2.6.5 PIPS (Sequential) IR 37
2.7 Conclusion 38

3 Concepts in Task Parallel Programming Languages **40**
3.1 Introduction 41
3.2 Mandelbrot Set Computation 42
3.3 Task Parallelism Issues 44

3.3.1 Task Creation 44
3.3.2 Synchronization 44
3.3.3 Atomicity 45
3.4 Parallel Programming Languages 45
3.4.1 Cilk 45
3.4.2 Chapel 47
3.4.3 X10 and Habanero-Java 48
3.4.4 OpenMP 50
3.4.5 MPI 52
3.4.6 OpenCL 54
3.5 Discussion and Comparison 56
3.6 Conclusion 59

4 SPIRE: A Generic Sequential to Parallel Intermediate Representation Extension Methodology 62
4.1 Introduction 64
4.2 SPIRE, a Sequential to Parallel IR Extension 65
4.2.1 Design Approach 66
4.2.2 Execution 67
4.2.3 Synchronization 69
4.2.4 Data Distribution 74
4.3 SPIRE Operational Semantics 76
4.3.1 Sequential Core Language 76
4.3.2 SPIRE as a Language Transformer 77
4.3.3 Rewriting Rules 80
4.4 Validation: SPIRE Application to LLVM IR 82
4.5 Related Work: Parallel Intermediate Representations 85
4.6 Conclusion 87

5 BDSC: A Memory-Constrained, Number of Processor-Bounded Extension of DSC 90
5.1 Introduction 91
5.2 The DSC List-Scheduling Algorithm 92
5.2.1 The DSC Algorithm 92
5.2.2 Dominant Sequence Length Reduction Warranty . . . 93
5.3 BDSC: A Resource-Constrained Extension of DSC 95
5.3.1 DSC Weaknesses 96
5.3.2 Resource Modeling 96
5.3.3 Resource Constraint Warranty 97
5.3.4 Efficient Task-to-Cluster Mapping 98
5.3.5 The BDSC Algorithm 100
5.4 Related Work: Scheduling Algorithms 103
5.4.1 Bounded Number of Clusters 103
5.4.2 Cluster Regrouping on Physical Processors 105

5.5 Conclusion 106

6 BDSC-Based Hierarchical Task Parallelization 108

6.1 Introduction 109
6.2 Hierarchical Sequence Dependence DAG Mapping 110
6.2.1 Sequence Dependence DAG (SDG) 110
6.2.2 Hierarchical SDG Mapping 115
6.3 Sequential Cost Models Generation 117
6.3.1 From Convex Polyhedra to Ehrhart Polynomials . . . 117
6.3.2 From Polynomials to Values 121
6.4 Reachability Analysis: The Path Transformer 124
6.4.1 Path Definition 124
6.4.2 Path Transformer Algorithm 125
6.4.3 Operations on Regions using Path Transformers . . . 132
6.5 BDSC-Based Hierarchical Scheduling (HBDSC) 134
6.5.1 Closure of SDGs 134
6.5.2 Recursive Top-Down Scheduling 135
6.5.3 Iterative Scheduling for Resource Optimization 139
6.5.4 Complexity of HBDSC Algorithm 140
6.5.5 Parallel Cost Models 142
6.6 Related Work: Task Parallelization Tools 144
6.7 Conclusion 146

7 SPIRE-Based Parallel Code Transformations and Generation 148

7.1 Introduction 149
7.2 Parallel Unstructured to Structured Transformation 150
7.2.1 Structuring Parallelism 152
7.2.2 Hierarchical Parallelism 155
7.3 From Shared Memory to Distributed Memory Transformation 157
7.3.1 Related Work: Communications Generation 158
7.3.2 Difficulties 159
7.3.3 Equilevel and Hierarchical Communications 161
7.3.4 Equilevel Communications Insertion Algorithm 165
7.3.5 Hierarchical Communications Insertion Algorithm . . 169
7.3.6 Communications Insertion Main Algorithm 169
7.3.7 Conclusion and Future Work 171
7.4 Parallel Code Generation 175
7.4.1 Mapping Approach 175
7.4.2 OpenMP Generation 176
7.4.3 SPMDization: MPI Generation 178
7.5 Conclusion 180

8 Experimental Evaluation with PIPS **182**
8.1 Introduction 183
8.2 Implementation of HBDSC- and SPIRE-Based Parallelization Processes in PIPS 184
8.2.1 Preliminary Passes 184
8.2.2 Task Parallelization Passes 187
8.2.3 OpenMP Related Passes 189
8.2.4 MPI Related Passes: SPMDization 190
8.3 Experimental Setting 191
8.3.1 Benchmarks: ABF, Harris, Equake, IS and FFT . . . 191
8.3.2 Experimental Platforms: Austin and Cmmcluster . . . 193
8.3.3 Effective Cost Model Parameters 194
8.4 Protocol 194
8.5 BDSC vs. DSC 195
8.5.1 Experiments on Shared Memory Systems 195
8.5.2 Experiments on Distributed Memory Systems 198
8.6 Scheduling Robustness 199
8.7 Faust Parallel Scheduling vs. BDSC 201
8.7.1 OpenMP Version in Faust 201
8.7.2 Faust Programs: Karplus32 and Freeverb 202
8.7.3 Experimental Comparison 202
8.8 Conclusion 203

9 Conclusion **206**
9.1 Contributions 206
9.2 Future Work 209

Conclusion (*en français*) **217**

List of Tables

2.1 Multicores proliferation 21
2.2 Execution time estimations for Harris functions using PIPS complexity analysis 37

3.1 Summary of parallel languages constructs 59

4.1 Mapping of SPIRE to parallel languages constructs (terms in parentheses are not currently handled by SPIRE) 67

6.1 Execution and communication time estimations for Harris using PIPS default cost model (N and M variables represent the input image size) 121
6.2 Comparison summary between different parallelization tools . 146

8.1 CPI for the Austin and Cmmcluster machines, plus the transfer cost of one byte β on the Cmmcluster machine (in #cycles) 194
8.2 Run-time sensitivity of BDSC with respect to static cost estimation (in ms for Harris and ABF; in s for equake and IS). 201

List of Figures

1 Une implementation séquentielle en C de la fonction `main` de l'algorithme de Harris . . . 4
2 Le graphe de flôts de données de l'algorithme de Harris . . . 4
3 Organisation de la thèse : le bleu indique les contributions de la thèse ; une ellipse, un processus; et un rectangle, résultats . 8

1.1 Sequential C implementation of the `main` function of Harris . 12
1.2 Harris algorithm data flow graph . . . 12
1.3 Thesis outline: blue indicates thesis contributions; an ellipse, a process; and a rectangle, results . . . 16

2.1 A typical multiprocessor architectural model . . . 20
2.2 Memory models . . . 22
2.3 Rewriting of data parallelism using the task parallelism model 24
2.4 Construction of the control dependence graph . . . 27
2.5 Example of a C code and its data dependence graph . . . 28
2.6 A Directed Acyclic Graph (left) and its associated data (right) 30
2.7 Example of transformer analysis . . . 34
2.8 Example of precondition analysis . . . 34
2.9 Example of array region analysis . . . 35
2.10 Simplified Newgen definitions of the PIPS IR . . . 37

3.1 Result of the Mandelbrot set . . . 43
3.2 Sequential C implementation of the Mandelbrot set . . . 43
3.3 Cilk implementation of the Mandelbrot set (P is the number of processors) . . . 46
3.4 Chapel implementation of the Mandelbrot set . . . 47
3.5 X10 implementation of the Mandelbrot set . . . 49
3.6 A clock in X10 (left) and a phaser in Habanero-Java (right) . 50
3.7 C OpenMP implementation of the Mandelbrot set . . . 51
3.8 MPI implementation of the Mandelbrot set (P is the number of processors) . . . 53
3.9 OpenCL implementation of the Mandelbrot set . . . 55
3.10 A hide-and-seek game (X10, HJ, Cilk) . . . 57
3.11 Example of an atomic directive in OpenMP . . . 58
3.12 Data race on `ptr` with Habanero-Java . . . 58

4.1 OpenCL example illustrating a parallel loop . . . 68
4.2 `forall` in Chapel, and its SPIRE core language representation 68
4.3 A C code, and its unstructured parallel control flow graph representation . . . 69

4.4 `parallel sections` in OpenMP, and its SPIRE core language representation . . . 69
4.5 OpenCL example illustrating spawn and barrier statements . 71
4.6 Cilk and OpenMP examples illustrating an atomically-synchronized statement . . . 71
4.7 X10 example illustrating a future task and its synchronization 72
4.8 A phaser in Habanero-Java, and its SPIRE core language representation . . . 73
4.9 Example of Pipeline Parallelism with phasers . . . 73
4.10 MPI example illustrating a communication, and its SPIRE core language representation . . . 75
4.11 SPIRE core language representation of a non-blocking send . 75
4.12 SPIRE core language representation of a non-blocking receive 75
4.13 SPIRE core language representation of a broadcast . . . 75
4.14 `Stmt` and `SPIRE(Stmt)` syntaxes . . . 76
4.15 `Stmt` sequential transition rules . . . 77
4.16 `SPIRE(Stmt)` synchronized transition rules . . . 80
4.17 `if` statement rewriting using `while` loops . . . 82
4.18 Simplified Newgen definitions of the LLVM IR . . . 83
4.19 A loop in C and its LLVM intermediate representation . . . 84
4.20 SPIRE (LLVM IR) . . . 84
4.21 An example of a `spawn`ed outlined sequence . . . 85

5.1 A Directed Acyclic Graph (left) and its scheduling (right); starred top levels (*) correspond to the selected clusters . . . 94
5.2 Result of DSC on the graph in Figure 5.1 without (left) and with (right) DSRW . . . 95
5.3 A DAG amenable to cluster minimization (left) and its BDSC step-by-step scheduling (right) . . . 100
5.4 DSC (left) and BDSC (right) cluster allocation . . . 100

6.1 Abstract syntax trees `Statement` syntax . . . 110
6.2 Parallelization process: blue indicates thesis contributions; an ellipse, a process; and a rectangle, results . . . 111
6.3 Example of a C code (left) and the DDG D of its internal S sequence (right) . . . 114
6.4 SDGs of S (top) and $S0$ (bottom) computed from the DDG (see the right of Figure 6.3); S and $S0$ are specified in the left of Figure 6.3 . . . 115
6.5 SDG G, for a part of equake S given in Figure 6.7; *Gbody* is the SDG of the body *Sbody* (logically included into G via H) 117

6.6 Example of the execution time estimation for the function `MultiplY` of the code Harris; each comment provides the complexity estimation of the statement below (N and M are assumed to be global variables) . . . 120
6.7 Instrumented part of equake (*Sbody* is the inner loop sequence) 122
6.8 Numerical results of the instrumented part of equake (`instrumented_equake.in`) . . . 123
6.9 An example of a subtree of root $forloop_0$ . . . 125
6.10 An example of a path execution trace from S_b to S_e in the subtree $forloop_0$ . . . 126
6.11 An example of a C code and the path transformer of the sequence between S_b and S_e; M is $\perp$, since there are no loops in the code . . . 129
6.12 An example of a C code and the path transformer of the sequence of calls and test between S_b and S_e; M is $\perp$, since there are no loops in the code . . . 130
6.13 An example of a C code and the path transformer between S_b and S_e . . . 133
6.14 Closure of the SDG $G0$ of the C code $S0$ in Figure 6.3, with additional import entry vertices . . . 136
6.15 `topsort`(G) for the hierarchical scheduling of sequences . . . 139
6.16 Scheduled SDG for Harris, using P=3 cores; the scheduling information via `cluster`(τ) is also printed inside each vertex of the SDG . . . 139
6.17 After the first application of BDSC on S = `sequence`(S_0;S_1), failing with "Not enough memory" . . . 141
6.18 Hierarchically (via H) Scheduled SDGs with memory resource minimization after the second application of BDSC (which succeeds) on `sequence`(S_0;S_1) (κ_i = `cluster`($\sigma(S_i)$)). To keep the picture readable, only communication edges are figured in these SDGs . . . 142

7.1 Parallel code transformations and generation: blue indicates this chapter contributions; an ellipse, a process; and a rectangle, results . . . 151
7.2 Unstructured parallel control flow graph (a) and event (b) and structured representations (c) . . . 152
7.3 A part of Harris, and one possible SPIRE core language representation . . . 156
7.4 SPIRE representation of a part of Harris, non optimized (left) and optimized (right) . . . 156
7.5 A simple C code (hierarchical), and its SPIRE representation 157

7.6 An example of a shared memory C code and its SPIRE code efficient communication primitive-based distributed memory equivalent . . . 160
7.7 Two examples of C code with non-uniform dependences . . . 160
7.8 An example of a C code and its read and write array regions analysis for two communications from S_0 to S_1 and to S_2 . . 161
7.9 An example of a C code and its in and out array regions analysis for two communications from S_0 to S_1 and to S_2 . . 162
7.10 Scheme of communications between multiple possible levels of hierarchy in a parallel code; the levels are specified with the superscripts on τ's . . . 163
7.11 Equilevel and hierarchical communications . . . 164
7.12 A part of an SDG and the topological sort order of predecessors of τ_4 . . . 166
7.13 Communications generation from a region using `region_to_coms` function . . . 168
7.14 Equilevel and hierarchical communications generation and SPIRE distributed memory representation of a C code 172
7.15 Graph illustration of communications made in the C code of Figure 7.14; equilevel in green arcs, hierarchical in black . . . 173
7.16 Equilevel and hierarchical communications generation after optimizations: the current cluster executes the last nested spawn (left) and barriers with one spawn statement are removed (right) . . . 174
7.17 SPIRE representation of a part of Harris, and its OpenMP task generated code . . . 177
7.18 SPIRE representation of a C code, and its OpenMP hierarchical task generated code . . . 178
7.19 SPIRE representation of a part of Harris, and its MPI generated code . . . 179

8.1 Parallelization process: blue indicates thesis contributions; an ellipse, a pass or a set of passes; and a rectangle, results . . . 185
8.2 Executable (harris.tpips) for harris.c . . . 186
8.3 A model of an MPI generated code . . . 191
8.4 Scheduled SDG of the main function of ABF . . . 192
8.5 Steps for the rewriting of data parallelism to task parallelism using the tiling and unrolling transformations . . . 196
8.6 ABF and equake speedups with OpenMP . . . 196
8.7 Speedups with OpenMP: impact of tiling (P=3) . . . 197
8.8 Speedups with OpenMP for different class sizes (IS) . . . 198
8.9 ABF and equake speedups with MPI . . . 199
8.10 Speedups with MPI: impact of tiling (P=3) . . . 199
8.11 Speedups with MPI for different class sizes (IS) . . . 200

8.12 Run-time comparison (BDSC vs. Faust for Karplus32) 203
8.13 Run-time comparison (BDSC vs. Faust for Freeverb) 203

Introduction (*en français*)

I live and don't know how long, I'll die and don't know when, I am going and don't know where, I wonder that I am happy. **Martinus Von Biberach**

Contexte

La loi de Moore [83] postule que, tout au long de l'histoire du matériel informatique, le nombre de transistors utilisés dans les circuits intégrés va doubler tous les deux ans environ. Cette croissance s'accompagnait, jusqu'à très récemment encore, de vitesses d'horloge toujours plus élevées, afin de d'améliorer encore plus les performances des processeurs (ou cœurs). Le passage de la fabrication de microprocesseurs uniques à la conception de machines parallèles est en partie dû à la croissance de la consommation exagérée d'énergie liée à cette augmentation de fréquence. Le nombre de transistors continue toutefois à augmenter afin d'intégrer plus de cœurs et assurer un passage à l'échelle proportionnel de performance pour des applications scientifiques toujours plus sophistiquées.

En dépit de la validité de la loi de Moore jusqu'à maintenant, la performance des cœurs a ainsi cessé d'augmenter après 2003. Du coup, la scalabilité des applications, qui correspond à l'idée que les performances sont accrues lorsque des ressources supplémentaires sont allouées à la résolution d'un problème, n'est plus garantie. Pour comprendre ce problème, il est nécessaire de jeter un coup d'œil à la loi dite d'Amdahl [16]. Selon cette loi, l'accélération d'un programme à l'aide de plusieurs processeurs est limitée par le temps nécessaire à l'exécution de sa partie séquentielle ; en plus du nombre de processeurs, l'algorithme lui-même limite également cette accélération.

Afin de profiter des performances que les multiprocesseurs peuvent fournir, il faut bien évidemment toujours arriver à exploiter efficacement le parallélisme présent dans les applications. C'est une tâche difficile pour les programmeurs, surtout si ce programmeur est un physicien, un mathématicien ou un informaticien pour qui la compréhension de l'application est difficile car elle n'est pas la sienne. Bien sûr, nous pourrions dire au programmeur : "penses parallèle" ! Mais les êtres humains ont tendance à penser, pour l'essentiel, de manière séquentielle. Par conséquent, détecter le parallélisme présent dans un code séquentiel et, automatiquement ou non, écrire un code parallèle efficace équivalent a été, et restera sans doute pendant un certain temps encore, un problème majeur.

Pour d'évidentes raisons économiques, une application parallèle, exprimée dans un modèle de programmation parallèle, doit être non seulement efficace mais également aussi portable que possible, c'est-à-dire être telle qu'il

ne faille pas avoir à réécrire ce code parallèle pour utiliser une autre machine que celle prévue initialement. Pourtant, la prolifération actuelle de modèles de programmation parallèle distincts fait que le choix d'un modèle général n'est, manifestement, pas évident, à moins de disposer d'un langage parallèle à la fois efficace et capable d'être compilé pour tous les types d'architectures actuelles et futures, ce qui n'est pas encore le cas. Par conséquent, cette écriture de code parallèle doit être fondée sur une approche générique, c'est à dire susceptible d'être facilement adaptée à un groupe de langages aussi large que possible, afin d'offrir une meilleure chance de portabilité.

Motivation

Pour que la programmation pour multiprocesseurs conduise à de bonnes performances sans aller jusqu'à avoir à penser "parallèle", une plate-forme logicielle de parallélisation automatique visant à exploiter efficacement les cœurs est nécessaire. Si divers modes de parallélisme existent, la prolifération de processeurs multi-cœurs offrant des pipelines courts et des fréquences d'horloge relativement basses et la pression que le simple modèle de parallélisme de données impose à la bande passante de la mémoire ont fait de telle sorte que la prise en compte du parallélisme à gros grain apparait comme inévitable pour améliorer les performances.

La première étape d'un processus de parallélisation nous semble donc être d'exposer la concurrence de tâche présente dans les programmes, en décomposant les applications en groupes pipelinés d'instructions, puisque les algorithmes peuvent souvent être décrits comme de tels ensembles de tâches. Pour illustrer l'importance de la parallélisation au niveau tâche, nous donnons ici un exemple simple : l'algorithme de recherche de coins dans une image proposé par Harris [96]: il est fondé sur de l'autocorrélation pixel par pixel, en utilisant une chaîne de fonctions, à savoir Sobel, Multiplication, Gauss, et Coarsity. La figure 1 illustre ce processus. Plusieurs chercheurs ont déjà parallélisé et adapté cet algorithme sur des architectures parallèles telles que le processeur CELL [95].

La figure 2 montre une instance de partitionnement possible, réalisée à la main, de l'algorithme de Harris (voir aussi [95]). L'algorithme de Harris peut ainsi être décomposé en une succession de deux à trois tâches simultanées (ellipses sur la figure), en faisant par ailleurs abstraction du parallélisme de données potentiellement présent. L'automatisation de cette approche intuitive, et donc l'extraction automatique de tâches et la parallélisation d'applications de ce type, est la motivation qui sous-tend cette thèse.

Cependant, la détection du parallélisme de tâches et la génération de code efficace s'appuient sur des analyses complexes de dépendances de données, de communication, de synchronisation, etc. Ces différentes analyses peuvent, *de facto*, être dès maintenant fournies en grande partie par des

```
void main(int argc, char *argv[]){
 float (*Gx)[N*M], (*Gy)[N*M], (*Ixx)[N*M],
       (*Iyy)[N*M], (*Ixy)[N*M], (*Sxx)[N*M],
       (*Sxy)[N*M], (*Syy)[N*M], (*in)[N*M];
 in = InitHarris();
 /* Now we run the Harris procedure */
 //Sobel
 SobelX(Gx, in);
 SobelY(Gy, in);
 //Multiply
 MultiplY(Ixx, Gx, Gx);
 MultiplY(Iyy, Gy, Gy);
 MultiplY(Ixy, Gx, Gy);
 //Gauss
 Gauss(Sxx, Ixx);
 Gauss(Syy, Iyy);
 Gauss(Sxy, Ixy);
 //Coarsity
 CoarsitY(out, Sxx, Syy, Sxy);
 return;
}
```

Figure 1: Une implementation séquentielle en C de la fonction `main` de l'algorithme de Harris

plateformes logicielles de compilation. Ainsi, la délégation au logiciel du "penser parallèle", construite sur la mobilisation d'analyses automatiques déjà existantes de dépendances de données, parait viable, et peut permettre d'espérer voir gérées tout à la fois la granularité présente dans les codes séquentiels et les contraintes de ressources telles que la taille de la mémoire ou le nombre de processeurs, contraintes qui ont un impact certain sur ce processus de parallélisation.

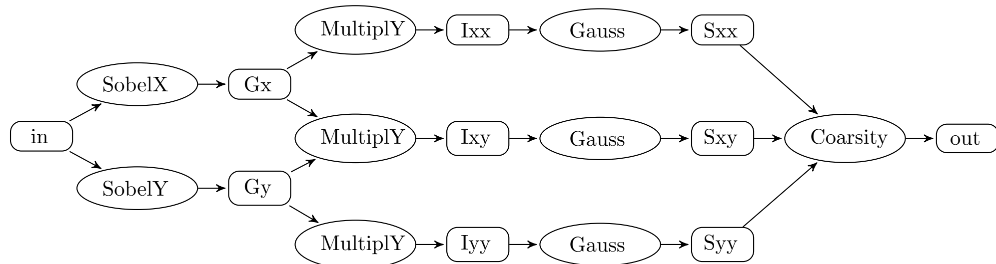


Figure 2: Le graphe de flôts de données de l'algorithme de Harris

La parallélisation automatique de tâches a été étudiée depuis presque un demi-siècle. Le problème de l'extraction d'un parallélisme optimal avec des

communications optimales est, dans toute sa généralité, un problème NP-complet [47]. Plusieurs travaux ont tenté d'automatiser la parallélisation de programmes en utilisant différents niveaux de granularité. Métis [62] et d'autres outils de partitionnement de graphe visent à attribuer la même quantité de travail aux processeurs, avec de petites quantités de communication entre ces processeurs, mais la structure du graphe ne fait pas de distinction entre les boucles, les appels de fonction, etc. Les étapes cruciales de construction de graphe et de génération de code parallèle sont absents. Sarkar [98] met en œuvre une méthode de compilation pour le problème de partitionnement pour multiprocesseurs. Un programme est divisé en tâches parallèles au moment de la compilation, puis celles-ci sont fusionnées jusqu'à ce qu'une partition avec le plus petit temps d'exécution parallèle, en présence des surcouts (ordonnancement et communication), soit trouvée. Malheureusement, cet algorithme ne prend pas en compte les contraintes de ressources, qui sont des facteurs importants pour cibler des architectures réelles. Tous ces outils de parallélisation sont dédiés à un modèle de programmation particulier : il y a manifestement un manque d'abstraction générique du parallélisme (exécution parallèle, synchronisation et distribution de données). Ils ne répondent donc pas à la question de la portabilité.

Dans cette thèse, nous développons une méthodologie de parallélisation de tâches automatique pour les compilateurs : les caractéristiques principales sur lesquelles nous nous concentrons sont les contraintes de ressources et l'ordonnancement statique. Elle comprend les techniques nécessaires pour décomposer les applications en tâches et générer le code parallèle équivalent, en utilisant une approche générique qui cible différents langages parallèles et donc différentes architectures. Nous appliquons cette méthodologie à l'outil existant PIPS [59], une plate-forme de compilation source-à-source.

Contributions

Notre objectif est de développer un nouvel outil et des algorithmes pour la parallélisation automatique de tâches ; nous souhaitons qu'ils prennent en compte certaines contraintes de ressources et également être utiles en général pour les langages parallèles existants. Ainsi, les principales contributions de cette thèse sont les suivantes :

1. un nouvel algorithme hiérarchique d'ordonnancement (HBDSC) qui utilise :
   - une nouvelle structure de données pour représenter les programmes parallèles partitionnés sous forme d'un graphe acyclique que nous nommons SDG,
   - une extension, appelée "DSC borné" (BDSC), de l'algorithme d'ordonnancement DSC [111] capable de gérer simultanément

deux contraintes de ressources, à savoir une taille mémoire bornée par processeur et un nombre borné de processeurs, qui sont des paramètres clés lors de l'ordonnancement des tâches sur les multiprocesseurs réels,

- un nouveau modèle de coût fondé sur l'estimation de la complexité en temps d'exécution, la définition d'approximations polyédriques convexes de la taille des tableaux de données et l'instrumentation de code pour l'étiquetage des sommets et les arêtes du SDG ;

2. une nouvelle approche pour l'adaptation des plates-formes de parallélisation automatique aux langages parallèles via :

   - SPIRE, une nouvelle méthodologie d'extension au parallélisme des représentations intermédiaires (RI) utilisées dans les compilateurs, pour la conception des RIs parallèles ,
   - le déploiement de SPIRE pour la parallélisation automatique au niveau tâche de programmes parallèles dans le compilateur PIPS [59] ;

3. une implémentation dans le compilateur source-à-source PIPS de :

   - la parallélisation fondée sur HBDSC des programmes encodés dans la RI parallèle de PIPS dérivée de SPIRE,
   - la génération de code parallèle fondée sur SPIRE pour deux langages parallèles : OpenMP [4] et MPI [3] ;

4. des mesures de performance de notre approche de parallélisation, sur la base de :

   - cinq programmes significatifs, ciblant à la fois les architectures à mémoire partagée et distribuée : deux benchmarks de traitement d'image et de signal, Harris [54] et ABF [52], le benchmark equake [22] de SPEC2001,le benchmark IS [87] extrait de NAS et un code de FFT [8],
   - leurs translations automatiques en deux langages parallèles : OpenMP [4] et MPI [3],
   - et, enfin, une étude comparative entre notre implémentation de parallélisme de tâche dans PIPS et celle du langage de traitement de signal audio Faust [88], en utilisant deux programmes de Faust : Karplus32 et Freeverb. Nous choisissons Faust puisqu'il s'agit d'un compilateur open-source qui génère également des tâches parallèles automatiquement en OpenMP.

Organisation

Cette thèse est organisée en neuf chapitres. La figure 3 montre comment ces chapitres peuvent être interprétés dans le contexte d'une chaîne de parallélisation.

Le chapitre 2 fournit un bref historique des concepts d'architectures, de parallélisme, des langages et des compilateurs utilisés dans les chapitres suivants.

Le chapitre 3 examine sept langages de programmation parallèles actuels et efficaces afin de déterminer et de classifier leurs constructions parallèles. Cela permet de définir un noyau de langage parallèle qui joue le rôle d'une représentation intermédiaire parallèle générique lors de la parallélisation de programmes séquentiels.

Cette proposition, appelée la méthodologie SPIRE, est développée dans le chapitre 4. SPIRE exploite les infrastructures de compilation existantes pour représenter les constructions à la fois de contrôle et de données présentes dans les langages parallèles tout en préservant autant que possible les analyses existantes pour les codes séquentiels. Pour valider cette approche dans la pratique, nous utilisons PIPS, une plate-forme de compilation source-à-source, comme un cas d'utilisation pour mettre à jour sa représentation intermédiaire séquentielle en une RI parallèle.

Puisque le but principal de cette thèse est la parallélisation automatique de tâches, l'extraction du parallélisme de tâche des codes séquentiels est une étape clé dans ce processus, que nous considérons comme un problème d'ordonnancement. Le chapitre 5 introduit ainsi une nouvelle heuristique automatique et efficace d'ordonnancement appelée BDSC pour les programmes parallèles en présence de contraintes de ressources sur le nombre de processeurs et la taille de leur mémoire locale.

Le processus de parallélisation que nous introduisons dans cette thèse utilise BDSC pour trouver un bon ordonnancement des tâches d'un programme sur les machines cibles et SPIRE pour générer le code source parallèle. Le chapitre 6 couple BDSC avec des modèles de coûts sophistiqués pour donner un nouvel algorithme de parallélisation.

Le chapitre 7 décrit comment nous pouvons générer des codes parallèles équivalents dans les deux langages cibles OpenMP et MPI, sélectionnés lors de l'étude comparative présentée dans le chapitre 3. Grâce à SPIRE, nous montrons comment la génération de code s'avère efficace tout en restant relativement simple.

Pour vérifier l'efficacité et la robustesse de notre travail, des résultats expérimentaux sont présentés dans le chapitre 8. Ils suggèrent que la parallélisation fondée sur SPIRE et BDSC, tout en gérant efficacement des ressources, conduit à des accélérations de parallélisation importantes sur les deux systèmes à mémoire partagée et distribuée. Nous comparons également notre implémentation de la génération de `omp task` dans PIPS avec la

génération de `omp sections` dans le compilateur Faust.

Nous concluons dans le chapitre 9.

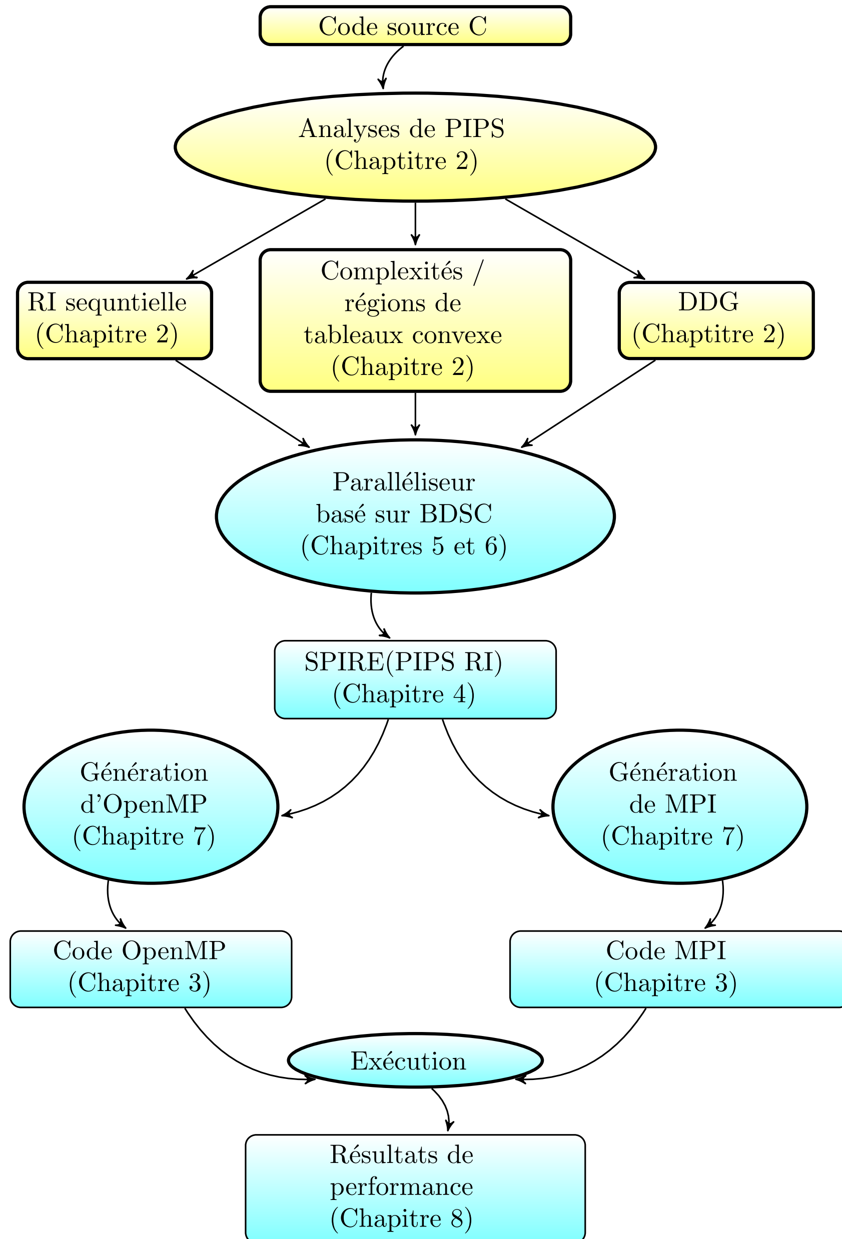


Figure 3: Organisation de la thèse : le bleu indique les contributions de la thèse ; une ellipse, un processus; et un rectangle, résultats

Chapter 1

Introduction

I live and don't know how long, I'll die and don't know when, I am going and don't know where, I wonder that I am happy. **Martinus Von Biberach**

1.1 Context

Moore's law [83] states that, over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years. Recently, the shift from fabricating microprocessors to parallel machines designs was partly because of the growth in power consumption due to high clock speeds, required to improve performance in single processor (or core) chips. The transistor count is still increasing in order to integrate more cores and ensure proportional performance scaling.

In spite of the validity of Moore's law till now, the performance of cores stopped increasing after 2003. Thus, the scalability of applications, which calls for increased performance when resources are added, is not guaranteed. To understand this issue, it is necessary to take a look at Amdahl's law [16]. This law states that the speedup of a program using multiple processors is bounded by the time needed for its sequential part; in addition to the number of processors, the algorithm also limits the speedup.

In order to enjoy the performance benefits that multiprocessors can provide, one should exploit efficiently the parallelism present in applications. This is a tricky task for programmers, especially if this programmer is a physicist or a mathematician or is a computer scientist for whom understanding the application is difficult since it is not his. Of course, we could say to the programmer: "think parallel"! But humans tend to think sequentially. Therefore, one past problem is and will be for some time to detect parallelism in a sequential code and, automatically or not, write its equivalent efficient parallel code.

A parallel application, expressed in a parallel programming model, should be as portable as possible, i.e. one should not have to rewrite parallel code when targeting an other machine. Yet, the proliferation of parallel programming models makes the choice of one general model not obvious, unless there is a parallel language that is efficient and able to be compiled to all types of current and future architectures, which is not yet the case. Therefore, this writing of a parallel code needs to be based on a generic approach, i.e. how well a range of different languages can be handled, in order to offer some chance of portability.

1.2 Motivation

In order to achieve good performance when programming for multiprocessors and go beyond having to "think parallel", a platform for automatic parallelization is required to exploit cores efficiently. Also, the proliferation of multi-core processors with shorter pipelines and lower clock rates and the pressure that the simpler data parallelism model imposes on their memory bandwidth have made coarse grained parallelism inevitable for improving performance.

The first step in a parallelization process is to expose concurrency by decomposing applications into pipelined tasks, since an algorithm is often described as a collection of tasks. To illustrate the importance of task parallelism, we give a simple example: the Harris algorithm [96] which is based on pixelwise autocorrelation, using a chain of functions, namely Sobel, Multiplication, Gauss, and Coarsity. Figure 1.1 shows this process. Several works already parallelized and mapped this algorithm on parallel architectures such as the CELL processor [95].

Figure 1.2 shows a possible instance of a manual partitioning of Harris (see also [95]). The Harris algorithm can be decomposed into a succession of two to three concurrent tasks (ellipses in the figure), assuming no exploitation of data parallelism. Automating this intuitive approach and thus the automatic task extraction and parallelization of such applications is our motivation in this thesis.

However, detecting task parallelism and generating efficient code rest on complex analyses of data dependences, communication, synchronization, etc. These different analyses can be provided by compilation frameworks. Thus, delegating the issue of "parallel thinking" to software is appealing, because it can be built upon existing sophisticated analyses of data dependences, and can handle the granularity present in sequential codes and the resource constraints such as memory size or the number of CPUs that also impact this process.

Automatic task parallelization has been studied around for almost half a century. The problem of extracting an optimal parallelism with optimal communications is NP-complete [47]. Several works have tried to automate the parallelization of programs using different granularities. Metis [62] and other graph partitioning tools aim at assigning the same amount of processor work with small quantities of interprocessor communication, but the structure of the graph does not differentiate between loops, function calls, etc. The crucial steps of graph construction and parallel code generation are missing. Sarkar [98] implements a compile-time method for the partitioning problem for multiprocessors. A program is partitioned into parallel tasks at compile time and then these are merged until a partition with the smallest parallel execution time in the presence of overhead (scheduling and communication overhead) is found. Unfortunately, this algorithm does not

```
void main(int argc, char *argv[]){
 float (*Gx)[N*M], (*Gy)[N*M], (*Ixx)[N*M],
       (*Iyy)[N*M], (*Ixy)[N*M], (*Sxx)[N*M],
       (*Sxy)[N*M], (*Syy)[N*M], (*in)[N*M];
 in = InitHarris();
 /* Now we run the Harris procedure */
 //Sobel
 SobelX(Gx, in);
 SobelY(Gy, in);
 //Multiply
 MultiplY(Ixx, Gx, Gx);
 MultiplY(Iyy, Gy, Gy);
 MultiplY(Ixy, Gx, Gy);
 //Gauss
 Gauss(Sxx, Ixx);
 Gauss(Syy, Iyy);
 Gauss(Sxy, Ixy);
 //Coarsity
 CoarsitY(out, Sxx, Syy, Sxy);
 return;
}
```

Figure 1.1: Sequential C implementation of the `main` function of Harris

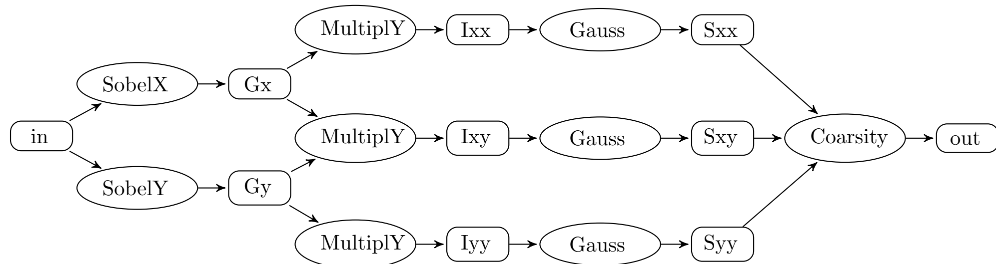


Figure 1.2: Harris algorithm data flow graph

address resource constraints which are important factors for targeting real architectures.

All parallelization tools are dedicated to a particular programming model: there is a lack of a generic abstraction of parallelism (multithreading, synchronization and data distribution). They thus do not usually address the issue of portability.

In this thesis, we develop an automatic task parallelization methodology for compilers: the key characteristics we focus on are resource constraints and static scheduling. It includes the techniques required to decompose applications into tasks and generate equivalent parallel code, using a generic approach that targets different parallel languages and thus different architec-

tures. We apply this methodology in the existing tool PIPS [59], a comprehensive source-to-source compilation platform.

1.3 Contributions

Our goal is to develop a new tool and algorithms for automatic task parallelization that take into account resource constraints and which can also be of general use for existing parallel languages. For this purpose, the main contributions of this thesis are:

1. a new BDSC-based hierarchical scheduling algorithm (HBDSC) that uses:
    - a new data structure, called the Sequence Data Dependence Graph (SDG), to represent partitioned parallel programs,
    - "Bounded DSC" (BDSC), an extension of the DSC [111] scheduling algorithm that simultaneously handles two resource constraints, namely a bounded amount of memory per processor and a bounded number of processors, which are key parameters when scheduling tasks on actual multiprocessors,
    - a new cost model based on execution time complexity estimation, convex polyhedral approximations of data array sizes and code instrumentation for the labeling of SDG vertices and edges;
2. a new approach for the adaptation of automatic parallelization platforms to parallel languages via:
    - SPIRE, a new, simple, parallel intermediate representation (IR) extension methodology for designing the parallel IRs used in compilation frameworks,
    - the deployment of SPIRE for automatic task-level parallelization of explicitly parallel programs on the PIPS [59] compilation framework;
3. an implementation in the PIPS source-to-source compilation framework of:
    - BDSC-based parallelization for programs encoded using SPIRE-based PIPS IR,
    - SPIRE-based parallel code generation into two parallel languages: OpenMP [4] and MPI [3];
4. performance measurements for our parallelization approach, based on:

- five significant programs, targeting both shared and distributed memory architectures: the image and signal processing benchmarks, actually sample constituent algorithms hopefully representative of classes of applications, Harris [54] and ABF [52], the SPEC2001 benchmark equake [22], the NAS parallel benchmark IS [87] and an FFT code [8],
- their automatic translations into two parallel languages: OpenMP [4] and MPI [3],
- and finally, a comparative study between our task parallelization implementation in PIPS and that of the audio signal processing language Faust [88], using two Faust applications: Karplus32 and Freeverb. We choose Faust since it is an open-source compiler and it also generates automatically parallel tasks in OpenMP.

1.4 Thesis Outline

This thesis is organized in nine chapters. Figure 1.3 shows how these chapters can be put into the context of a parallelization chain.

Chapter 2 provides a short background to the concepts of architectures, parallelism, languages and compilers used in the next chapters.

Chapter 3 surveys seven current and efficient parallel programming languages in order to determine and classify their parallel constructs. This helps us to define a core parallel language that plays the role of a generic parallel intermediate representation when parallelizing sequential programs.

Our proposal called the SPIRE methodology (Sequential to Parallel Intermediate Representation Extension) is developed in Chapter 4. SPIRE leverages existing infrastructures to address both control and data parallel languages while preserving as much as possible existing analyses for sequential codes. To validate this approach in practice, we use PIPS, a comprehensive source-to-source compilation platform, as a use case to upgrade its sequential intermediate representation to a parallel one.

Since the main goal of this thesis is automatic task parallelization, extracting task parallelism from sequential codes is a key issue in this process, which we view as a scheduling problem. Chapter 5 introduces a new efficient automatic scheduling heuristic called BDSC (Bounded Dominant Sequence Clustering) for parallel programs in the presence of resource constraints on the number of processors and their local memory size.

The parallelization process we introduce in this thesis uses BDSC to find a good scheduling of program tasks on target machines and SPIRE to generate parallel source code. Chapter 6 equips BDSC with a sophisticated cost model to yield a new parallelization algorithm.

Chapter 7 describes how we can generate equivalent parallel codes in two target languages, OpenMP and MPI, selected from the survey presented in

Chapter 3. Thanks to SPIRE, we show how code generation is simple and efficient.

To verify the efficiency and robustness of our work, experimental results are presented in Chapter 8. They suggest that BDSC- and SPIRE-based parallelization focused on efficient resource management leads to significant parallelization speedups on both shared and distributed memory systems. We also compare our implementation of the generation of `omp task` in PIPS with the generation of `omp sections` in Faust.

We conclude in Chapter 9.

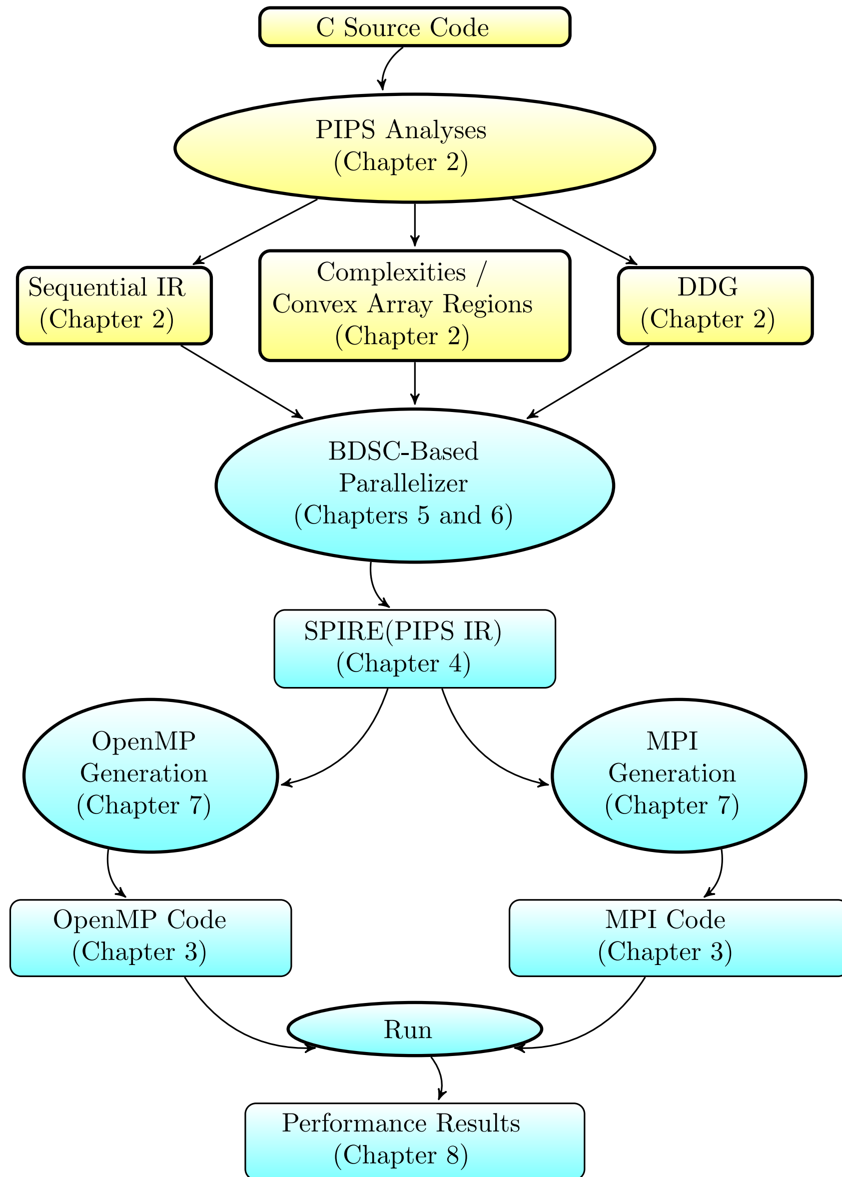


Figure 1.3: Thesis outline: blue indicates thesis contributions; an ellipse, a process; and a rectangle, results

Chapter 2

Perspectives: Bridging Parallel Architectures and Software Parallelism

You talk when you cease to be at peace with your thoughts. **Kahlil Gibran**

"Anyone can build a fast CPU. The trick is to build a fast system." Attributed to Seymour Cray, this quote is even more pertinent when looking at multiprocessor systems that contain several fast processing units; parallel system architectures introduce subtle system features to achieve good performance. Real world applications, which operate on large amounts of data, must be able to deal with constraints such as memory requirements, code size and processor features. These constraints must also be addressed by parallelizing compilers that are related to such applications, from the domains of scientific, signal and image processing, and translate sequential codes into efficient parallel ones. The multiplication of hardware intricacies increases the importance of software in order to achieve adequate performance.

This thesis was carried out under this perspective. Our goal is to develop a prototype for automatic task parallelization that generates a parallel version of an input sequential code using an algorithm of scheduling. We also address code generation and parallel intermediate representation issues. More generally, we study how parallelism can be detected, represented and finally generated.

"Anyone can build a fast CPU. The trick is to build a fast system." Attribuée à Seymour Cray, cette citation[1] *est d'autant plus pertinente quand on considère les systèmes multiprocesseurs qui contiennent plusieurs unités de traitement rapide ; les architectures parallèles présentent des caractéristiques subtiles qu'il convient de bien gérer pour obtenir de bonnes performances. Les applications du monde réel, qui nécessitent de grandes quantités de données, doivent en plus être en mesure de faire face à des contraintes telles que les besoins en mémoire, la taille du code et les fonctionnalités du processeur. Ces contraintes doivent également être prises en compte par*

[1] Tout un chacun peut construire un CPU rapide. Le plus difficile est de construire un système rapide.

les compilateurs de parallélisation, qui, surtout dans les domaines des applications scientifiques et de traitement du signal et d'image, s'efforcent de traduire des codes séquentiels en des codes parallèles efficaces. L'accroissement de la complexité des matériels augmente l'importance du logiciel chargé de les gérer, et ce afin d'obtenir une performance adéquate.

Cette thèse a été réalisée dans la perspective que nous venons d'esquisser. Notre objectif est de développer un prototype logiciel de parallélisation automatique de tâches qui génère une version parallèle d'un code séquentiel d'entrée en utilisant un algorithme d'ordonnancement. Nous abordons également la génération de code parallèle, en particulier les problèmes de représentation intermédiaire parallèle. Plus généralement, nous étudions comment le parallélisme peut être détecté, représenté et finalement généré.

2.1 Parallel Architectures

Up to 2003, the performance of machines has been improved by increasing clock frequencies. Yet, relying on single-thread performance has been increasingly disappointing due to access latency to main memory. Moreover, other aspects of performance have been of importance lately, such as power consumption, energy dissipation, and number of cores. For these reasons, the development of parallel processors, with a large number of cores that run at slower frequencies to reduce energy consumption, is adopted and has had a significant impact on performance.

The market of parallel machines is wide, from vector platforms, accelerators, Graphical Processing Unit (GPUs) and dualcore to manycore[2] with thousands of on-chip cores. However, gains obtained when parallelizing and executing applications are limited by input-output issues (disk read/write), data communications among processors, memory access time, load unbalance, etc. When parallelizing an application, one must detect the available parallelism and map different parallel tasks on the parallel machine. In this dissertation, we consider a task as a static notion, i.e., a list of instructions, while processes and threads (see Section 2.1.2) are running instances of tasks.

In our parallelization approach, we make trade-offs between the available parallelism and communications/locality; this latter point is related to the memory model in the target machine. Besides, extracted parallelism should be correlated to the number of processors/cores available in the target machine. We also take into account the memory size and the number of processors.

[2]Contains 32 or more cores.

2.1.1 Multiprocessors

The generic model of multiprocessors is a collection of cores, each potentially including both CPU and memory, attached to an interconnection network. Figure 2.1 illustrates an example of a multiprocessor architecture that contains two clusters; each cluster is a quadcore multiprocessor. In a multicore, cores can transfer data between their cache memory (level 2 cache generally) without having to go through the main memory. Moreover, they may or may not share caches (CPU-local level 1 cache in Figure 2.1 for example). This is not possible at the cluster level, where processor cache memories are not linked. Data can be transfered to and from the main memory only.

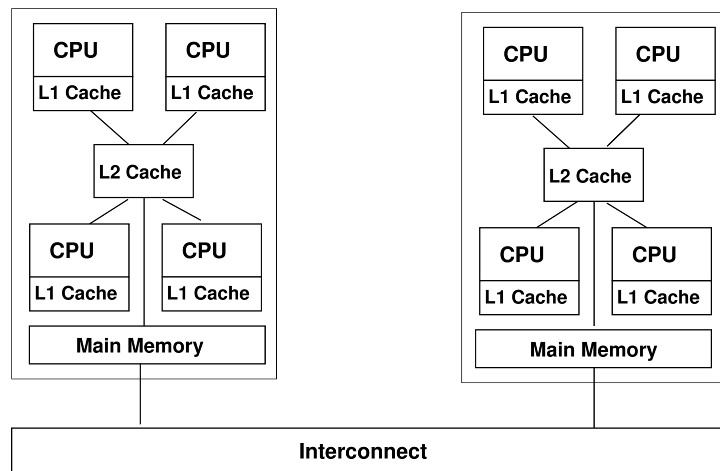


Figure 2.1: A typical multiprocessor architectural model

Flynn [46] classifies multiprocessors in three main groups: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). Multiprocessors are generally recognized to be MIMD architectures. We present the important branches of current multiprocessors: MPSoCs and multicore processors, since they are widely used in many applications such as signal processing and multimedia.

Multiprocessors System-on-Chip

A multiprocessor system-on-chip (MPSoC) integrates multiple CPUs in a hardware system. MPSoCs are commonly used for applications such as embedded multimedia on cell phones. The first MPSoC often considered to be is the Lucent Daytona [9]. One distinguishes two important forms of MPSoCs: homogeneous and heterogeneous multiprocessors. Homogeneous architectures include multiple cores that are similar in every aspect. On the other side, heterogeneous architectures use processors that are different in terms of the instruction set architecture, functionality, frequency, etc. Field

Programmable Gate Array (FPGA) platforms and the CELL processor[3] [43] are examples of heterogeneous architectures.

Multicore Processors

This model implements the shared memory model. The programming model in a multicore processor is the same for all its CPUs. The first multicore general purpose processors were introduced in 2005 (Intel Core Duo). Intel Xeon Phi is a recent multicore chip with 60 homogeneous cores. The number of cores on a chip is expected to continue to grow to construct more powerful computers. This emerging hardware approach will deliver petaflop and exaflop performance with the efficient capabilities needed to handle high-performance emerging applications. Table 2.1 illustrates the rapid proliferation of multicores and summarizes the important existing ones; note that IBM Sequoia, the second system in the Top500 list (`top500.org`) in November 2012, is a petascale Blue Gene/Q supercomputer that contains 98,304 compute nodes where each computing node is a 16-core PowerPC A2 processor chip (sixth entry in the table). See [25] for an extended survey of multicore processors.

Year	Number of cores	Company	Processor
The early 1970s	The 1st microprocessor	Intel	4004
2005	2-core machine	Intel	Xeon Paxville DP
2005	2	AMD	Opteron E6-265
2009	6	AMD	Phenom II X6
2011	8	Intel	Xeon E7-2820
2011	16	IBM	PowerPC A2
2007	64	Tilera	TILE64
2012	60	Intel	Xeon Phi

Table 2.1: Multicores proliferation

In this thesis, we provide tools that automatically generate parallel programs from sequential ones, targeting homogeneous multiprocessors or multicores. Since increasing the number of cores on a single chip creates challenges with memory and cache coherence, as well as communication between the cores, our aim in this thesis is to deliver the benefit and efficiency of multicore processors, by the introduction of an efficient scheduling algorithm, to map parallel tasks of a program on CPUs, in the presence of resource constraints on the number of processors and their local memory size.

[3]The Cell configuration has one Power Processing Element (PPE) on the core, acting as a controller for eight physical Synergistic Processing Elements (SPEs).

2.1.2 Memory Models

The choice of a proper memory model to express parallel programs is an important issue in parallel language design. Indeed, the ways processes and threads communicate using the target architecture and impact the programmer's computation specifications affect both performance and ease of programming. There are currently three main approaches (see Figure 2.2 extracted from [64]).

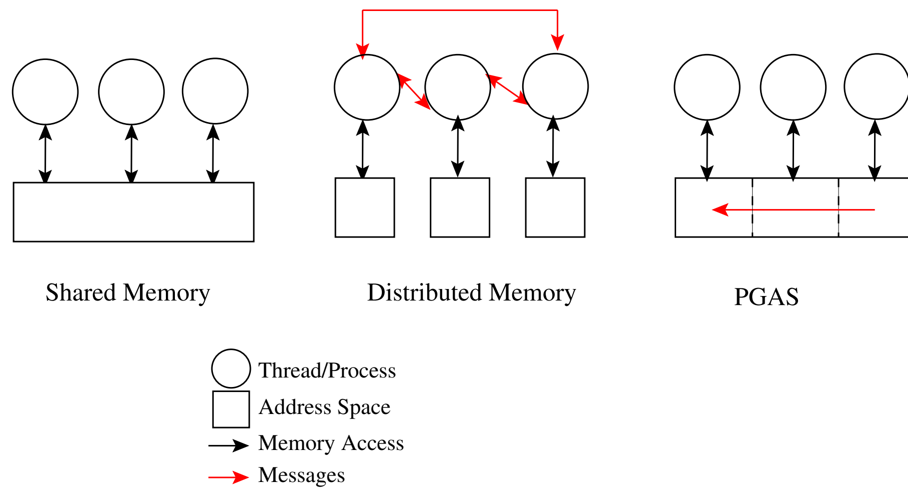


Figure 2.2: Memory models

Shared Memory

Also called global address space, this model is the simplest one to use [11]. Here, the address spaces of the threads are mapped onto the global memory; no explicit data passing between threads is needed. However, synchronization is required between the threads that are writing and reading the same data to and from the shared memory. The OpenMP [4] and Cilk [26] languages use the shared memory model. Intel's Larrabee [101] is homogeneous general-purpose many-core machine with cache-coherent shared memory.

Distributed Memory

In a distributed-memory environment, each processor is only able to address its own memory. The message passing model that uses communication libraries is usually adopted to write parallel programs for distributed-memory systems. These libraries provide routines to initiate and configure the messaging environment as well as sending and receiving data packets. Currently, the most popular high-level message-passing system for scientific and engineering applications is MPI (Message Passing Interface) [3]. OpenCL [70]

uses a variation of the message passing memory model for GPUs. Intel Single-chip Cloud Computer (SCC) [56] is a homogeneous, general-purpose, many-core (48 cores) chip implementing the message passing memory model.

Partitioned Global Address Space (PGAS)

PGAS-based languages combine the programming convenience of shared memory with the performance control of message passing by partitioning logically a global address space into places; each thread is local to a place. From the programmer's point of view programs have a single address space and one task of a given thread may refer directly to the storage of a different thread. UPC [34], Fortress [13], Chapel [6], X10 [5] and Habanero-Java [30] use the PGAS memory model.

In this thesis, we target both shared and distributed memory systems since efficiently allocating the tasks of an application on a target architecture requires reducing communication overheads and transfer costs for both shared and distributed memory architectures. By convention, we call running instances of tasks, processes, in the case of distributed memory systems, and threads, for shared memory and PGAS systems.

If reducing communications is obviously meaningful for distributed memory systems, it is also worthwhile on shared memory architectures since this may impact cache behavior. Also, locality is an other important issue; a parallelization process should be able to make trade-offs between locality in cache memory and communications. Indeed, mapping two tasks to the same processor keeps the data in its local memory, and even possibly its cache. This avoids its copying over the shared memory bus. Therefore, transmission costs are decreased and bus contention is reduced.

Moreover, we take into account the memory size, since this is an important factor when parallelizing applications that operate on large amount of data or when targeting embedded systems.

2.2 Parallelism Paradigms

Parallel computing has been looked at since computers exist. The market dominance of multi- and many-core processors and the growing importance and increasing number of clusters in the Top500 list are making parallelism a key concern when implementing large applications such as weather modeling [40] or nuclear simulations [37]. These important applications require a lot of computational power and thus need to be programmed to run on parallel supercomputers.

Parallelism expression is the fundamental issue when breaking an application into concurrent parts in order to take advantage of a parallel computer

and to execute these simultaneously on different CPUs. Parallel architectures support multiple levels of parallelism within a single chip: the PE (Processing Element) level [65], the thread level (e.g. Symmetric multiprocessing (SMT)), the data level (e.g. Single Instruction Multiple Data (SIMD)), the instruction level (e.g. Very Long Instruction Word (VLIW)), etc. We present here the three main types of parallelism (data, task and pipeline).

Task Parallelism

This model corresponds to the distribution of the execution processes or threads across different parallel computing cores/processors. A task can be defined as a unit of parallel work in a program. We can classify its implementation of scheduling in two categories: dynamic and static.

- With dynamic task parallelism, tasks are dynamically created at runtime and added to the work queue. The runtime scheduler is responsible for scheduling and synchronizing the tasks across the cores.
- With static task parallelism, the set of tasks is known statically and the scheduling is applied at compile time; the compiler generates and assigns different tasks for the thread groups.

Data Parallelism

This model implies that the same independent instruction is performed repeatedly and simultaneously on different data. Data parallelism or loop level parallelism corresponds to the distribution of the data across different parallel cores/processors, dividing the loops into chunks[4]. Data-parallel execution puts a high pressure on the memory bandwidth of multi-core processors. Data parallelism can be expressed using task parallelism constructs; an example is provided in Figure 2.3 where loop iterations are distributed differently among the threads: contiguous order generally in the left and interleaved order in the right.

```
forall (i = 0; i < height; ++i)
  for (j = 0; j < width; ++j)
    kernel();
```

(a) Data parallel example

```
for (i = 0; i < height; ++i) {
  launchTask{
    for (j = 0; j < width; ++j)
      kernel();
  }
}
```

(b) Task parallel example

Figure 2.3: Rewriting of data parallelism using the task parallelism model

[4] A chunk is a set of iterations.

Pipeline Parallelism

Pipeline parallelism applies to chains of producers and consumers that contain many steps depending on each other. These dependences can be removed by interleaving these steps. Pipeline parallelism execution leads to reduced latency and buffering but it introduces extra synchronization between producers and consumers to maintain them coupled in their execution.

In this thesis we focus on task parallelism, often considered more difficult to handle than data parallelism, since it lacks the regularity present in the latter model; processes (threads) run simultaneously different instructions, leading to different execution schedules and memory access patterns. Besides, the pressure of the data parallelism model on memory bandwidth and the extra synchronization introduced in the pipeline parallelism have made coarse grained parallelism inevitable for improving performance.

Moreover, data parallelism can be implemented using the task parallelism (see Figure 2.3). Therefore, our implementation of task parallelism is also able to handle data parallelism after transformation of the sequential code (see Section 8.4 that explains the protocol applied to task parallelize sequential applications).

Task management must address both control and data dependences in order to reduce execution and communication times. This is related to another key enabling issue, compilation, which we detail in the following section.

2.3 Mapping Paradigms to Architectures

In order to map efficiently a parallel program on a parallel architecture, one needs to write parallel programs in the presence of constraints of data and control dependences, communications and resources. Writing a parallel application implies the use of a target parallel language; we survey in Chapter 3 different, recent and popular task parallel languages. We can split this parallelization process into two steps: understanding the sequential application, by analyzing it, to extract parallelism, and then generating the equivalent parallel code. Before reaching the writing step, decisions should be made about an efficient task schedule (Chapter 5). To perform these operations automatically, a compilation framework is necessary to provide a base for analyses, implementation and experimentation. Moreover, to keep this parallel software development process simple, generic and language-independent, a general parallel core language is needed (Chapter 4).

Programming parallel machines requires a parallel programming language with one of the previous paradigms of parallelism. Since in this thesis we focus on extracting and generating task parallelism, we are interested in task parallel languages, which we detail in Chapter 3. We distinguish two

approaches for mapping languages to parallel architectures. In the first case, the entry is a parallel language: it means the user wrote his parallel program by hand; this requires a great subsequent effort for detecting dependences and parallelism and writing an equivalent efficient parallel program. In the second case, the entry is a sequential language; this calls for an automatic parallelizer which is responsible for dependence analyses and scheduling in order to generate an efficient parallel code.

In this thesis, we adopt the second approach: we use the PIPS compiler presented in Section 2.6 as a practical tool for code analysis and parallelization. We target both shared and distributed memory systems using two parallel languages: OpenMP and MPI. They provide high-level programming constructs to express task creation, termination, synchronization, etc. In Chapter 7, we show how we generate parallel code for these two languages.

2.4 Program Dependence Graph (PDG)

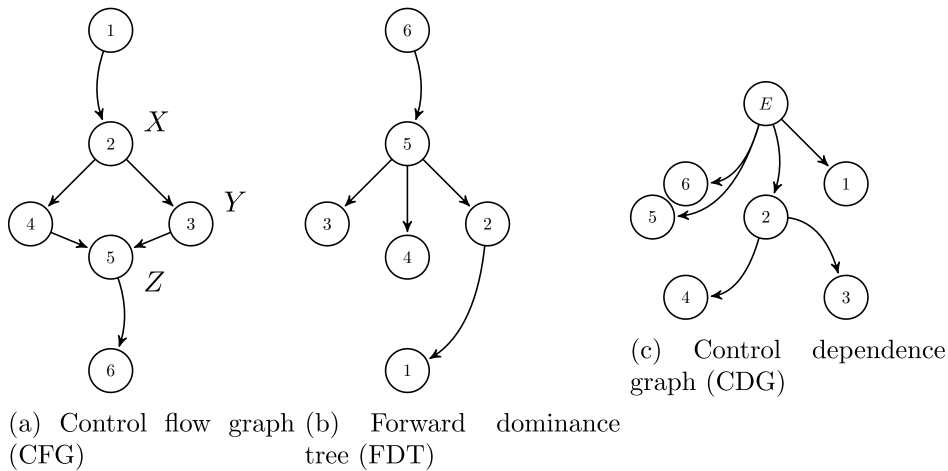
Parallelism detection is based on the analyses of data and control dependences between instructions encoded in an intermediate representation format. Parallelizers use various dependence graphs to represent such constraints. A program dependence graph (PDG) is a directed graph where the vertices represent blocks of statements and the edges represent essential control or data dependences. It encodes both control and data dependence information.

2.4.1 Control Dependence Graph (CDG)

The CDG represents a program as a graph of statements that are control dependent on the entry to the program. Control dependences represent control flow relationships that must be respected by any execution of the program, whether parallel or sequential.

Ferrante et al [45] define control dependence as follows: Statement Y is control dependent on X when there is a path in the control flow graph (CFG) (see Figure 2.4a) from X to Y that does not contain the immediate forward dominator Z of X. Z forward dominates Y if all paths from Y include Z. The immediate forward dominator is the first forward dominator (closest to X). In the forward dominance tree (FDT) (see Figure 2.4b), vertices represent statements; edges represent the immediate forward dominance relation; the root of the tree is the exit of the CFG. For the CDG (see Figure 2.4c) construction, one relies on explicit control flow and postdominance information. Unfortunately, when programs contain *goto* statements, explicit control flow and postdominance information are prerequisites for calculating control dependences.

Harrold et al [55] have proposed an algorithm that does not require explicit control flow or postdominator information to compute exact control



(a) Control flow graph (CFG)

(b) Forward dominance tree (FDT)

(c) Control dependence graph (CDG)

Figure 2.4: Construction of the control dependence graph

dependences. Indeed, for structured programs, the control dependences are obtained directly from Abstract Syntax Tree (AST). Programs that contain what are called by the authors structured transfers of control such as `break`, `return` or `exit` can be handled as special cases [55]. For more unstructured transfers of control, via arbitrary `goto`s, additional processing based on the control flow graph needs to be performed.

In this thesis, we only handle structured parts of a code, i.e. the ones that do not contain `goto` statements. Therefore, regarding this context, PIPS implements control dependences in its IR since it is in form of an AST as above (for structured programs CDG and AST are equivalent).

2.4.2 Data Dependence Graph (DDG)

The DDG is a subgraph of the program dependence graph that encodes data dependence relations between instructions that use/define the same data element. Figure 2.5 shows an example of a C code and its data dependence graph. This DDG has been generated automatically with PIPS. Data dependences in a DDG are basically of three kinds [90] as shown in the figure:

- true dependences or true data-flow dependences, when a value of an instruction is used by a subsequent one (aka RAW), colored red in the figure;
- output dependences, between two definitions of the same item (aka WAW), colored blue in the figure;
- anti-dependences, between a use of an item and a subsequent write (aka WAR), colored green in the figure.

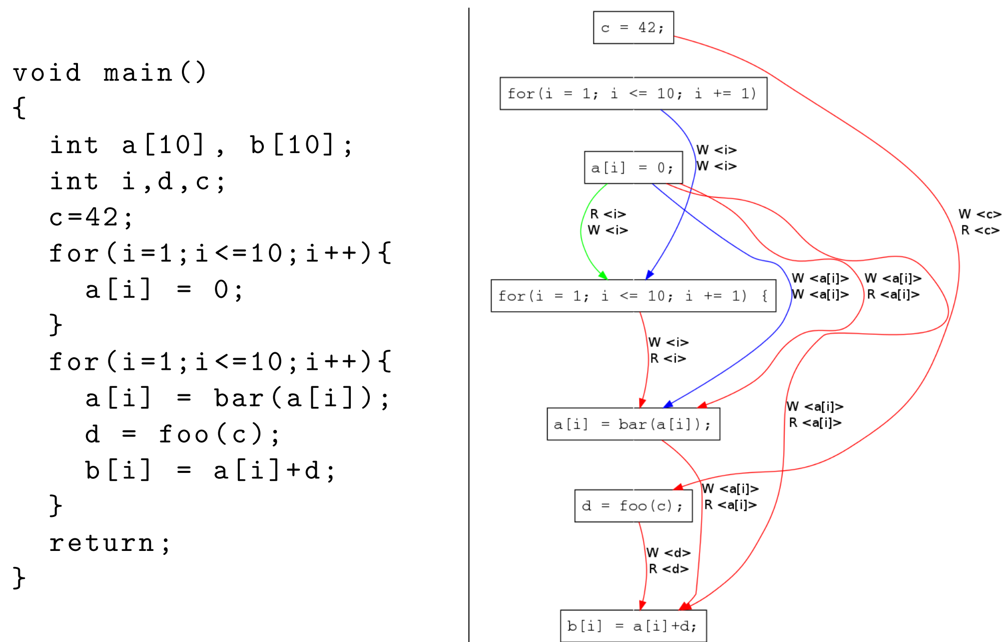


Figure 2.5: Example of a C code and its data dependence graph

Dependences are needed to define precedence constraints in programs. PIPS implements both control (its IR in form of an AST) and data (in form of a dependence graph) dependences. These data structures are the basis for scheduling instructions, an important step in any parallelization process.

2.5 List Scheduling

Task scheduling is the process of specifying the order and assignment of start and end times to a set of tasks, to be run concurrently on a multiprocessor, such that the completion time of the whole application is as small as possible while respecting the dependence constraints of each task. Usually, the number of tasks exceeds the number of processors; thus some processors are dedicated to multiple tasks. Since finding the optimal solution of a general scheduling problem is NP-complete [47], providing an efficient heuristic to find a good solution is needed. This is a problem we address in Chapter 5.

2.5.1 Background

Scheduling approaches can be categorized in many ways. In preemptive techniques, the current executing task can be preempted by an other higher-priority task, while, in a non-preemptive scheme, a task keeps its processor until termination. Preemptive scheduling algorithms are instrumental in avoiding possible deadlocks and for implementing real-time systems, where

tasks must adhere to specified deadlines; however, preemptions generate run-time overhead. Moreover, when static predictions of task characteristics such as execution time and communication cost exist, task scheduling can be performed statically (offline). Otherwise, dynamic (online) schedulers must make run-time mapping decisions whenever new tasks arrive; this also introduces run-time overhead.

In the context of the automatic parallelization of scientific applications we focus on in this thesis, we are interested in non-preemptive static scheduling policies of parallelized code[5]. Even though the subject of static scheduling is rather mature (see Section 5.4), we believe the advent and widespread use of multi-core architectures, with the constraints they impose, warrant to take a fresh look at its potential. Indeed, static scheduling mechanisms have, first, the strong advantage of reducing run-time overheads, a key factor when considering execution time and energy usage metrics. One other important advantage of these schedulers over dynamic ones, at least over those not equipped with detailed static task information, is that the existence of efficient schedules is ensured prior to program execution. This is usually not an issue when time performance is the only goal at stake, but much more so when memory constraints might prevent a task being executed at all on a given architecture. Finally, static schedules are predictable, which helps both at the specification (if such a requirement has been introduced by designers) and debugging levels. Given the breadth of the literature on scheduling, we introduce in this section the notion of list-scheduling heuristics [74], since we use it in this thesis, which is a class of scheduling heuristics.

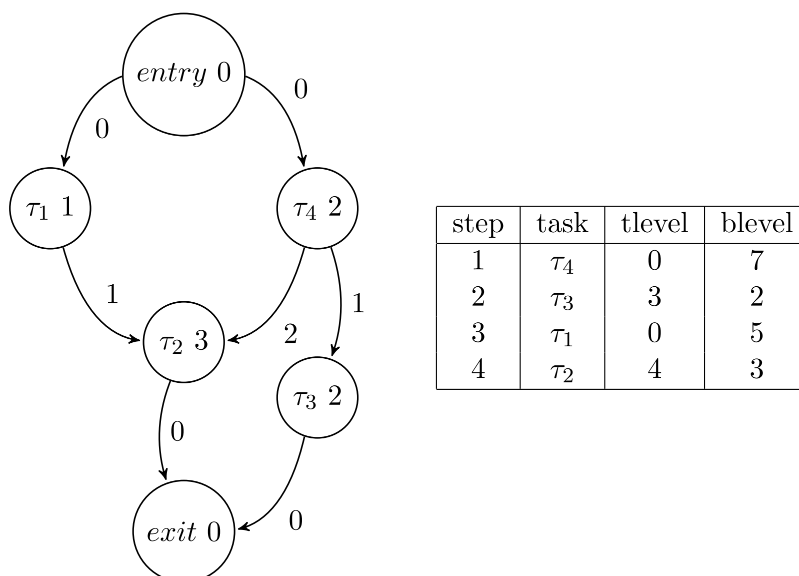
2.5.2 Algorithm

A labeled direct acyclic graph (DAG) G is defined as $G = (T, E, C)$, where (1) $T =$ `vertices`(G) is a set of n tasks (vertices) τ annotated with an estimation of their execution time `task_time`(τ), (2) $E = edges(G)$, a set of m directed edges $e = (\tau_i, \tau_j)$ between two tasks annotated with their dependences `edge_regions`(e)[6], and (3) C, a $n \times n$ sparse communication edge cost matrix `edge_cost`(e); `task_time`(τ) and `edge_cost`(e) are assumed to be numerical constants, although we show how we lift this restriction in Section 6.3. The functions `successors`$(\tau,\ G)$ and `predecessors`$(\tau,\ G)$ return the list of immediate successors and predecessors of a task τ in the DAG G. Figure 2.6 provides an example of a simple graph, with vertices τ_i; vertex times are listed in the vertex circles while edge costs label arrows.

A list scheduling process provides, from a DAG G, a sequence of its vertices that satisfies the relationship imposed by E. Various heuristics try

[5] Note that more expensive preemptive schedulers would be required if fairness concerns were high, which is not frequently the case in the applications we address here.

[6] A region is defined in Section 2.6.3.



step	task	tlevel	blevel
1	τ_4	0	7
2	τ_3	3	2
3	τ_1	0	5
4	τ_2	4	3

Figure 2.6: A Directed Acyclic Graph (left) and its associated data (right)

to minimize the schedule total length, possibly allocating the various vertices in different clusters, which ultimately will correspond to different processes or threads. A cluster κ is thus a list of tasks; if $\tau \in \kappa$, we note `cluster`(τ) $= \kappa$. List scheduling is based on the notion of vertex priorities. The priority for each task τ is computed using the following attributes:

- The top level `tlevel`(τ, G) of a vertex τ is the length of the longest path from the entry vertex[7] of G to τ. The length of a path is the sum of the communication cost of the edges and the computational time of the vertices along the path. Top levels are used to estimate the start times of vertices on processors: the top level is the earliest possible start time. Scheduling in an ascending order of top levels schedules vertices in a topological order. The algorithm for computing the top level of a vertex τ in a graph is given in Algorithm 1.

- The bottom level `blevel`(τ, G) of a vertex τ is the length of the longest path from τ to the exit vertex of G. The maximum of the bottom level of vertices is the length `cpl`(G) of a graph's critical path, which has the longest path in the DAG G. The latest start time of a vertex τ is the difference (`cpl`$(G)-$`blevel`(τ, G)) between the critical path length and the bottom level of τ. Scheduling in a descending order of bottom levels tends to schedule critical path vertices first. The algorithm for computing the bottom level of τ in a graph is given in Algorithm 2.

[7] One can always assume that a unique entry vertex exists, by adding it if need be and connecting it to the original ones with null-cost edges. This remark also applies to the exit vertex (see Figure 2.6).

ALGORITHM 1: The top level of Task τ in Graph G

```
function tlevel(τ, G)
  tl = 0;
  foreach τ_i ∈ predecessors(τ, G)
    level = tlevel(τ_i, G)+task_time(τ_i)+edge_cost(τ_i, τ);
    if (tl < level) then tl = level;
  return tl;
end
```

ALGORITHM 2: The bottom level of Task τ in Graph G

```
function blevel(τ, G)
  bl = 0;
  foreach τ_j ∈ successors(τ, G)
    level = blevel(τ_j, G)+edge_cost(τ, τ_j);
    if (bl < level) then bl = level;
  return bl+task_time(τ);
end
```

To illustrate these notions, the top levels and bottom levels of each vertex of the graph presented in the left of Figure 2.6 are provided in the adjacent table.

The general algorithmic skeleton for list scheduling a graph G on P clusters (P can be infinite and is assumed to be always strictly positive) is provided in Algorithm 3: first, priorities `priority`(τ) are computed for all currently unscheduled vertices; then, the vertex with the highest priority is selected for scheduling; finally, this vertex is allocated to the cluster that offers the earliest start time. Function f characterizes each specific heuristic, while the set of clusters already allocated to tasks is $clusters$. Priorities need to be computed again for (a possibly updated) graph G after each scheduling of a task: task times and communication costs change when tasks are allocated to clusters. This is performed by the `update_priority_values` function call. Part of the `allocate_task_to_cluster` procedure is to ensure that `cluster`(τ) = κ, which indicates that Task τ is now scheduled on Cluster κ.

In this thesis, we introduce a new non-preemptive static list-scheduling heuristic, called BDSC (Chapter 5), to extract task-level parallelism in the presence of resource constraints on the number of processors and their local memory size; it is based on a precise cost model and addresses both shared and distributed parallel memory architectures. We implement BDSC in PIPS.

ALGORITHM 3: List scheduling of Graph G on P processors

```
procedure list_scheduling(G, P)
  clusters = ∅;
  foreach τi ∈ vertices(G)
    priority(τi) = f(tlevel(τi, G), blevel(τi, G));
  UT = vertices(G);            // unscheduled tasks
  while UT ≠ ∅
    τ = select_task_with_highest_priority(UT);
    κ = select_cluster(τ, G, P, clusters);
    allocate_task_to_cluster(τ, κ, G);
    update_graph(G);
    update_priority_values(G);
    UT = UT-{τ};
end
```

2.6 PIPS: Automatic Parallelizer and Code Transformation Framework

We chose PIPS [59] to showcase our parallelization approach, since it is readily available, well-documented and encodes both control and data dependences. PIPS is a source-to-source compilation framework for analyzing and transforming C and Fortran programs. PIPS implements polyhedral analyses and transformations that are inter-procedural and provides detailed static information about programs such as use-def chains, data dependence graph, symbolic complexity, memory effects[8], call graph, interprocedural control flow graph, etc.

Many program analyses and transformations can be applied; the most important are listed below.

Parallelization Several algorithms are implemented such as Allen & Kennedy's parallelization algorithm [14] that performs loop distribution and vectorization selecting innermost loops for vector units, and coarse grain parallelization, based on the convex array regions of loops (see below) to avoid loop distribution and restrictions on loop bodies.

Scalar and array privatization This analysis discovers variables whose values are local to a particular scope, usually a loop iteration.

Loop transformations [90] These include unrolling, interchange, normalization, distribution, strip mining, tiling, index set splitting...

Function transformations PIPS supports for instance the inlining, outlining and cloning of functions.

[8] A representation of data read and written by a given statement.

Other transformations These classical code transformations include dead-code elimination, partial evaluation, 3-address code generation, control restructuring, loop invariant code motion, forward substitution, etc.

We use many of these analyses for implementing our parallelization methodology in PIPS, e.g. "complexity" and data dependence analyses. These static analyses use convex polyhedra to represent abstractions of the values within various domains . A convex polyhedron over $\mathbb{Z}$ is a set of integer points in the n-dimensional space $\mathbb{Z}^n$, and can be represented by a system of linear inequalities. These analyses give impressive information and results which can be leveraged to perform automatic task parallelization (Chapter 6).

We present below the precondition, transformer, array region and complexity analyses that are directly used in this thesis to compute dependences, execution and communication times; then, we present the intermediate representation (IR) of PIPS, since we extend it in this thesis to a parallel intermediate representation (Chapter 4), which is necessary for simple and generic parallel code generation.

2.6.1 Transformer Analysis

A transformer is an affine relation between variables values, in the memory store, before (old values) and after (new values) the execution of a statement. This analysis [59] is implemented in PIPS where transformers are represented by convex polyhedra. Indeed, a transformer T is defined by a list of arguments and a predicate system labeled by T; a transformer is empty ($\perp$) when the set of the affine constraints of the system is not feasible; a transformer is equal to the identity, when no variables are modified.

This relation models the transformation, by the execution of a statement, of an input memory store into an output memory store. For instance, in Figure 2.7, the transformer of k, $T(k)$, before and after the execution of the statement `k++;` is the relation $\{k = k\#\text{init}+1\}$, where $k\#\text{init}$ is the value before executing the increment statement; its value before the loop is $k\#\text{init} = 0$.

The Transformer analysis is used to compute preconditions (see next section, Section 2.6.2) and to compute the Path Transformer (see Section 6.4) that connects two memory stores. This analysis is necessary for our dependence test.

2.6.2 Precondition Analysis

A precondition analysis is a function that labels each statement in a program with its precondition. A precondition provides information about the values of variables; it is a condition that holds true before the execution of the statement.

```
                                // T(i,k) {k = 0}
int i, k=0;
                                // T(i,k) {i = k + 1, k#init= 0}
for(i = 1; i<= 10; i++) {
                                // T() {}
  a[i] = f(a[i]);
                                // T(k) {k = k#init+1}
  k++;
}
```

Figure 2.7: Example of transformer analysis

This analysis [59] is implemented in PIPS, where preconditions are represented by convex polyhedra. Preconditions are determined before the computing of array regions (see next section, Section 2.6.3), in order to have precise information on the variables values. They are also used to perform a sophisticated static complexity analysis (see Section 2.6.4). In Figure 2.8, for instance, the precondition P over the variables i and j before the second loop is $\{i = 11\}$, which is indeed the value of i after the execution of the first loop. Note that no information is available for j at this step.

```
int i,j;
                                // P(i,j) {}
for(i = 1; i <= 10; i++)
                                // P(i,j) {1 ≤ i, i ≤ 10}
  a[i]=f(a[i]);
                                // P(i,j) {i = 11}
for(j = 6; j <= 20; j++)
                                // P(i,j) {i = 11, 6 ≤ j, j ≤ 20}
  b[j]=g(a[j],a[j+1]);
                                // P(i,j) {i = 11, j = 21}
...
```

Figure 2.8: Example of precondition analysis

2.6.3 Array Region Analysis

PIPS provides a precise intra- and inter-procedural analysis of array data flow. This latter is important for computing dependences for each array element. This analysis of array regions [36] is implemented in PIPS. A region r of an array T is an abstraction of a subset of its elements. Formally, a region r is a quadruplet of (1) a reference v (variable referenced in the

region), (2) a region type t, (3) an approximation a for the region, *EXACT* when the region exactly represents the requested set of array elements or *MAY* if it is an over-or under-approximation, and (4) c a convex polyhedron containing equalities and inequalities where parameters are the variables values in the current memory store. We write $< v - t - a - c >$ for Region r.

We distinguish four types of sets of regions: Rr, Rw, Ri and Ro. *Read* Rr and *Write* Rw regions contain the array elements respectively read or written by a statement. *In* regions Ri contain the arrays elements read and imported by a statement. *Out* regions Ro contain the array elements written and exported by a statement.

For instance, in Figure 2.9, PIPS is able to infer the following sets of regions:

$$
\begin{aligned}
Rw_1 = \{ &< a(\phi_1) - W - \ EXACT - \{1 \leq \phi_1, \phi_1 \leq 10\} >, \\
&< b(\phi_1) - W - \ EXACT - \{1 \leq \phi_1, \phi_1 \leq 10\} > \} \\
Rr_2 = \{ &< a(\phi_1) - R - \ EXACT - \{6 \leq \phi_1, \phi_1 \leq 21\} > \}
\end{aligned}
$$

where the Write regions Rw_1 of arrays a and b, modified in the first loop, are EXACTly the array elements of a with indices in the interval. The Read regions Rr_2 of arrays a in the second loop represents EXACTly the elements with indices in [6,21].

```
                                // < a(φ1) − R − EXACT − {1 ≤ φ1, φ1 ≤ 10} >
                                // < a(φ1) − W − EXACT − {1 ≤ φ1, φ1 ≤ 10} >
                                // < b(φ1) − W − EXACT − {1 ≤ φ1, φ1 ≤ 10} >
for(i = 1; i <= 10; i++) {
  a[i] = f(a[i]);
  b[i] = 42;
}
                                // < a(φ1) − R − EXACT − {6 ≤ φ1, φ1 ≤ 21} >
                                // < b(φ1) − W − EXACT − {6 ≤ φ1, φ1 ≤ 20} >
for(j = 6; j <= 20; j++)
  b[j]=g(a[j],a[j+1]);
```

Figure 2.9: Example of array region analysis

We define the following set of operations on set of regions R_i (convex polyhedra).

1. `regions_intersection`(R_1,R_2) is a set of regions; each region r in this set is the intersection of two regions $r_1 \in R_1$ of type t_1 and $r_2 \in R_2$ of type t_2 with the same reference. The type of r is $t_1 t_2$ and the convex

polyhedron of r is the intersection of the two convex polyhedra of r_1 and r_2 which is also a convex polyhedron.

2. `regions_difference(`R_1`,`R_2`)` is a set of regions; each region r in this set is the difference of two regions $r_1 \in R_1$ and $r_2 \in R_2$ with the same reference. The type of r is t_1t_2 and the convex polyhedron of r is the set difference between the two convex polyhedra of r_1 and r_2. Since this is not generally a convex polyhedron, an under approximation is computed in order to obtain a convex representation.

3. `regions_union(`R_1`,`R_2`)` is a set of regions; each region r in this set is the union of two regions $r_1 \in R_1$ of type t_1 and $r_2 \in R_2$ of type t_2 with the same reference. The type of r is t_1t_2 and the convex polyhedron of r is the union of the two convex polyhedra r_1 and r_2, which is again not necessarily a convex polyhedron. An approximated convex hull is thus computed in order to return the smallest enclosing polyhedron.

In this thesis, array regions are used for communication cost estimation, dependence computation between statements, and data volume estimation for statements. For example, if we need to compute array dependences between two compound statements S_1 and S_2, we search for the array elements that are accessed in both statements. Therefore, we are dealing with two statements set of regions R_1 and R_2; the result is the intersection between the two sets of regions. However, two sets of regions should be in the same memory store to make it possible to compare them; we should to bring a set of region R_1 to the store of R_2 by combining R_1 with the changes performed, i.e. the transformer that connects the two memory stores. Thus, we compute the path transformer between S_1 and S_2 (see Section 6.4). Moreover, we use operations on array regions for our cost model generation (see Section 6.3) and communications generation (see Section 7.3).

2.6.4 "Complexity" Analysis

"Complexities" are symbolic approximations of the static execution time estimation of statements in term of number of cycles. They are implemented in PIPS using polynomial approximations of execution times. Each statement is thus labeled with an expression, represented by a polynomial over program variables, assuming that each basic operation (addition, multiplication...) has a fixed, architecture-dependent execution time. Table 2.2 shows the complexities formulas generated for each function of Harris (see the code in Figure 1.1) using PIPS complexity analysis, where the N and M variables represent the input image size.

We use this analysis provided by PIPS to determine an approximate execution time for each statement in order to guide our scheduling heuristic.

Function	Complexity (polynomial)
`InitHarris`	$9 \times N \times M$
`SobelX`	$60 \times N \times M$
`SobelY`	$60 \times N \times M$
`MultiplY`	$17 \times N \times M$
`Gauss`	$85 \times N \times M$
`CoarsitY`	$34 \times N \times M$

Table 2.2: Execution time estimations for Harris functions using PIPS complexity analysis

2.6.5 PIPS (Sequential) IR

PIPS intermediate representation (IR) [33] of sequential programs is a hierarchical data structure that embeds both control flow graphs and abstract syntax trees. We provide in this section a high-level description of the intermediate representation of PIPS, which targets the Fortran and C imperative programming languages; it is specified using Newgen [61], a Domain Specific Language for the definition of set equations, from which a dedicated API is automatically generated to manipulate (creation, access, IO operations...) data structures implementing these set elements. This section contains only a slightly simplified subset of the intermediate representation of PIPS, the part that is directly related to the parallel paradigms addressed in this thesis. The Newgen definition of this part is given in Figure 2.10:

```
instruction  = call + forloop + sequence + unstructured;
statement    = instruction x declarations:entity*;
entity       = name:string x type x initial:value;
call         = function:entity x arguments:expression*;
forloop      = index:entity x
               lower:expression x upper:expression x
               step:expression x body:statement;
sequence     = statements:statement*;
unstructured = entry:control x exit:control;
control      = statement x
               predecessors:control* x successors:control*;
expression   = syntax x normalized;
syntax       = reference + call + cast;
reference    = variable:entity x indices:expression*;
type         = area + void;
```

Figure 2.10: Simplified Newgen definitions of the PIPS IR

- Control flow in PIPS IR is represented via instructions, members of the disjoint union (using the "+" symbol) set `instruction`. An instruction

can be either a simple call or a compound instruction, i.e., a `for` loop, a sequence or a control flow graph. A call instruction represents built-in or user-defined function calls; for instance, assign statements are represented as calls to the "`:=`" function.

- Instructions are included within statements, which are members of the cartesian product set `statement` that also incorporates the declarations of local variables; a whole function body is represented in PIPS IR as a statement. All named objects such as user variables or built-in functions in PIPS are members of the `entity` set (the `value` set denotes constants while the "`*`" symbol introduces Newgen list sets).

- Compound instructions can be either (1) a loop instruction, which includes an iteration index variable with its lower, upper and increment expressions and a loop body, (2) a sequence, i.e., a succession of statements, encoded as a list, or (3) a control flow graph (unstructured). In Newgen, a given set component such as `expression` can be distinguished using a prefix such as `lower` and `upper` here.

- Programs that contain structured (`exit`, `break` and `return`) and unstructured (`goto`) transfers of control are handled in the PIPS intermediate representation via the `unstructured` set. An unstructured instruction has one entry and one exit `control` vertices; a control is a vertex in a graph labeled with a statement and its lists of predecessor and successor control vertices. Executing an unstructured instruction amounts to following the control flow induced by the graph successor relationship, starting at the entry vertex, while executing the vertex statements, until the exit vertex is reached, if at all.

In this thesis, we use the IR of PIPS [59] to showcase our approach SPIRE as a parallel *extension* formalism for existing sequential intermediate representations (Chapter 4).

2.7 Conclusion

The purpose of this chapter is to present the context of existing hardware architectures and parallel programming models according for which our thesis work is developed. We also outline how to bridge these two levels using two approaches for writing parallel languages: manual and automatic approaches. Since in this thesis we adopt the automatic approach, this chapter presents also the notions of intermediate representation, control and data dependences, scheduling, and the framework of compilation PIPS where we implement our automatic parallelization methodology.

In the next chapter, we survey in detail seven recent and popular parallel programming languages and different parallel constructs. This analysis will

help us to find a uniform core language that can be designed as a unique but generic parallel intermediate representation by the development of the methodology SPIRE (see Chapter 4).

Chapter 3

Concepts in Task Parallel Programming Languages

L'inconscient est structuré comme un langage. **Jacques Lacan**

Existing parallel languages present portability issues such as OpenMP that targets only shared memory systems. Parallelizing sequential programs to be run on different types of architectures requires the writing of the same application in different parallel programming languages. Unifying the wide variety of existing programming models in a generic core language, as a parallel intermediate representation, can simplify the portability problem. To design this parallel IR, we use a survey of existing parallel language constructs which provide a trade-off between expressibility and conciseness of representation to design a generic parallel intermediate representation. This chapter surveys seven popular and efficient parallel language designs that tackle this difficult issue: Cilk, Chapel, X10, Habanero-Java, OpenMP, MPI and OpenCL. Using as single running example a parallel implementation of the computation of the Mandelbrot set, this chapter describes how the fundamentals of task parallel programming, i.e., collective and point-to-point synchronization and mutual exclusion, are dealt with in these languages. We discuss how these languages allocate and distribute data over memory. Our study suggests that, even though there are many keywords and notions introduced by these languages, they all boil down, as far as control issues are concerned, to three key task concepts: creation, synchronization and atomicity. Regarding memory models, these languages adopt one of three approaches: shared memory, distributed memory and PGAS (Partitioned Global Address Space).

Les langages parallèles existants tels que OpenMP, qui est un langage utilisé pour les systèmes à mémoire partagée, posent des problèmes de portabilité. La parallélisation de programmes séquentiels pour une exécution sur des types différents d'architectures parallèles nécessite l'écriture de la même application dans différents langages de programmation parallèle. L'unification de la grande variété de modèles de programmation existants dans un cœur de langage générique, comme une représentation intermédiaire (RI) parallèle, devrait permettre de simplifier le problème de la portabilité. Pour concevoir cette RI parallèle, nous effectuons une étude comparative des constructions de certains langages parallèles existants, langages qui fournissent un com-

promis entre capacité d'expression et concision de représentation, pour concevoir une représentation intermédiaire parallèle générique. Ce chapitre passe en revue sept langages parallèles populaires et efficaces qui abordent la question difficile de la représentation du parallélisme : Cilk, Chapel, X10, Habanero-Java, OpenMP, MPI et OpenCL. En utilisant comme exemple récurrent le calcul de l'ensemble de Mandelbrot, ce chapitre décrit comment les bases de la programmation parallèle de tâches, y compris la synchronisation collective et point-à-point et l'exclusion mutuelle, sont traitées dans ces langages. Nous discutons comment ces langages répartissent et distribuent des données sur la mémoire. Notre étude suggère que, même si beaucoup de mots-clés et notions sont introduits par ces langages, ils se résument tous, en ce qui concerne les questions relatives au contrôle, à trois concepts de tâche clés : création, synchronisation et atomicité. En ce qui concerne les modèles de mémoire, ces langages adoptent une parmi trois approches : mémoire partagée, mémoire distribuée et PGAS.

3.1 Introduction

This chapter is a state-of-the-art of important existing task parallel programming languages to help us to achieve two of our goals in this thesis of (1) the development of a parallel intermediate representation which should be as generic as possible to handle different parallel programming languages. This is possible using the taxonomy produced at the end of this chapter, (2) the languages we use as a back-end of our automatic parallelization process. These are determined by the knowledge of existing target parallel languages, their limits and their extensions.

Indeed, programming parallel machines as effectively as sequential ones would ideally require a language that provides high-level programming constructs to avoid the programming errors frequent when expressing parallelism. Programming languages adopt one of two ways to deal with this issue: (1) high-level languages hide the presence of parallelism at the software level, thus offering a code easy to build and port, but the performance of which is not guaranteed, and (2) low-level languages use explicit constructs for communication patterns and specifying the number and placement of threads, but the resulting code is difficult to build and not very portable, although usually efficient. Recent programming models explore the best trade-offs between expressiveness and performance when addressing parallelism.

Traditionally, there are two general ways to break an application into concurrent parts in order to take advantage of a parallel computer and execute them simultaneously on different CPUs: data and task parallelisms. In data parallelism, the same instruction is performed repeatedly and simultaneously on different data. In task parallelism, the execution of different

processes (threads) is distributed across multiple computing nodes. Task parallelism is often considered more difficult to specify than data parallelism, since it lacks the regularity present in the latter model; processes (threads) run simultaneously different instructions, leading to different execution schedules and memory access patterns. Task management must address both control and data dependences, in order to minimize execution and communication times.

This chapter describes how seven popular and efficient parallel programming language designs, either widely used or recently developed at DARPA[1], tackle the issue of task parallelism specification: Cilk, Chapel, X10, Habanero-Java, OpenMP, MPI and OpenCL. They are selected based on the richness of their constructs and their popularity; they provide simple high-level parallel abstractions that cover most of the parallel programming language design spectrum. We use a popular parallel problem, the computation of the Mandelbrot set [1], as a running example. We consider this an interesting test case, since it exhibits a high-level of embarrassing parallelism while its iteration space is not easily partitioned, if one wants to have tasks of balanced run times. Our goal with this chapter is to study and compare aspects around task parallelism for our automatic parallelization aim.

After this introduction, Section 3.2 presents our running example. We discuss the parallel language features specific to task parallelism, namely task creation, synchronization and atomicity in Section 3.3. In Section 3.4, a selection of current and important parallel programming languages are described: Cilk, Chapel, X10, Habanero Java, OpenMP, MPI and OpenCL. For each language, an implementation of the Mandelbrot set algorithm is presented. Section 3.5 contains comparison, discussion and classification of these languages. We conclude in Section 3.6.

3.2 Mandelbrot Set Computation

The Mandelbrot set is a fractal set. For each complex $c \in \mathbb{C}$, the set of complex numbers $z_n(c)$ is defined by induction as follows: $z_0(c) = c$ and $z_{n+1}(c) = z_n^2(c) + c$. The Mandelbrot set M is then defined as $\{c \in \mathbb{C} / \lim_{n\to\infty} z_n(c) < \infty\}$; thus, M is the set of all complex numbers c for which the series $z_n(c)$ converges. One can show [1] that a finite limit for $z_n(c)$ exists only if the modulus of $z_m(c)$ is less than 2, for some positive m. We give a sequential C implementation of the computation of the Mandelbrot set in Figure 3.2. Running this program yields Figure 3.1, in which each complex c is seen as a pixel, its color being related to its convergence property: the Mandelbrot set is the black shape in the middle of the figure.

[1]Three programming languages developed as part of the DARPA HPCS program: Chapel, Fortress [13], X10.

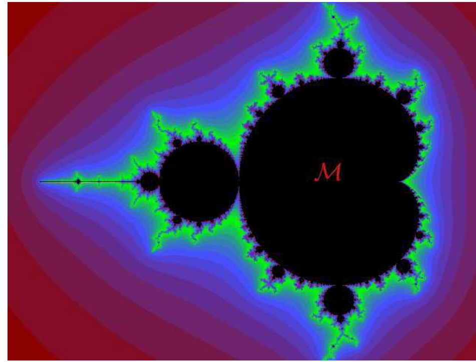


Figure 3.1: Result of the Mandelbrot set

```
unsigned long min_color = 0, max_color = 16777215;
unsigned int width = NPIXELS, height = NPIXELS,
             N = 2, maxiter = 10000;
double r_min = -N, r_max = N, i_min = -N, i_max = N;
double scale_r = (r_max - r_min)/width;
double scale_i = (i_max - i_min)/heigth;
double scale_color = (max_color - min_color)/maxiter;
Display *display; Window win; GC gc;
for (row = 0; row < height; ++row) {
  for (col = 0; col < width; ++col) {
    z.r = z.i = 0;
    /* Scale c as display coordinates of current point */
    c.r = r_min + ((double) col * scale_r);
    c.i = i_min + ((double) (height-1-row) * scale_i);
    /* Iterates z = z*z+c while |z| < N, or maxiter is reached */
    k = 0;
    do {
      temp = z.r*z.r - z.i*z.i + c.r;
      z.i = 2*z.r*z.i + c.i; z.r = temp;
      ++k;
    } while (z.r*z.r + z.i*z.i < (N*N) && k < maxiter);
    /* Set color and display point */
    color = (ulong) ((k-1) * scale_color) + min_color;
    XSetForeground (display, gc, color);
    XDrawPoint (display, win, gc, col, row);
  }
}
```

Figure 3.2: Sequential C implementation of the Mandelbrot set

We use this base program as our test case in our parallel implementations, in Section 3.4, for the parallel languages we selected. This is an interesting case for illustrating parallel programming languages: (1) it is an embarrassingly parallel problem, since all computations of pixel colors can be performed simultaneously, and thus is obviously a good candidate for expressing parallelism, but (2) its efficient implementation is not obvious,

since good load balancing cannot be achieved by simply grouping localized pixels together because convergence can vary widely from one point to the next, due to the fractal nature of the Mandelbrot set.

3.3 Task Parallelism Issues

Among the many issues related to parallel programming, the questions of task creation, synchronization, atomicity are particularly important when dealing with task parallelism, our focus in this thesis.

3.3.1 Task Creation

In this thesis, a task is a static notion, i.e., a list of instructions, while processes and threads are running instances of tasks. Creation of system-level task instances is an expensive operation, since its implementation, via processes, requires allocating and later possibly releasing system-specific resources. If a task has a short execution time, this overhead might make the overall computation quite inefficient. Another way to introduce parallelism is to use lighter, user-level tasks, called threads. In all languages addressed in this chapter, task management operations refer to such user-level tasks. The problem of finding the proper size of tasks, and hence the number of tasks, can be decided at compile or run times, using heuristics.

In our Mandelbrot example, the parallel implementations we provide below use a static schedule that allocates a number of iterations of the loop `row` to a particular thread; we interleave successive iterations into distinct threads in a round-robin fashion, in order to group loop body computations into chunks, of size `height`$/P$, where P is the (language-dependent) number of threads. Our intent here is to try to reach a good load balancing between threads.

3.3.2 Synchronization

Coordination in task-parallel programs is a major source of complexity. It is dealt with using synchronization primitives, for instance when a code fragment contains many phases of execution where each phase should wait for the precedent ones to proceed. When a process or a thread exits before synchronizing on a barrier that other processes are waiting on or when processes operate on different barriers using different orders, a deadlock occurs. Programmers must avoid these situations (and be deadlock-free). Different forms of synchronization constructs exist, such as mutual exclusion when accessing shared resources using locks, join operations that terminate child threads, multiple synchronizations using barriers[2], and point-to-point syn-

[2]The term "barrier" is used in various ways by authors [89]; we consider here that barriers are synchronization points that wait for the termination of sets of threads, defined

chronization using counting semaphores [97].

In our Mandelbrot example, we need to synchronize all pixel computations before exiting; we also need to use synchronization to deal with the atomic section (see the next section). Even though synchronization is rather simple in this example, caution is always needed; an example that may lead to deadlocks is mentioned in Section 3.5.

3.3.3 Atomicity

Access to shared resources requires atomic operations that, at any given time, can be executed by only one process or thread. Atomicity comes in two flavors: weak and strong [75]. A weak atomic statement is atomic only with respect to other explicitly atomic statements; no guarantee is made regarding interactions with non-isolated statements (not declared as atomic). By opposition, strong atomicity enforces non-interaction of atomic statements with all operations in the entire program. It usually requires specialized hardware support (e.g., atomic “compare and swap” operations), although a software implementation that treats non-explicitly atomic accesses as implicitly atomic single operations (using a single global lock) is possible.

In our Mandelbrot example, display accesses require connection to the X server; drawing a given pixel is an atomic operation since GUI-specific calls need synchronization. Moreover, two simple examples of atomic sections are provided in Section 3.5.

3.4 Parallel Programming Languages

We present here seven parallel programming languages and describe how they deal with the concepts introduced in the previous section. We also study how these languages distribute data over different processors. Given the large number of parallel languages that exist, we focus primarily on languages that are in current use and popular and that support simple high-level task-oriented parallel abstractions.

3.4.1 Cilk

Cilk [105], developed at MIT, is a multithreaded parallel language based on C for shared memory systems. Cilk is designed for exploiting dynamic and asynchronous parallelism. A Cilk implementation of the Mandelbrot set is provided in Figure 3.3[3].

in a language-dependent manner.

[3]From now on, variable declarations are omitted, unless required for the purpose of our presentation.

```
{
  cilk_lock_init(display_lock);
  for (m = 0; m < P; m++)
    spawn compute_points(m);
  sync;
}
cilk void compute_points(uint m) {
  for (row = m; row < height; row += P)
    for (col = 0; col < width; ++col) {
      //Initialization of c, k and z
      do {
        temp = z.r*z.r - z.i*z.i + c.r;
        z.i = 2*z.r*z.i + c.i; z.r = temp;
        ++k;
      } while (z.r*z.r + z.i*z.i < (N*N) && k < maxiter);
      color = (ulong) ((k-1) * scale_color) + min_color;
      cilk_lock(display_lock);
      XSetForeground (display, gc, color);
      XDrawPoint (display, win, gc, col, row);
      cilk_unlock(display_lock);
    }
}
```

Figure 3.3: Cilk implementation of the Mandelbrot set (P is the number of processors)

Task Parallelism

The `cilk` keyword identifies functions that can be spawned in parallel. A Cilk function may create threads to execute functions in parallel. The `spawn` keyword is used to create child tasks, such as *compute_points* in our example, when referring to Cilk functions.

Cilk introduces the notion of inlets [2], which are local Cilk functions defined to take the result of spawned tasks and use it (performing a reduction). The result should not be put in a variable in the parent function. All the variables of the function are available within an inlet. `Abort` allows to abort a speculative work by terminating all of the already spawned children of a function; it must be called inside an inlet. Inlets are not used in our example.

Synchronization

The `sync` statement is a local barrier, used in our example to ensure task termination. It waits only for the spawned child tasks of the current procedure to complete, and not for all tasks currently being executed.

Atomic Section

Mutual exclusion is implemented using locks of type `cilk_lockvar`, such as *display_lock* in our example. The function `cilk_lock` is used to test a lock and block if it is already acquired; the function `cilk_unlock` is used to release a lock. Both functions take a single argument which is an object of type `cilk_lockvar`. `cilk_lock_init` is used to initialize the lock object before it is used.

Data Distribution

In Cilk's shared memory model, all variables declared outside Cilk functions are shared. Also, variables addressed indirectly by passing of pointers to spawned functions are shared. To avoid possible non-determinism due to data races, the programmer should avoid the situation when a task writes a variable that may be read or written concurrently by another task, or use the primitive `cilk_fence` that ensures that all memory operations of a thread are committed before the next operation execution.

3.4.2 Chapel

Chapel [6], developed by Cray, supports both data and control flow parallelism and is designed around a multithreaded execution model based on PGAS for shared and distributed-memory systems. A Chapel implementation of the Mandelbrot set is provided in Figure 3.4.

```
coforall loc in Locales do
  on loc {
    for row in loc.id..height by numLocales do {
      for col in 1..width do {
        //Initialization of c, k and z
        do {
          temp = z.r*z.r - z.i*z.i + c.r;
          z.i = 2*z.r*z.i + c.i; z.r = temp;
          k = k+1;
        } while (z.r*z.r + z.i*z.i < (N*N) && k < maxiter);
        color = (ulong) ((k-1) * scale_color) + min_color;
        atomic {
          XSetForeground (display, gc, color);
          XDrawPoint (display, win, gc, col, row);
        }
}}}
```

Figure 3.4: Chapel implementation of the Mandelbrot set

Task Parallelism

Chapel provides three types of task parallelism [6], two structured ones and one unstructured. `cobegin`{*stmts*} creates a task for each statement in *stmts*; the parent task waits for the *stmts* tasks to be completed. `coforall` is a loop variant of the `cobegin` statement, where each iteration of the `coforall` loop is a separate task and the main thread of execution does not continue until every iteration is completed. Finally, in `begin`{*stmt*}, the original parent task continues its execution after spawning a child running *stmt*.

Synchronization

In addition to `cobegin` and `coforall`, used in our example, which have an implicit synchronization at the end, synchronization variables of type `sync` can be used for coordinating parallel tasks. A `sync` [6] variable is either empty or full, with an additional data value. Reading an empty variable and writing in a full variable suspends the thread. Writing to an empty variable atomically changes its state to full. Reading a full variable consumes the value and atomically changes the state to empty.

Atomic Section

Chapel supports atomic sections; `atomic`{*stmt*} executes *stmt* atomically with respect to other threads. The precise semantics is still ongoing work [6].

Data Distribution

Chapel introduces a type called `locale` to refer to a unit of the machine resources on which a computation is running. A locale is a mapping of Chapel data and computations to the physical machine. In Figure 3.4, Array `Locales` represents the set of locale values corresponding to the machine resources on which this code is running; `numLocales` refers to the number of locales. Chapel also introduces new `domain` types to specify array distribution; they are not used in our example.

3.4.3 X10 and Habanero-Java

X10 [5], developed at IBM, is a distributed asynchronous dynamic parallel programming language for multi-core processors, symmetric shared-memory multiprocessors (SMPs), commodity clusters, high end supercomputers, and even embedded processors like Cell. An X10 implementation of the Mandelbrot set is provided in Figure 3.5.

Habanero-Java [30], under development at Rice University, is derived from X10, and introduces additional synchronization and atomicity primitives surveyed below.

```
finish {
  for (m = 0; m < place.MAX_PLACES; m++) {
    place pl_row = place.places(m);
    async at (pl_row) {
      for (row = m; row < height; row += place.MAX_PLACES){
        for (col = 0; col < width; ++col) {
          //Initialization of c, k and z
          do {
            temp = z.r*z.r - z.i*z.i + c.r;
            z.i = 2*z.r*z.i + c.i; z.r = temp;
            ++k;
          } while (z.r*z.r + z.i*z.i < (N*N) && k < maxiter);
          color = (ulong) ((k-1) * scale_color) + min_color;
          atomic {
            XSetForeground (display, gc, color);
            XDrawPoint (display, win, gc, col, row);
          }
}}}}}
```

Figure 3.5: X10 implementation of the Mandelbrot set

Task Parallelism

X10 provides two task creation primitives: (1) the `async` *stmt* construct creates a new asynchronous task that executes *stmt*, while the current thread continues, and (2) the `future` *exp* expression launches a parallel task that returns the value of *exp*.

Synchronization

With `finish` *stmt*, the current running task is blocked at the end of the `finish` clause, waiting till all the children spawned during the execution of *stmt* have terminated. The expression `f.force()` is used to get the actual value of the "future" task `f`.

X10 introduces a new synchronization concept: the clock. It acts as a barrier for a dynamically varying set of tasks [102] that operate in phases of execution where each phase should wait for previous ones before proceeding. A task that uses a clock must first register with it (multiple clocks can be used). It then uses the statement `next` to signal to all the tasks that are registered with its clocks that it is ready to move to the following phase, and waits until all the clocks with which it is registered can advance. A clock can advance only when all the tasks that are registered with it have executed a `next` statement (see the left side of Figure 3.6).

Habanero-Java introduces phasers to extend this clock mechanism. A phaser is created and initialized to its first phase using the function `new`. The scope of a phaser is limited to the immediately enclosing `finish` statement; this constraint guarantees the deadlock-freedom safety property of

phasers [102]. A task can be registered with zero or more phasers, using one of four registration modes: the first two are the traditional SIG and WAIT signal operations for producer-consumer synchronization; the SIG_WAIT mode implements barrier synchronization, while SIG_WAIT_SINGLE ensures, in addition, that its associated statement is executed by only one thread. As in X10, a `next` instruction is used to advance each phaser that this task is registered with to its next phase, in accordance with this task's registration mode, and waits on each phaser that task is registered with, with a WAIT submode (see the right side of Figure 3.6). Note that clocks and phasers are not used in our Mandelbrot example, since a collective barrier based on the `finish` statement is sufficient.

```
finish async {
 clock cl = clock.make();
 for(j = 1;j <= n;j++) {
  async clocked(cl) {
   S;
   next;
   S';
  }
 }
}
```

```
finish {
 phaser ph=new phaser();
 for(j = 1;j <= n;j++) {
  async phased(ph<SIG_WAIT>) {
   S;
   next;
   S';
  }
 }
```

Figure 3.6: A clock in X10 (left) and a phaser in Habanero-Java (right)

Atomic Section

When a thread enters an `atomic` statement, no other thread may enter it until the original thread terminates it.

Habanero-Java supports weak atomicity using the `isolated` *stmt* primitive for mutual exclusion. The Habanero-Java implementation takes a single-lock approach to deal with isolated statements.

Data Distribution

In order to distribute data across processors, X10 and HJ introduce a type called `place`. A place is an address space within which a task may run; different places may however refer to the same physical processor and share physical memory. The program address space is partitioned into logically distinct places. `Place.MAX_PLACES`, used in Figure 3.5, is the number of places available to a program.

3.4.4 OpenMP

OpenMP [4] is an application program interface providing a multi-threaded programming model for shared memory parallelism; it uses directives to ex-

tend sequential languages. A C OpenMP implementation of the Mandelbrot set is provided in Figure 3.7.

```
P = omp_get_num_threads();
#pragma omp parallel shared(height,width,scale_r,\
         scale_i,maxiter,scale_color,min_color,r_min,i_min)\
         private(row,col,k,m,color,temp,z,c)
#pragma omp single
{
  for (m = 0; m < P; m++)
#pragma omp task
    for (row = m; row < height; row += P) {
      for (col = 0; col < width; ++col) {
        //Initialization of c, k and z
        do {
          temp = z.r*z.r - z.i*z.i + c.r;
          z.i = 2*z.r*z.i + c.i; z.r = temp;
          ++k;
        } while (z.r*z.r + z.i*z.i < (N*N) && k < maxiter);
        color = (ulong) ((k-1) * scale_color) + min_color;
#pragma omp critical
        {
          XSetForeground (display, gc, color);
          XDrawPoint (display, win, gc, col, row);
        }
}}}
```

Figure 3.7: C OpenMP implementation of the Mandelbrot set

Task Parallelism

OpenMP allows dynamic (`omp task`) and static (`omp section`) scheduling models. A task instance is generated each time a thread (the encountering thread) encounters an `omp task` directive. This task may either be scheduled immediately on the same thread or deferred and assigned to any thread in a thread team, which is the group of threads created when an `omp parallel` directive is encountered. The `omp sections` directive is a non-iterative work-sharing construct. It specifies that the enclosed sections of code, declared with `omp section`, are to be divided among the threads in the team; these sections are independent blocks of code that the compiler can execute concurrently.

Synchronization

OpenMP provides synchronization constructs that control the execution inside a team thread: `barrier` and `taskwait`. When a thread encounters a `barrier` directive, it waits until all other threads in the team reach the same point; the scope of a barrier region is the innermost enclosing paral-

lel region. The `taskwait` construct is a restricted barrier that blocks the thread until all child tasks created since the beginning of the current task are completed. The `omp single` directive identifies code that must be run by only one thread.

Atomic Section

The `critical` and `atomic` directives are used for identifying a section of code that must be executed by a single thread at a time. The `atomic` directive works faster than `critical`, since it only applies to single instructions, and can thus often benefit from hardware support. Our implementation of the Mandelbrot set in Figure 3.7 uses `critical` for the drawing function (not a single instruction).

Data Distribution

OpenMP variables are either global (`shared`) or local (`private`); see Figure 3.7 for examples. A shared variable refers to one unique block of storage for all threads in the team. A private variable refers to a different block of storage for each thread. More memory access modes exist, such as `firstprivate` or `lastprivate`, that may require communication or copy operations.

3.4.5 MPI

MPI [3] is a library specification for message-passing used to program shared and distributed memory systems. Both point-to-point and collective communication are supported. The C MPI implementation of the Mandelbrot set is provided in Figure 3.8.

Task Parallelism

The starter process may be a separate process that is not part of the MPI application, or the rank 0 process may act as a starter process to launch the remaining MPI processes of the MPI application. MPI processes are created when `MPI_Init` is called that defines also the initially universe intracommunicator for all processes to drive various communications. A communicator determines the scope of communications. Processes are terminated using `MPI_Finalize`.

Synchronization

`MPI_Barrier` blocks until all processes in the communicator passed as an argument have reached this call.

```
int rank;
MPI_Status status;
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &P);
for (m = 1; m < P; m++){
  if(rank == P){
    for (row = m; row < height; row += P){
      for (col = 0; col < width; ++col){
        //Initialization of c, k and z
        do {
          temp = z.r*z.r - z.i*z.i + c.r;
          z.i = 2*z.r*z.i + c.i; z.r = temp;
          ++k;
        } while (z.r*z.r + z.i*z.i < (N*N) && k < maxiter);
        color_msg[col] = (ulong) ((k-1) * scale_color) + min_color;
       }
       ierr = MPI_Send(color_msg, width, MPI_UNSIGNED_LONG, 0,0,
                       MPI_COMM_WORLD);
     }
   }
}
if(rank == 0){
  for (row = 0; row < height; row++){
    ierr = MPI_Recv(data_msg, width, MPI_UNSIGNED_LONG,
                    MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    for (col = 0; col < width; ++col){
      color = data_msg[col];
      XSetForeground (display, gc, color);
      XDrawPoint (display, win, gc, col, row);
    }
  }
}
ierr = MPI_Barrier(MPI_COMM_WORLD);
ierr = MPI_Finalize();
```

Figure 3.8: MPI implementation of the Mandelbrot set (P is the number of processors)

Atomic Section

MPI does not support atomic sections. Indeed, the atomic section is a concept intrinsically linked to the shared memory. In our example, the drawing function is executed by the master process (the rank 0 process).

Data Distribution

MPI supports the distributed memory model where it proceeds by point-to-point and collective communications to transfer data between processors. In our implementation of the Mandelbrot set, we used the point-to-point routines `MPI_Send` and `MPI_Recv` to communicate between the master process

(`rank = 0`) and the other processes.

3.4.6 OpenCL

OpenCL (Open Computing Language) [70] is a standard for programming heterogeneous multiprocessor platforms where programs are divided into several parts: some called “the kernels” that execute on separate devices, e.g., GPUs, with their own memories and the others that execute on the host CPU. The main object in OpenCL is the command queue, which is used to submit work to a device by enqueueing OpenCL commands to be executed. An OpenCL implementation of the Mandelbrot set is provided in Figure 4.5.

Task Parallelism

OpenCL provides the parallel construct `clEnqueueTask`, which enqueues a command requiring the execution of a kernel on a device by a work item (OpenCL thread). OpenCL uses two different models of execution of command queues: in-order, used for data parallelism, and out-of-order. In an out-of-order command queue, commands are executed as soon as possible, and no order is specified, except for wait and barrier events. We illustrate the out-of-order execution mechanism in Figure 4.5, but currently this is an optional feature and is thus not supported by many devices.

Synchronization

OpenCL distinguishes between two types of synchronization: coarse and fine. Coarse grained synchronization, which deals with command queue operations, uses the construct `clEnqueueBarrier`, which defines a barrier synchronization point. Fine grained synchronization, which covers synchronization at the GPU function call granularity level, uses OpenCL events via `ClEnqueueWaitForEvents` calls.

Data transfers between the GPU memory and the host memory, via functions such as `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`, also induce synchronization between blocking or non-blocking communication commands. Events returned by `clEnqueue` operations can be used to check if a non-blocking operation has completed.

Atomic Section

Atomic operations are only supported on integer data, via functions such as `atom_add` or `atom_xchg`. Currently, these are only supported by some devices as part of an extension of the OpenCL standard. Since OpenCL lacks support for general atomic sections, the drawing function is executed by the host in Figure 4.5.

```
__kernel void kernel_main(complex c, uint maxiter, double scale_color,
                    uint m, uint P, ulong color[NPIXELS][NPIXELS]) {
  for (row = m; row < NPIXELS; row += P)
    for (col = 0; col < NPIXELS; ++col) {
      //Initialization of c, k and z
      do {
        temp = z.r*z.r-z.i*z.i+c.r;
        z.i = 2*z.r*z.i+c.i; z.r = temp;
        ++k;
      } while (z.r*z.r+z.i*z.i<(N*N) && k<maxiter);
      color[row][col] = (ulong) ((k-1)*scale_color);
    }
}
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
                    &device_id, &ret_num_devices);
cl_context context = clCreateContext( NULL, 1, &device_id,
                                    NULL, NULL, &ret);
cQueue=clCreateCommandQueue(context,device_id,
                          OUT_OF_ORDER_EXEC_MODE_ENABLE,NULL);
P = CL_DEVICE_MAX_COMPUTE_UNITS;
memc = clCreateBuffer(context, CL_MEM_READ_ONLY , sizeof(complex), c);
// ... Create read-only buffers with maxiter, scale_color and P too
memcolor = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                        sizeof(ulong)*height*width, NULL, NULL);
clEnqueueWriteBuffer(cQueue, memc, CL_TRUE, 0,
                    sizeof(complex), &c, 0, NULL, NULL);
// ... Enqueue write buffer with maxiter, scale_color and P too
program = clCreateProgramWithSource(context, 1, &program_source,
                                  NULL, NULL);
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "kernel_main", NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem),(void *)&memc);
// ... Set kernel argument with
// memmaxiter, memscale_color, memP and memcolor too
for(m = 0; m < P; m++) {
  memm = clCreateBuffer(context, CL_MEM_READ_ONLY , sizeof(uint), m);
  clEnqueueWriteBuffer(cQueue, memm, CL_TRUE, 0, sizeof(uint), &m, 0,
                      NULL, NULL);
  clSetKernelArg(kernel, 0, sizeof(cl_mem),(void *)&memm);
  clEnqueueTask(cQueue, kernel, 0, NULL, NULL);
}
clFinish(cQueue);
clEnqueueReadBuffer(cQueue,memcolor,CL_TRUE,0,space,color,0,NULL,NULL);
for (row = 0; row < height; ++row)
  for (col = 0; col < width; ++col) {
    XSetForeground (display, gc, color[col][row]);
    XDrawPoint (display, win, gc, col, row);
  }
```

Figure 3.9: OpenCL implementation of the Mandelbrot set

Data Distribution

Each work item can either use (1) its private memory, (2) its local memory, which is shared between multiple work items, (3) its constant memory, which is closer to the processor than the `__global` memory, and thus much faster to access, although slower than `__local` memory, and (4) global

memory, shared by all work items. Data is only accessible after being transferred from the host, using functions such as `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` that move data in and out of a device.

3.5 Discussion and Comparison

This section discusses the salient features of our surveyed languages. More specifically, we look at their design philosophy and the new concepts they introduce, how point-to-point synchronization is addressed in each of these languages, the various semantics of atomic sections and the data distribution issues. We end up summarizing the key features of all the languages covered in this chapter.

Design Paradigms

Our overview study, based on a single running example, namely the computation of the Mandelbrot set, is admittedly somewhat biased, since each language has been designed with a particular application framework in mind, which may, or may not, be well adapted to a given application. Cilk is well suited to deal with divide-and-conquer strategies, something not put into practice in our example. On the contrary, X10, Chapel and Habanero-Java are high-level Partitioned Global Address Space languages that offer abstract notions such as places and locales, which were put to good use in our example. OpenCL is a low-level, verbose language that works across GPUs and CPUs; our example clearly illustrates that this approach is not providing much help here in terms of shrinking the semantic gap between specification and implementation. The OpenMP philosophy is to add compiler directives to parallelize parts of code on shared-memory machines; this helps programmers move incrementally from a sequential to a parallel implementation. While OpenMP suffers from portability issues, MPI tackles both shared and distributed memory systems using a low-level support for communications.

New Concepts

Even though this thesis does not address data parallelism per se, note that Cilk is the only language that does not provide support for data parallelism; yet, spawned threads can be used inside loops to simulate SIMD processing. Also, Cilk adds a facility to support speculative parallelism, enabling spawned tasks abort operations via the `abort` statement. Habanero-Java introduces the `isolated` statement to specify the weak atomicity property. Phasers, in Habanero-Java, and clocks, in X10, are new high-level constructs for collective and point-to-point synchronization between varying sets of threads.

Point-to-Point Synchronization

We illustrate the way the surveyed languages address the difficult issue of point-to-point synchronization via a simple example, a hide-and-seek children game in Figure 3.10. X10 clocks or Habanero-Java phasers help express easily the different phases between threads. The notion of point-to-point synchronization cannot be expressed easily using OpenMP or Chapel[4]. We were not able to implement this game using Cilk high-level synchronization primitives, since `sync`, the only synchronization construct, is a local barrier for recursive tasks: it synchronizes only threads spawned in the current procedure, and thus not the two searcher and hider tasks. As mentioned above, this is not surprising, given Cilk's approach to parallelism.

```
finish async {
 clock cl = clock.make();
 async clocked(cl) {
  count_to_a_number();
  next;
  start_searching();
 }
 async clocked(cl) {
  hide_oneself();
  next;
  continue_to_be_hidden();
 }
}
```

```
finish async{
 phaser ph = new phaser();
 async phased(ph) {
  count_to_a_number();
  next;
  start searching();
 }
 async phased(ph) {
  hide_oneself();
  next;
  continue_to_be_hidden();
 }
}
```

```
cilk void searcher() {
 count_to_a_number();
 point_to_point_sync();//missing
 start_searching();
}
cilk void hidder() {
 hide_oneself();
 point_to_point_sync();//missing
 continue_to_be_hidden();
}
void main() {
  spawn searcher();
  spawn hidder();
}
```

Figure 3.10: A hide-and-seek game (X10, HJ, Cilk)

Atomic Section

The semantics and implementations of the various proposals for dealing with atomicity are rather subtle.

[4] Busy-waiting techniques can be applied.

Atomic operations, which apply to single instructions, can be efficiently implemented, e.g. in X10, using non-blocking techniques such as the atomic instruction `compare-and-swap`. In OpenMP, the `atomic` directive can be made to work faster than the `critical` directive, when atomic operations are replaced with processor commands such as GLSC [72] instructions, or "gather-linked and scatter-conditional", which extend scatter-gather hardware to support advanced atomic memory operations; therefore, it is better to use this directive when protecting shared memory during elementary operations. Atomic operations can be used to update different elements of a data structure (arrays, records) in parallel without using many explicit locks. In the example of Figure 3.11, the updates of different elements of Array `x` are allowed to occur in parallel. General atomic sections, on the other hand, serialize the execution of updates to elements via one lock.

```
#pragma omp parallel for shared(x, index, n)
for (i=0; i<n; i++) {
#pragma omp atomic
   x[index[i]] += f(i);         // index is supposed injective
}
```

Figure 3.11: Example of an atomic directive in OpenMP

With the weak atomicity model of Habanero-Java, the `isolated` keyword is used instead of `atomic` to make explicit the fact that the construct supports weak rather than strong isolation. In Figure 3.12, Threads 1 and 2 may access to `ptr` simultaneously; since weakly atomic accesses are used, an atomic access to `temp->next` is not enforced.

```
// Thread 1
ptr = head;//non isolated statement
isolated {
  ready = true;
}
```

```
// Thread 2
isolated {
  if(ready)
    temp->next = ptr;
}
```

Figure 3.12: Data race on `ptr` with Habanero-Java

Data Distribution

PGAS languages offer a compromise between the fine level of control of data placement provided by the message passing model and the simplicity of the shared memory model. However, the physical reality is that different PGAS portions, although logically distinct, may refer to the same physical processor and share physical memory. Practical performance might thus not be as good as expected.

Regarding the shared memory model, despite its simplicity of programming, programmers have scarce support for expressing data locality, which

could help improve performance in many cases. Debugging is also difficult when data races or deadlocks occur.

Finally, the message passing memory model, where processors have no direct access to the memories of other processors, can be seen as the most general one, in which programmers can both specify data distribution and control locality. Shared memory (where there is only one processor managing the whole memory) and PGAS (where one assumes that each portion is located on a distinct processor) models can be seen as particular instances of the message passing model, when converting implicit write and read operations with explicit send/receive message passing constructs.

Summary Table

We collect in Table 3.1 the main characteristics of each language addressed in this chapter. Even though we have not discussed the issue of data parallelism in this chapter, we nonetheless provide the main constructs used in each language to launch data parallel computations.

Language	Task creation	Synchronization			Data parallelism	Memory model
		Task join	Point-to-point	Atomic section		
Cilk (MIT)	spawn	sync abort	—	cilk_lock cilk_unlock	—	*Shared*
Chapel (Cray)	begin cobegin	sync	sync	sync atomic	forall coforall	*PGAS (Locales)*
X10 (IBM)	async future	finish	next force	atomic	foreach	*PGAS (Places)*
Habanero-Java (Rice)	async future	finish	next get	atomic isolated	foreach	*PGAS (Places)*
OpenMP	omp task omp section	omp taskwait omp barrier	—	omp critical omp atomic	omp for	*Shared*
OpenCL	EnqueueTask	Finish EnqueueBarrier	*events*	atom_add, ...	EnqueueND-RangeKernel	*Message passing*
MPI	MPI_spawn	MPI_Finalize MPI_Barrier	—	—	MPI_Init	*Message passing*

Table 3.1: Summary of parallel languages constructs

3.6 Conclusion

Using the Mandelbrot set computation as a running example, this chapter presents an up-to-date comparative overview of seven parallel programming languages: Cilk, Chapel, X10, Habanero-Java, OpenMP, MPI and OpenCL. These languages are in current use, popular, offer rich and highly abstract functionalities, and most support both data and task parallel execution models. The chapter describes how, in addition to data distribution and locality,

the fundamentals of task parallel programming, namely task creation, collective and point-to-point synchronization and mutual exclusion are dealt with in these languages.

This study serves as the basis for the design of SPIRE, a sequential to parallel intermediate representation extension that we use to upgrade the intermediate representations of PIPS [19] source-to-source compilation framework to represent task concepts in parallel languages. SPIRE is presented in the next chapter.

An earlier version of the work presented in this chapter was published in [67].

Chapter 4

SPIRE: A Generic Sequential to Parallel Intermediate Representation Extension Methodology

I was working on the proof of one of my poems all the morning, and took out a comma. In the afternoon I put it back again. **Oscar Wilde**

The research goal addressed in this thesis is the design of efficient automatic parallelization algorithms that use as a backend a uniform, simple and generic parallel intermediate core language. Instead of designing from scratch one such particular intermediate language, we take an indirect, more general, route. We use the survey presented in Chapter 3 to design SPIRE, a new methodology for the design of parallel extensions of the intermediate representations used in compilation frameworks of sequential languages. It can be used to leverage existing infrastructures for sequential languages to address both control and data parallel constructs while preserving as much as possible existing analyses for sequential code. In this chapter, we suggest to view this upgrade process as an "intermediate representation transformer" at the syntactic and semantic levels; we show this can be done via the introduction of only ten new concepts, collected in three groups, namely execution, synchronization and data distribution, precisely defined via a formal semantics and rewriting rules.

We use the sequential intermediate representation of PIPS, a comprehensive source-to-source compilation platform, as a use case for our approach of the definition of parallel intermediate languages. We introduce our SPIRE parallel primitives, extend PIPS intermediate representation and show how example code snippets from the OpenCL, Cilk, OpenMP, X10, Habanero-Java, MPI and Chapel parallel programming languages can be represented this way. A formal definition of SPIRE operational semantics is provided, built on top of the one used for the sequential intermediate representation. We assess the generality of our proposal by showing how a different sequential IR, namely LLVM, can be extended to handle parallelism using the SPIRE methodology.

Our primary goal with the development of SPIRE is to provide, at a low cost, powerful parallel program representations that ease the design of effi-

cient automatic parallelization algorithms. More precisely, in this chapter, our intent is to describe SPIRE as a uniform, simple and generic core language transformer and apply it to PIPS IR. We illustrate the applicability of the resulting parallel IR by using it as parallel code target in Chapter 7, where we also address code generation issues.

L'objectif de la recherche abordée dans cette thèse est la conception d'algorithmes de parallélisation efficaces automatiques qui utilisent comme langage-cible une représentation intermédiaire uniforme, simple et générique. Au lieu de concevoir à partir de zéro un tel langage intermédiaire particulier, nous prenons une voie indirecte et plus générale. Nous utilisons l'étude présentée dans le chapitre 3 pour concevoir SPIRE, une nouvelle méthodologie pour la conception d'extensions parallèles des représentations intermédiaires utilisées dans les compilateurs des langages séquentiels. Elle peut être utilisée pour exploiter les infrastructures existant pour les langages séquentiels afin de traiter à la fois les constructions parallèles de contrôle et de données tout en préservant autant que possible les analyses existantes pour le code séquentiel. Dans ce chapitre, nous suggérons de voir ce processus de mise à niveau comme un "transformateur de représentation intermédiaire" aux niveaux syntaxique et sémantique ; nous montrons que cela peut être fait par l'introduction de dix nouveaux concepts, collectés dans trois groupes, à savoir exécution, synchronisation et distribution de données, définis avec précision par une sémantique formelle et des règles de réécriture.

Nous utilisons la représentation intermédiaire séquentielle de PIPS, une plate-forme de compilation source-à-source, comme un cas d'utilisation de notre approche de la définition des langages intermédiaires parallèles. Nous présentons nos primitives parallèles déduites de SPIRE, étendons la représentation intermédiaire de PIPS et montrons comment des extraits de codes écrits dans les langages de programmation parallèles OpenCL, Cilk, OpenMP, X10, Habanero Java, MPI et Chapel peuvent être représentés de cette façon. Une définition formelle de la sémantique opérationnelle de SPIRE est présentée, construite à partir de celle utilisée pour la représentation intermédiaire séquentielle. Nous évaluons la généralité de notre proposition en montrant comment une RI séquentielle différente, à savoir LLVM, peut être étendue pour prendre en compte le parallélisme en utilisant la méthodologie SPIRE.

Notre principal objectif avec le développement de SPIRE est de fournir, à un faible coût, de puissantes représentations de programmes parallèles qui facilitent la conception d'algorithmes de parallélisation automatique efficaces. Plus précisément, dans ce chapitre, notre intention est de décrire SPIRE comme un transformateur de langages uniforme, simple et générique et de l'appliquer à la RI de PIPS. Nous illustrons l'applicabilité de la RI parallèle résultante en l'utilisant comme cible de code parallèle au chapitre 7, où nous abordons également les questions de génération de code.

4.1 Introduction

The growing importance of parallel computers and the search for efficient programming models led, and is still leading, to the proliferation of parallel programming languages such as, currently, Cilk [105], Chapel [6], X10 [5], Habanero-Java [30], OpenMP [4], OpenCL [70] or MPI [3], presented in Chapter 3. Our automatic task parallelization algorithm which we integrate within PIPS [59], a comprehensive source-to-source compilation and optimization platform, should help in the adaptation of PIPS to such an evolution. To reach this goal, we introduce an internal intermediate representation (IR) to represent parallel programs into PIPS. The choice of a proper parallel IR is of key importance, since the efficiency and power of the transformations and optimizations the compiler PIPS can perform are closely related to the selection of a proper program representation paradigm. This seems to suggest the introduction of a dedicated IR for each parallel programming paradigm. Yet, it would be better, from a software engineering point of view, to find a unique parallel IR, as general and simple as possible.

Existing proposals for program representation techniques already provide a basis for the exploitation of parallelism via the encoding of control and/or data flow information. HPIR [114], PLASMA [91] or InsPIRe [58] are instances that operate at a high abstraction level, while the hierarchical task, stream or program dependence graphs (we survey these notions in Section 4.5) are better suited to graph-based approaches. Unfortunately many existing compiler frameworks such as PIPS use traditional representations for sequential-only programs, and changing their internal data structures to deal with parallel constructs is a difficult and time-consuming task.

In this thesis, we choose to develop a methodology for the design of parallel extensions of the intermediate representations used in compilation frameworks of sequential languages in general, instead of developing a parallel intermediate representation for PIPS only. The main motivation behind the design of the methodology introduced in this chapter is to preserve the many years of development efforts invested in huge compiler platforms such as GCC (more than 7 million lines of code), PIPS (600 000 lines of code), LLVM (more than 1 million lines of code),... when upgrading their intermediate representations to handle parallel languages, as source languages or as targets for source-to-source transformations. We provide an evolutionary path for these large software developments via the introduction of the Sequential to Parallel Intermediate Representation Extension (SPIRE) methodology. We show that it can be plugged into existing compilers in a rather simple manner. SPIRE is based on only three key concepts: (1) the parallel vs. sequential execution of groups of statements such as sequences, loops and general control-flow graphs, (2) the global synchronization characteristics of statements and the specification of finer grain synchronization

via the notion of events and (3) the handling of data distribution for different memory models. To illustrate how this approach can be used in practice, we use SPIRE to extend the intermediate representation (IR) [33] of PIPS.

The design of SPIRE is the result of many trade-offs between generality and precision, abstraction and low-level concerns. On the one hand, and in particular when looking at source-to-source optimizing compiler platforms adapted to multiple source languages, one needs to (1) represent as many of the existing (and, hopefully, future) parallel constructs, (2) minimize the number of additional built-in functions that hamper compilers optimization passes, and (3) preserve high-level structured parallel constructs to keep their properties such as deadlock free, while minimizing the number of new concepts introduced in the parallel IR. On the other hand, keeping only a limited number of hardware-level notions in the IR, while good enough to deal with all parallel constructs, would entail convoluted rewritings of high-level parallel flows. We used an extensive survey of key parallel languages, namely Cilk, Chapel, X10, Habanero-Java, OpenMP, OpenCL and MPI, to guide our design of SPIRE, while showing how to express their relevant parallel constructs within SPIRE.

The remainder of this chapter is structured as follows. Our parallel extension proposal, SPIRE, is introduced in Section 4.2, where we illustrate its application to PIPS sequential IR, presented in Section 2.6.5; we also show how simple illustrative examples written in OpenCL, Cilk, OpenMP, X10, Habanero-Java, MPI and Chapel can be easily represented within SPIRE(PIPS IR)[1]. The formal operational semantics of SPIRE is given in Section 4.3. Section 4.4 shows the generality of SPIRE methodology by showing its use on LLVM. We survey existing parallel IRs in Section 4.5. We conclude in Section 4.6.

4.2 SPIRE, a Sequential to Parallel IR Extension

In this section, we present in detail the SPIRE methodology, which is used to add parallel concepts to sequential IRs. After introducing our design philosophy, we describe the application of SPIRE on the PIPS IR presented in Section 2.6.5. We illustrate these SPIRE-derived constructs with code excerpts from various parallel programming languages; our intent is not to provide here general rewriting techniques from these to SPIRE, but to provide hints on how such rewritings might possibly proceed. Note that, in Section 4.4, using LLVM, we show that our methodology is general enough to be adapted to other IRs.

[1]SPIRE(PIPS IR) means that the function of transformation SPIRE is applied on the sequential IR of PIPS; it yields its parallel IR.

4.2.1 Design Approach

SPIRE intends to be a practical methodology to extend existing sequential IRs to parallelism constructs, either to generate parallel code from sequential programs or compile explicitly parallel programming languages. Interestingly, the idea of seeing the issue of parallelism as an extension over sequential concepts is in sync with Dijkstra's view that "parallelism or concurrency are operational concepts that refer not to the program, but to its execution." [41]. If one accepts such a vision, adding parallelism extensions to existing IRs, as advocated by our approach with SPIRE, can thus, at a fundamental level, not be seen as an afterthought but as a consequence of the fundamental nature of parallelism.

Our design of SPIRE does not intend to be minimalist but to be as seamlessly as possible integrable within actual IRs and general enough to handle as many parallel programming constructs as possible. To be successful, our design point must provide proper trade-offs between generality, expressibility and conciseness of representation. We used an extensive survey of existing parallel languages to guide us during this design process. Table 4.1, which extends the one provided in Chapter 3, summarizes the main characteristics of seven recent and widely used parallel languages: Cilk, Chapel, X10, Habanero-Java, OpenMP, OpenCL and MPI. The main constructs used in each language to launch task and data parallel computations, perform synchronization, introduce atomic sections and transfer data in the various memory models are listed. Our main finding from this analysis is that, to be able to deal with parallel programming, one simply needs to add to a given sequential IR the ability to specify (1) the parallel execution mechanism of groups of statements, (2) the synchronization behavior of statements and (3) the layout of data, i.e., how memory is modeled in the parallel language.

The last line of Table 4.1 summarizes the approach we propose to map these programming concepts to our parallel intermediate representation extension. SPIRE is based on the introduction of only ten key notions, collected in three groups:

- execution, via the `sequential` and `parallel` constructs;
- synchronization, via the `spawn`, `barrier`, `atomic`, `single`, `signal` and `wait` constructs;
- data distribution, via `send` and `recv` constructs.

Small code snippets are provided below to sketch how the key constructs of these parallel languages can be encoded in practice within a SPIRE-extended parallel IR.

	Execution	Synchronization				Memory	
Language	Parallelism	Task creation	Task join	Point-to-point	Atomic section	model	Data distribution
Cilk (MIT)	—	spawn	sync	—	cilk_lock	*Shared*	—
Chapel (Cray)	forall coforall cobegin	begin	sync	sync	sync atomic	*PGAS (Locales)*	(on)
X10 (IBM)	foreach	async future	finish	next force	atomic	*PGAS (Places)*	(at)
Habanero-Java (Rice)	foreach	async future	finish	next get	atomic isolated	*PGAS (Places)*	(at)
OpenMP	omp for omp sections	omp task omp section	omp taskwait omp barrier	—	omp critical omp atomic	*Shared*	private, shared...
OpenCL	EnqueueND-RangeKernel	EnqueueTask	Finish EnqueueBarrier	*events*	atom_add, ...	*Distributed*	ReadBuffer WriteBuffer
MPI	MPI_Init	MPI_spawn	MPI_Finalize MPI_Barrier	—	—	*Distributed*	MPI_Send MPI_Recv...
SPIRE	`sequential`, `parallel`	`spawn`	`barrier`	`signal`, `wait`	`atomic`	*Shared,* *Distributed*	`send`, `recv`

Table 4.1: Mapping of SPIRE to parallel languages constructs (terms in parentheses are not currently handled by SPIRE)

4.2.2 Execution

The issue of parallel vs. sequential execution appears when dealing with groups of statements, which in our case study correspond to members of the `forloop`, `sequence` and `unstructured` sets presented in Section 2.6.5. To apply SPIRE to PIPS sequential IR, an `execution` attribute is added to these sequential set definitions:

```
forloop'      = forloop x execution;
sequence'     = sequence x execution;
unstructured' = unstructured x execution;
```

The primed sets `forloop'` (expressing data parallelism) and `sequence'` and `unstructured'` (implementing control parallelism) represent SPIREd-up sets for the PIPS parallel IR. Of course, the 'prime' notation is used here for pedagogical purpose only; in practice, an execution field is added in the existing IR representation. The definition of `execution` is straightforward:

```
execution = sequential:unit + parallel:unit;
```

where `unit` denotes a set with one single element; this encodes a simple enumeration of cases for execution. A `parallel` execution attribute asks for all loop iterations, sequence statements and control nodes of unstructured instructions to be run concurrently. Note that there is no implicit barrier

after the termination of these parallel statements; in Section 4.2.3, we introduce the annotation `barrier` that can be used to add an explicit barrier on these parallel statements, if need be.

For instance, a parallel `execution` construct can be used to represent data parallelism on GPUs, when expressed via the OpenCL `clEnqueueNDRangeKernel` function (see the top of Figure 4.1). This function call could be encoded within PIPS parallel IR as a parallel loop (see the bottom of Figure 4.1), each iteration executing the `kernel` function as a separate task, receiving the proper index value as an argument.

```
//Execute 'n' kernels in parallel
global_work_size[0] = n;
err = clEnqueueNDRangeKernel(cmd_queue, kernel,
                     1, NULL, global_work_size,
                     NULL, 0, NULL, NULL);
```

```
forloop(I, 1,
        global_work_size,
        1,
        kernel(...),
        parallel)
```

Figure 4.1: OpenCL example illustrating a parallel loop

An other example, in the left side of Figure 4.2, from Chapel, illustrates its `forall` data parallelism construct, which will be encoded with a SPIRE parallel loop.

`forall i in 1..n do` `  t[i] = 0;`	`forloop(i, 1, n, 1,` `        t[i] = 0, parallel)`

Figure 4.2: `forall` in Chapel, and its SPIRE core language representation

The main difference between a parallel sequence and a parallel unstructured is that, in a parallel unstructured, synchronizations are explicit using arcs that represent dependences between statements. An example of an unstructured code is illustrated in Figure 4.3, where the graph in the right side is a parallel unstructured representation of the code in the left side. Nodes are statements and edges are data dependences between them; for example the statement `u=x+1` cannot be launched before the statement `x=3` is terminated. Figure 4.4, from OpenMP, illustrates its `parallel sections` task parallelism construct, which will be encoded with a SPIRE parallel sequence. We can say that a parallel sequence is only suitable for fork-join parallelism fashion whereas any other unstructured form of parallelism can be encoded using the control flow graph (`unstructured`) set.

```
x = 3;      // statement 1
y = 5;      // statement 2
z = x + y;  // statement 3
u = x + 1;  // statement 4
```

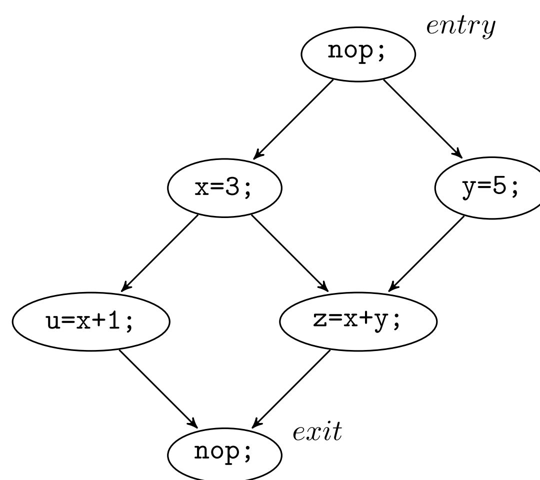


Figure 4.3: A C code, and its unstructured parallel control flow graph representation

```
#pragma omp sections nowait
{
  #pragma omp section
  x = foo();
  #pragma omp section
  y = bar();
}
z = baz(x,y);
```

```
sequence(
  sequence(x = foo(); y = bar(),
           parallel);
  z = baz(x,y),
  sequential)
```

Figure 4.4: `parallel sections` in OpenMP, and its SPIRE core language representation

Representing the presence of parallelism using only the annotation of `parallel` constitutes a voluntary trade-off between conciseness and expressibility of representation. For example, representing the code of OpenCL in Figure 4.1 in the way we suggest may induce a loss of precision regarding the use of queues of tasks adopted in OpenCL and lead to errors; great care must be taken when analyzing programs to avoid errors in translation between OpenCL and SPIRE such as scheduling of these tasks.

4.2.3 Synchronization

The issue of synchronization is a characteristic feature of the run-time behavior of one statement with respect to other statements. In parallel code, one usually distinguishes between two types of synchronization: (1) collective synchronization between threads using barriers, and (2) point-to-point synchronization between participating threads. We suggest this can be done

in two parts.

Collective Synchronization

The first possibility is via the new synchronization attribute, which can be added to the specification of statements. SPIRE extends sequential intermediate representations in a straightforward way by adding a synchronization attribute to the specification of statements:

```
statement' = statement x synchronization;
```

Coordination by synchronization in parallel programs is often dealt with via coding patterns such as barriers, used for instance when a code fragment contains many phases of parallel execution where each phase should wait for the precedent ones to proceed. We define the `synchronization` set via high-level coordination characteristics useful for optimization purposes:

```
synchronization =
                  none:unit + spawn:entity + barrier:unit +
                  single:bool + atomic:reference;
```

where S is the statement with the synchronization attribute:

- `none` specifies the default behavior, i.e., independent with respect to other statements, for S;
- `spawn` induces the creation of an asynchronous task S, while the value of the corresponding entity is the user-chosen number of the thread that executes S. SPIRE thus, in addition to decorating parallel tasks using the `parallel execution` attribute to handle coarse grain parallelism, provides the possibility of specifying each task via the `spawn synchronization` attribute in order to enable the encoding of finer grain level of parallelism;
- `barrier` specifies that all the child threads spawned by the execution of S are suspended before exiting until they are all finished – an OpenCL example illustrating spawn to encode the `clEnqueueTask` instruction and barrier to encode the `clEnqueueBarrier` instruction is provided in Figure 4.5;
- `single` ensures that S is executed by only one thread in its thread team (a thread team is the set of all the threads spawned within the innermost parallel `forloop` statement) and a barrier exists at the end of a single operation if its `synchronization_single` value is true;
- `atomic` predicates the execution of S via the acquisition of a lock to ensure exclusive access; at any given time, S can be executed by only

```
mode = OUT_OF_ORDER_EXEC_MODE_ENABLE;
commands = clCreateCommandQueue(context,
                        device_id, mode, &err);
clEnqueueTask(commands, kernel_A, 0, NULL, NULL);
clEnqueueTask(commands, kernel_B, 0, NULL, NULL);
// synchronize so that Kernel C starts only
// after Kernels A and B have finished
clEnqueueBarrier(commands);
clEnqueueTask(commands, kernel_C, 0, NULL, NULL);
```

```
barrier(
  spawn(zero, kernel_A(...));
  spawn(one, kernel_B(...))
);
spawn(zero,kernel_C(...))
```

Figure 4.5: OpenCL example illustrating spawn and barrier statements

one thread. Locks are logical memory addresses, represented here by a member of the PIPS IR `reference` set presented in Section 2.6.5. An example illustrating how an atomic synchronization on the reference `l` in a SPIRE statement accessing Array `x` can be translated in Cilk (via `Cilk_lock` and `Cilk_unlock`) and OpenMP (`critical`) is provided in Figure 4.6.

```
Cilk_lockvar l;
Cilk_lock_init(l);
...
Cilk_lock(l);
 x[index[i]] += f(i);
Cilk_unlock(l);
```

```
#pragma omp critical
  x[index[i]] += f(i);
```

Figure 4.6: Cilk and OpenMP examples illustrating an atomically-synchronized statement

Event API: Point-to-Point Synchronization

The second possibility addresses the case of point-to-point synchronization between participating threads. Handling point-to-point synchronization using decorations on abstract syntax trees is too constraining when one has to deal with a varying set of threads that may belong to different parallel parent nodes. Thus, SPIRE suggests to deal with this last class of coordination using a new class of values, of the `event` type. SPIRE extends the

underlying type system of the existing sequential IRs with a new basic type, namely `event`:

```
type' = type + event:unit ;
```

Values of type `event` are counters, in a manner reminiscent of semaphores. The programming interface for events is defined by the following atomic functions:

- `event newEvent(int i)` is the creation function of events, initialized with the integer `i` that specifies how many threads can execute `wait` on this event without being blocked;
- `void freeEvent(event e)` is the deallocation function of the event `e`;
- `void signal(event e)` increments by one the event value[2] of `e`;
- `void wait(event e)` blocks the thread that calls it until the value of `e` is strictly greater than 0. When the thread is released, this value is decremented by one.

In a first example of possible use of this event API, the construct `future` used in X10 (see Figure 4.7) can be seen as the spawning of the computation of `foo()`. The end result is obtained via the call to the `force` method; such a mechanism can be easily implemented in SPIRE using an event attached to the running task; it is `signal`ed when the task is completed and `wait`ed by the `force` method.

```
future<int> Fi = future{foo()};
int i = Fi.force();
```

Figure 4.7: X10 example illustrating a future task and its synchronization

A second example, taken from Habanero-Java, illustrates how point-to-point synchronization primitives such as phasers and the `next` statement can be dealt with using the Event API (see Figure 4.8, left). The `async phased` keyword can be replaced by `spawn`. In this example, the `next` statement is equivalent to the following sequence:

```
signal(ph);
wait(ph);
signal(ph);
```

where the event `ph` is supposed initialized to `newEvent (-(n-1))`; the second `signal` is used to resume the suspended tasks in a chain-like fashion.

[2]The `void` return type will be replaced by `int` in practice, to enable the handling of error values.

```
finish{
 phaser ph=new phaser();
 for(j = 1;j <= n;j++){
  async phased(ph<SIG_WAIT>){
   S;
   next;
   S′;
  }
 }
}
```

```
barrier(
  ph=newEvent(-(n-1));
  j = 1;
  loop(j <= n,
    spawn(j,
          S;
          signal(ph);
          wait(ph);
          signal(ph);
          S');
    j = j+1);
  freeEvent(ph)
)
```

Figure 4.8: A phaser in Habanero-Java, and its SPIRE core language representation

In order to illustrate the importance of the constructs implementing point-to-point synchronization and thus the necessity to handle them, we show the code in Figure 4.9 extracted from [99], where phasers can be used to implement one of main types of parallelism namely the pipeline parallelism (see Section 2.2).

```
finish {
  phaser [] ph = new phaser[m+1];
  for (int i = 1; i < m; i++)
    async phased (ph[i]<SIG>, ph[i-1]<WAIT>){
      for (int j = 1; j < n; j++) {
        a[i][j] = foo(a[i][j], a[i][j-1], a[i-1][j-1]);
        next;
      }
    }
  }
}
```

Figure 4.9: Example of Pipeline Parallelism with phasers

Representing high-level point-to-point synchronization such as phasers and futures using low-level functions of `event` type constitutes a trade-off between generality and conciseness: we can represent any type of synchronization via the low-level interface using events, and expressibility of representation: we lose some of the expressiveness and precision of these high-level constructs such as the deadlock-freedom safety property of phasers [102]. Indeed, an Habanero-Java parallel code avoids deadlock since the scope of each used phaser is the immediately enclosing `finish`. However, the scope of the

events is less restrictive, the user can put them wherever he wants, and this may lead to deadlock appearance.

4.2.4 Data Distribution

The choice of a proper memory model to express parallel programs is an important issue when designing a generic intermediate representation. There are usually two main approaches to memory modeling: shared and message passing models. Since SPIRE is designed to extend existing IRs for sequential languages, it can be straightforwardly seen as using a shared memory model when parallel constructs are added. By convention, we say that `spawn` creates processes, in the case of message passing memory models, and threads, in the other case.

In order to take into account the explicit distribution required by the message passing memory model used in parallel languages such as MPI, SPIRE introduces the `send` and `recv` blocking functions for implementing communication between processes:

- `void send(int dest, entity buf)` transfers the value in Entity `buf` to the process numbered `dest`;

- `void recv(int source, entity buf)` receives in `buf` the value sent by Process `source`.

The MPI example in Figure 4.10 can be represented in SPIRE as a parallel loop with index `rank` of `size` iterations whose body is the MPI code from `MPI_Comm_size` to `MPI_Finalize`. The communication of Variable `sum` from Process 1 to Process 0 can be handled with SPIRE `send/recv` functions.

Note that non-blocking communications can be easily implemented in SPIRE using the above primitives within `spawn`ed statements (see Figures 4.11 and 4.12).

A non-blocking receive should indicate the completion of the communication. We may use for instance `finish_recv` (see Figure 4.12) that can be accessed later by the receiver to test the status of the communication.

Also, broadcast collective communications, such as defined in MPI, can be represented via wrappers around `send` and `recv` operations. When the master process and receiver processes want to perform a broadcast operation, then, if this process is the master, its broadcast operation is equivalent to a loop over receivers, with a call to `send` as body; otherwise (receiver), the broadcast is a `recv` function. Figure 4.13 illustrates the SPIRE representation of a a broadcast collective communication of the value of sum.

Another interesting memory model for parallel programming has been introduced somewhat recently: the Partitioned Global Address Space [113]. The uses of the PGAS memory model in languages such as UPC [34],

```
MPI_Init(&argc, &argv[]);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank == 0)
  MPI_Recv(sum, sizeof(sum), MPI_FLOAT, 1, 1,
           MPI_COMM_WORLD, &stat);
else if(rank == 1){
  sum = 42;
  MPI_Send(sum, sizeof(sum), MPI_FLOAT, 0, 1,
           MPI_COMM_WORLD);
}
MPI_Finalize();
```

```
forloop(rank,0,size,1,
        if(rank==0,
             recv(one,sum),
             if(rank==1,
                  sum=42;
                  send(zero,sum),
                  nop)
            ),
        parallel)
```

Figure 4.10: MPI example illustrating a communication, and its SPIRE core language representation

```
spawn(new, send(zero, sum))
```

Figure 4.11: SPIRE core language representation of a non-blocking send

```
atomic(finish_recv = false);
spawn(new, recv(one, sum);
           atomic(finish_recv = true)
     )
```

Figure 4.12: SPIRE core language representation of a non-blocking receive

```
if(rank==0,
     forloop(rank, 1, size, 1, send(rank,sum),
             parallel),
     recv(zero,sum))
```

Figure 4.13: SPIRE core language representation of a broadcast

Habanero-Java, X10 and Chapel introduce various notions such as `Place` or `Locale` to label portions of a logically-shared memory that threads may access, in addition to complex APIs for distributing objects/arrays over these portions. Given the wide variety of current proposals, we leave the issue of integrating the PGAS model within the general methodology of SPIRE as future work.

4.3 SPIRE Operational Semantics

The purpose of the formal definition given in this section is to provide a solid basis for program analyses and transformations. It is a systematic way to specify our IR extension mechanism, something seldom present in IR definitions. It also illustrates how SPIRE leverages the syntactic and semantic level of sequential constructs to parallel ones, preserving the traits of the sequential operational semantics.

Fundamentally, at the syntactic and semantic levels, SPIRE is a methodology for expressing representation transformers, mapping the definition of a sequential language IR to a parallel version. We define the operational semantics of SPIRE in a two-step fashion: we introduce (1) a minimal core parallel language that we use to model fundamental SPIRE concepts and for which we provide a small-step operational semantics and (2) rewriting rules that translate the more complex constructs of SPIRE into this core language.

4.3.1 Sequential Core Language

Illustrating the transformations induced by SPIRE requires the definition of a sequential IR basis, as is done above, via PIPS IR. Since we focus here on the fundamentals, we use as core language a simpler, minimal sequential language, `Stmt`. Its syntax is given in Figure 4.14, where we assume that the sets `Ide` of identifiers `I` and `Exp` of expressions `E` are given.

```
S ∈ Stmt::=
            nop | I=E | S1;S2 | loop(E,S)

S ∈ SPIRE(Stmt)::=
                   nop | I=E | S1;S2 | loop(E,S) |
                   spawn(I,S) |
                   barrier(S) | barrier_wait(n) |
                   wait(I) | signal(I) |
                   send(I,I') | recv(I,I')
```

Figure 4.14: `Stmt` and `SPIRE(Stmt)` syntaxes

Sequential statements are: (1) `nop` for no operation, (2) `I=E` for an assignment of `E` to `I`, (3) $\texttt{S}_1\texttt{;S}_2$ for a sequence and (4) `loop(E,S)` for a while loop.

At the semantic level, a statement in `Stmt` is a very simple memory transformer. A memory $m \in Memory$ is a mapping in $\texttt{Ide} \to Value$, where values $v \in Value = N + Bool$ can either be integers $n \in N$ or booleans $b \in Bool$. The sequential operational semantics for `Stmt`, expressed as transition rules over configurations $\kappa \in Configuration = Memory \times \texttt{Stmt}$, is given in Figure 4.15; we assume that the program is syntax- and type-correct. A transition $(m, \texttt{S}) \to (m', \texttt{S}')$ means that executing the statement S in a memory m yields a new memory m' and a new statement $\texttt{S}'$; we posit that the "$\to$" relation is transitive. Rules 4.1 to 4.5 encode typical sequential small-step operational semantic rules for the sequential part of the core language. We assume that $\zeta \in \texttt{Exp} \to Memory \to Value$ is the usual function for expression evaluation.

$$\frac{v = \zeta(\texttt{E})m}{(m, \texttt{I} = \texttt{E}) \to (m[\texttt{I} \to v], \texttt{nop})} \tag{4.1}$$

$$(m, \texttt{nop}; \texttt{S}) \to (m, \texttt{S}) \tag{4.2}$$

$$\frac{(m, \texttt{S}_1) \to (m', \texttt{S}_1')}{(m, \texttt{S}_1; \texttt{S}_2) \to (m', \texttt{S}_1'; \texttt{S}_2)} \tag{4.3}$$

$$\frac{\zeta(\texttt{E})m}{(m, \texttt{loop(E,S)}) \to (m, \texttt{S}; \texttt{loop(E,S)})} \tag{4.4}$$

$$\frac{\neg\zeta(\texttt{E})m}{(m, \texttt{loop(E,S)}) \to (m, \texttt{nop})} \tag{4.5}$$

Figure 4.15: `Stmt` sequential transition rules

The semantics of a whole sequential program `S` is defined as the memory m such that $(\perp, \texttt{S}) \to (m, \texttt{nop})$, if the execution of `S` terminates.

4.3.2 SPIRE as a Language Transformer

Syntax

At the syntactic level, SPIRE specifies how a grammar for a sequential language such as `Stmt` is transformed, i.e., extended, with synchronized parallel statements. The grammar of `SPIRE(Stmt)` in Figure 4.14 adds to the sequential statements of `Stmt` (from now on, synchronized using the default

`none`) new parallel statements: a task creation `spawn`, a termination `barrier` and two `wait` and `signal` operations on events or `send` and `recv` operations for communication. Synchronizations `single` and `atomic` are defined via rewriting (see Subsection 4.3.3). The statement `barrier_wait(`n`)`, added here for specifying the multiple-step behavior of the `barrier` statement in the semantics, is not accessible to the programmer. Figure 4.8 provides the SPIRE representation of a program example.

Semantic Domains

SPIRE is an intermediate representation transformer; As it extends grammars, it also extends semantics. The set of values manipulated by `SPIRE(Stmt)` statements extends the sequential $Value$ domain with events $e \in Event = N$, that encode events current values; we posit that $\zeta(\texttt{newEvent(E)})m = \zeta(\texttt{E})m$.

Parallelism is managed in SPIRE via processes (or threads). We introduce control state functions $\pi \in State = Proc \rightarrow Configuration \times Procs$ to keep track of the whole computation, mapping each process $i \in Proc = N$ to its current configuration (i.e., the statement it executes and its own view of memory m and the set $c \in Procs = \mathcal{P}(Proc)$ of the process children it has spawned during its execution.

In the following, we note $domain(\pi) = \{i \in Proc/\pi(i) \text{ is defined}\}$ the set of currently running processes, and $\pi[i \rightarrow (\kappa, c)]$ the state π extended at i with (κ, c). A process is said to be *finished* if and only if all its children processes, in c, are also finished, i.e., when only `nop` is left to execute: $finished(\pi, c) = (\forall i \in c, \exists c_i \in Procs, \exists m_i \in Memory/\pi(i) = ((m_i, \texttt{nop}), c_i) \wedge finished(\pi, c_i))$.

Memory Models

The memory model of sequential languages is a unique address space for identifier values. In our parallel extension, a configuration for a given process or thread includes its view of memory. We suggest to use the same semantic rules, detailed below, to deal with both shared and message passing memory rules. The distinction between these models, beside the additional use of send/receive constructs in the message passing model versus events in the shared one, is included in SPIRE via constraints we impose on the control states π used in computations. Namely, we introduce the variable $\texttt{_pids} \in \texttt{Ide}$, such that $m(\texttt{_pids}) \in \mathcal{P}(\texttt{Ide})$, that is used to harvest the process identities and we posit that, in the shared memory model, for all threads i and i' with $\pi(i) = ((m, \texttt{S}), c)$ and $\pi(i') = ((m', \texttt{S}'), c')$, one has $\forall \texttt{I} \in (domain(m) \cap domain(m')) - (m(\texttt{_pids}) \cup m'(\texttt{_pids})), m(\texttt{I}) = m'(\texttt{I})$. No such constraint is needed for the message passing model. Regarding the notion of memory equality, note that the issue of private variables in threads would have to be introduced in full-fledged languages. As mentioned above,

PGAS is left for future work; some sort of constraints based on the characteristics of the address space partitioning for places/locales would have to be introduced.

Semantic Rules

At the semantic level, SPIRE is thus a transition system transformer, mapping rules such as the ones in Figure 4.15 to parallel, synchronized transition rules in Figure 4.16. A transition $(\pi[i \to ((m, \texttt{S}), c)]) \hookrightarrow (\pi'[i \to ((m', \texttt{S}'), c')])$ means that the i-th process, when executing S in a memory m, yields a new memory m' and a new control state $\pi'[i \to ((m', \texttt{S}'), c')]$ in which this process now will execute $\texttt{S}'$; additional children processes may have been created in c' compared to c. We posit that the "$\hookrightarrow$" relation is transitive.

Rule 4.6 is a key rule to specify SPIRE transformer behavior, providing a bridge between the sequential and the SPIRE-extended parallel semantics; all processes can non-deterministically proceed along their sequential semantics "$\to$", leading to valid evaluation steps along the parallel semantics "$\hookrightarrow$". The interleaving between parallel processes in `SPIRE(Stmt)` is a consequence of (1) the non-deterministic choice of the value of i within $domain(\pi)$ when selecting the transition to perform and (2) the number of steps executed by the sequential semantics. Note that one might want to add even more non-determinism in our semantics; indeed, Rule 4.1 is atomic: loading the variables in `E` and performing the store operation to `I` are performed in one sequential step. To simplify this presentation, we do not provide the simple intermediate steps in the sequential evaluation semantics of Rule 4.1 that would have removed this artificial atomicity.

The remaining rules focus on parallel evaluation. In Rule 4.7, if Process n does not already exist, `spawn` adds to the state a new process n that inherits the parent memory m in a fork-like manner; the set of processes spawned by n is initially equal to $\emptyset$. Otherwise, n keeps its memory m_n and its set of spawned processes c_n. In both cases, the value of `_pids` in the memory of n is updated by the identifier `I`, n executes `S` and is added to the set of processes c spawned by i. Rule 4.8 implements a rendezvous: a new process n executes S, while process i is suspended as long as $finished$ is not true; indeed, the rule 4.9 resumes execution of process i when all the child processes spawned by n have finished. In Rules 4.10 and 4.11, `I` is an `event`, that is a counting variable used to control access to a resource or to perform a point-to-point synchronization, initialized via `newEvent` to a value equal to the number of processes that will be granted access to it. Its current value n is decremented every time a `wait(I)` statement is executed and, when $\pi(\texttt{I}) = n$ with $n > 0$, the resource can be used or the barrier can be crossed. In Rule 4.11, the current value n' of `I` is incremented; this is a non-blocking operation. In Rule 4.12, p and p' are two processes that

$$\frac{\kappa \to \kappa'}{\pi[i \to (\kappa, c)] \hookrightarrow \pi[i \to (\kappa', c)]} \tag{4.6}$$

$$\frac{\begin{array}{c} n = \zeta(\texttt{I})m \\ ((m_n, S_n), c_n) = (n \in domain(\pi)) \ ? \ \pi(n) : ((m, \texttt{S}), \emptyset) \\ m'_n = m_n[\texttt{_pids} \to m_n(\texttt{_pids}) \cup \{\texttt{I}\}] \end{array}}{\begin{array}{l} \pi[i \to ((m, \texttt{spawn(I,S)}), c)] \hookrightarrow \\ \qquad \pi[i \to ((m, \texttt{nop}), c \cup \{n\})][n \to ((m'_n, \texttt{S}), c_n)] \end{array}} \tag{4.7}$$

$$\frac{n \notin domain(\pi) \cup \{i\}}{\begin{array}{l} \pi[i \to ((m, \texttt{barrier(S)}), c)] \hookrightarrow \\ \qquad \pi[i \to (m, \texttt{barrier_wait}(n)), c)][n \to ((m, \texttt{S}), \emptyset)] \end{array}} \tag{4.8}$$

$$\frac{\begin{array}{c} finished(\pi, \{n\}) \\ \pi(n) = ((m', \texttt{nop}), c') \end{array}}{\pi[i \to ((m, \texttt{barrier_wait}(n)), c)] \hookrightarrow \pi[i \to ((m', \texttt{nop}), c)]} \tag{4.9}$$

$$\frac{n = \zeta(\texttt{I})m \qquad n > 0}{\pi[i \to ((m, \texttt{wait(I)}), c)] \hookrightarrow \pi[i \to ((m[\texttt{I} \to n-1], \texttt{nop}), c)]} \tag{4.10}$$

$$\frac{n = \zeta(\texttt{I})m}{\pi[i \to ((m, \texttt{signal(I)}), c)]) \hookrightarrow \pi[i \to ((m[\texttt{I} \to n+1], \texttt{nop}), c)]} \tag{4.11}$$

$$\frac{p' = \zeta(\texttt{P}')m \qquad p = \zeta(\texttt{P})m'}{\begin{array}{l} \pi[p \to ((m, \texttt{send(P',I)}), c)][p' \to ((m', \texttt{recv(P,I')}), c')] \hookrightarrow \\ \qquad \pi[p \to ((m, \texttt{nop}), c)][p' \to ((m'[\texttt{I'} \to m(\texttt{I})], \texttt{nop}), c')] \end{array}} \tag{4.12}$$

Figure 4.16: `SPIRE(Stmt)` synchronized transition rules

communicate: p sends the datum `I` to p', while this latter consumes it in `I'`.

The semantics of a whole parallel program `S` is defined as the set of memories m such that $\bot[0 \to ((\bot, \texttt{S}), \emptyset)] \hookrightarrow \pi[0 \to ((m, \texttt{nop}), c)]$, if `S` terminates.

4.3.3 Rewriting Rules

The SPIRE concepts not dealt with in the previous section are defined via their rewriting into the core language. This is the case for both the treatment

of the `execution` attribute and the remaining coarse-grain synchronization constructs.

Execution

A parallel `sequence` of statements S_1 and S_2 is a pair of independent substatements executed simultaneously by spawned processes I_1 and I_2 respectively, i.e., is equivalent to:

```
spawn(I1,S1);spawn(I2,S2)
```

A parallel `forloop` (see Figure 4.2) with index `I`, lower expression `low`, upper expression `up`, step expression `step` and body `S` is equivalent to:

```
I=low;loop(I<=up,spawn(I,S);I=I+step)
```

A parallel `unstructured` is rewritten as follows. All control nodes present in the transitive closure of the successor relation are rewritten in the same manner. Each control node `C` is characterized by a statement `S`, predecessor list `ps` and successor list `ss`. For each edge (`c,C`), where `c` is a predecessor of `C` in `ps`, an event $e_{c,C}$ initialized at `newEvent(0)` is created, and similarly for `ss`. The whole unstructured construct is replaced by a sequential sequence of `spawn(I,`S_c`)`, one for each `C` of the transitive closure of the successor relation starting at the `entry` control node, where S_c is defined as follows:

```
barrier(spawn(one,wait(e_{ps[1],C}));...;
        spawn(m,wait(e_{ps[m],C})));
S;
signal(e_{C,ss[1]});...;signal(e_{C,ss[m']})
```

where `m` and `m'` are the lengths of the `ps` and `ss` lists; `L[j]` is the `j`-th element of `L`.

The code corresponding to the statement `z=x+y` taken from the parallel unstructured graph showed in Figure 4.3 is rewritten as:

```
spawn(three,
      barrier(spawn(one,wait(e_{1,3}));
              spawn(two,wait(e_{2,3})));
      z = x + y;  // statement 3
      signal(e_{3,exit})
      )
```

Synchronization

A statement `S` with synchronization `atomic(I)` is rewritten as:

```
wait(I);S;signal(I)
```

assuming that the assignment `I = newEvent(1)` is performed on the `event` identifier `I` at the very beginning of the `main` function. A `wait` on an event variable sets it to zero if it is currently equal to one to prohibit other threads to enter the atomic section; the `signal` resets the event variable to one to permit further access.

A statement `S` with a blocking synchronization `single`, i.e., equal to `true`, is equivalent, when it occurs within an enclosing innermost parallel `forloop`, to:

```
barrier(wait(I_S);
        if(first_S,
           S; first_S = false,
           nop);
        signal(I_S))
```

where `first_S` is a boolean variable that ensures that only one process among those spawned by the parallel loop will execute `S`; access to this variable is protected by the event `I_S`. Both `first_S` and `I_S` are respectively initialized before loop entry to `true` and `newEvent(1)`.

The conditional `if(E,S,S')` can be rewritten using the core `loop` construct as illustrated in Figure 4.17. The same rewriting can be used when the `single` synchronization is equal to `false`, corresponding to a non-blocking synchronization construct, except that no `barrier` is needed.

`if(E,S,S′)`	`stop = false;`
	`loop(E ∧ ¬stop, S; stop = true);`
	`loop(¬E ∧ ¬stop, S′; stop = true)`

Figure 4.17: `if` statement rewriting using `while` loops

4.4 Validation: SPIRE Application to LLVM IR

Assessing the quality of a methodology that impacts the definition of a data structure as central for compilation frameworks as an intermediate representation is a difficult task. This section illustrates how it can be used on a different IR, namely LLVM, with minimal changes, thus providing support regarding the generality of our methodology. We show how simple it is to upgrade the LLVM IR to a "parallel LLVM IR" via SPIRE transformation methodology.

LLVM [77] (Low-Level Virtual Machine) is an open source compilation framework that uses an intermediate representation in Static Single Assignment (SSA) [38] form. We chose the IR of LLVM to illustrate a second time our approach because LLVM is widely used in both academia and industry; another interesting feature of LLVM IR, compared to PIPS's, is that it

sports a graph approach, while PIPS is mostly abstract syntax tree-based; each function is structured in LLVM as a control flow graph (CFG).

Figure 4.18 provides the definition of a significant subset of the sequential LLVM IR described in [77] (to keep notations simple in this thesis, we use the same Newgen language to write this specification):

- a function is a list of basic blocks, which are portions of code with one entry and one exit points;
- a basic block has an entry label, a list of ϕ nodes and a list of instructions, and ends with a terminator instruction;
- ϕ nodes, which are the key elements of SSA, are used to merge the values coming from multiple basic blocks. A ϕ node is an assignment (represented here as a call expression) that takes as arguments an identifier and a list of pairs (value, label); it assigns to the identifier the value corresponding to the label of the block preceding the current one at run time;
- every basic block ends with a terminator which is a control flow-altering instruction that specifies which block to execute after termination of the current one.

An example of the LLVM encoding of a simple for loop in three basic blocks is provided in Figure 4.19 where, in the first assignment, which uses a ϕ node, `sum` is `42` if it is the first time we enter the `bb1` block (from `entry`) or `sum` otherwise (from the branch `bb`).

```
function              = blocks:block*;
block                 = label:entity x phi_nodes:phi_node* x
                        instructions:instruction* x
                        terminator;
phi_node              = call;
instruction           = call;
terminator            = conditional_branch +
                        unconditional_branch +
                        return;
conditional_branch    = value:entity x label_true:entity x
                        label_false:entity;
unconditional_branch  = label:entity;
return                = value:entity;
```

Figure 4.18: Simplified Newgen definitions of the LLVM IR

Applying SPIRE to LLVM IR is, as illustrated above with PIPS, achieved in three steps, yielding the SPIREd parallel extension of the LLVM sequential IR provided in Figure 4.20:

```
sum = 42;
for(i=0; i<10; i++)
  sum = sum + 2;
```

```
entry:
  br label %bb1
bb:           ; preds = %bb1
  %0 = add nsw i32 %sum.0, 2
  %1 = add nsw i32 %i.0, 1
  br label %bb1
bb1:          ; preds = %bb, %entry
  %sum.0 = phi i32 [42,%entry],[%0,%bb]
  %i.0 = phi i32 [0,%entry],[%1,%bb]
  %2 = icmp sle i32 %i.0, 10
  br i1 %2, label %bb, label %bb2
```

Figure 4.19: A loop in C and its LLVM intermediate representation

- an `execution` attribute is added to `function` and `block`: a parallel basic block sees all its instructions launched in parallel, while all the blocks of a parallel function are seen as parallel tasks to be executed concurrently;
- a `synchronization` attribute is added to `instruction`; therefore, an instruction can be annotated with `spawn`, `barrier`, `single` or `atomic` synchronization attributes. When one wants to consider a sequence of instructions as a whole, this sequence is first outlined in a "function", to be called instead; this new call instruction is then annotated with the proper synchronization attribute, such as `spawn`, if the sequence must be considered as an asynchronous task; we provide an example in Figure 4.21. A similar technique is used for the other synchronization constructs `barrier`, `single` and `atomic`;
- as LLVM provides a set of intrinsic functions [77], SPIRE functions `newEvent`, `freeEvent`, `signal` and `wait` for handling point-to-point synchronization, and `send` and `recv` for handling data distribution, are added to this set.

```
function′    = function x execution;
block′       = block x execution;
instruction′ = instruction x synchronization;
Intrinsic Functions:
       send, recv, signal, wait, newEvent, freeEvent
```

Figure 4.20: SPIRE (LLVM IR)

Note that the use of SPIRE on the LLVM IR is not able to express parallel loops as easily as was the case on PIPS IR. Indeed, the notion of a

```
main()
{
  int A, B, C;
  A = foo();
  B = bar(A);
  C = baz();
  ...
}
```

```
call_AB(int *A,int *B){
  *A = foo();
  *B = bar(*A);
}
main(){
  int A, B, C;
  spawn(zero,call_AB(&A,&B));
  C = baz();
  ...
}
```

Figure 4.21: An example of a `spawn`ed outlined sequence

loop does not always exist in the definition of intermediate representations based on control flow graphs, including LLVM; it is an attribute of some of its nodes, which has to be added later on by a loop-detection program analysis phase. Of course, such analysis could also be applied on the SPIRE-derived IR, to thus recover this information. Once one knows that a particular loop is parallel, this can be encoded within SPIRE using the same technique as presented in Section 4.3.3.

More generally, even though SPIRE uses traditional parallel paradigms for code generation purposes, SPIRE-derived IRs are able to deal with more specific parallel constructs such as DOACROSS loops [57] or HELIX-like [29] approaches. HELIX is a technique to parallelize a loop in the presence of loop-carried dependences by executing sequential segments with exploiting TLP between them and using prefetching of signal synchronization. Basically, a compiler would parse a given sequential program into sequential IR elements. Optimization compilation phases specific to particular parallel code generation paradigms such as those above will translate, whenever possible (specific data and control-flow analyses will be needed here), these sequential IR constructs into parallel loops, with the corresponding synchronization primitives, as need be. Code generation will then recognize such IR patterns and generate specific parallel instructions such as DOACROSS.

4.5 Related Work: Parallel Intermediate Representations

In this section, we review several different possible representations of parallel programs, both at the high, syntactic, and mid, graph-based, levels. We provide synthetic descriptions of the key existing IRs addressing similar issues to our thesis's. Bird's eye view comparisons with SPIRE are also given here.

Syntactic approaches to parallelism expression use abstract syntax tree nodes, while adding specific built-in functions for parallelism. For instance, the intermediate representation of the implementation of OpenMP in GCC (GOMP) [86] extends its three-address representation, GIMPLE [79]. The OpenMP parallel directives are replaced by specific built-ins in low- and high-level GIMPLE, and additional nodes in high-level GIMPLE, such as the `__sync_fetch_and_add` built-in function for an atomic memory access addition. Similarly, Sarkar and Zhao introduce the high-level parallel intermediate representation HPIR [114] that decomposes Habanero-Java programs into region syntax trees, while maintaining additional data structures on the side: region control-flow graphs and region dictionaries that contain information about the use and the definition of variables in region nodes. New abstract syntax tree nodes are introduced: `AsyncRegionEntry` and `AsyncRegionExit` are used to delimit tasks, while `FinishRegionEntry` and `FinishRegionExit` can be used in parallel sections. SPIRE borrows some of the ideas used in GOMP or HPIR, but frames them in more structured settings while trying to be more language-neutral. In particular, we try to minimize the number of additional built-in functions, which have the drawback of hiding the abstract high-level structure of parallelism and affecting compiler optimization passes. Moreover, we focus on extending existing AST nodes rather than adding new ones (such as in HPIR) in order to not fatten the IR and avoid redundant analyses and transformations on the same basic constructs. Applying the SPIRE approach to systems such as GCC would have provided a minimal set of extensions that could have also been used for other implementations of parallel languages that rely on GCC as a backend, such as Cilk.

PLASMA is a programming framework for heterogeneous SIMD systems, with an IR [91] that abstracts data parallelism and vector instructions. It provides specific operators such as `add` on vectors and special instructions such as `reduce` and `par`. While PLASMA abstracts SIMD implementation and compilation concepts for SIMD accelerators, SPIRE is more architecture-independent and also covers control parallelism.

InsPIRe is the parallel intermediate representation at the core of the source-to-source Insieme compiler [58] for C, C++, OpenMP, MPI and OpenCL programs. Parallel constructs are encoded using built-ins. SPIRE intends to also cover source-to-source optimization. We believe it could have been applied to Insieme sequential components, parallel constructs being defined as extensions of the sequential abstract syntax tree nodes of InsPIRe instead of using numerous built-ins.

Turning now to mid-level intermediate representations, many systems rely on graph structures for representing sequential code, and extend them for parallelism. The Hierarchical Task Graph [49] represents the program control flow. The hierarchy exposes the loop nesting structure; at each loop nesting level, the loop body is hierarchically represented as a single node

that embeds a subgraph that has control and data dependence information associated with it. SPIRE is able to represent both structured and unstructured control-flow dependence, thus enabling recursively-defined optimization techniques to be applied easily. The hierarchical nature of underlying sequential IRs can be leveraged, via SPIRE, to their parallel extensions; this feature is used in the PIPS case addressed below.

A stream graph [31] is a dataflow representation introduced specifically for streaming languages. Nodes represent data reorganization and processing operations between streams, and edges, communications between nodes. The number of data samples defined and used by each node is supposed to be known statically. Each time a node is fired, it consumes a fixed number of elements of its inputs and produces a fixed number of elements on its outputs. SPIRE provides support for both data and control dependence information; streaming can be handled in SPIRE using its point-to-point synchronization primitives.

The parallel program graph (PPDG) [100] extends the program dependence graph [45], where vertices represent blocks of statements and edges, essential control or data dependences; *mgoto* control edges are added to represent task creation occurrences, and synchronization edges, to impose ordering on tasks. Kimble IR [24] uses an intermediate representation designed along the same lines, i.e., as a hierarchical direct acyclic graphs (DAG) on top of GCC IR, GIMPLE. Parallelism is expressed there using new types of nodes: *region*, which is a subgraph of *cluster*s, and *cluster*, a sequence of statements to be executed by one thread. Like PPDG and Kimble IR, SPIRE adopts an extension approach to "parallelize" existing sequential intermediate representations; this chapter showed that this can be defined as a general mechanism for parallel IR definitions and provides a formal specification of this concept.

LLVM IR [77] represents each function as control flow graphs. To encode parallel constructs, LLVM introduces the notion of metadata such as `llvm.loop.parallel` for implementing parallel loops. A metadata is a string used as an annotation on the LLVM IR nodes. LLVM IR lacks support for other parallel constructs such as starting parallel threads, synchronizing them, etc. We presented in Section 4.4 our proposal for a parallel IR for LLVM via the application of SPIRE to LLVM.

4.6 Conclusion

SPIRE is a new and general 3-step extension methodology for mapping any intermediate representation (IR) used in compilation platforms for representing sequential programming constructs to a parallel IR; one can leverage it for the source-to-source and high- to mid-level optimization of control-parallel languages and constructs.

The extension of an existing IR introduces (1) a parallel execution attribute for each group of statements, (2) a high-level synchronization attribute on each statement node and an API for low-level synchronization events and (3) two built-ins for implementing communications in message passing memory systems. The formal semantics of SPIRE transformational definitions is specified using a two-tiered approach: a small-step operational semantics for its base parallel concepts and a rewriting mechanism for high-level constructs.

The SPIRE methodology is presented via a use case, the intermediate representation of PIPS, a powerful source-to-source compilation infrastructure for Fortran and C. We illustrate the generality of our approach by showing how SPIRE can be used to represent the constructs of the current parallel languages Cilk, Chapel, X10, Habanero-Java, OpenMP, OpenCL and MPI. To validate SPIRE, we show the application of SPIRE on another IR, namely the one of the widely-used LLVM compilation infrastructure.

The extension of the sequential IR of PIPS using SPIRE has been implemented in the PIPS middle-end and used for parallel code transformations and generation (see Chapter 7). We added the execution set (`parallel` and `sequential` attributes), the synchronization set (`none`, `spawn`, `barrier`, `atomic` and `single` attributes), events and data distribution calls. Chapter 8 provides performance data, gathered using our implementation of SPIRE on PIPS IR, to illustrate the ability of the resulting parallel IR to efficiently express parallelism present in scientific applications. The next chapter addresses the crucial step in any parallelization process, which is scheduling.

An earlier version of the work presented in this chapter was presented at CPC [68] and at the HiPEAC Computing Systems Week.

Chapter 5

BDSC: A Memory-Constrained, Number of Processor-Bounded Extension of DSC

Life is too unpredictable to live by a schedule. **Anonymous**

In the previous chapter, we have introduced an extension methodology for sequential intermediate representations to handle parallelism. In this chapter, as we are interested in this thesis in task parallelism, we focus on the next step: finding or extracting this parallelism from a sequential code, to be eventually expressed using a SPIRE-derived parallel intermediate representation. To reach that goal, we introduce BDSC, which extends Yang and Gerasoulis's Dominant Sequence Clustering (DSC) algorithm. BDSC is a new efficient automatic scheduling algorithm for parallel programs in the presence of resource constraints on the number of processors and their local memory size; it is based on the static estimation of both execution time and communication cost. Thanks to the handling of these two resource parameters in a single framework, BDSC can address both shared and distributed parallel memory architectures. As a whole, BDSC provides a trade-off between concurrency gain and communication overhead between processors, under two resource constraints: finite number of processors and amount of memory.

Dans le chapitre précédent, nous avons mis en place une méthodologie d'extension pour les représentations intermédiaires séquentielles afin de pouvoir gérer le parallélisme. Dans ce chapitre, puisque nous nous intéressons dans cette thèse à la parallélisation de tâches, nous nous concentrons sur l'étape suivante : trouver ou extraire ce parallélisme d'un code séquentiel, pour être finalement exprimé en utilisant une représentation intermédiaire parallèle dérivée de SPIRE. Pour atteindre ce but, nous introduisons BDSC, qui étend l'algorithme DSC (Dominant Sequence Clustering) de Yang et Gerasoulis. BDSC est un nouvel algorithme d'ordonnancement automa-

tique et efficace pour les programmes parallèles en présence de contraintes de ressources sur le nombre de processeurs et la taille de leur mémoire locale. Ces contraintes sont fondées sur l'estimation statique à la fois des temps d'exécution et des coûts de communication. Grâce à la manipulation de ces deux paramètres de ressources dans un cadre unique, BDSC peut traiter à la fois les architectures parallèles à mémoire partagée et distribuée. Globalement, BDSC offre un compromis entre le gain de concurrence et le coût de communication entre les processeurs, sous deux contraintes de ressources : un nombre fini de processeurs et une quantité limitée de mémoire.

5.1 Introduction

One key problem when attempting to parallelize sequential programs is to find solutions to graph partitioning and scheduling problems, where vertices represent computational tasks and edges, data exchanges. Each vertex is labeled with an estimation of the time taken by the computation it performs; similarly, each edge is assigned a cost that reflects the amount of data that need to be exchanged between its adjacent vertices. Efficiency is strongly dependent here on the accuracy of the cost information encoded in the graph to be scheduled. Gathering such information is a difficult process in general, in particular in our case where tasks are automatically generated from program code.

We introduce thus a new non-preemptive static scheduling heuristic that strives to give as small as possible schedule lengths, i.e., parallel execution time, in order to exploit the task-level parallelism possibly present in sequential programs targeting homogeneous multiprocessors or multicores, while enforcing architecture-dependent constraints: the number of processors, the computational cost and memory use of each task and the communication cost for each task exchange, to provide hopefully significant speedups on shared and distributed computer architectures. Our technique, called BDSC, is based on an existing best-of-breed static scheduling heuristic, namely Yang and Gerasoulis's DSC (Dominant Sequence Clustering) list-scheduling algorithm [111] [48], that we equip to deal with new heuristics that handle resource constraints. One key advantage of DSC over other scheduling policies (see Section 5.4), besides its already good performance when the number of processors is unlimited, is that it has been proven optimal for fork/join graphs: this is a serious asset given our focus on the program parallelization process, since task graphs representing parallel programs often use this particular graph pattern. Even though this property may be lost when constraints are taken into account, our experiments on scientific benchmarks suggest that our extension still provides good performance (see Chapter 8).

This chapter is organized as follows. Section 5.2 presents the original

DSC algorithm that we intend to extend. We detail our heuristic extension, BDSC, in Section 5.3. Section 5.4 compares the main existing scheduling algorithms with BDSC. Finally Section 5.5 concludes the chapter.

5.2 The DSC List-Scheduling Algorithm

In this section, we present the list-scheduling heuristic called DSC [111] that we extend and enhance to be useful for automatic task parallelization under resource constraints.

5.2.1 The DSC Algorithm

DSC (Dominant Sequence Clustering) is a task list-scheduling heuristic for an unbounded number of processors. The objective is to minimize the top level of each task (see Section 2.5.2 for a review of the main notions of list scheduling). A DS (Dominant Sequence) is a path that has the longest length in a partially scheduled DAG; a graph critical path is thus a DS for the totally scheduled DAG. The DSC heuristic computes a Dominant Sequence (DS) after each vertex is processed, using $\texttt{tlevel}(\tau, G) + \texttt{blevel}(\tau, G)$ as $\texttt{priority}(\tau)$. Since priorities are updated after each iteration, DSC computes dynamically the critical path based on both top level and bottom level information. A ready vertex τ, i.e., all vertex all of whose all predecessors have already been scheduled, on one of the current DSs, i.e., with the highest priority, is clustered with a predecessor τ_p when this reduces the top level of τ by zeroing, i.e., setting to zero, the cost of communications on the incident edge (τ_p, τ). As said in Section 2.5.2, $task_time(\tau)$ and $edge_cost(e)$ are assumed to be numerical constants, although we show how we lift this restriction in Section 6.3.

The DSC instance of the general list-scheduling Algorithm 3, presented in Section 2.5.2, is provided in Algorithm 4 where `select_cluster` is replaced by the code in Algorithm 5 (`new_cluster` extends *clusters* with a new empty cluster; its cluster time is set to 0). The table in Figure 5.1 represents the result of scheduling the DAG in the same figure using the DSC algorithm.

To decide which predecessor τ_p to select to be clustered with τ, DSC applies the minimization procedure `tlevel_decrease` presented in Algorithm 6, which returns the predecessor that leads to the highest reduction of top level for τ if clustered together, and the resulting top level; if no zeroing is accepted, the vertex τ is kept in a new single vertex cluster. More precisely, the minimization procedure `tlevel_decrease` for a task τ, in Algorithm 6, tries to find the cluster $\texttt{cluster}(min_\tau)$ of one of its predecessors τ_p that reduces the top level of τ as much as possible by zeroing the cost of the edge (min_τ, τ). All clusters start at the same time, and each cluster is characterized by its running time, $\texttt{cluster_time}(\kappa)$, which is the cumulated time of all tasks scheduled into κ; idle slots within clusters may exist

ALGORITHM 4: DSC scheduling of Graph G on P processors

```
procedure DSC(G, P)
  clusters = ∅;
  foreach τ_i ∈ vertices(G)
    priority(τ_i) = tlevel(τ_i, G) + blevel(τ_i, G);
  UT = vertices(G);                // unscheduled tasks
  while UT ≠ ∅
    τ = select_task_with_highest_priority(UT);
    κ = select_cluster(τ, G, P, clusters);
    allocate_task_to_cluster(τ, κ, G);
    update_graph(G);
    update_priority_values(G);
    UT = UT-{τ};
end
```

ALGORITHM 5: DSC cluster selection for Task τ for Graph G on P processors

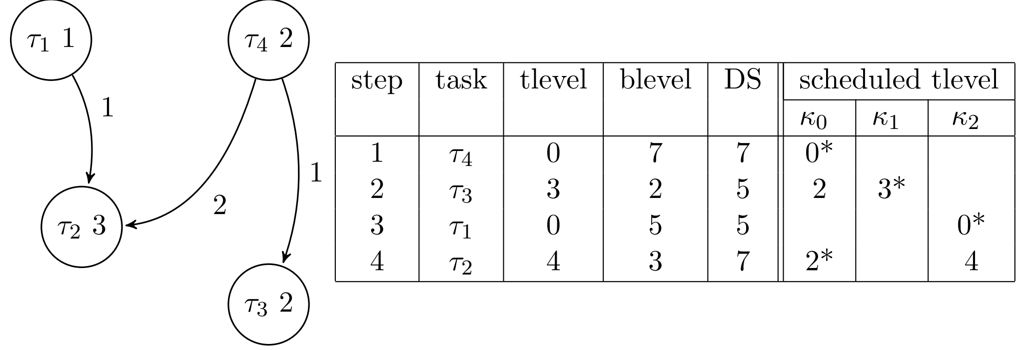
```
function select_cluster(τ, G, P, clusters)
  (min_τ, min_tlevel) = tlevel_decrease(τ, G);
  return (cluster(min_τ) ≠ cluster_undefined) ?
                cluster(min_τ) : new_cluster(clusters);
end
```

and are also taken into account in this accumulation process. The condition `cluster`(τ_p) $\neq$ `cluster_undefined` is tested on predecessors of τ in order to make it possible to apply this procedure for ready and unready vertices; an unready vertex has at least one unscheduled predecessor.

5.2.2 Dominant Sequence Length Reduction Warranty

Dominant Sequence Length Reduction Warranty (DSRW) is a greedy heuristic within DSC that aims to further reduce the scheduling length. A vertex on the DS path with the highest priority can be ready or not ready. With the DSRW heuristic, DSC schedules the ready vertices first, but, if such a ready vertex τ_r is not on the DS path, DSRW verifies, using the procedure in Algorithm 7, that the corresponding zeroing does not affect later the reduction of the top levels of the DS vertices τ_u that are partially ready, i.e., such that there exists at least one unscheduled predecessor of τ_u. To do this, we check if the "partial top level" of τ_u, which does not take into account unexamined (unscheduled) predecessors and is computed using `tlevel_decrease`, is reducible, once τ_r is scheduled.

The table in Figure 5.1 illustrates an example where it is useful to apply the DSRW optimization. There, the DS column provides, for the task



step	task	tlevel	blevel	DS	scheduled tlevel		
					κ_0	κ_1	κ_2
1	τ_4	0	7	7	0*		
2	τ_3	3	2	5	2	3*	
3	τ_1	0	5	5			0*
4	τ_2	4	3	7	2*		4

Figure 5.1: A Directed Acyclic Graph (left) and its scheduling (right); starred top levels (*) correspond to the selected clusters

ALGORITHM 6: Minimization DSC procedure for Task τ in Graph G

```
function tlevel_decrease(τ, G)
  min_tlevel = tlevel(τ, G);
  min_τ = τ;
  foreach τp ∈ predecessors(τ, G)
          where cluster(τp) ≠ cluster_undefined
    start_time = cluster_time(cluster(τp))),
    foreach τ'p ∈ predecessors(τ, G) where
            cluster(τ'p) ≠ cluster_undefined
      if(τp ≠ τ'p) then
        level = tlevel(τ'p, G)+task_time(τ'p)+edge_cost(τ'p, τ);
        start_time = max(level, start_time);
    if(min_tlevel > start_time) then
      min_tlevel = start_time;
      min_τ = τp;
  return (min_τ, min_tlevel);
end
```

scheduled at each step, its priority, i.e., the length of its dominant sequence, while the last column represents, for each possible zeroing, the corresponding task top level; starred top levels (*) correspond to the selected clusters. Task τ_4 is mapped to Cluster κ_0 in the first step of DSC. Then, τ_3 is selected because it is the ready task with the highest priority. The mapping of τ_3 to Cluster κ_0 would reduce its top level from 3 to 2. But the zeroing of (τ_4, τ_3) affects the top level of τ_2, τ_2 being the unready task with the highest priority. Since the partial top level of τ_2 is 2 with the zeroing of (τ_4,τ_2) but 4 after the zeroing of (τ_4,τ_3), DSRW will fail, and DSC allocates τ_3 to a new cluster, κ_1. Then, τ_1 is allocated to a new cluster, κ_2, since it has no predecessors. Thus, the zeroing of (τ_4,τ_2) is kept thanks to the DSRW optimization; the total scheduling length is 5 (with DSRW) instead

ALGORITHM 7: DSRW optimization for Task τ_u when scheduling Task τ_r for Graph G

```
function DSRW(τr, τu, clusters, G)
  (min_τ, min_tlevel) = tlevel_decrease(τr, G);

  // before scheduling τr
  (τb,ptlevel_before) = tlevel_decrease(τu, G);

  // scheduling τr
  allocate_task_to_cluster(τr, cluster(min_τ), G);
  saved_edge_cost = edge_cost(min_τ, τr);
  edge_cost(min_τ,τr) = 0;

  // after scheduling τr
  (τa,ptlevel_after) = tlevel_decrease(τu, G);
  if (ptlevel_after > ptlevel_before) then

    // (min_τ,τr) zeroing not accepted
    edge_cost(min_τ, τr) = saved_edge_cost;
    return false;
  return true;
end
```

κ_0	κ_1
τ_4	τ_1
τ_3	τ_2

κ_0	κ_1	κ_2
τ_4		τ_1
τ_2	τ_3	

Figure 5.2: Result of DSC on the graph in Figure 5.1 without (left) and with (right) DSRW

of 7 (without DSRW) (Figure 5.2).

5.3 BDSC: A Resource-Constrained Extension of DSC

This section details the key ideas at the core of our new scheduling process BDSC, which extends DSC with a number of important features, namely (1) by verifying predefined memory constraints, (2) by targeting a bounded number of processors and (3) by trying to make this number as small as possible.

5.3.1 DSC Weaknesses

A good scheduling solution is a solution that is built carefully, by having knowledge about previous scheduled tasks and tasks to execute in the future. Yet, as stated in [74], "an algorithm that only considers bottom level or only top level cannot guarantee optimal solutions". Even though tests carried out on a variety of scheduling algorithms show that the best competitor among them is the DSC algorithm [103], and DSC is a policy that uses the critical path for computing dynamic priorities based on both the bottom level and the top level for each vertex, it has some limits in practice.

The key weakness of DSC for our purpose is that the number of processors cannot be predefined; DSC yields blind clusterings, disregarding resource issues. Therefore, in practice, a thresholding mechanism to limit the number of generated clusters should be introduced. When allocating new clusters, one should verify that the number of clusters does not exceed a predefined threshold P (Section 5.3.3). Also, zeroings should handle memory constraints, by verifying that the resulting clustering does not lead to cluster data sizes that exceed a predefined cluster memory threshold M (Section 5.3.3).

Finally, DSC may generate a lot of idle slots in the created clusters. It adds a new cluster when no zeroing is accepted without verifying the possible existence of gaps in existing clusters. We handle this case in Section 5.3.4, adding an efficient idle cluster slot allocation routine in the task-to-cluster mapping process.

5.3.2 Resource Modeling

Since our extension deals with computer resources, we assume that each vertex in a task DAG is labeled with an additional information, `task_data`(τ), which is an over-approximation of the memory space used by Task τ; its size is assumed to be always strictly less than M. A similar `cluster_data` function applies to clusters, where it represents the collective data space used by the tasks scheduled within it. Since BDSC, as DSC, needs execution times and communication costs to be numerical constants, we discuss in Section 6.3 how this information is computed.

Our improvement to the DSC heuristic intends to reach a tradeoff between the gained parallelism and the communication overhead between processors, under two resource constraints: finite number of processors and amount of memory. We track these resources in our implementation of `allocate_task_to_cluster` given in Algorithm 8; note that the aggregation function `regions_union` is defined in Section 2.6.3.

Efficiently allocating tasks on the target architecture requires reducing the communication overhead and transfer cost for both shared and distributed memory architectures. If zeroing operations, that reduce the start

ALGORITHM 8: Task allocation of Task τ in Graph G to Cluster κ, with resource management

```
procedure allocate_task_to_cluster(τ, κ, G)
  cluster(τ) = κ;
  cluster_time(κ) = max(cluster_time(κ), tlevel(τ, G)) +
                    task_time(τ);
  cluster_data(κ) = regions_union(cluster_data(κ),
                                  task_data(τ));
end
```

time of each task and nullify the corresponding `edge_cost`, are obviously meaningful for distributed memory systems, they are also worthwhile on shared memory architectures. Merging two tasks in the same cluster keeps the data in the local memory, and even possibly cache, of each thread and avoids their copying over the shared memory bus. Therefore, transmission costs are decreased and bus contention is reduced.

5.3.3 Resource Constraint Warranty

Resource usage affects the performance of running programs. Thus, parallelization algorithms should try to limit the size of the memory used by tasks. BDSC introduces a new heuristic to control the amount of memory used by a cluster, via the user-defined memory upper bound parameter M. The limitation of the memory size of tasks is important when (1) executing large applications that operate on large amount of data, (2) M represents the processor local[1] (or cache) memory size, since, if the memory limitation is not respected, transfer between the global and local memories may occur during execution and may result in performance degradation, and (3) targeting embedded systems architecture. For each task τ, BDSC uses an over-approximation of the amount of data that τ allocates to perform read and write operations; it is used to check that the memory constraint of Cluster κ is satisfied whenever τ is included in κ. Algorithm 9 implements this memory constraint warranty (MCW); `regions_union` and `data_size` are functions that respectively merge data and yield the size (in bytes) of data (see Section 6.3 in the next chapter).

The previous line of reasoning is well adapted to a distributed memory architecture. When dealing with a multicore equipped with a purely shared memory, such per-cluster memory constraint is less meaningful. We can nonetheless keep the MCW constraint check within the BDSC algorithm even in this case, if we set M to the size of the global shared memory. A positive by-product of this design choice is that BDSC is able, in the shared

[1] We handle one level of the memory hierarchy of the processors.

ALGORITHM 9: Resource constraint warranties, on memory size M and processor number P

```
function MCW(τ, κ, M)
  merged = regions_union(cluster_data(κ), task_data(τ));
  return data_size(merged) ≤ M;
end

function PCW(clusters, P)
  return |clusters| < P;
end
```

memory case, to reject computations that need more memory space than available, even within a single cluster.

Another scarce resource is the number of processors. In the original policy of DSC, when no zeroing for τ is accepted, i.e. that would decrease its start time, τ is allocated to a new cluster. In order to limit the number of created clusters, we propose to introduce a user-defined cluster threshold P. This processor constraint warranty PCW is defined in Algorithm 9.

5.3.4 Efficient Task-to-Cluster Mapping

In the original policy of DSC, when no zeroings are accepted because none would decrease the start time of Vertex τ or DSRW failed, τ is allocated to a new cluster. This cluster creation is not necessary when idle slots are present at the end of other clusters; thus, we suggest to select instead one of these idle slots, if this cluster can decrease the start time of τ, without affecting the scheduling of the successors of the vertices already scheduled in this cluster. To ensure this, these successors must have already been scheduled or they must be a subset of the successors of τ. Therefore, in order to efficiently use clusters and not introduce additional clusters without needing it, we propose to schedule τ to this cluster that verifies this optimizing constraint, if no zeroing is accepted.

This extension of DSC we introduce in BDSC amounts thus to replacing each assignment of the cluster of τ to a new cluster by a call to `end_idle_clusters`. The `end_idle_clusters` function given in Algorithm 10 returns, among the idle clusters, the ones that finished the most recently before τ's top level or the empty set, if none is found. This assumes, of course, that τ's dependencies are compatible with this choice.

To illustrate the importance of this heuristic, suppose we have the DAG presented in Figure 5.3. The tables in Figure 5.4 exhibit the difference in scheduling obtained by DSC and our extension on this graph. We observe here that the number of clusters generated using DSC is 3, with 5 idle slots, while BDSC needs only 2 clusters, with 2 idle slots. Moreover, BDSC

ALGORITHM 10: Efficiently mapping Task τ in Graph G to clusters, if possible

```
function end_idle_clusters(τ, G, clusters)
  idle_clusters = clusters;
  foreach κ ∈ clusters
    if(cluster_time(κ) ≤ tlevel(τ, G)) then
      end_idle_p = TRUE;
      foreach τκ ∈ vertices(G) where cluster(τκ) = κ
        foreach τs ∈ successors(τκ, G)
          end_idle_p ∧= cluster(τs) ≠ cluster_undefined ∨
                          τs ∈ successors(τ, G);
      if(¬end_idle_p) then
        idle_clusters = idle_clusters-{κ};
  last_clusters = argmax_{κ∈idle_clusters} cluster_time(κ);
  return (idle_clusters != ∅) ? last_clusters : ∅;
end
```

achieves a better load balancing than DSC, since it reduces the variance of the clusters' execution loads, defined, for a given cluster, as the sum of the costs of all its tasks $X_\kappa = \sum_{\tau\in\kappa} \texttt{task_time}(\tau)$. Indeed, we compute first the average of costs

$$E[X_\kappa] = \sum_{\kappa=0}^{|clusters|-1} \frac{X_\kappa}{|clusters|}$$

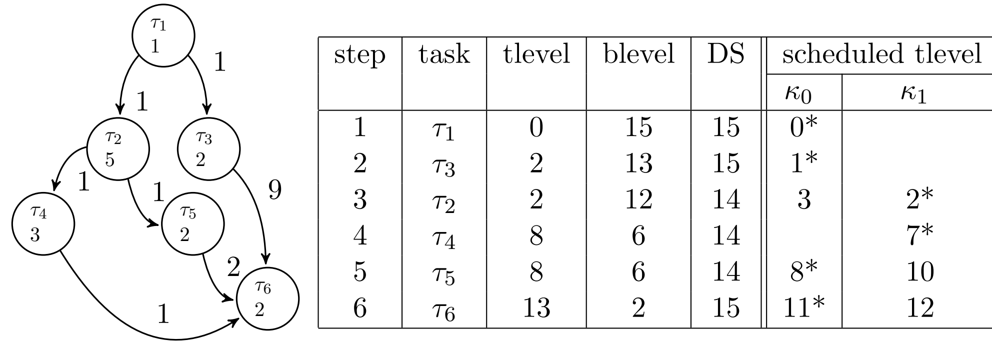
and then the variance

$$V(X_\kappa) = \frac{\sum_{\kappa=0}^{|clusters|-1}(X_\kappa - E[X_\kappa])^2}{|clusters|}.$$

For BDSC, $E(X_\kappa) = \frac{15}{2} = 7.5$ and $V(X_\kappa) = \frac{(7-7.5)^2+(8-7.5)^2}{2} = 0.25$. For DSC, $E(X_\kappa) = \frac{15}{3} = 5$ and $V(X_\kappa) = \frac{(5-5)^2+(8-5)^2+(2-5)^2}{3} = 6$.

Finally, with our efficient task-to-cluster mapping, in addition to decreasing the number of generated clusters, we gain also in the total execution time since our approach reduces communication costs by allocating tasks to the same cluster, e.g., by eliminating $\texttt{edge_cost}(\tau_5, \tau_6) = 2$ in Figure 5.4; thus, as shown in this figure (the last column accumulates the execution time for the clusters), the execution time with DSC is 14, but is 13 with BDSC.

To get a feeling for the way BDSC operates, we detail the steps taken to get this better scheduling in the table of Figure 5.3. BDSC is equivalent to DSC until Step 5, where κ_0 is chosen by our cluster mapping heuristic, since $\texttt{successors}(\tau_3, G) \subset \texttt{successors}(\tau_5, G)$; no new cluster needs to be allocated.



step	task	tlevel	blevel	DS	scheduled tlevel	
					κ_0	κ_1
1	τ_1	0	15	15	0*	
2	τ_3	2	13	15	1*	
3	τ_2	2	12	14	3	2*
4	τ_4	8	6	14		7*
5	τ_5	8	6	14	8*	10
6	τ_6	13	2	15	11*	12

Figure 5.3: A DAG amenable to cluster minimization (left) and its BDSC step-by-step scheduling (right)

κ_0	κ_1	κ_2	cumulative time
τ_1			1
τ_3	τ_2		7
	τ_4	τ_5	10
τ_6			14

κ_0	κ_1	cumulative time
τ_1		1
τ_3	τ_2	7
τ_5	τ_4	10
τ_6		13

Figure 5.4: DSC (left) and BDSC (right) cluster allocation

5.3.5 The BDSC Algorithm

BDSC extends the list scheduling template DSC provided in Algorithm 4 by taking into account the various extensions discussed above. In a nutshell, the BDSC `select_cluster` function, which decides in which cluster κ a task τ should be allocated, tries successively the four following strategies:

1. choose κ among the clusters of τ's predecessors that decrease the start time of τ, under MCW and DSRW constraints;

2. or, assign κ using our efficient task-to-cluster mapping strategy, under the additional constraint MCW;

3. or, create a new cluster if the PCW constraint is satisfied;

4. otherwise, choose the cluster among all clusters in `MCW_clusters_min` under the constraint MCW. Note that, in this worst case scenario, the top level of τ can be increased, leading to a decrease in performance since the length of the graph critical path is also increased.

BDSC is described in Algorithms 11 and 12; the entry graph G_u is the whole unscheduled program DAG, P, the maximum number of processors, and M, the maximum amount of memory available in a cluster. UT denotes the set of unexamined tasks at each BDSC iteration, RL, the set of

ready tasks and URL, the set of unready ones. We schedule the vertices of G according to the four rules above in a descending order of the vertices' priorities. Each time a task τ_r has been scheduled, all the newly readied vertices are added to the set RL (ready list) by the `update_ready_set` function.

ALGORITHM 11: BDSC scheduling Graph G_u, under processor and memory bounds P and M

```
function BDSC(Gᵤ, P, M)
  if (P ≤ 0) then
    return error('Not enough processors', Gᵤ) ;
  G = graph_copy(Gᵤ);
  foreach  τᵢ ∈ vertices(G)
    priority(τᵢ) = tlevel(τᵢ, G) + blevel(τⱼ, G);
  UT = vertices(G);
  RL = {τ ∈ UT / predecessors(τ, G) = ∅};
  URL = UT - RL;
  clusters = ∅;
  while UT ≠ ∅
    τᵣ = select_task_with_highest_priority(RL);
    (τₘ, min_tlevel) = tlevel_decrease(τᵣ, G);
    if (τₘ ≠ τᵣ ∧ MCW(τᵣ, cluster(τₘ), M)) then
      τᵤ = select_task_with_highest_priority(URL);
      if (priority(τᵣ) < priority(τᵤ)) then
        if (¬DSRW(τᵣ, τᵤ, clusters, G)) then
          if (PCW(clusters, P)) then
            κ = new_cluster(clusters);
            allocate_task_to_cluster(τᵣ, κ, G);
          else
            if (¬find_cluster(τᵣ, G, clusters, P, M)) then
              return error('Not enough memory', Gᵤ);
      else
        allocate_task_to_cluster(τᵣ, cluster(τₘ), G);
        edge_cost(τₘ, τᵣ) = 0;
    else if (¬find_cluster(τᵣ, G, clusters, P, M)) then
      return error('Not enough memory', Gᵤ);
    update_priority_values(G);
    UT = UT-{τᵣ};
    RL = update_ready_set(RL, τᵣ, G);
    URL = UT-RL;
  clusters(G) = clusters;
  return G;
end
```

BDSC returns a *scheduled* graph, i.e., an updated graph where some zeroings may have been performed and for which the *clusters* function yields the clusters needed by the given schedule; this schedule includes, beside the

ALGORITHM 12: Attempt to allocate κ in clusters for Task τ in Graph G, under processor and memory bounds P and M, returning true if successful

```
function find_cluster(τ, G, clusters, P, M)
  MCW_idle_clusters =
             {κ ∈ end_idle_clusters(τ, G, clusters, P) /
              MCW(τ, κ, M)};
  if (MCW_idle_clusters ≠ ∅) then
    κ = choose_any(MCW_idle_clusters);
    allocate_task_to_cluster(τ, κ, G);
  else if (PCW(clusters, P)) then
    allocate_task_to_cluster(τ, new_cluster(clusters), G);
  else
    MCW_clusters = {κ ∈ clusters / MCW(τ, κ, M)};
    MCW_clusters_min = argmin_{κ ∈ MCW_clusters} cluster_time(κ);
    if (MCW_clusters_min ≠ ∅) then
      κ = choose_any(MCW_clusters_min);
      allocate_task_to_cluster(τ, κ, G);
    else
      return false;
  return true;
end

function error(m, G)
  clusters(G) = ∅;
  return G;
end
```

new graph, the cluster allocation function on tasks, *cluster*. If not enough memory is available, BDSC returns the original graph, and signals its failure by setting *clusters* to the empty set. In Chapter 6, we introduce an alternative solution in case of failure using an iterative, hierarchical scheduling approach.

We suggest to apply here an additional heuristic, in which, if multiple vertices have the same priority, the vertex with the greatest bottom level is chosen for τ_r (likewise for τ_u) to be scheduled first to favor the successors that have the longest path from τ_r to the exit vertex. Also, an optimization could be performed when calling `update_priority_values(G)`; indeed, after each cluster allocation, only the top levels of the successors of τ_r need to be recomputed instead of those of the whole graph.

Theorem 1. *The time complexity of Algorithm 11 (BDSC) is $\mathcal{O}(n^3)$, n being the number of vertices in Graph G.*

Proof. In the "while" loop of BDSC, the most expensive computation among different `if` or `else` branches is the function `end_idle_cluster` used

in `find_cluster` that locates an existing cluster suitable to allocate there Task τ; such reuse intends to optimize the use of the limited of processors. Its complexity is proportional to

$$\sum_{\tau \in vertices(G)} |\texttt{successors}(\tau, G)|,$$

which is of worst case complexity $\mathcal{O}(n^2)$. Thus the total cost for n iterations of the "while" loop is $\mathcal{O}(n^3)$. □

Even though BDSC's worst case complexity is larger than DSC's, which is $\mathcal{O}(n^2 log(n))$ [111], it remains polynomial, with a small exponent. Our experiments (see Chapter 8) showed that theoretical slowdown not to be a significant factor in practice.

5.4 Related Work: Scheduling Algorithms

In this section, we survey the main existing scheduling algorithms; we compare them to BDSC. Given the breadth of the literature on this subject, we limit this presentation to heuristics that implement static scheduling of task DAGs. Static scheduling algorithms can be divided into three principal classes: guided random, metaheuristic and heuristic techniques.

Guided randomization and metaheuristic techniques use possibly random choices with guiding information gained from previous search results to construct new possible solutions. Genetic algorithms are examples of these techniques; they proceed by performing a global search, working in parallel on a population to increase the chance to achieve a global optimum. The Genetic Algorithm for Task Scheduling (GATS 1.0) [39] is an example of GAs. Overall, these techniques usually take a larger than necessary time to find a good solution [74].

Cuckoo Search (CS) [112] is an example of the metaheuristic class. CS combines the behavior of cuckoo species (throwing their eggs in the nests of other birds) with the Lévy flight [28]. In metaheuristics, finding an initial population and deciding when a solution is good are problems that affect the quality of the solution and the complexity of the algorithm.

The third class is heuristics, including mostly list-scheduling-based algorithms (see Section 2.5.2). We first compare BDSC with six scheduling algorithms for a bounded number of clusters, namely HLFET, ISH, MCP, HEFT, CEFT and ELT. Then, we compare BDSC with four scheduling algorithms or techniques that regroup clusters on physical processors, i.e., LPGS, LSGP, Triplet and PYRROS.

5.4.1 Bounded Number of Clusters

The Highest Level First with Estimated Times (HLFET) [10] and Insertion Scheduling Heuristic (ISH) [71] algorithms use static blevels for order-

ing; scheduling is performed according to a descending order of blevels. To schedule a task, they select the cluster that offers the earliest execution time, using a non-insertion approach, i.e., not taking into account idle slots within existing clusters to insert that task. If scheduling a given task introduces an idle slot, ISH adds the possibility of inserting from the ready list tasks that can be scheduled to this idle slot. Since, in both algorithms, only blevels are used for scheduling purposes, optimal schedules for fork/join graphs cannot be guaranteed.

The Modified Critical Path (MCP) algorithm [109] uses the latest start times, i.e., the critical path length minus blevel, as task priorities. It constructs a list of tasks in an ascending order of latest start times, and searches for the cluster yielding the earliest execution using the insertion approach. As before, it cannot guarantee optimal schedules for fork/join structures.

The Heterogeneous Earliest-Finish-Time (HEFT) algorithm [107] selects the cluster that minimizes the earliest finish time using the insertion approach. The priority of a task, its upward rank, is the task blevel. Since this algorithm is based on blevels only, it cannot guarantee optimal schedules for fork/join structures.

The Constrained Earliest Finish Time (CEFT) [69] algorithm schedules tasks on heterogeneous systems. It uses the concept of constrained critical paths (CCPs) that represent the tasks ready at each step of the scheduling process. CEFT schedules the tasks in the CCPs using the finish time in the entire CCP. The fact that CEFT schedules critical path tasks first cannot guarantee optimal schedules for fork and join structures even if sufficient processors are provided.

Contrarily to the five proposals above, BDSC preserves, when no resource constraints exist, the DSC characteristics of optimal scheduling for fork/join structures, since it uses the critical path length for computing dynamic priorities, based on blevels and tlevels. HLFET, ISH and MCP guarantee that the current critical path will not increase, but they do not attempt to decrease the critical path length; BDSC decreases the length of each task DS and starts with a ready node to simplify the computation time of new priorities. When resource scarcity is a factor, BDSC introduces a simple, two-step heuristics for task allocation: to schedule tasks, it first searches for possible idle slots in already existing clusters and, otherwise, picks a cluster with enough memory. Our experiments suggest that such an approach provides good schedules.

Extended Latency Time (ELT) algorithm [104] assigns tasks to a parallel machine with shared memory. It uses the attribute of synchronization time instead of communication time because this does not exist in a machine with shared memory. BDSC targets both shared and distributed memory systems.

Kwork and Ahmad [73] have implemented and compared 15 scheduling algorithms. They found that, among the critical-path-based algorithms,

dynamic-list algorithms such as DSC perform better than static-list ones. The insertion technique, which puts tasks within idle slots, improves scheduling. DSC does not implement this technique, while, thanks to our efficient task-to-cluster mapping strategy, which uses an insertion technique, BDSC yields better performance.

5.4.2 Cluster Regrouping on Physical Processors

The Locally Parallel-Globally Sequential (LPGS) [81] and Locally Sequential-Globally Parallel (LSGP) [60] algorithms are two techniques that, from a schedule for an unbounded number of clusters, remap the solutions to a bounded number of clusters. In LSGP, clusters are partitioned into blocks, each block being assigned to one cluster (locally sequential). The blocks are handled separately by different clusters, which can be run in parallel (globally parallel). LPGS links each original one-block cluster to one processor (locally parallel); blocks are executed sequentially (globally sequential). BDSC computes a bounded schedule on the fly and covers many more other possibilities of scheduling than LPGS and LSGP.

Triplet [32] is a clustering algorithm for heterogeneous architectures. It proceeds, first, by clustering tasks while assuming an unbounded number of clusters and, then, a second clustering of these first clusters is performed to merge them on actual processors. Here, the sorting of tasks is based on tlevel estimates only, contrarily to BDSC, which uses better information.

PYRROS [110] is also based on a two-step method for scheduling. The first step assumes an unbounded number of clusters and uses the DSC algorithm. Then, in the second step, an other clustering, on P processors, is performed, using cluster merging. This mapping sorts the clusters in the ascending order of their loads and then maps each cluster to a processor in such a way that all processors are as well balanced as possible. An another mapping is also used in order to minimize the communication costs between the P processors, using a pairwise interchange and task reordering heuristic. Note that, in this method, each step may change the decisions taken in the precedent step. For example, the ordering of tasks performed using DSC is modified during the second step. Also, in this second step, load balancing may be lost when performing the communication reduction heuristic.

The main difference between BDSC and the techniques that remap clusters on physical processors is that BDSC computes, by design, a bounded schedule. This is efficient in term of algorithmic complexity. Moreover, communication minimization is done once, while load balancing is ensured by our efficient task-to-cluster mapping heuristic. BDSC handles completion time and communication cost minimization and load balancing as parts of a single process. Finally, BDSC is also the only scheduling algorithm that takes into account the memory parameter, ignored in all the works surveyed here.

Note that all these works do not explain how information about costs of communication or execution times of tasks is obtained; they also do not address the construction of the DAG. Our paper provides a much broader picture, from analyzing sequential codes up to the generation of scheduled task graphs.

5.5 Conclusion

This chapter presents the memory-constrained, number of processor-bounded, list-scheduling heuristic BDSC, which extends the DSC (Dominant Sequence Clustering) algorithm. This extension improves upon DSC by dealing with memory- and number-of-processors-constrained parallel architectures, and introducing an efficient task-to-cluster mapping strategy.

This chapter leaves pending many questions about the practical use of BDSC and how to estimate the execution times and communication costs as numerical constants. The next chapter answers these questions and others by showing (1) the practical use of BDSC within a hierarchical scheduling algorithm called HBDSC for parallelizing scientific applications, and (2) how BDSC will benefit from a sophisticated execution and communication cost model, based on either static code analysis results provided by a compiler infrastructure such as PIPS or a dynamic-based instrumentation assessment tool.

Chapter 6

BDSC-Based Hierarchical Task Parallelization

All sciences are now under the obligation to prepare the ground for the future task of the philosopher, which is to solve the problem of value, to determine the true hierarchy of values. **Friedrich Nietzsche**

To make a recapitulation of what we presented so far in the previous chapters, recall we have handled two key issues in our automatic task parallelization process, namely (1) the design of SPIRE to provide parallel program representations and (2) the introduction of the BDSC list-scheduling heuristic for mapping DAGs in the presence of resource constraints on the number of processors and their local memory size. In order to reach our goal of automatically task parallelizing sequential codes, we develop in this chapter an approach based on BDSC that, first, maps this code into a DAG, then, generates a cost model to label edges and vertices of this DAG and, finally, applies BDSC on sequential applications.

Therefore, this chapter introduces a new BDSC-based hierarchical scheduling algorithm that we call HBDSC. It introduces a new DAG data structure, called the Sequence Dependence DAG (SDG), to represent partitioned parallel programs. Moreover, in order to label this SDG vertices and edges, HBDSC uses a new cost model based on time complexity measures, convex polyhedral approximations of data array sizes and code instrumentation.

Pour récapituler ce que nous avons présenté jusqu'ici dans les chapitres précédents, rappelons que nous avons traité deux questions clés de notre processus de parallélisation automatique de tâches, à savoir (1) la conception de SPIRE pour fournir des représentations de programmes parallèles et (2) l'introduction de l'heuristique d'ordonnancement par listes BDSC pour associer DAGs et processeurs en présence de contraintes de ressources sur le nombre de processeurs et la taille de leur mémoire locale. Afin d'atteindre notre objectif de paralléliser automatiquement des codes séquentiels, nous développons dans ce chapitre une approche fondée sur BDSC qui, tout d'abord, représente ce code sous forme de DAG, ensuite, génère un modèle de coût pour pondérer les sommets et arêtes de ce DAG et, enfin, applique BDSC sur les applications séquentielles.

En pratique, ce chapitre introduit un nouvel algorithme d'ordonnancement hiérarchique fondé sur BDSC, que nous appelons HBDSC. Il introduit

une nouvelle structure de données de DAG, appelée SDG (Sequence Dependence DAG), pour représenter les programmes parallèles partitionnés. En outre, afin de pondérer les sommets et arêtes du SDG, HBDSC utilise un nouveau modèle de coût fondé sur des mesures de complexité en temps, des approximations polyédriques convexes de la taille des tableaux de données et de l'instrumentation de code.

6.1 Introduction

If static scheduling algorithms are key issues in sequential program parallelization, they need to be properly integrated into compilation platforms to be used in practice. These platforms are in particular expected to provide the data required for scheduling purposes. However, gathering such information is a difficult problem in general, in particular in our case where tasks are automatically generated from program code. Therefore, in addition to BDSC, this chapter introduces also new static program analyses to gather the information required to perform scheduling under resource constraints, namely a static execution and communication cost model, a data dependence graph to provide scheduling constraints and static information regarding the volume of data exchanged between program tasks.

In this chapter, we detail how BDSC can be used, in practice, to schedule applications. We show (1) how to build from an existing program source code what we call a Sequence Dependence DAG (SDG), which plays the role of DAG G used in Chapter 5, (2) how to generate the numerical costs of vertices and edges in SDGs, and (3) how to perform what we call Hierarchical Scheduling (HBDSC) for SDGs to generate parallel code encoded in SPIRE(PIPS IR). We use PIPS to illustrate how these new techniques can be implemented in an optimizing compilation platform.

PIPS represents user code as abstract syntax trees (see Section 2.6.5). We define a subset of its grammar in Figure 6.1, limited to the statements S at stake in this chapter. $\mathtt{E}_{cond}$, $\mathtt{E}_{lower}$ and $\mathtt{E}_{upper}$ are expressions, while `I` is an identifier. The semantics of these constructs is straightforward. Note that, in PIPS, assignments are seen as function calls, where left hand sides are parameters passed by reference. We use the notion of `unstructured` with an `execution` attribute equal to `parallel` (defined in Chapter 4) to represent parallel code.

The remainder of this chapter is structured as follows. Section 6.2 introduces the partitioning of source codes into Sequence Dependence DAGs (SDG). Section 6.3 presents our cost models for the labeling of vertices and edges of SDGs. We introduce in Section 6.4 an important analysis, the path transformer, that we use within our cost models. A new BDSC-based hierarchical scheduling algorithm (HBDSC) is introduced in Section 6.5. Section 6.6 compares the main existing parallelization platforms with our

```
S ∈ Statement ::= sequence(S_1;....;S_n) |
                  test(E_cond,S_t,S_f) |
                  forloop(I, E_lower, E_upper, S_body) |
                  call |
                  unstructured(C_entry, C_exit, parallel)
C ∈ Control ::= control(S, L_succ, L_pred)
L ∈ Control*
```

Figure 6.1: Abstract syntax trees `Statement` syntax

BDSC-based hierarchical parallelization approach. Finally Section 6.7 concludes the chapter. Figure 6.2 summarizes the processes of parallelization applied in this chapter.

6.2 Hierarchical Sequence Dependence DAG Mapping

In order to partition into tasks real applications, which include loops, tests and other structured constructs[1], into dependence DAGs, our approach is to first build a Sequence Dependence DAG (SDG) which will be the input for the BDSC algorithm. Then, we use the code presented in form of an AST to define a hierarchical mapping function, that we call H, to map each sequence statement of the code to its SDG. H is used for the input of the HBDSC algorithm. We present in this section what SDGs are and how an H is built upon them.

6.2.1 Sequence Dependence DAG (SDG)

A Sequence Dependence DAG G is a data dependence DAG where task vertices τ are labeled with statements, while control dependences are encoded in the abstract syntax trees of statements. Any statement S can label a DAG vertex, i.e. each vertex τ contains a statement S, which corresponds to the code it runs when scheduled. We assume that there exist two functions `vertex_statement` and `statement_vertex` such that, on their respective domains of definition, they satisfy $S =$ `vertex_statement(`τ`)` and `statement_vertex(`S`,`G`)` $= \tau$. In contrast to the usual program dependence graph defined in [45], an SDG is thus not built only on simple instructions, represented here as *call* statements; compound statements such

[1]In this thesis, we only handle structured parts of a code, i.e., the ones that do not contain `goto` statements. Therefore, within this context, PIPS implements control dependences in its IR since it is equivalent to an AST (for structured programs, CDG and AST are equivalent).

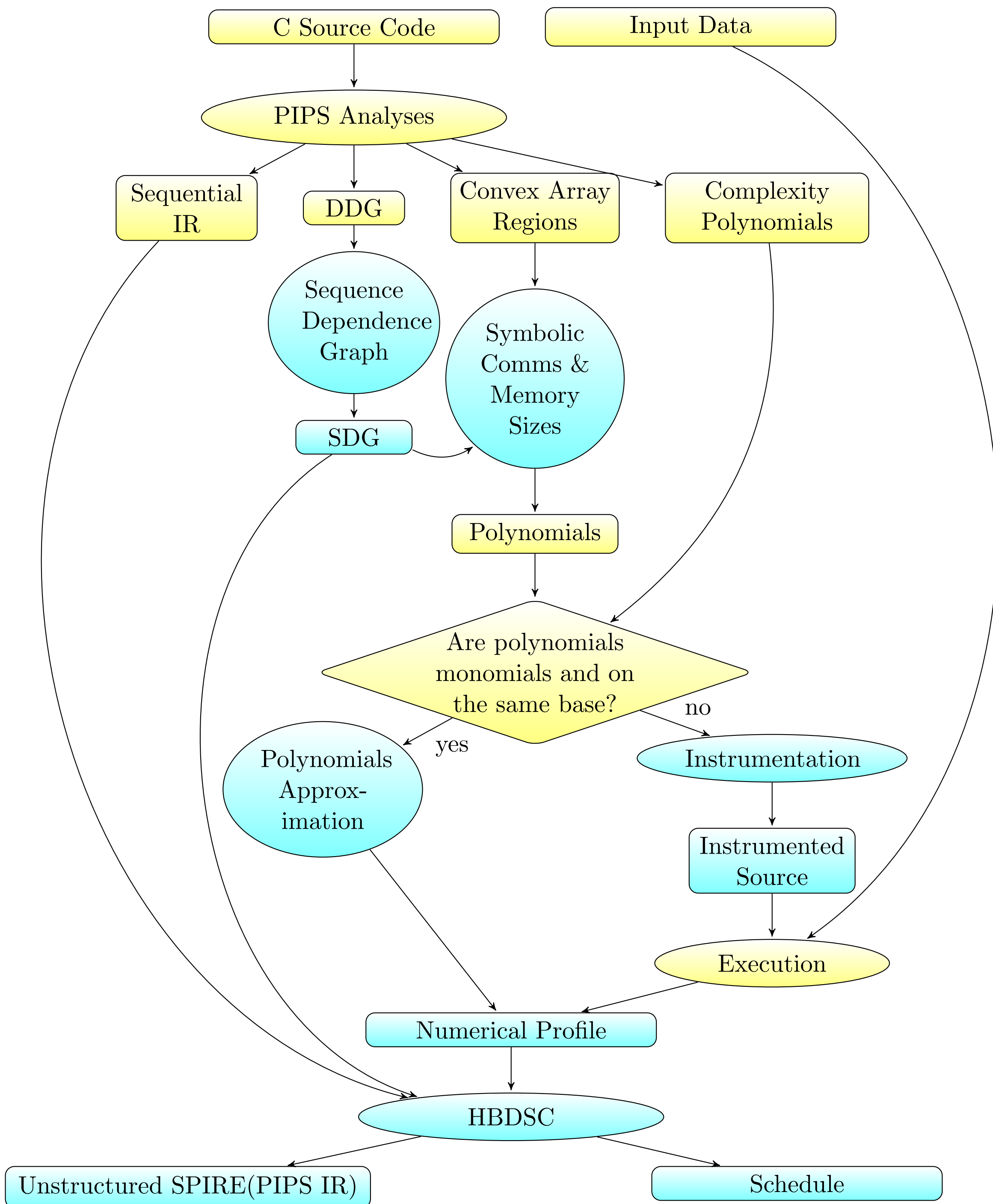


Figure 6.2: Parallelization process: blue indicates thesis contributions; an ellipse, a process; and a rectangle, results

as test statements (both true and false branches) and loop nests may constitute indivisible vertices of the SDG.

Design Analysis

For a statement S, one computes its sequence dependence DAG G using a recursive descent (depth-first traversal) along the structure of S, creating a vertex for each statement in a sequence and building loop body, true branch and false branch statements as recursively nested graphs. Definition 1 specifies this relation of hierarchical nesting between statements, which we call "enclosed".

Definition 1. A statement S_2 is *enclose*d into the statement S_1, noted $S_2 \subset S_1$, if and only if either:

1. $S_2 = S_1$, i.e. S_1 and S_2 have the same lexicographical order position within the whole AST to which they belong; this ordering of statements based on their statement number render statements textually comparable;
2. or S_1 is a loop, S_2' its body statement and $S_2 \subset S_2'$;
3. or S_1 is a test, S_2' its true or false branch and $S_2 \subset S_2'$;
4. or S_1 is a sequence and there exists S_2', one of the statements in this sequence, such that $S_2 \subset S_2'$.

This relation can also be applied to the vertices τ_i of these statements S_i in a graph such that $\texttt{vertex_statement}(\tau_i) = S_i$; we note $S_2 \subset S_1 \iff \tau_2 \subset \tau_1$

In Definition 2, we provide a functional specification of the construction of a Sequence Dependence DAG G where the vertices are compound statements and thus under the enclosed relation. To create connections within this set of vertices, we keep an edge between two vertices if the possibly compound statements in the vertices are data dependent using the recursive function *prune*; we rely on static analyses such as the one provided in PIPS to get this data dependence information in the form of a data dependence graph that we call D in the definition.

Definition 2. Given a data dependence graph $D = (T, E)$[2] and a sequence of statements $S = \texttt{sequence}(S_1; S_2; ...; S_m)$, its SDG is $G = (N, prune(A), 0_{m^2})$, where:

[2] $T = vertices(D)$ is a set of n tasks (vertices) τ and E is a set of m edges (τ_i, τ_j) between two tasks.

- $N = \{n_1, n_2, ..., n_m\}$, where $n_j = \texttt{statement_vertex}(S_j, G)$, $\forall j \in \{1, ..., m\}$
- $A = \{(n_i, n_j)/\exists(\tau, \tau') \in E, \tau \subset n_i \text{ and } \tau' \subset n_j\}$
- $prune(A' \cup \{(n, n')\}) = \begin{cases} prune(A') & \text{if } \exists(n_0, n'_0) \in A', \\ & (n_0 \subset n \text{ and } n'_0 \subset n') \\ prune(A') \cup \{(n, n')\} & \text{otherwise} \end{cases}$
- $prune(\emptyset) = \emptyset$
- 0_{m^2} is a $m \times m$ communication edge cost matrix, initially equal to the zero matrix.

Note that G is a quotient graph of D; moreover, we assume that D does not contain inter iteration dependencies.

Construction Algorithm

Algorithm 13 specifies the construction of the sequence dependence DAG G of a statement that is a sequence of S_1,S_2,... and S_n substatements. G initially is an empty graph, i.e., its sets of vertices and edges are empty, and its communication matrix is the zero matrix.

ALGORITHM 13: Construction of the sequence dependence DAG G from a statement $S = \texttt{sequence}(S_1;S_2;.....;S_m)$

```
function SDG(sequence(S1;S2;.....;Sm))
  G = (∅,∅,0_{m²});
  D = DDG(sequence(S1;S2;.....;Sm));
  foreach Si ∈ {S1,S2,.....,Sm}
    τi = statement_vertex(Si,D);
    add_vertex(τi, G);
    foreach Se / Se ⊂ Si
      τe = statement_vertex(Se,D);
      foreach τj ∈ successors(τe, D)
        Sj = vertex_statement(τj);
        if (∃ k ∈ [1,m] / Sj ⊂ Sk) then
          τk = statement_vertex(Sk,D);
          edge_regions(τi,τk) ∪= edge_regions(τe,τj);
          if(¬∃ (τi,τk) ∈ edges(G))
            add_edge((τi,τk), G);
  return G;
end
```

From the data dependence graph D, provided via the function `DDG` that we suppose given, Function `SDG`(S) adds vertices and edges to G, according

to Definition 2, using the functions `add_vertex` and `add_edge`. First, we group dependences between compound statements S_i, to form cumulated dependences, using dependences between their enclosed statements S_e; this yields the successors τ_j. Then, we search for another compound statement S_k to form the second side of the dependence τ_k, the first being τ_i. We keep then the dependences that label the edges of D using `edge_regions` of the new edge. Finally, we add dependence edge (τ_i, τ_k) to G.

Figure 6.4 illustrates the construction, from the DDG given in Figure 6.3 (right, meaning of colors given in Section 2.4.2), the SDG of the C code (left). The figure contains two SDGs corresponding to the two sequences in the code; the body $S0$ of the first loop (in blue) has also an SDG $G0$. Note how the dependences between the two loops have been deduced from the dependences of their enclosed statements (their loop bodies). These SDGs and their printouts have been generated automatically with PIPS[3].

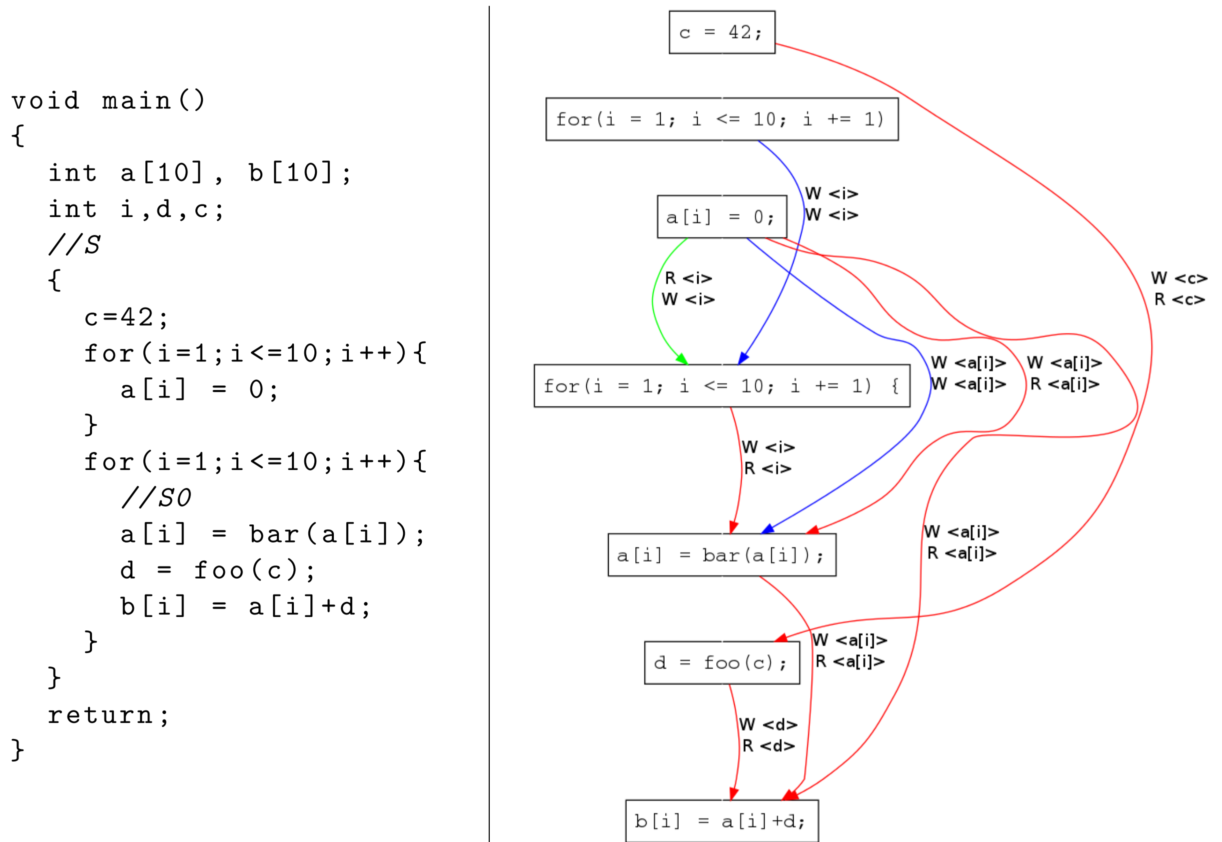


Figure 6.3: Example of a C code (left) and the DDG D of its internal S sequence (right)

[3] We assume that declarations and codes are not mixed.

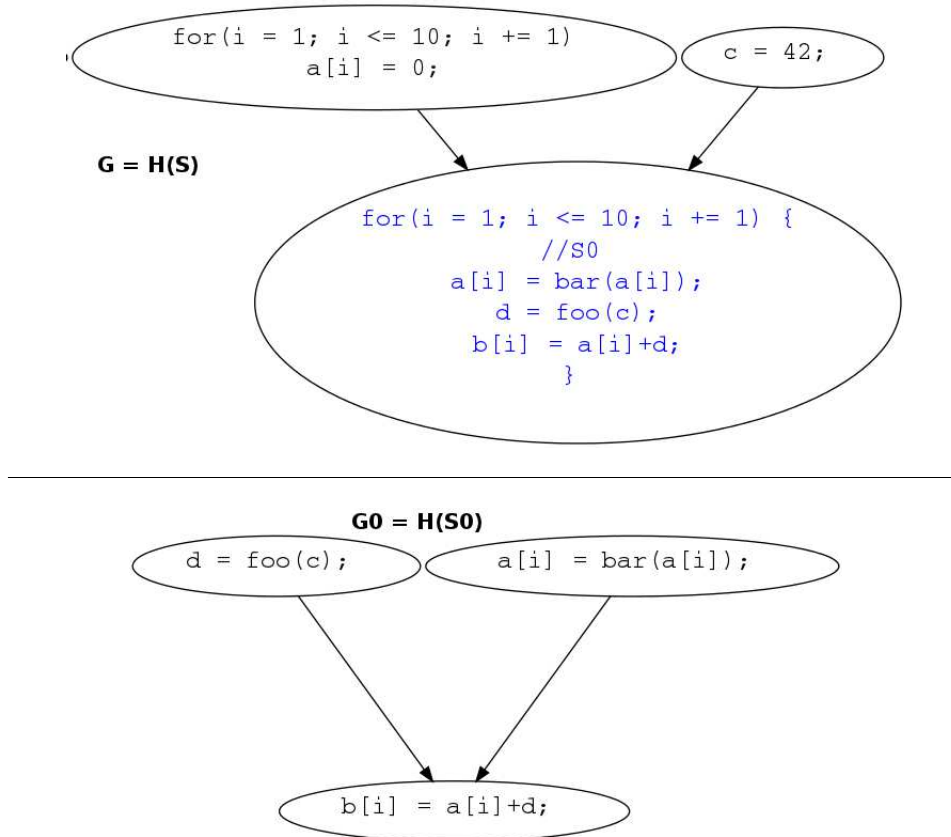


Figure 6.4: SDGs of S (top) and $S0$ (bottom) computed from the DDG (see the right of Figure 6.3); S and $S0$ are specified in the left of Figure 6.3

6.2.2 Hierarchical SDG Mapping

A *hierarchical SDG mapping* function H maps each statement S to an SDG $G = H(S)$ if S is a sequence statement; otherwise G is equal to $\perp$.

Proposition 1. $\forall \tau \in \texttt{vertices}(H(S')), \texttt{vertex_statement}(\tau) \subset S'$

Our introduction of SDG and its hierarchical mapping to different statements via H is motivated by the following observations, which also support our design decisions:

1. The true and false statements of a test are control dependent upon the condition of the test statement, while every statement within a loop (i.e., statements of its body) is control dependent upon the loop statement header. If we define a *control area* as a set of statements transitively linked by the control dependence relation, our SDG construction process insures that the control area of the statement of a

given vertex is in the vertex. This way, we keep all the control dependences of a task in our SDG within itself.

2. We decided to consider test statements as single vertices in the SDG to ensure that they are scheduled on one cluster[4], which guarantees the execution of the enclosed code (true or false statements), whichever branch is taken, on this cluster.

3. We do not group successive simple *call* instructions into a single "basic block" vertex in the SDG in order to let BDSC fuse the corresponding statements so as to maximize parallelism and minimize communications. Note that PIPS performs interprocedural analyses, which will allow call sequences to be efficiently scheduled whether these calls represent trivial assignments or complex function calls.

In Algorithm 13, Function `SDG` yields the sequence dependence DAG of a sequence statement S. We use it in Function `SDG_mapping` presented in Algorithm 14 to update the mapping function H for statements S that can be a sequence or a non-sequence; the goal is to build a hierarchy between these SDGs via the recursive call $H = \texttt{SDG_mapping}(S, \perp)$.

ALGORITHM 14: Recursive mapping of an SDG G to each sequence statement S via the function H

```
function SDG_mapping(S, H)
  switch (S)
    case call:
      return H[S → ⊥];
    case sequence(S_1;...;S_n):
      foreach S_i ∈ {S_1,...,S_n}
        H = SDG_mapping(S_i, H);
      return H[S → SDG(S)];
    case forloop(I, E_lower, E_upper, S_body):
      H = SDG_mapping(S_body, H);
      return H[S → ⊥];
    case test(E_cond, S_t, S_f):
      H = SDG_mapping(S_t, H);
      H = SDG_mapping(S_f, H);
     return H[S → ⊥];
end
```

Figure 6.5 shows the construction of H where S is the last part of the equake code excerpt given in Figure 6.7: note how the body *Sbody* of the last loop is also an SDG *Gbody*. Note that $H(\mathit{Sbody}) = \mathit{Gbody}$ and $H(S) = G$. These SDGs have been generated automatically with PIPS; we use the Graphviz tool for pretty printing [51].

[4] A cluster is a logical entity which will correspond to one process or thread.

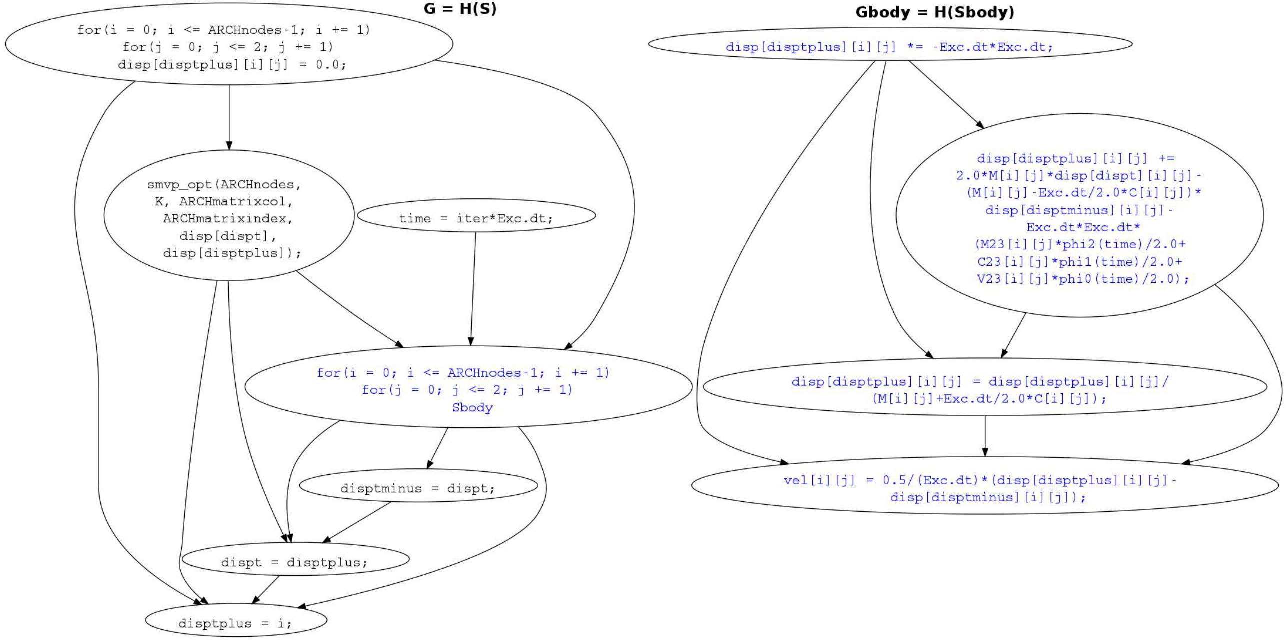


Figure 6.5: SDG G, for a part of equake S given in Figure 6.7; *Gbody* is the SDG of the body *Sbody* (logically included into G via H)

6.3 Sequential Cost Models Generation

Since the volume of data used or exchanged by SDG tasks and their execution times are key factors in the BDSC scheduling process, we need to assess this information as precisely as possible. PIPS provides an intra- and inter-procedural analysis of array data flow called array regions analysis (see Section 2.6.3) that computes dependences for each array element access. For each statement S, two types of set of regions are considered: `read_regions(`S`)` and `write_regions(`S`)`[5] contain the array elements respectively read and written by S. Moreover, in PIPS, array region analysis is not limited to array variables but is extended to scalar variables which can be considered as arrays with no dimensions [35].

6.3.1 From Convex Polyhedra to Ehrhart Polynomials

Our analysis uses the following set of operations on sets of regions R_i of Statement S_i presented in Section 2.6.3: `regions_intersection` and `regions_union`. Note that regions must be defined with respect to a common memory store for these operations to be properly defined. Two sets of

[5] `in_regions(`S`)` and `out_regions(`S`)` are not suitable to be used for our context unless, after each statement schedule, `in_regions` and `out_regions` are recomputed. That is why, we use `read_regions` and `write_regions` that are independent of the schedule.

regions R_1 and R_2 should be defined in the same memory store to make it possible to compare them; we should bring a set of regions R_1 to the store of R_2 by combining R_2 with the possible store changes introduced in between, what we call "path transformer" that connects the two memory stores. We introduce the path transformer analysis in Section 6.4.

Since we are interested in the size of array regions to precisely assess communication costs and memory requirements, we compute Ehrhart polynomials [42], which represent the number of integer points contained in a given parameterized polyhedron, from this region. To manipulate these polynomials, we use various operations using the Ehrhart API provided by the polylib library [92].

Communication: Edge Cost

To assess the communication cost between two SDG vertices, τ_1 as source and τ_2 as sink vertices, we rely on the number of bytes involved in dependences of type "read after write" (RAW) data, using the read and write regions as follows:

$$R_{w1} = \texttt{write_regions(vertex_statement(}\tau_1\texttt{))}$$
$$R_{r2} = \texttt{read_regions(vertex_statement(}\tau_2\texttt{))}$$
$$\texttt{edge_cost_bytes}(\tau_1, \tau_2) = \sum_{r \in \texttt{regions_intersection}(R_{w1}, R_{r2})} \texttt{ehrhart}(r)$$

For example, the array dependences between the two statements $S_1 = \texttt{MultiplY}(Ixx, Gx, Gx)$ and $S_2 = \texttt{Gauss}(Sxx, Ixx)$, in the code of Harris presented in Figure 1.1, is Ixx of size $N \times M$, where the N and M variables represent the input image size. Therefore, S_1 should communicate the array Ixx to S_2 when completed. We look for the array elements that are accessed in both statements (written in S_1 and read in S_2); we obtain the following two sets of regions R_{w1} and R_{r2} information:

$$R_{w1} = \{< Ixx(\phi_1) - W - \ EXACT - \{1 \leq \phi_1, \phi_1 \leq N \times M\} >\}$$
$$R_{r2} = \{< Ixx(\phi_1) - R - \ EXACT - \{1 \leq \phi_1, \phi_1 \leq N \times M\} >\}$$

The result of the intersection between these two sets of regions is as follows:

$$R_{w1} \cap R_{r2} = \\ \{< Ixx(\phi_1) - WR - \ EXACT - \{1 \leq \phi_1, \phi_1 \leq N \times M\} >\}$$

The sum of the Ehrhart polynomials resulting from the set $R_{w1} \cap R_{r2}$ of regions that yields the volume of this set of regions, representing the number of its elements in bytes, is as follows, where 4 is the number of bytes in a float:

$$\begin{aligned}\texttt{edge_cost_bytes}(\tau_1, \tau_2) &= \texttt{ehrhart}(< Ixx(\phi_1) - WR - \ EXACT - \\ &\qquad \{1 \leq \phi_1, \phi_1 \leq N \times M\} >) \\ &= 4 \times N \times M\end{aligned}$$

In practice, in our experiments, in order to compute the communication time, this polynomial that represents the message size to communicate, in number of bytes, is multiplied by a transfer time of one byte (β); it is then added to the latency time (α). These two coefficients are dependent on the specific target machine; we note:

$\texttt{edge_cost}(\tau_1, \tau_2) = \alpha + \beta \times \texttt{edge_cost_bytes}(\tau_1, \tau_2)$

The default cost model in PIPS considers that $\alpha = 0$ and $\beta = 1$. Moreover, in some cases the array region analysis cannot provide the required information. Therefore, in the case of shared memory codes, BDSC can proceed and generate an efficient schedule without the information of `edge_cost`; otherwise, in the case of distributed memory codes, $\texttt{edge_cost}(\tau_1, \tau_2) = \infty$ is generated as a result for such cases. We will decide thus to schedule τ_1 and τ_2 in the same cluster.

Local Storage: Task Data

To provide an estimation of the volume of data used by each task τ, we use the number of bytes of data read and written by the task statement, via the following definitions assuming that the task statement contains no internal allocation of data:

```
S = vertex_statement(τ)
task_data(τ) = regions_union(read_regions(S),
                             write_regions(S))
```

As above, we define `data_size` as follows, for any set of regions R:

$\texttt{data_size}(R) = \sum_{r \in R} \texttt{ehrhart}(r)$

For instance, in Harris `main` function, `task_data(`τ`)`, where τ is the vertex labeled with the call statement $S_1 = \texttt{MultiplY}(Ixx, Gx, Gx)$, is equal to:

$$R = \{< Gx(\phi_1) - R - \ EXACT - \{1 \leq \phi_1, \phi_1 \leq N \times M\} >, \\ < Ixx(\phi_1) - W - \ EXACT - \{1 \leq \phi_1, \phi_1 \leq N \times M\} >\}$$

The size of this region is:

$\texttt{data_size}(R) = 2 \times N \times M$

When the array region analysis cannot provide the required information, the size of the array contained in its declaration is returned as an upper bound of the accessed array elements for such cases.

Execution: Task Time

In order to determine an average execution time for each vertex in the SDG, we use a static execution time approach based on a program complexity analysis provided by PIPS. There, each statement S is automatically labeled with an expression, represented by a polynomial over program variables via the `complexity_estimation(`S`)` function, that denotes an estimation of the execution time of this statement, assuming that each basic operation (addition, multiplication...) has a fixed, architecture-dependent execution time. In practice, to perform our experiments on a specific target machine, we use a table *COMPLEXITY_COST_TABLE* that, for each basic operation, provides a coefficient of its time execution on this machine. This sophisticated static complexity analysis is based on inter-procedural information such as preconditions. Using this approach, one can define `task_time` as:

$$\texttt{task_time}(\tau) = \texttt{complexity_estimation(vertex_statement(}\tau\texttt{))}$$

For instance, for the call statement $S_1 = \texttt{MultiplY}(Ixx, Gx, Gx)$ in Harris `main` function, `task_time(`τ`)`, where τ is the vertex labeled with S_1, is equal to $N \times M \times 17 + 2$ as detailed in Figure 6.6. Thanks to the interprocedural analysis of PIPS, this information can be reported to the call for the function `MultiplY` in the `main` function of Harris. Note that we use here a default cost model where the coefficient of each basic operation is equal to 1.

```
void MultiplY(float M[N*M], float X[N*M], float Y[N*M])
{
                                                   //N*M*17+2
  for(i = 0; i < N; i += 1)
                                                   //M*17+2
    for(j = 0; j < M; j += 1)
                                                   //17
      M[z(i,j)] = X[z(i,j)]*Y[z(i,j)];
}
```

Figure 6.6: Example of the execution time estimation for the function `MultiplY` of the code Harris; each comment provides the complexity estimation of the statement below (N and M are assumed to be global variables)

When this analysis cannot provide the required information: currently, the complexity estimation of a while loop is not computed. However, our algorithm computes a naive schedule in such cases, where we assume that `task_time`$(\tau) = \infty$, but the quality (in term of efficiency) of this schedule is not guaranteed.

6.3.2 From Polynomials to Values

We have just seen how to represent cost, data and time information in terms of polynomials; yet, running BDSC requires actual values. This is particularly a problem when computing tlevels and blevels, since cost and time are cumulated there. We convert cost information into time by assuming that communication times are proportional to costs, which amounts in particular to setting the communication latency α to zero. This assumption is validated by our experimental results, and the fact that data arrays are usually large in the application domain we target.

Static Approach: Polynomials Approximation

When program variables used in the above-defined polynomials are numerical values, each polynomial is a constant; this happens to be the case for one of our applications, ABF. However, when input data are unknown at compile time (as for the Harris application), we suggest to use a very simple heuristic to replace the polynomials by numerical constants. When all polynomials at stake are monomials on the same base, we simply keep the coefficient of these monomials as costs. Even though this heuristic appears naive at first, it actually is quite useful in the Harris application: Table 6.1 shows the complexities for time estimation and communication (the last line) costs generated for each function of Harris using PIPS default operation and communication cost model, where the N and M variables represent the input image size.

Function	Complexity and Transfer (polynomial)	Numerical estimation
`InitHarris`	$9 \times N \times M$	9
`SobelX`	$60 \times N \times M$	60
`SobelY`	$60 \times N \times M$	60
`MultiplY`	$17 \times N \times M$	17
`Gauss`	$85 \times N \times M$	85
`CoarsitY`	$34 \times N \times M$	34
One image transfer	$4 \times N \times M$	4

Table 6.1: Execution and communication time estimations for Harris using PIPS default cost model (N and M variables represent the input image size)

In our implementation (see Section 8.3.3), we use a more precise operation and communication cost model that depends on the target machine. It relies on a number of parameters of a given CPU such as the cost of an addition, a multiplication, or of the communication cost of one byte for an Intel CPU, all these in numbers of cycle units (CPI). In Section 8.3.3, we detail these parameters for the used CPUs.

Dynamic Approach: Instrumentation

The general case deals with polynomials that are functions of many variables, such as the ones that occur in the SPEC2001 benchmark equake and that depend on the variables *ARCHelems* or *ARCHnodes* and *timessteps* (see the non-bold code in Figure 6.7 that shows a part of the equake code).

```
FILE * finstrumented = fopen("instrumented_equake.in", "w");
...
fprintf(finstrumented, "task_time 62 = %ld\n", 179 * ARCHelems + 3);
for (i = 0; i < ARCHelems; i++){
  for (j = 0; j < 4; j++)
    cor[j] = ARCHvertex[i][j];
  ...
}
...
fprintf(finstrumented, "task_time 163 = %ld\n", 20 * ARCHnodes + 3);
for(i = 0; i <= ARCHnodes-1; i += 1)
  for(j = 0; j <= 2; j += 1)
    disp[disptplus][i][j] = 0.0;
fprintf(finstrumented, "edge_cost 163 → 166 = %ld\n", ARCHnodes * 9);
fprintf(finstrumented, "task_time 166 = %ld\n", 110 * ARCHnodes + 106);
smvp_opt(ARCHnodes, K, ARCHmatrixcol, ARCHmatrixindex,
         disp[dispt], disp[disptplus]);
fprintf(finstrumented, "task_time 167 = %d\n", 6);
time = iter*Exc.dt;
fprintf(instrumented, "task_time 168 = %ld\n", 510 * ARCHnodes + 3);
// S
for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++){
    // Sbody
    disp[disptplus][i][j] *= -Exc.dt*Exc.dt;
    disp[disptplus][i][j] +=
      2.0*M[i][j]*disp[dispt][i][j]-M[i][j] -
      Exc.dt/2.0*C[i][j])*disp[disptminus][i][j] -
      Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0 +
      C23[i][j] * phi1(time) / 2.0 +
      V23[i][j] * phi0(time) / 2.0);
    disp[disptplus][i][j] =
     disp[disptplus][i][j] / (M[i][j] + Exc.dt / 2.0 * C[i][j]);
    vel[i][j] = 0.5 / Exc.dt * (disp[disptplus][i][j] -
                disp[disptminus][i][j]);
  }
fprintf(finstrumented, "task_time 175 = %d\n", 2);
disptminus = dispt;
fprintf(finstrumented, "task_time 176 = %d\n", 2);
dispt = disptplus;
fprintf(finstrumented, "task_time 177 = %d\n", 2);
disptplus = i;
```

Figure 6.7: Instrumented part of equake (*Sbody* is the inner loop sequence)

In such cases, we first instrument automatically the input sequential code and run it once in order to obtain the numerical values of the polynomials. The instrumented code contains the initial user code plus instructions that compute the numerical values of the cost polynomials for each statement. BDSC is then applied, using this cost information, to yield the final parallel program. Note that this approach is sound since BDSC ensures that the value of a variable (and thus a polynomial) is the same, whichever scheduling is used. Of course, this approach works well, as our experiments show (see Chapter 8), when a program's complexity and communication polynomials do not change when some of its input parameters are modified (partial evaluation). This is the case for many signal processing benchmarks, where performance is mostly a function of structure parameters such as image size, and is independent of the actual signal (pixel) values upon which the program acts.

We show an example of this final case using a part of the instrumented equake code[6] in Figure 6.7. The added instrumentation instructions are `fprintf` statements, the second parameter of which represents the statement number of the following statement, and the third, the value of its execution time for task time instrumentation. For edge cost instrumentation, the second parameter is the number of the incident statements of the edge, and the third, the edge cost polynomial value. After execution of the instrumented code, the numerical results of the polynomials are printed in the file `instrumented_equake.in` presented in Figure 6.8. This file is an input for the PIPS implementation of BDSC. We parse this file in order to extract execution and communication estimation times (in number of cycles) to be used as parameters for the computation of top levels and bottom levels necessary for the BDSC algorithm (see Chapter 5).

```
...
task_time 62 = 35192835
...
task_time 163 = 718743
edge_cost 163 -> 166 =  323433
task_time 166 = 3953176
task_time 167 =  6
task_time 168 = 18327873
...
task_time 175 =  2
task_time 176 = 2
task_time 177 = 2
```

Figure 6.8: Numerical results of the instrumented part of equake (`instrumented_equake.in`)

[6] We do not show the instrumentation on the statements inside the loops for readability purposes.

This section supplies BDSC scheduling algorithm that relies upon weights on vertices (time execution and storage data) and edges (communication cost) of its entry graph. Indeed, we exploit complexity and array region analyses provided by PIPS to harvest all this information statically. However, since each two sets of regions should be defined in the same memory store to make it possible comparing them, the second set of regions must be combined with the path transformer, that we define in the next section, and that connects the two memory stores of these two sets of regions.

6.4 Reachability Analysis: The Path Transformer

Our cost model defined in Section 6.3 uses affine relations such as convex array regions to estimate the communications and data volumes induced by the execution of statements. We used a set of operations on array regions such as the function `regions_intersection` (see Section 6.3.1). However, this is not sufficient because regions should be combined with what we call path transformers in order to propagate the memory stores used in them. Indeed, regions must be defined with respect to a common memory store for region operations to be properly performed.

A path transformer permits to compare array regions of statements originally defined in different memory stores. The path transformer between two statements computes the possible changes performed by a piece of code delimited by two statements S_{begin} and S_{end} enclosed within a statement S. A path transformer is represented by a convex polyhedron over program variables and its computation is based on transformers (see Section 2.6.1); thus, it is a system of linear inequalities.

Our path transformer analysis can be seen as a special case of reachability analysis, a particular kind of graph analysis where one checks whether a given final state can be reached from an initial one. In the domain of programming languages, one can find practical application of this technique in the computation, for general control-flow graphs, of the transitive closure of the relation corresponding to statement transitions (see for instance [108]). Since the scientific applications we target in this thesis use mostly for loops, we propose in this section a dedicated analysis, called path transformer, that provides, in the restricted case of for loops and tests, analytical expressions for such transitive closures.

6.4.1 Path Definition

A "path" specifies a statement execution instance, including the mapping of loop indices to values and specific syntactic statement of interest. A path is defined by a pair $P = (M, S)$ where M is a mapping function from

`Ide` to `Exp`[7] and S is a statement, specified by its syntactic path from the top program syntax tree to the particular statement. Indeed, a particular statement S_i may correspond to a particular iteration $M(i)$ within a set of iterations (created by a loop of index i). Paths are used to store the values of indices of the loops enclosing statements. Every loop on the path has its entry in M; indices of loops not related to it are not present.

We provide a simple program example in Figure 6.9 that shows the AST of a code that contains a loop. We search for the path transformer between S_b and S_e; we suppose that S, the smallest subtree that encloses the two statements, is $forloop_0$. As the two statements are enclosed into a loop, we have to specify the indices of these statements inside the loop e.g. $M(i) = 2$ for S_b and $M(i) = 5$ for S_e. The path execution trace between S_b and S_e is illustrated in Figure 6.10. The chosen path runs through S_b for the second iteration of the loop until S_e for the fifth iteration.

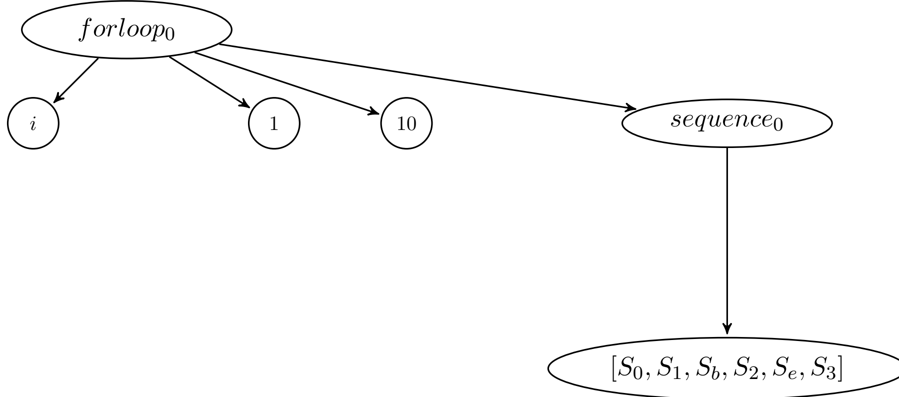


Figure 6.9: An example of a subtree of root $forloop_0$

In this section, we assume that S_{end} is reachable [44] from S_{begin}. This means that, for each enclosing loop S ($S_{begin} \subset S$ and $S_{end} \subset S$), where $S =$ `forloop`$(I, E_{lower}, E_{upper}, S_{body})$, $E_{lower} \leq E_{upper}$ - loops are always executed - and, for each mapping of an index of these loops, $E_{lower} \leq i_{begin} \leq i_{end} \leq E_{upper}$. Moreover, since there is no execution trace from the true S_t branch to the false branch S_f of a test statement S_{test}, S_{end} is not reachable from S_{begin} if for each mapping of an index of a loop that encloses S_{test}: $i_{begin} = i_{end}$, or there is no loop enclosing the test.

6.4.2 Path Transformer Algorithm

In this section, we present an algorithm that computes affine relations between program variables at any two points, or "paths", in the program, from the memory store of S_{begin} to the memory store of S_{end}, along any sequential

[7] `Ide` and `Exp` are the set of identifiers `I` and expressions `E` respectively.

$S_b(2)$, $S_2(2)$, $S_e(2)$, $S_3(2)$,
$S_0(3)$, $S_1(3)$, $S_b(3)$, $S_2(3)$, $S_e(3)$, $S_3(3)$,
$S_0(4)$, $S_1(4)$, $S_b(4)$, $S_2(4)$, $S_e(4)$, $S_3(4)$,
$S_0(5)$, $S_1(5)$, $S_b(5)$, $S_2(5)$, $S_e(5)$.

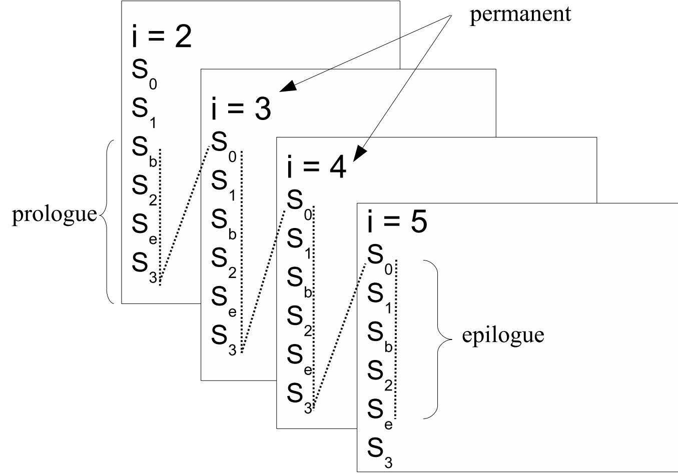


Figure 6.10: An example of a path execution trace from S_b to S_e in the subtree $forloop_0$

execution that links the two memory states. `path_transformer(`S`,` P_{begin}`,` P_{end}`)` is the path transformer of the subtree of S with all vertices not on the P_{begin} path to P_{end} path execution trace pruned. It is the path transformer between statements S_{begin} and S_{end}, if $P_{begin} = (M_{begin}, S_{begin})$ and $P_{end} = (M_{end}, S_{end})$.

`path_transformer` uses a variable *mode* to indicate the current context on the compile-time version of the execution trace. Modes are propagated along the depth-first left-to-right traversal of statements: (1) `mode` $=$ `sequence` when there is no loop on the trace at the current statement level; (2) `mode` $=$ `prologue` on the prologue part of an enclosing loop; (3) `mode` $=$ `permanent` on the full part of an enclosing loop; and (4) `mode` $=$ `epilogue` on the epilogue part of an enclosing loop. The algorithm is described in Algorithm 15.

A first call to the recursive function `pt_on(`S`,` P_{begin}`,` P_{end}`, mode)` with `mode` $=$ `sequence` is made. In this function, many path transformer operations are performed. A *path transformer* T is recursively defined by:

$$T = \texttt{id} \mid \bot \mid T_1 \diamond T_2 \mid T_1 \sqcup T_2 \mid T^k \mid T(S)$$

These operations on path transformers are specified as follows.

- `id` is the transformer identity, used when no variables are modified, i.e., they are all constrained by an equality.

ALGORITHM 15: AST-based Path Transformer algorithm

```
function path_transformer(S,(M_begin,S_begin),(M_end,S_end))
  return pt_on(S,(M_begin,S_begin),(M_end,S_end),sequence);
end

function pt_on(S, (M_begin,S_begin),(M_end,S_end), mode)
  if (S_end = S) then return id ;
  elseif (S_begin = S) then return T(S) ;
  switch (S)
    case call:
      (mode = permanent
      ∨ mode = sequence ∧ S_begin ≤ S ∧ S < S_end
      ∨ mode = prologue ∧ S_begin ≤ S
      ∨ mode = epilogue ∧ S < S_end)
      ? return T(S) : return id;
    case sequence(S_1;S_2;...;S_n):
      if (|S_1,S_2,...,S_n|=0) then return id;
      else
        rest = sequence(S_2;......;S_n);
        return pt_on(S_1, (M_begin,S_begin),(M_end,S_end), mode) ⋄
               pt_on(rest, (M_begin,S_begin),(M_end,S_end), mode);
    case test(E_cond,S_t,S_f):
      T_t = pt_on(S_t, (M_begin,S_begin),(M_end,S_end), mode);
      T_f = pt_on(S_f, (M_begin,S_begin),(M_end,S_end), mode);
      if((S_begin ⊂ S_t ∧ S_end ⊂ S_f) ∨
         (S_end ⊂ S_t ∧ S_begin ⊂ S_f) then
        if(mode = permanent) then
          return  T_t ⊔ T_f;
        else
          return ⊥;
      else if (S_begin ⊂ S_t ∨ S_end ⊂ S_t) then
        return T(E_cond) ⋄ T_t;
      else if (S_begin ⊂ S_f ∨ S_end ⊂ S_f) then
        return T(¬E_cond) ⋄ T_f;
      else
        return T(S);
    case forloop(I, E_lower, E_upper, S_body):
      return
        pt_on_loop(S,(M_begin,S_begin),(M_end,S_end), mode);
end
```

- $\perp$ denotes the undefined (empty) transformer, i.e., in which the set of affine constraints of the system is not feasible.

- '$\diamond$' denotes the operation of composition of transformers T_1 and T_2. $T_1 \diamond T_2$ is overapproximated by the union of constraints in T_1 (on variables

$x_1, ..., x_n, x''_1, ..., x''_p$) with constraints in T_2 (on variables $x''_1, ..., x''_p, x'_1, ..., x'_{n'}$), then projected on $x_1, ..., x_n, x'_1, ..., x'_{n'}$ to eliminate the "intermediate" variables $x''_1, ..., x''_p$ (this definition is extracted from [78]). Note that $T \diamond \texttt{id} = T$, for all T.

- '$\sqcup$' denotes the convex hull of each transformer argument. The usual union of two transformers is a union between two convex polyhedra, and is not necessarily a convex polyhedron. A convex hull polyhedron approximation is required in order to obtain a convex result; a convex hull operation provides the smallest polyhedron enclosing the union.
- '^{k}' denotes the effect of a k iterations of a transformer T. In practice, heuristics are used to compute a transitive closure approximation T^*; thus, this operation generally provides an approximation of the transformer T^k. Note that PIPS uses here the Affine Derivative Closure algorithm [18].
- $T(S)$ is the transformer of a statement S, defined in Section 2.6.1; we suppose these transformers already exist. T is defined by a list of arguments and a predicate system labeled by T.

We distinguish several cases in the `pt_on`(S, (M_{begin}, S_{begin}), (M_{end}, S_{end}), *mode*) function. The computation of the path transformer begins when S is S_{begin}, i.e. the path is beginning; we return the transformer of S_{begin}. It ends when S is S_{end}; there, we close the path and we return the identity transformer, `id`. The other cases depend on the type of the statement as follows.

Call

When S is a function call, we return the transformer of S, if S belongs to the paths from S_{begin} to S_{end}, and `id` otherwise; we use the lexicographic ordering $<$ to compare statement paths.

Sequence

When S is a sequence of n statements, we return the composition of the transformers of each statement in this sequence. An example of the result provided by the path transformer algorithm for a sequence is illustrated in Figure 6.11. The left side presents a code fragment that contains a sequence of four instructions. The right side shows the resulting path transformer between the two statements S_b and S_e. It shows the result of the composition of the transformers of the three first statements; note that the transformer of S_e is not included.

```
// S
{
  Sb: i = 42;
  i = i+5;
  i = i*3;
  Se: i = i+4;
}
```

```
path_transformer(S,(⊥,Sb),(⊥,Se)) =
    T(i) {
      i = 141
    }
```

Figure 6.11: An example of a C code and the path transformer of the sequence between S_b and S_e; M is $\perp$, since there are no loops in the code

Test

When S is a test, the path transformer is computed according to the location of S_{begin} and S_{end} in the branches of the test. If one is in the true statement and the other in the false statement of the test, the path transformer is the union of the true and false transformers iff the mode is permanent, i.e, there is at least one loop enclosing S. If only one of the two statements is enclosed in the true or in the false branch of the test, the transformer of the condition (true or false) is composed with the transformer of the corresponding enclosing branch (true or false) and is returned. An example of the result provided by the path transformer algorithm for a test is illustrated in Figure 6.12. The left side presents a code fragment that contains a test. The right side shows the resulting path transformer between the two statements S_b and S_e. It shows the result of the composition of the transformers of statements from S_b to the statement just before the test and the transformer of the condition true (`i>0`) and the statements in the true branch just before S_e; note that the transformer of S_e is not included.

Loop

When S is a loop, we call the function `pt_on_loop` (see Algorithm 16). Note that if loop indices are not specified for a statement in the `map` function, the algorithm posits i_{begin} equal to the lower bound of the loop and i_{end} to the upper one. In this function, a composition of two transformers are computed, depending on the value of *mode*: (1) the prologue transformer from the iteration i_{begin} until the upper bound of the loop, or the iteration i_{end}, if S_{end} belongs to this loop, and (2) the epilogue transformer from the iteration lower bound of the loop until the upper bound of the loop, or the iteration i_{end}, if S_{end} belongs to this loop. These two transformers are computed using the function `iter` that we show in Algorithm 17.

`iter` computes the transformer of a number of iterations of a loop S when loop bounds are constant or symbolic functions, via the predicate function `is_constant_p` on an expression that returns true if this expres-

```
// S
{
  Sb: i=4;
  i = i*3;
  if(i>0){
    k1++;
    Se: i = i+4;
  }
  else{
    k2++;
    i--;
  }
}
```

```
path_transformer(S,(⊥,Sb),(⊥,Se)) =
    T(i, k1) {
      i = 12,
      k1 = k1#init + 1,
    }
```

Figure 6.12: An example of a C code and the path transformer of the sequence of calls and test between S_b and S_e; M is $\perp$, since there are no loops in the code

sion is constant. It computes a loop transformer that is the transformer of the proper number of iterations of this loop. It is equal to the transformer of the looping condition T_{enter}, composed with the transformer of the loop body, composed with the transformer of the incrementation statement T_{inc}. In practice, when $h - l$ is not a constant, a transitive closure approximation T^* is computed using the function already implemented in PIPS, `affine_derivative_closure`, and generally provides an approximation of the transformer T^{h-l+1}.

The case where $i_{begin} = i_{end}$ means that we execute only one iteration of the loop from S_{begin} to S_{end}. Therefore, we compute T_s, i.e, is a transformer on the loop body with the sequence mode using Function `pt_on_loop_one_iteration` presented in Algorithm 17. This eliminates the statements outside the path from S_{begin} to S_{end}. Moreover, since with symbolic parameters, we cannot have this precision, we execute one iteration or more; a union of T and T_s is returned. The result should be composed with the transformer of the index initialization T_{init} and/or the exit condition of the loop T_{exit}, depending on the position of S_{begin} and S_{end} in the loop.

An example of the result provided by the path transformer algorithm is illustrated in Figure 6.13. The left side presents a code that contains a sequence of two loops and an incrementation instruction that makes the computation of the path transformer required when performing parallelization. Indeed, comparing array accesses in the two loops must take into account the fact that n, used in the first loop, is not the same as n in the second one, because it is incremented between the two loops. The right side shows the resulting path transformer between the two statements S_b and S_e. It shows the result of the call to `path_transformer(`S`, (`M_b`,`S_b`),(`M_e`,`S_e`))`. We

ALGORITHM 16: Case for loops of the path transformer algorithm

```
function pt_on_loop(S, (M_begin,S_begin),(M_end,S_end), mode)
  forloop(I, E_lower, E_upper, S_body) = S;
  low = i_begin = (undefined(M_begin(I))) ? E_lower : M_begin(I);
  high = i_end = (undefined(M_end(I))) ? E_upper : M_end(I);
  T_init = T(I = E_lower);
  T_enter = T(I <= E_upper);
  T_inc = T(I=I+1);
  T_exit = T(I > E_upper);
  if (mode = permanent) then
    T = iter(S,(M_begin,S_begin),(M_end,S_end),E_lower,E_upper);
    return  T_init ⋄ T ⋄ T_exit;
  T_s = pt_on_loop_one_iteration(S,(M_begin,S_begin),(M_end,S_end));
  T_p = T_e = p = e = id;
  if (mode = prologue ∨ mode = sequence) then
    if(S_begin ⊂ S)
      p = pt_on(S_body,(M_begin,S_begin),(M_end,S_end), prologue) ⋄ T_inc;
      low =  i_begin+1;
    high = (mode = sequence) ? i_end-1 : E_upper;
    T_p = p ⋄ iter(S,(M_begin,S_begin),(M_end,S_end),low,high);
  if (mode = epilogue ∨ mode = sequence) then
    if(S_end ⊂ S)
      e = T_enter ⋄ pt_on(S_body,(M_begin,S_begin),(M_end,S_end),epilogue);
      high = (mode = sequence) ? -1 : i_end-1;
    low = E_lower;
    T_e = iter(S, (M_begin,S_begin),(M_end,S_end),low,high) ⋄ e;
  T = T_p ⋄ T_e;
  T = ((S_begin ⊄ S) ? T_init : id) ⋄ T ⋄ ((S_end ⊄ S) ? T_exit : id);
  if(is_constant_p(E_upper) ∧ is_constant_p(E_lower)) then
    if (mode = sequence ∧ i_begin = i_end) then
      return  T_s;
    else
      return T;
  else
    return  T ⊔ T_s;
end
```

suppose here that $M_b(i)$=0 and $M_e(j)= n-1$. Notice that the constraint $\{n = n\#\ init+1\}$ is important to detect that n is changed between the two loops via the `n++;` instruction.

An optimization for the cases of a statement S of type sequence or loop would be to return the transformer of S, $T(S)$, if S does not contain neither S_{begin} nor S_{end}.

ALGORITHM 17: Transformers for one iteration and a constant/symbolic number of iterations $h - l + 1$ of a loop S

```
function pt_on_loop_one_iteration(S,(M_begin,S_begin),(M_end,S_end))
  forloop(I, E_lower, E_upper, S_body) = S;
  T_init = T(I = E_lower);
  T_enter = T(I <= E_upper);
  T_inc = T(I=I+1);
  T_exit = T(I > E_upper);
  T_s = pt_on(S_body,(M_begin,S_begin),(M_end,S_end),sequence);
  T_s = (S_begin ⊂ S ∧ S_end ⊄ S) ? T_s ◇ T_inc ◇ T_exit: T_s;
  T_s = (S_end ⊂ S ∧ S_begin ⊄ S) ? T_init ◇ T_enter ◇ T_s : T_s;
  return T_s;
end

function iter(S, (M_begin,S_begin) , (M_end, S_end), l, h)
  forloop(I, E_lower, E_upper, S_body) = S;
  T_enter = T(I <= h);
  T_inc = T(I=I+1);
  T_body = pt_on(S_body,(M_begin,S_begin),(M_end,S_end), permanent);
  T_complete_body = T_enter ◇ T_body ◇ T_inc;
  if(is_constant_p(E_upper) ∧ is_constant_p(E_lower))
    return T(I=l) ◇ (T_complete_body)^(h-l+1);
  else
    T_star = affine_derivative_closure(T_complete_body);
    return T_star;
end
```

6.4.3 Operations on Regions using Path Transformers

In order to make possible operations combining sets of regions of two statements S_1 and S_2, the set of regions R_1 of S_1 must be brought to the store of the second set of regions R_2 of S_2. Therefore, we first compute the path transformer that links the two statements of the two sets of regions as follows:

$$T_{12} = \texttt{path_transformer}(S, (M_1, S_1), (M_2, S_2))$$

where S is a statement that verifies $S_1 \subset S$ and $S_2 \subset S$. M_1 and M_2 contain information about the indices of these statements in the potentially existing loops in S. Then, we compose the second set of regions R_2 with T_{12} to bring R_2 to a common memory store with R_1. We use thus the operation `region_transformer_compose` already implemented in PIPS [35], as follows:

$$R'_2 = \texttt{region_transformer_compose}(R_2, T_{12})$$

We thus update the operations defined in Section 2.6.3: (1) the intersection of the two sets of regions R_1 and R_2 is obtained via this call:

```
//S
{
  //l1
  for(i=0;i<n;i++){
    Sb:
     a[i] = a[i]+1;
    k1++;
  }
  n++;
  //l2
  for(j=0;j<n;j++){
    k2++;
    Se:
     b[j] = a[j]+a[j+1];
  }
}
```

```
path_transformer(S,(Mb,Sb),(Me,Se)) =
    T(i,j,k1,k2,n) {
      i+k1#init = i#init+k1,
      j+k2#init = k2-1,
      n = n#init+1,
      i#init+1 ≤ i,
      n#init ≤ i,
      k2#init+1 ≤ k2,
      k2 ≤ k2#init+n
    }
```

Figure 6.13: An example of a C code and the path transformer between S_b and S_e

`regions_intersection(`R_1`, `R_2'`)`

and (2) the difference between R_1 and R_2 is obtained via this call:

`regions_difference(`R_1`, `R_2'`)`

and (3) the union of R_1 and R_2 is obtained via this call:

`regions_union(`R_1`, `R_2'`)`

For example, if we want to compute RAW dependences between the $Write$ regions R_{w1} of Loop l_1 and *Read* regions R_{r2} of Loop l_2 in the code in Figure 6.13, we must first find T_{12}, the path transformer between the two loops. The result is then:

$$RAW(l_1, l_2) = \texttt{regions_intersection}(R_{w1}, \texttt{regions_transformer_compose}(R_{r2}, T_{12}))$$

If we assume the size of Arrays a and b is $n = 20$, for example, we obtain the following results:

$$\begin{aligned}
R_{w1} &= \{< a(\phi_1) - W - \; EXACT - \{0 \leq \phi_1, \phi_1 \leq 19, \phi_1 + 1 \leq n\} >\} \\
R_{r2} &= \{< a(\phi_1) - R - \; EXACT - \{0 \leq \phi_1, \phi_1 \leq 19, \phi_1 \leq n\} >\} \\
T_{12} &= T(i, k1, n)\{k1 = k\#init + i, n = n\#init + 1, i \geq 0, i \geq n\# \; init\} \\
R'_{r2} &= \{< a(\phi_1) - R - \; EXACT - \{0 \leq \phi_1, \phi_1 \leq 19, \phi_1 \leq n + 1\} >\} \\
R_{w1} \cap R'_{r2} &= \{< a(\phi_1) - WR - \; EXACT - \{0 \leq \phi_1, \phi_1 \leq 19, \phi_1 + 1 \leq n\} >\}
\end{aligned}$$

Notice how ϕ_1 accesses to Array a differ in R_{r2} and R'_{r2} (after composition with T_{12}), because of the equality $n = n\#\ init + 1$ in T_{12}, due to the modification of n between the two loops via the `n++;` instruction.

In our context of combining path transformers with regions to compute communication costs, dependences and data volumes between the vertices of a DAG, for each loop of index i, $M(i)$ must be set to the lower bound of the loop and i_{end} to the upper one (the default values). We are not interested here in computing the path transformer between statements of specific iterations. However, our algorithm is general enough to handle loop parallelism, for example, or transformations on loops where one wants access to statements of a particular iteration.

6.5 BDSC-Based Hierarchical Scheduling (HBDSC)

Now that all the information needed by the basic version of BDSC presented in the previous chapter has been gathered, we detail in this section how we suggest to adapt it to different SDGs linked hierarchically via the mapping function H introduced in Section 6.2.2 on page 115 in order to eventually generate nested parallel code when possible. We adopt in this section the unstructured parallel programming model of Section 4.2.2 since it offers the freedom to implement arbitrary parallel patterns and since SDGs implement this model. Therefore, we use the `parallel unstructured` construct of SPIRE to encode the generated parallel code.

6.5.1 Closure of SDGs

An SDG G of a nested sequence S may depend upon statements outside S. We introduce the function `closure` defined in Algorithm 18 to provide a self-contained version of G. There, G is completed with a set of entry vertices and edges in order to recover dependences coming from outside S. We use the predecessors τ_p in the DDG D to find these dependences Rs; for each dependence we create an entry vertex and we schedule it in the same cluster as τ_p; Function `reference` is used here to find the reference expression of the dependence. Note that these *import* vertices will be used to schedule S using BDSC in order to compute the updated values of `tlevel` and `blevel` impacted by the communication cost of each vertex in G relatively to all SDGs in H. We set their `task_time` to 0.

For instance, in the C code given in the left of Figure 6.3, parallelizing $S0$ may decrease its execution time but it may also increase the execution time of the loop indexed by i; this could be due to communications from outside $S0$, if we presume, to simplify, that every statement is scheduled on a different cluster. The corresponding SDG of $S0$ presented in Figure 6.4 is not complete because of dependences not explicit in this DAG coming from outside $S0$. In order to make $G0$ self-contained, we add these dependences via

ALGORITHM 18: Computation of the closure of the SDG G of a statement $S = \texttt{sequence}(S_1; ...; S_n)$

```
function closure(sequence(S_1;...;S_n), G)
  D = DDG(sequence(S_1;...;S_n));
  foreach S_i ∈ {S_1,S_2,.....,S_m}
    τ_d = statement_vertex(S_i, D);
    τ_k = statement_vertex(S_i, G);
    foreach τ_p / (τ_p ∈ predecessors(τ_d, D) ∧ τ_p ∉ vertices(G))
      Rs = regions_intersection(read_regions(S_i),
                        write_regions(vertex_statement(τ_p)));
      foreach R ∈ Rs
        s_entry = import(reference(R));
        τ_entry = new_vertex(s_entry);
        cluster(τ_entry) = cluster(τ_p);
        add_vertex(τ_entry, G);
        add_edge((τ_entry, τ_k), G);
  return G;
end
```

a set of additional import entry vertices τ_{entry} scheduled in the same cluster as the corresponding statements outside $S0$. We extract these dependences from the DDG presented in the right of Figure 6.3 as illustrated in Function `closure`; the closure of $G0$ is given in Figure 6.14. The scheduling of $S0$ needs to take into account all the dependences coming from outside $S0$; we add thus three vertices to encode the communication cost of $\{i, a[i], c\}$ at the entry of $S0$.

6.5.2 Recursive Top-Down Scheduling

Hierarchically scheduling a given statement S of SDG $H(S)$ in a cluster κ is seen here as the definition of a *hierarchical schedule* σ which maps each substatement s of S to $\sigma(s) = (s', \kappa, n)$. If there are enough processor and memory resources to schedule S using BDSC, (s', κ, n) is a triplet made of a *parallel statement* $s' = \texttt{parallel}(\sigma(s))$, the cluster $\kappa = \texttt{cluster}(\sigma(s))$ where s is being allocated and the number $n = \texttt{nbclusters}(\sigma(s))$ of clusters the inner scheduling of s' requires. Otherwise, scheduling is impossible, and the program stops. In a scheduled statement, all sequences are replaced by parallel unstructured statements.

A successful call to the $\texttt{HBDSC}(S, H, \kappa, P, M, \sigma)$ function defined in Algorithm 19, which assumes that P is strictly positive, yields a new version of σ that schedules S into κ and takes into account all substatements of S; only P clusters, with a data size at most M each, can be used for scheduling. $\sigma[S \to (S', \kappa, n)]$ is the function equal to σ except for S, where its value is

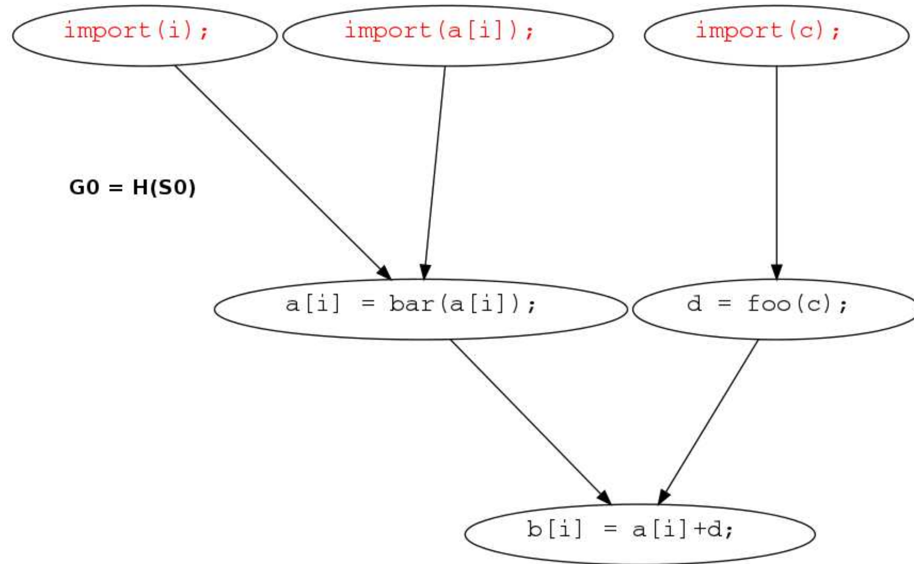


Figure 6.14: Closure of the SDG $G0$ of the C code $S0$ in Figure 6.3, with additional import entry vertices

(S', κ, n). H is the function that yields an SDG for each S to be scheduled using BDSC.

Our approach is top-down in order to yield tasks that are as coarse grained as possible when dealing with sequences. In the `HBDSC` function, we distinguish four cases of statements. First, the constructs of loops[8] and tests are simply traversed, scheduling information being recursively gathered in different SDGs. Then, for a call statement, there is no descent in the call graph, the call statement is returned. In order to handle the corresponding call function, one has to treat separately the different functions. Finally, for a sequence S, one first accesses its SDG and computes a closure of this DAG G_{seq} using the function `closure` defined above. Next, G_{seq} is scheduled using BDSC to generate a scheduled SDG G'.

The hierarchical scheduling process is then recursively performed, to take into account substatements of S, within Function `HBDSC_step` defined in Algorithm 20 on each statement s of each task of G'. There, G' is traversed along a topological sort-ordered descent using the function `topsort`(G') yields a list of stages of computation, each *cluster_stage* being a list of independent lists L of tasks τ, one L for each cluster κ generated by BDSC

[8]Regarding parallel loops, since we adopt the task parallelism paradigm, note that, initially, it may be useful to apply the tiling transformation and then perform full unrolling of the outer loop (we give more details in the protocol of our experiments in Section 8.4). This way, the input code contains more potentially parallel tasks resulting from the initial (parallel) loop.

ALGORITHM 19: BDSC-based update of Schedule σ for Statement S of SDG $H(S)$, with P and M constraints

```
function HBDSC(S, H, κ, P, M, σ)
  switch (S)
    case call:
      return σ[S → (S, κ, 0)];
    case sequence(S_1;...;S_n):
      G_seq = closure(S, H(S))
      G′ = BDSC(G_seq, P, M, σ);
      iter = 0;
      do
        σ′ = HBDSC_step(G′, H, κ, P, M, σ);
        G = G′;
        G′ = BDSC(G_seq, P, M, σ′);
        if (clusters(G′) = ∅) then
          abort(′Unable to schedule′);
        iter++;
        σ = σ′;
      while (completion_time(G′) < completion_time(G) ∧
             |clusters(G′)| ≤ |clusters(G)| ∧
             iter ≤ MAX_ITER)
      return σ[S →(dag_to_unstructured(G),
                   κ, |clusters(G)|)];
    case forloop(I, E_lower, E_upper, S_body):
      σ′ = HBDSC(S_body, H, κ, P, M, σ);
      (S′_body, κ_body, nbclusters_body) = σ′(S_body);
      return σ′[S → (forloop(I, E_lower, E_upper, S′_body),
                    κ, nbclusters_body)];
    case test(E_cond, S_t, S_f):
      σ = HBDSC(S_t, H, κ, P, M, σ′);
      σ″ = HBDSC(S_f, H, κ, P, M, σ′);
      (S′_t, κ_t, nbclusters_t) = σ″(S_t);
      (S′_f, κ_f, nbclusters_f) = σ″(S_f);
      return σ″[S → (test(E_cond, S′_t, S′_f),
                     κ, max(nbclusters_t, nbclusters_f))];
end
```

for this particular stage in the topological order.

The recursive hierarchical scheduling via `HBDSC`, within the function `HBDSC_step`, of each statement s = `vertex_statement`(τ) may take advantage of at most P' available clusters, since $|cluster_stage|$ clusters are already reserved to schedule the current stage $cluster_stage$ of tasks for Statement S. It yields a new scheduling function σ_s. Otherwise, if no clusters are available, all statements enclosed into s are scheduled on the same cluster as their parent, κ. We use the straightforward function `same_cluster_mapping` (not

provided here) to affect recursively (s_e, κ, 0) to $\sigma(s_e)$ for each s_e enclosed into s.

ALGORITHM 20: Iterative hierarchical scheduling step for DAG fixpoint computation

```
function HBDSC_step(G′, H, κ, P, M, σ)
  foreach cluster_stage ∈ topsort(G′)
    P′ = P - |cluster_stage|;
    foreach L ∈ cluster_stage
      nbclusters_L = 0;
      foreach τ ∈ L
        s = vertex_statement(τ);
        if (P′ ≤ 0) then
          σ = σ[s → same_cluster_mapping(s, κ, σ)];
        else
          nbclusters_s = (s ∈ domain(σ)) ?
                                   nbclusters(σ(s)) : 0;
          σ_s = HBDSC(s, H, cluster(τ), P′, M, σ);
          nbclusters′_s = nbclusters(σ_s(s));
          if (nbclusters′_s ≥ nbclusters_s ∧
               task_time(τ, σ) ≥ task_time(τ, σ_s) then
            nbclusters_s = nbclusters′_s;
            σ = σ_s;
          nbclusters_L = max(nbclusters_L, nbclusters_s);
      P′ -= nbclusters_L;
  return σ;
end
```

Figure 6.15 illustrates the various entities involved in the computation of such a scheduling function. Note that one needs to be careful in `HBDSC_step` to ensure that each rescheduled substatement s is allocated a number of clusters consistent with the one used when computing its parallel execution time; we check the condition $nbclusters'_{\mathrm{s}} \geq nbclusters_{\mathrm{s}}$, which ensures that the parallelism assumed when computing time complexities within s remains available.

Cluster allocation information for each substatement s whose vertex in G' is τ is maintained in σ via the recursive call to `HBDSC`, this time with the current cluster κ = `cluster`(τ). For the non-sequence constructs in Function `HBDSC`, cluster information is set to κ, the current cluster.

The scheduling of a sequence yields a parallel unstructured statement; we use the function `dag_to_unstructured`(G) that returns a SPIRE unstructured statement S_u from the SDG G, where the vertices of G are the statement control vertices s_u of S_u, and the edges of G constitute the list of successors $\mathtt{L}_{succ}$ of s_u while the list of predecessors $\mathtt{L}_{pred}$ of s_u is deduced from $\mathtt{L}_{succ}$. Note that vertices and edges of G are not changed before and

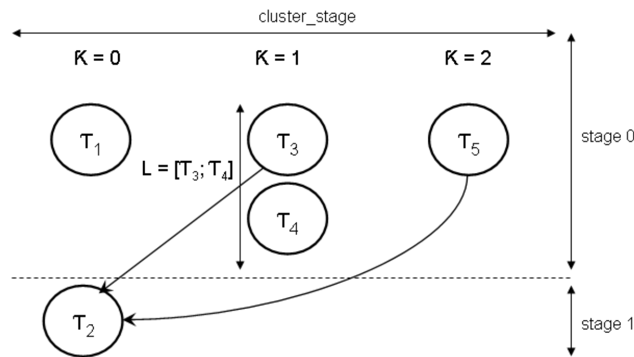


Figure 6.15: `topsort`(G) for the hierarchical scheduling of sequences

after scheduling; however, information of scheduling is saved in σ.

As an application of our scheduling algorithm on a real application, Figure 6.16 shows the scheduled SDG for Harris obtained using $P = 3$, generated automatically with PIPS using the Graphviz tool.

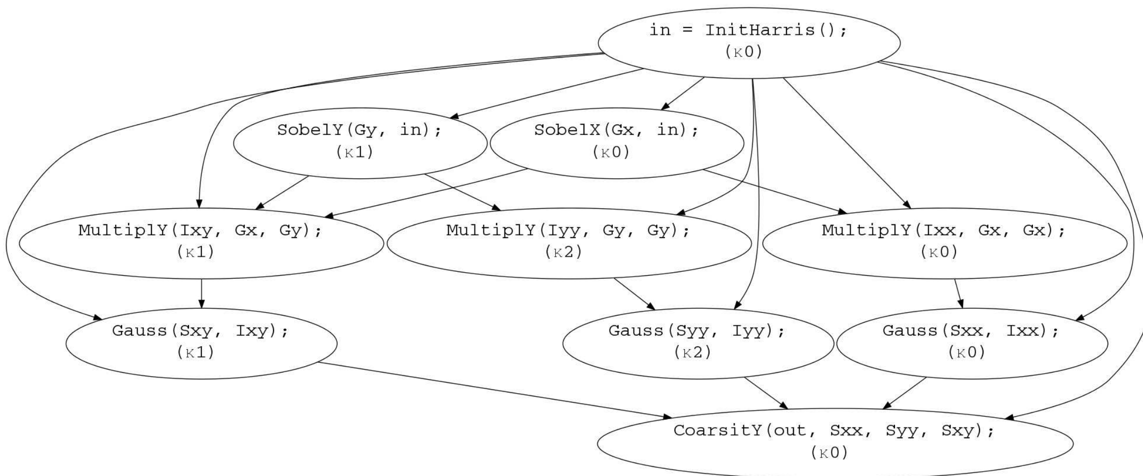


Figure 6.16: Scheduled SDG for Harris, using P=3 cores; the scheduling information via `cluster`(τ) is also printed inside each vertex of the SDG

6.5.3 Iterative Scheduling for Resource Optimization

BDSC is called in `HBDSC` before substatements are hierarchically scheduled. However, a unique pass over substatements could be suboptimal, since parallelism may exist within substatements. It may be discovered by later recursive calls to `HBDSC`. Yet, if this parallelism had been known ahead of time, previous values of `task_time` used by BDSC would have been possibly smaller, which could have had an impact on the higher-level scheduling. In

order to address this issue, our hierarchical scheduling algorithm iterates the top down pass `HBDSC_step` on the new DAG G' in which BDSC takes into account these modified task complexities; iteration continues while G' provides a smaller DAG scheduling length than G and the iteration limit *MAX_ITER* has not been reached. We compute the completion time of the DAG G, as follows:

$$\texttt{completion_time}(G) = \max_{\kappa \in clusters(G)} \texttt{cluster_time}(\kappa)$$

One constraint due to the iterative nature of the hierarchical scheduling is that, in BDSC, zeroing cannot be made between the import entry vertices and their successors. This keeps an independence in terms of allocated clusters between the different levels of the hierarchy. Indeed, at a higher level, for S, if we assume that we have scheduled the parts S_e inside (hierarchically) S; attempting to reschedule S iteratively cancels the precedent schedule of S but maintains the schedule of S_e and vice versa. Therefore, for each sequence, we have to deal with a new set of clusters; and thus, zeroing cannot be made between these entry vertices and their successors.

Note that our top-down, iterative, hierarchical scheduling approach also helps dealing with limited memory resources. If BDSC fails at first because not enough memory is available for a given task, the `HBDSC_step` function is nonetheless called to schedule nested statements, possibly loosening up the memory constraints by distributing some of the work on less memory-challenged additional clusters. This might enable the subsequent call to BDSC to succeed.

For instance, in Figure 6.17, we assume that $P = 3$ and the memory M of each available cluster κ_0, κ_1 or κ_2 is less that the size of Array a. Therefore, the first application of BDSC on $S = \texttt{sequence}(S_0;S_1)$ will fail (Not enough memory). Thanks to the while loop introduced in the hierarchical schedule algorithm, the scheduling of the hierarchical parts inside S_0 and S_1 hopefully minimizes the `task_data` of each statement. Thus, a second schedule succeeds, as is illustrated in Figure 6.18. Note that the `edge_cost` between tasks S_0 and S_1 is equal to 0 (the region on the edge in the figure is equal to the empty set). Indeed, the communication is made at an inner level, between κ_1 and κ_3, due to the entry vertex added to $H(S_{1body})$. This cost is obtained after the hierarchical scheduling of S; it is thus a parallel `edge_cost`; we define parallel `task_data` and `edge_cost` of a parallel statement task in Section 6.5.5.

6.5.4 Complexity of HBDSC Algorithm

Theorem 2. *The time complexity of Algorithm 19 (HBDSC) over Statement S is $\mathcal{O}(k^n)$, where n is the number of call statements in S and k a constant greater than 1.*

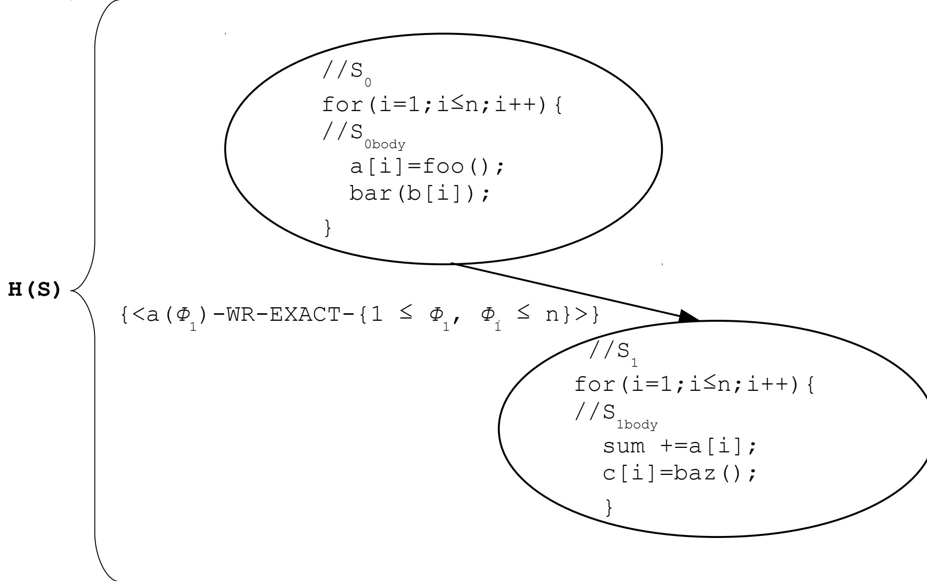


Figure 6.17: After the first application of BDSC on $S =$ `sequence`$(S_0;S_1)$, failing with "Not enough memory"

Proof. Let $t(l)$ be the worst-case time complexity for our hierarchical scheduling algorithm on the structured statement S of hierarchical level[9] l. Time complexity increases significantly only in sequences, loops and tests being simply managed by straightforward recursive calls of `HBDSC` on sub-statements. For a sequence S, $t(l)$ is proportional to the time complexity of BDSC followed by a call to `HBDSC_step`; the proportionality constant is $k = MAX_ITER$ (supposed to be greater than 1).

The time complexity of BDSC for a sequence of m statements is at most $\mathcal{O}(m^3)$ (see Theorem 1). Assuming that all subsequences have a maximum number m of (possibly compound) statements, the time complexity for the hierarchical scheduling step function is the time complexity of the topological sort algorithm followed by a recursive call to `HBDSC`, and is thus $\mathcal{O}(m^2 + mt(l-1))$. Thus $t(l)$ is at most proportional to $k(m^3 + m^2 + mt(l-1)) \sim km^3 + kmt(l-1)$. Since $t(l)$ is an arithmetico-geometric series, its analytical value $t(l)$ is $\frac{(km)^l(km^3+km-1)-km^3}{km-1} \sim (km)^l m^2$. Let l_S be the level for the whole Statement S. The worst performance occurs when the structure of S is flat, i.e., when $l_S \sim n$ and m is $\mathcal{O}(1)$; hence $t(n) = t(l_S) \sim k^n$. □

Even though the worst case time complexity of `HBDSC` is exponential, we expect and our experiments suggest that it behaves more tamely on actual, properly structured code. Indeed, note that $l_S \sim log_m(n)$ if S is balanced for some large constant m; in this case, $t(n) \sim (km)^{log(n)}$, showing

[9]Levels represent the hierarchy structure between statements of the AST and are counted up from leaves to the root.

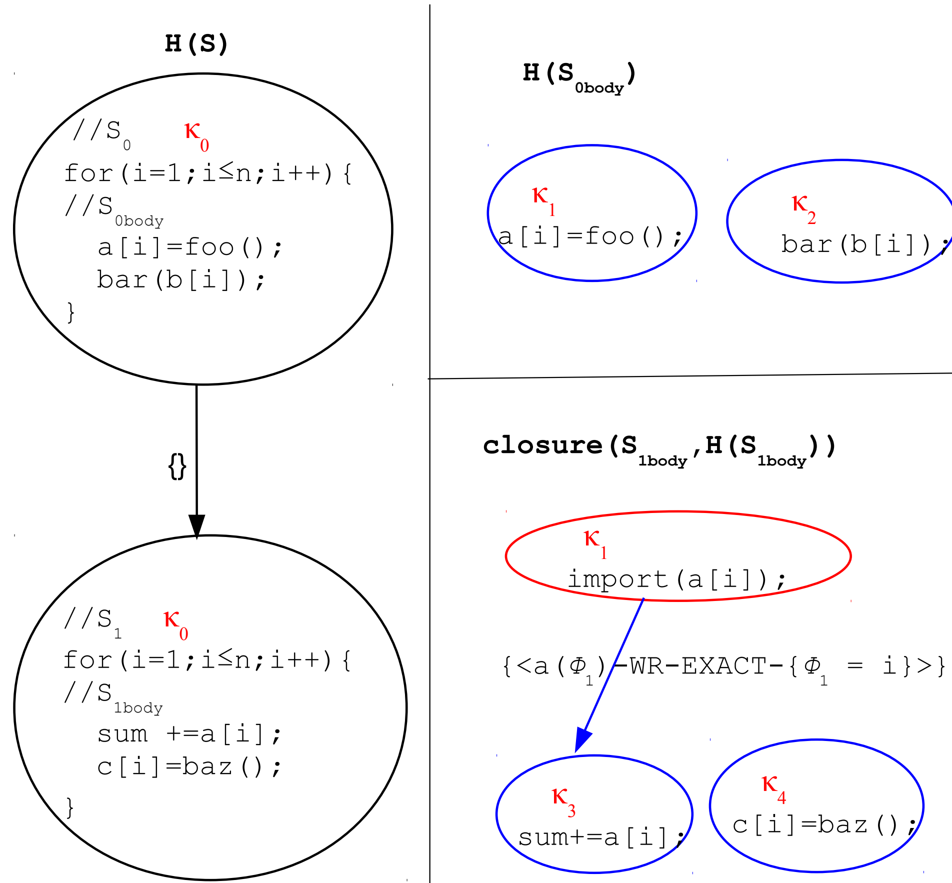


Figure 6.18: Hierarchically (via H) Scheduled SDGs with memory resource minimization after the second application of BDSC (which succeeds) on `sequence`(S_0;S_1) (κ_i = `cluster`($\sigma(S_i)$)). To keep the picture readable, only communication edges are figured in these SDGs

a subexponential time complexity.

6.5.5 Parallel Cost Models

In Section 6.3, we present the sequential cost models usable in the case of sequential codes, i.e, for each first call to BDSC. When a substatement S_e of S ($S_e \subset S$) is parallelized, the parameters `task_time`, `task_data` and `edge_cost` are modified for S_e and thus for S. Thus, hierarchical scheduling must use extended definitions of `task_time`, `task_data` and `edge_cost` for tasks τ using statements S = `vertex_statement`(τ) that are parallel unstructured statements, extending the definitions provided in Section 6.3, which still applies to non-unstructured statements. For such a case, we as-

sume that BDSC and other relevant functions take σ as an additional argument to access the scheduling result associated to statement sequences and handle the modified definitions of `task_time`, `edge_cost` and `task_data`. These functions are defined as follows.

Parallel Task Time

When statements S = `vertex_statement`(τ) correspond to parallel unstructured statements, lets S_u = `parallel`($\sigma(S)$) be the parallel unstructured (i.e., such that $S \in domain(\sigma)$). We define `task_time` as follows, where Function `clusters` returns the set of clusters used to schedule S_u and Function `control_statements` is used to return the statements from S_u:

$$\begin{aligned} \texttt{task_time}(\tau, \sigma) &= \max_{\kappa \in \texttt{clusters}(S_u, \sigma)} \texttt{cluster_time}(\kappa) \\ \texttt{clusters}(S_u, \sigma) &= \{\texttt{cluster}(\sigma(s)) / s \in \texttt{control_statements}(S_u)\} \end{aligned}$$

Otherwise, the previous definitions of `task_time` on S provided in Section 6.3 is used. The recursive computation of complexities accesses this new definition in case of parallel unstructured statements.

Parallel Task Data and Edge Cost

To define `task_data` and `edge_cost` parameters that depend on the computation of array regions[10], we need extended versions of the `read_regions` and `write_regions` functions that take into account the allocation information to clusters. These functions take σ as an additional argument to access the allocation information `cluster(`σ`)`:

$$\begin{aligned} S &= \texttt{vertex_statement}(\tau); \\ \texttt{task_data}(\tau, \sigma) &= \texttt{regions_union}(\texttt{read_regions}(S, \sigma), \\ &\qquad\qquad \texttt{write_regions}(S, \sigma)) \\ R_{w1} &= \texttt{write_regions}(\texttt{vertex_statement}(\tau_1), \sigma); \\ R_{r2} &= \texttt{read_regions}(\texttt{vertex_statement}(\tau_2), \sigma); \\ \texttt{edge_cost_bytes}(\tau_1, \tau_2, \sigma) &= \sum_{r \in \texttt{regions_intersection}(R_{w1}, R_{r2})} \texttt{ehrhart}(r); \\ \texttt{edge_cost}(\tau_1, \tau_2, \sigma) &= \alpha + \beta \times \texttt{edge_cost_bytes}(\tau_1, \tau_2, \sigma) \end{aligned}$$

When statements are parallel unstructured statements S_u, we propose to add a condition on the accumulation of regions in the sequential recursive computation functions of these regions as follows, where Function `stmts_in_cluster`

[10] Note that every operation combining regions of two statements, used in this chapter, assumes the use of the path transformer via the function `region_transformer_compose` as illustrated in Section 6.4.3 in order to bring the regions of the two statements in a common memory store.

returns the set of statements scheduled in the same cluster as S_u:

$$\texttt{read_regions}(S_u, \sigma) = \\ \quad \texttt{regions_union}_{s \in \texttt{stmts_in_cluster}(S_u,\sigma)} \texttt{read_regions}(s, \sigma)$$

$$\texttt{write_regions}(S_u, \sigma) = \\ \quad \texttt{regions_union}_{s \in \texttt{stmts_in_cluster}(S_u,\sigma)} \texttt{write_regions}(s, \sigma)$$

$$\texttt{stmts_in_cluster}(S_u, \sigma) = \\ \{s \in \texttt{control_statements}(S_u) / \texttt{cluster}(\sigma(s)) = \texttt{cluster}(\sigma(S_u))\}$$

Otherwise, the sequential definitions of `read_regions` and `write_regions` on S provided in Section 6.3 is used. The recursive computation of array regions accesses this new definition in case of parallel unstructured statements.

6.6 Related Work: Task Parallelization Tools

In this section, we review several other approaches that intend to automate the parallelization of programs using different granularities and scheduling policies. Given the breadth of literature on this subject, we limit this presentation to approaches that focus on static list-scheduling methods and compare them with BDSC.

Sarkar's work on the partitioning and scheduling of parallel programs [98] for multiprocessors introduced a compile-time method where a GR (Graphical Representation) program is partitioned into parallel tasks at compile time. A GR graph has four kinds of vertices: "simple", to represent an indivisible sequential computation, "function call", "parallel", to represent parallel loops, and "compound", for conditional instructions. Sarkar presents an approximation parallelization algorithm. Starting with an initial fine granularity partition, P_0, tasks (chosen by heuristics) are iteratively merged till the coarsest partition P_n (with one task containing all vertices), after n iterations, is reached. The partition P_{min} with the smallest parallel execution time in the presence of overhead (scheduling and communication overhead) is chosen. For scheduling, Sarkar introduces the EZ (Edge-Zeroing) algorithm that uses bottom levels for ordering: it is based on edge weights for clustering; all edges are examined from the largest edge weight to the smallest; it then proceeds by zeroing the highest edge weight if the completion time decreases. While this algorithm is based only on the bottom level for an unbounded number of processors and does not recompute the priorities after zeroings, BDSC adds resource constraints and is based on both bottom levels and dynamic top levels.

The OSCAR Fortran Compiler [63] is used as a multigrain parallelizer from Fortran to parallelized OpenMP Fortran. OSCAR partitions a program into a macro-task graph, where vertices represent macro-tasks of three

kinds, namely basic, repetition and subroutine blocks. The coarse grain task parallelization proceeds as follows. First, the macro-tasks are generated by decomposition of the source program. Then, a macro-flow graph is generated to represent data and control dependences on macro-tasks. The macro-task graph is subsequently generated via the analysis of parallelism among macro-tasks using an earliest executable condition analysis that represents the conditions on which a given macro-task may begin its execution at the earliest time, assuming precedence constraints. If a macro-task graph has only data dependence edges, macro-tasks are assigned to processors by static scheduling. If a macro-task graph has both data and control dependence edges, macro-tasks are assigned to processors at run time by a dynamic scheduling routine. In addition to dealing with a richer set of resource constraints, BDSC targets both shared and distributed memory systems with a cost model based on communication, used data and time estimations.

Pedigree [85] is a compilation tool based on the program dependence graph (PDG). The PDG is extended by adding a new type of vertex, a "Par" vertex, which groups children vertices reachable via the same branch conditions. Pedigree proceeds by estimating a latency for each vertex and data dependences edge weights in the PDG. The scheduling process orders the children and assigns them to a subset of the processors. For scheduling, vertices with minimum early and late times are given highest priority; the highest priority ready vertex is selected for scheduling based on the synchronization overhead and latency. While Pedigree operates on assembly code, PIPS and our extension for task-parallelism using BDSC offer a higher-level, source-to-source parallelization framework. Moreover, Pedigree generated code is specialized for only symmetric multiprocessors, while BDSC targets many architecture types, thanks to its resource constraints and cost models.

The SPIR (Signal Processing Intermediate Representation) compiler [31] takes a sequential dataflow program as input and generates a multithreaded parallel program for an embedded multicore system. First, SPIR builds a stream graph where a vertex corresponds to a kernel function call or to the condition of an "if" statement; an edge denotes a transfer of data between two kernel function calls or a control transfer by an "if" statement (true or false). Then, for task scheduling purposes, given a stream graph and a target platform, the task scheduler assigns each vertex to a processor in the target platform. It allocates stream buffers, and generates DMA operations under given memory and timing constraints. The degree of automation of BDSC is larger than SPIR's, because this latter system needs several keywords extensions plus the C code denoting the streaming scope within applications. Also, the granularity in SPIR is a function, whereas BDSC uses several granularity levels.

We collect in Table 6.2 the main characteristics of each parallelization tool addressed in this section.

	blevel	tlevel	Resource constraints	Dependence control	Dependence data	Execution time estimation	Communication time estimation	Memory model
HBDSC	√	√	√	X	√	√	√	Shared, distributed
Sarkar's work	√	X	X	X	√	√	√	Shared, distributed
OSCAR	X	√	X	√	√	√	X	Shared
Pedigree	√	√	X	√	√	√	X	Shared
SPIR	X	√	√	√	√	√	√	Shared

Table 6.2: Comparison summary between different parallelization tools

6.7 Conclusion

This chapter presents our BDSC-based hierarchical task parallelization approach, which uses a new data structure called SDG and a symbolic execution and communication cost model, based on either static code analysis or a dynamic-based instrumentation assessment tool. The HBDSC scheduling algorithm maps hierarchically, via a mapping function H, each task of the SDG to one cluster. The result is the generation of a parallel code represented in SPIRE(PIPS IR) using the parallel `unstructured` construct. We have integrated this approach within the PIPS automatic parallelization compiler infrastructure (See Section 8.2 for the implementation details).

The next chapter illustrates two parallel transformations of SPIRE(PIPS IR) parallel code and the generation of OpenMP and MPI codes from SPIRE(PIPS IR).

An earlier version of the work presented in the previous chapter (Chapter 5) and a part of the work presented in this chapter are submitted to *Parallel Computing* [66].

Chapter 7

SPIRE-Based Parallel Code Transformations and Generation

Satisfaction lies in the effort, not in the attainment; full effort is full victory.
Gandhi

In the previous chapter, we describe how sequential programs are analyzed, scheduled and represented in a parallel intermediate representation. Their parallel versions are expressed as graphs using the `parallel unstructured` attribute of SPIRE; but such graphs are not directly expressible in parallel languages. To test the flexibility of this parallelization approach, this chapter covers three aspects of parallel code transformations and generation: (1) the transformation of SPIRE(PIPS IR)[1] code from the unstructured parallel programing model to a structured (high level) model using the `spawn` and `barrier` constructs, (2) the transformation from the shared to the distributed memory programming model, and (3) the code generation targeting of parallel languages from SPIRE(PIPS IR). We have implemented our SPIRE-derived parallel IR in the PIPS middle-end, and HBDSC-based task parallelization algorithm. We generate both OpenMP and MPI code from the same parallel IR using the PIPS backend and its prettyprinters.

Dans le chapitre précédent, nous décrivons comment des programmes séquentiels sont analysés, ordonnancés et représentés dans une représentation intermédiaire parallèle. Leurs versions parallèles sont exprimées sous forme de graphes à l'aide de la construction `parallel unstructured` de SPIRE. Toutefois, ces graphes ne sont pas encore directement exprimés en langages parallèles. Pour tester la souplesse de cette approche de parallélisation, ce chapitre porte sur trois aspects de transformation et de génération de codes parallèles : (1) la transformation de codes représentés dans SPIRE(PIPS IR)[2] du modèle de programmation parallèle non-structurée à un modèle structuré (de haut niveau) en utilisant les constructions `spawn` et `barrier`, (2) la transformation du modèle de programmation à mémoire partagée au

[1]Recall SPIRE(PIPS IR) means that the function of transformation SPIRE is applied on the sequential IR of PIPS; it yields a parallel IR of PIPS.

[2]Rappelons que SPIRE(PIPS IR) signifie que la fonction de transformation SPIRE est appliquée sur la RI séquentielle de PIPS ; cela donne une RI parallèle de PIPS.

modèle à mémoire distribuée et (3) la génération de code ciblant des langages parallèles à partir de SPIRE(PIPS IR). Nous avons implanté notre RI parallèle dérivée de SPIRE dans le coeur de PIPS, ainsi que l'algorithme de parallélisation de tâches fondé sur HBDSC. Nous générons des codes OpenMP et MPI à partir de la même RI parallèle en utilisant le générateur de code de PIPS et ses afficheurs.

7.1 Introduction

Generally, the process of source-to-source code generation is a difficult task. In fact, compilers use at least two intermediate representations when transforming a source code written in a programming language abstracted in a first IR to a source code written in an other programming language abstracted in a second IR. Moreover, this difficulty is compounded whenever the resulting input source code is to be executed on a machine not targeted by its language. The portability issue was a key factor when we designed SPIRE as a generic approach (Chapter 4). Indeed, representing different parallel constructs from different parallel programming languages makes the code generation process generic. In the context of this thesis, we need to generate the equivalent parallel version of a sequential code; representing the parallel code in SPIRE facilitates parallel code generation in different parallel programming languages.

The parallel code encoded in SPIRE(PIPS IR), that we generate in Chapter 6, uses an unstructured parallel programming model based on static task graphs. However, structured programming constructs increase the level of abstraction when writing parallel codes, and structured parallel programs are easier to analyze. Moreover, parallel unstructured does not exist in parallel languages. In this chapter, we use SPIRE both for the input, unstructured parallel code, and the output, structured parallel code, by translating the SPIRE `unstructured` construct to `spawn` and `barrier` constructs. Another parallel transformation we address is to transform a SPIRE shared memory code to a SPIRE distributed memory one in order to adapt the SPIRE code to message-passing libraries/languages such as MPI and thus distributed memory system architectures.

A parallel program transformation takes the parallel intermediate representation of a code fragment and yields a new equivalent parallel representation for it. Such transformations are useful for optimization purposes or to enable other optimizations that intend to improve execution time, data locality, energy, memory use, etc. While parallel code generation outputs a final code written into a parallel programming language that can be compiled by its compiler and run on a target machine, a parallel transformation generates an intermediate code written into a parallel intermediate representation.

In this chapter, a parallel transformation takes a *hierarchical schedule* σ, defined in Section 6.5.2 on page 135, and yields another schedule σ', which maps each statement S in `domain`(σ) to $\sigma'(S) = (S', \kappa, n)$ such that S' belongs to the resulting parallel IR, $\kappa =$ `cluster`$(\sigma(S))$ is the cluster where S is allocated, and $n =$ `nbclusters`$(\sigma(S))$ is the number of clusters the inner scheduling of S' requires. A cluster stands for a processor.

Today's typical architectures have multiple hierarchical levels. A multiprocessor has many nodes, each node has multiple clusters, each cluster has multiple cores, and each core has multiple physical threads. Therefore, in addition to distinguishing the parallelism in terms of abstraction of parallel constructs such as structured and unstructured, we can also classify parallelism in two types in terms of code structure: *equilevel*, when it is generated within a sequence, i.e, zero hierarchy level, and *hierarchical* or *multilevel*, when one manages the code hierarchy, i.e. multiple levels. In this chapter, we handle these two types of parallelism in order to enhance the performance of the generated parallel code.

The remainder of this chapter is structured into three sections. Firstly, Section 7.2 presents the first of the two parallel transformations we designed: the unstructured to structured parallel code transformation. Secondly, the transformation from shared memory to distributed memory codes is presented in Section 7.3. Moving to code generation issues, Section 7.4 details the generation of parallel programs from SPIRE(PIPS IR); first, Section 7.4.1 shows the mapping approach between SPIRE and different parallel languages, and then, as illustrating cases, Section 7.4.2 introduces the generation of OpenMP and Section 7.4.3 the generation of MPI, from SPIRE(PIPS IR). Finally, we conclude in Section 7.5. Figure 7.1 summarizes the parallel code transformations and generation applied in this chapter.

7.2 Parallel Unstructured to Structured Transformation

We present in Chapter 4 the application of the SPIRE methodology to the PIPS IR case. Chapter 6 shows how to construct unstructured parallel code presented in SPIRE(PIPS IR). In this section, we detail the algorithm transforming unstructured (mid, graph-based, level) parallel code to structured (high, syntactic, level) code expressed in SPIRE(PIPS IR) and the implementation of the structured model in PIPS. The goal behind structuring parallelism is to increase the level of abstraction of parallel programs and simplify analysis and optimization passes. Another reason is the fact that parallel unstructured does not exist in parallel languages. However, this has a cost in terms of performance if the code contains arbitrary parallel patterns. For instance, the example already presented in Figure 4.3, which uses `unstructured` constructs to form a graph, and copied again for con-

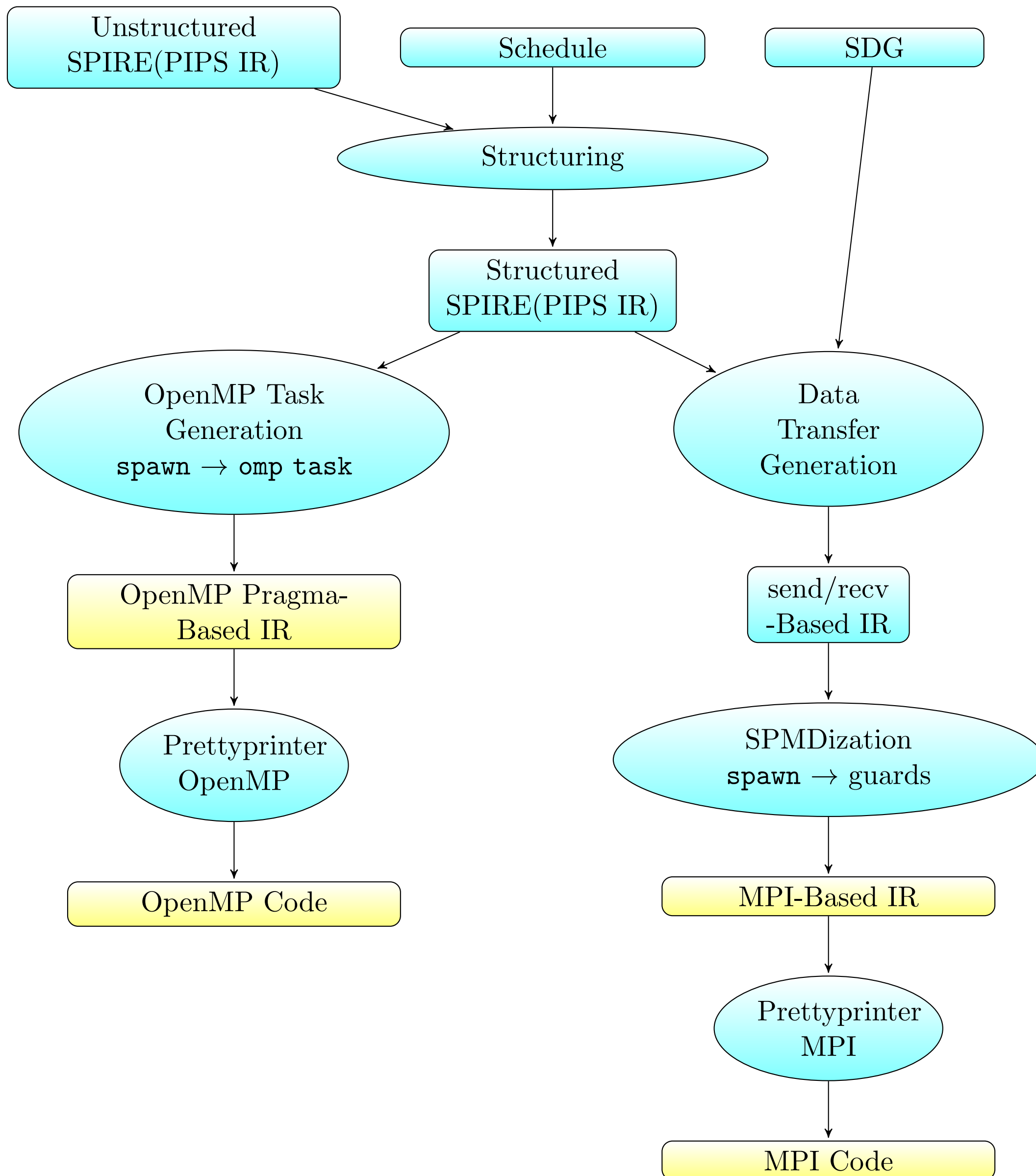


Figure 7.1: Parallel code transformations and generation: blue indicates this chapter contributions; an ellipse, a process; and a rectangle, results

venience in Figure 7.2, can be represented in a low-level unstructured way using events ((b) of Figure 7.2) and in a structured way using `spawn`[3] and `barrier` constructs ((c) of Figure 7.2). The structured representation (c)

[3]Note that, in this chapter, we assume for simplification purposes that constants, instead of temporary identifiers having the corresponding value, are used as the first parameter of `spawn` statements.

minimizes control overhead. Moreover, it is a trade-off between structuring and performance. Indeed, if the tasks have the same execution time, the structured way is better than the unstructured one since no synchronizations using events are necessary. Therefore, depending on the run-time properties of the tasks, it may be better to run one form or the other.

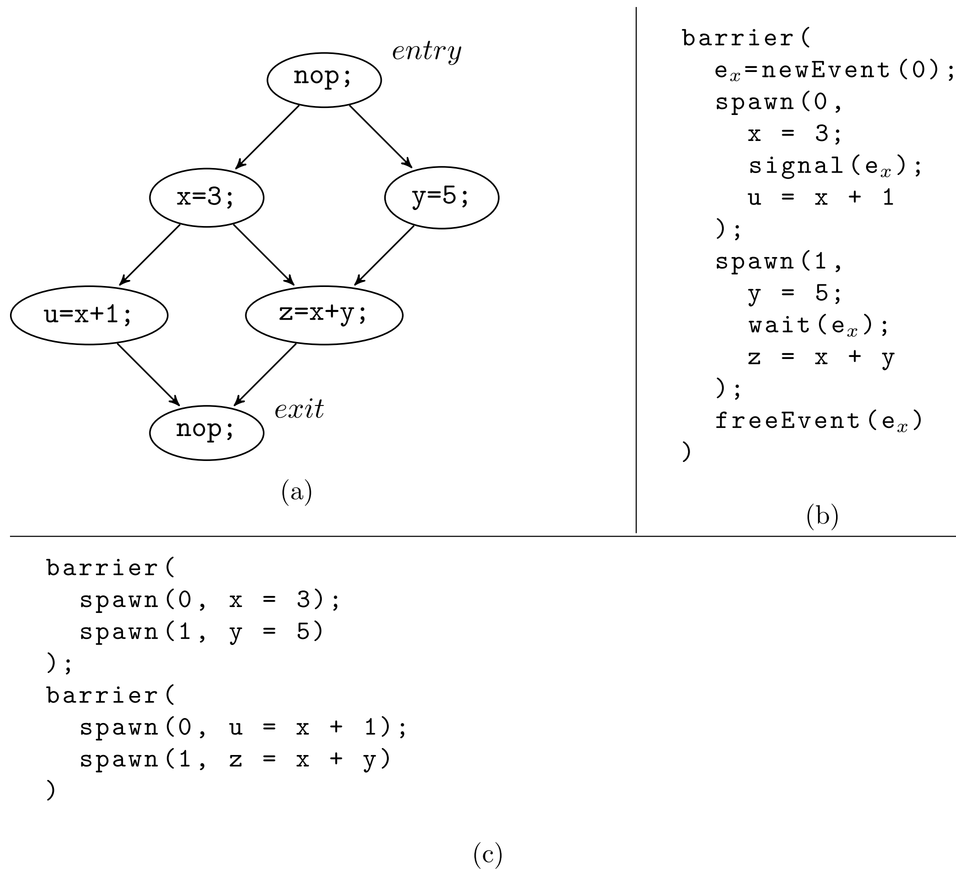


Figure 7.2: Unstructured parallel control flow graph (a) and event (b) and structured representations (c)

7.2.1 Structuring Parallelism

Function `unstructured_to_structured` presented in Algorithm 21 uses a hierarchical schedule σ, that maps each substatement s of S to $\sigma(s) = (s', \kappa, n)$ (see Section 6.5), to handle new statements. It specifies how the unstructured scheduled statement `parallel`($\sigma(S)$) presented in Section 6.5 is used to generate a structured parallel statement encoded also in SPIRE(PIPS IR) using `spawn` and `barrier` constructs. Note that another transformation would be to generate events and `spawn` constructs from the `unstructured` constructs (as in (b) of Figure 7.2).

ALGORITHM 21: Bottom-up hierarchical unstructured to structured SPIRE transformation updating a parallel schedule σ for Statement S

```
function unstructured_to_structured(S, p, σ)
  κ = cluster(σ(S));
  n = nbclusters(σ(S));
  switch (S)
    case call:
      return σ;
    case unstructured(C_entry, C_exit, parallel):
      S_structured = [];
      foreach cluster_stage ∈ topsort_unstructured(S)
        (S_barrier,σ) = spawns_barrier(cluster_stage, κ, p, σ);
        n_barrier = |structured_clusters(statements(S_barrier), σ)|;
        σ = σ[S_barrier → (S_barrier, κ, n_barrier)];
        S_structured ||= [S_barrier];
      S′ = sequence(S_structured);
      return σ[S →(S′, κ, n)];
    case forloop(I, E_lower, E_upper, S_body):
      σ = unstructured_to_structured(S_body, p, σ);
      S′ = forloop(I, E_lower, E_upper, parallel(σ(S_body)));
      return σ[S → (S′, κ, n)];
    case test(E_cond, S_t, S_f):
      σ = unstructured_to_structured(S_t, p, σ);
      σ = unstructured_to_structured(S_f, p, σ);
      S′ = test(E_cond, parallel(σ(S_t)), parallel(σ(S_f)));
      return σ[S → (S′, κ, n)];
end

function structured_clusters({S_1,S_2,...,S_n}, σ)
  clusters = ∅;
  foreach S_i ∈ {S_1,S_2,...,S_n}
    if(cluster(σ(S_i)) ∉ clusters)
      clusters ∪= {cluster(σ(S_i))};
  return clusters;
end
```

For an unstructured statement S (other constructs are simply traversed; parallel statements are reconstructed using the original statements), its control vertices are traversed along a topological sort-ordered descent. As an unstructured statement, by definition, is also a graph (control statements are the vertices, while predecessors (resp. successors) control statements S_n of a control statement S are edges from S_n to S (resp. from S to S_n)), we use the same idea as in Section 6.5 `topsort`, but this time, from an unstructured statement. We use thus Function `topsort_unstructured(`S`)` that yields a list of stages of computation, each cluster stage being a list of

independent lists L of statements s.

Cluster stages are seen as a set of fork-join structures; a fork-join structure can be seen as a set of spawn statements (to be launched in parallel) enclosed in a barrier statement. It can be seen also as a parallel `sequence`. In our implementation of structured code generation, we choose to generate `spawn` and `barrier` constructs instead of parallel `sequence` constructs because, in contrast to the parallel `sequence` construct, the `spawn` construct allows the implementation of recursive parallel tasks. Besides, we can represent a parallel loop via the `spawn` construct but it cannot be implemented using parallel `sequence`.

New statements have to be added to `domain`(σ). Therefore, in order to set the third argument of σ (`nbclusters`(σ)) of these new statements, we use the function `structured_clusters` that harvests the clusters used to schedule the list of statements in a sequence. Recall that Function `statements` of a sequence returns its list of statements.

We use Function `spawns_barrier` detailed in Algorithm 22 to form spawn and barrier statements. In a cluster stage, each list L for each cluster κ_s corresponds to a spawn statement S_{spawn}. In order to affect a cluster number as an entity to the spawn statement as required by SPIRE, we associate a cluster number to the cluster κ_s returned by `cluster`($\sigma(s)$) via the `cluster_number` function; we posit thus that two clusters are equal if they have the same cluster number. However, since BDSC maps tasks to logical clusters in Algorithm 19, with numbers in $[0, p-1]$, we convert this logical number to a physical one using the formula $n_{physical} = P - p +$`cluster_number`(κ_s). P is the total number of clusters available in the target machine.

Spawn statements S_L of one cluster stage are collected in a barrier statement $S_{barrier}$ as follows. First, the list of statements S_L is constructed using the `||` symbol for the concatenation of two lists of arguments. Then, Statement $S_{barrier}$ is created using Function `sequence` from the list of statements S_L. Synchronization attributes should be set for the new statements S_{spawn} and $S_{barrier}$. We use thus the functions `spawn(`I`)` and `barrier()` to set the synchronization attribute `spawn` or `barrier` of different statements.

Figure 7.3 depicts an example of a SPIRE structured representation (in the right hand side) of a part of the C implementation of the main function of Harris presented in the left hand side. Note that the scheduling is resulting from our BDSC-based task parallelization process using $P = 3$ (we use three clusters because the maximum parallelism in Harris is three). Here, up to three parallel tasks are enclosed in a barrier for each chain of functions: `Sobel`, `Multiplication`, and `Gauss`. The synchronization annotations `spawn` and `barrier` are printed within the SPIRE generated code; remember that the synchronization attribute is added to each statement (see Section 4.2.3 on page 69).

Multiple parallel code configuration schemes can be generated. In a first

ALGORITHM 22: Spawns-barrier generation for a cluster stage

```
function spawns_barrier(cluster_stage, κ, p, σ)
  p′ = p - |cluster_stage|;
  S_L = [];
  foreach L ∈ cluster_stage
    L′ = [];
    nbclusters_L = 0;
    foreach s ∈ L
      κ_s = cluster(σ(s)); //all s have the same schedule κ_s
      if(p′ > 0)
        σ = unstructured_to_structured(s, p′, σ);
      L′ ||= [parallel(σ(s))];
      nbclusters_L = max(nbclusters_L, nbclusters(σ(s)));
    S_spawn = sequence(L′);
    n_physical = P - p + cluster_number(κ_s);
    cluster_number(κ_s) = n_physical;
    synchronization(S_spawn) = spawn(n_physical);
    σ = σ[S_spawn →(S_spawn, κ_s, |structured_clusters(L′, σ)|)];
    S_L ||= [S_spawn];
    p′ -= nbclusters_L;
  S_barrier = sequence(S_L);
  synchronization(S_barrier) = barrier();
  return (S_barrier,σ);
end
```

scheme, illustrated in Function `spawns_barrier`, we activate other clusters from an enclosing one, i.e, the enclosing cluster executes the spawn instructions (launch the tasks). A second scheme would be to use the enclosing cluster to execute one of the enclosed tasks; there is no need to generate the last task as a spawn statement, since it is executed by the current cluster. We show an example in Figure 7.4.

Although barrier synchronization of statements with only one spawn statement can be omitted as another optimization: `barrier`(`spawn`(i, S)) $=$ S, we choose to keep these synchronizations for pedagogical purposes and plan to apply this optimization in a separate phase as a parallel transformation or handle it at the code generation level.

7.2.2 Hierarchical Parallelism

Function `unstructured_to_structured` is recursive in order to handle hierarchy in code. An example is provided in Figure 7.5, where we assume that $P = 4$. We use two clusters κ_0 and κ_1 to schedule the cluster stage $\{S_1, S_2\}$; the remaining clusters are then used to schedule statements enclosed in S_1 and S_2. Note that Statement S_{12}, where $p = 2$, is mapped to the physical

```
in = InitHarris();
//Sobel
SobelX(Gx, in);
SobelY(Gy, in);
//Multiply
MultiplY(Ixx, Gx, Gx);
MultiplY(Iyy, Gy, Gy);
MultiplY(Ixy, Gx, Gy);
//Gauss
Gauss(Sxx, Ixx);
Gauss(Syy, Iyy);
Gauss(Sxy, Ixy);
//Coarsity
CoarsitY(out,
         Sxx, Syy, Sxy);
```

```
barrier(
  spawn(0, InitHarris(in))
);
barrier(
  spawn(0, SobelX(Gx, in));
  spawn(1, SobelY(Gy, in))
);
barrier(
  spawn(0, MultiplY(Ixx, Gx, Gx));
  spawn(1, MultiplY(Iyy, Gy, Gy));
  spawn(2, MultiplY(Ixy, Gx, Gy))
);
barrier(
  spawn(0, Gauss(Sxx, Ixx));
  spawn(1, Gauss(Syy, Iyy));
  spawn(2, Gauss(Sxy, Ixy))
);
barrier(
  spawn(0, CoarsitY(out,
                    Sxx, Syy, Sxy))
)
```

Figure 7.3: A part of Harris, and one possible SPIRE core language representation

```
...
spawn(0, Gauss(Sxx, Ixx));
spawn(1, Gauss(Syy, Iyy));
spawn(2, Gauss(Sxy, Ixy))
...
```

```
...
spawn(1, Gauss(Sxx, Ixx));
spawn(2, Gauss(Syy, Iyy));
Gauss(Sxy, Ixy)
...
```

Figure 7.4: SPIRE representation of a part of Harris, non optimized (left) and optimized (right)

cluster number 3 for its logical number 3 ($3 = 4 - 2 + 1$).

Barrier statements $S_{structured}$ formed from all cluster stages $S_{barrier}$ in the sequence S constitute the new sequence S' that we build in Function `unstructured_to_structured`; by default, the `execution` attribute of this sequence is sequential.

The code in the right hand side of Figure 7.5 contains barrier statements with only one spawn statement that can be omitted (as an optimization). Moreover, in the same figure, note how S_{01} and S_{02} are apparently reversed; this is a consequence, in our algorithm 21, of the function `topsort_unstructured`, where statements are generated in the order of the graph traversal.

```
//S
{
 //S0
 for(i = 1; i <= N; i++){
   A[i] = 5; //S01
   B[i] = 3; //S02
 }
 //S1
 for(i = 1; i <= N; i++){
   A[i] = foo(A[i]); //S11
   C[i] = bar();     //S12
 }
 //S2
 for(i = 1; i <= N; i++)
   B[i] = baz(B[i]); //S21
 //S3
 for(i = 1; i <= N; i++)
   C[i] += A[i]+ B[i];//S31
}
```

```
barrier(
  spawn(0, forloop(i, 1, N, 1,
    barrier(
      spawn(1, B[i] = 3);
      spawn(2, A[i] = 5)
    ),
    sequential))
);
barrier(
  spawn(0, forloop(i, 1, N, 1,
    barrier(
      spawn(2, A[i] = foo(A[i]));
      spawn(3, C[i] = bar())
    ),
    sequential));
  spawn(1,
    forloop(i, 1, N, 1,
      B[i] = baz(B[i]),
      sequential))
);
barrier(
  spawn(0, forloop(i, 1, N, 1,
    C[i] += A[i]+B[i],
    sequential))
)
```

Figure 7.5: A simple C code (hierarchical), and its SPIRE representation

The `execution` attribute of the loops in Figure 7.5 is `sequential`, but they are actually parallel. Note that, regarding these loops, since we adopt the task parallelism paradigm, it may be useful to apply the tiling transformation and then perform full unrolling of the outer loop. We give more details of this transformation in the protocol of our experiments in Section 8.4. This way, the input code contains more parallel tasks, `spawn` loops with `sequential` execution attribute, resulting from the parallel loops.

7.3 From Shared Memory to Distributed Memory Transformation

This section provides a second illustration of the type of parallel transformations that can be performed thanks to SPIRE: from shared memory to distributed memory transformation. Given the complexity of this issue, as illustrated by the vast existing related work (see Section 7.3.1), the work reported in this section is preliminary and exploratory.

The `send` and `recv` primitives of SPIRE provide a simple communication

API adaptable for message-passing libraries/languages such as MPI and thus distributed memory system architectures. In these architectures, the data domain of whole programs should be partitioned into disjoint parts and mapped to different processes that have separate memory spaces. If a process accesses a non-local variable, communications must be performed. In our case, `send` and `recv` SPIRE functions have to be added to the parallel intermediate representation of distributed memory code.

7.3.1 Related Work: Communications Generation

Many researchers have been generating communication routines between several processors (general purpose, graphical, accelerator,...) and have been trying to optimize the generated code. A large amount of literature already exists and large amount of work has been done to address this problem since the early 1990s. However, no practical and efficient general solution currently exists [27].

Amarasinghe and Lam [15] generate send and receive instructions necessary for communication between multiple processors. Goumas et al. [50] generate automatically message-passing code for tiled iteration spaces. However, these two works handle only perfectly nested loops with uniform dependences. Moreover, the data parallelism code generation proposed in [15] and [50] produces an SPMD (single program multiple data) program to be run on each processor; this is different from our context of task parallelism with its hierarchical approach.

Bondhugula [27], in their automatic distributed memory code generation technique, use the polyhedral framework to compute communication sets between computations under a given iteration of the parallel dimension of a loop. Once more, the parallel model is nested loops. Indeed, he “only” needs to transfer written data from one iteration to another of read mode of the same loop, or final writes for all data, to be aggregated once the entire loop is finished. In our case, we need to transfer data from one or many iterations or many of a loop to different loop(s), knowing that data spaces of these loops may be interlaced.

Cetus contains a source-to-source OpenMP to MPI transformation [23], where at the end of a parallel construct, each participating process communicates the shared data it has produced and that other processes may use.

STEP [80] is a tool that transforms a parallel program containing high-level directives of OpenMP into a message-passing program. To generate MPI communications, STEP constructs (1) send updated array regions needed at the end of a parallel section, and (2) receive array regions needed at the beginning of a parallel section. Habel et al.[53] present a directive-based programming model for hybrid distributed and shared memory systems. Using pragmas `distribute` and `gridify`, they distribute nested loop

dimensions and array data among processors.

The OpenMP to GPGPU compiler framework [76] transforms OpenMP to GPU code. In order to generate communications, it inserts memory transfer calls for all shared data accessed by the kernel functions and defined by the host, and data written in the kernels and used by the host.

Amini *et al* [17] generate memory transfers between a computer host and its hardware GPU (CPU-GPU) using data flow analysis; this latter is also based on array regions.

The main difference with our context of task parallelism with hierarchy is that these works generate communications between different iterations of nested loops or between one host and GPUs. Indeed, we have to transfer data between two different entire loops or two iterations of different loops.

In this section, we describe how to figure out which pieces of data have to be transferred. We construct array region transfers in the context of a distributed-memory homogeneous architecture.

7.3.2 Difficulties

Distributed-memory code generation becomes a problem much harder than the ones addressed in the above related works (see Section 7.3.1) because of the hierarchical aspect of our methodology. In this section, we illustrate how communication generation impose a set of challenges in terms of efficiency and correctness of the generated code and the coherence of data communications.

Inter-Level Communications

In the example in Figure 7.6, we suppose given this schedule which, although not efficient, is useful to illustrate the difficulty of sending an array, element by element, from the source to the sink dependence extracted from the DDG. The difficulty occurs when these source and sink belong to different levels of hierarchy in the code i.e, the loop tree or test branches.

Communicating element by element is simple in the case of array B in S_{12} since no loop-carried dependence exists. But, when a dependence distance is greater than zero, such as is the case of Array A in S_{11}, sending only $A[0]$ to S_{11} is required, although generating exact communications in such cases requires more sophisticated static analyses that we do not have. Note that `asend` is a macro instruction for an asynchronous send; it is detailed in Section 7.3.3.

Non-Uniform Dependence Distance

In previous works (see Section 7.3.1), only uniform (constant) loop-carried dependences were considered. When a dependence is non-uniform, i.e, the

```
//S
{
 //S0
 for(i = 0; i < N; i++){
   A[i] = 5; //S01
   B[i] = 3; //S02
 }
 //S1
 for(i = 1; i < N; i++){
   A[i] = foo(A[i-1]);//S11
   B[i] = bar(B[i]);  //S12
 }
}
```

```
forloop(i, 0, N-1, 1,
  barrier(
    spawn(0, A[i] = 5;
             if(i==0,
               asend(1, A[i]),
               nop));
    spawn(1, B[i] = 3,
             asend(0, B[i])
), sequential);
forloop(i, 1, N-1, 1,
  barrier(
    spawn(1, if(i==1,
               recv(0, A[i-1]),
               nop);
             A[i] = foo(A[i-1]));
    spawn(0, recv(1, B[i])),
             B[i] = bar(B[i]))
  )
), sequential)
```

Figure 7.6: An example of a shared memory C code and its SPIRE code efficient communication primitive-based distributed memory equivalent

dependence distance is not constant as in the two codes in Figure 7.7, knowing how long dependences are for Array A and generating exact communications in such cases is complex. The problem is even more complicated when nested loops are used and arrays are multidimensional. Generating exact communications in such cases requires more sophisticated static analyses that we do not have.

```
//S
{
  //S0
  for(i = 0; i < N; i++){
    A[i] = 5; //S01
    B[i] = 3; //S02
  }
  //S1
  for(i = 0; i < N; i++){
    A[i] = A[i-p] + a; //S11
    B[i] = A[i] * B[i];//S12
    p = B[i];
  }
}
```

```
//S
{
  //S0
  for(i = 0; i < N; i++){
    A[i] = 5; //S01
    B[i] = 3; //S02
  }
  //S1
  for(i = 0; i < N; i++){
    A[i] = A[p*i-q] + a;//S11
    B[i] = A[i] * B[i]; //S12
  }
}
```

Figure 7.7: Two examples of C code with non-uniform dependences

Exact Analyses

We presented in Section 6.3 our cost models: we rely on array region analysis to overapproximate communications. This analysis is not always exact. PIPS generates approximations (see Section 2.6.3 on page 34), noted by the *MAY* attribute, if an over- or under-approximation has been used because the region cannot be exactly represented by a convex polyhedron. While BDSC scheduling is still robust in spite of these approximations (see Section 8.6), they can affect the correctness of the generated communications. For instance, in the code presented in Figure 7.8, the quality of array regions analysis may pass from *EXACT* to *MAY* if a region contains holes, as is the case of S_0. Also, while the region of A in the case of S_1 remains *EXACT* because the exact region is rectangular, the region of A in the case of S_2 is *MAY* since it contains holes.

```
//S0
                  // < A(φ1) − R − MAY − {1 ≤ φ1, φ1 ≤ N} >
                  // < A(φ1) − W − MAY − {1 ≤ φ1, φ1 ≤ N} >
for(i = 1 ; i <= N ; i++)
  if(i != N/2)
    A[i] = foo(A[i]);
//S1
                  // < A(φ1) − R − EXACT − {2 ≤ φ1, φ1 ≤ 2 × N} >
                  // < B(φ1) − W − EXACT − {1 ≤ φ1, φ1 ≤ N} >
for(i = 1 ; i <= N ; i++)
                  // < A(φ1) − R − EXACT − {i + 1 ≤ φ1, φ1 ≤ i + N} >
                  // < B(φ1) − W − EXACT − {φ1 = i} >
  for(j = 1 ; j <= N ; j++)
    B[i] = bar(A[i+j]);
//S2
                  // < A(φ1) − R − MAY − {2 ≤ φ1, φ1 ≤ 2 × N} >
                  // < B(φ1) − W − EXACT − {1 ≤ φ1, φ1 ≤ N} >
for(j = 1 ; j <= N ; j++)
  B[j] = baz(A[2*j]);
```

Figure 7.8: An example of a C code and its read and write array regions analysis for two communications from S_0 to S_1 and to S_2

Approximations are also propagated when computing in and out regions for Array A (see Figure 7.9). In this chapter, communications are computed using in and out regions such as communicating A from S_0 to S_1 and to S_2 in Figure 7.9.

7.3.3 Equilevel and Hierarchical Communications

As transferring data from one task to another when they are not at the same level of hierarchy is unstructured and too complex, designing a compilation

```
//S0
                // < A[φ1] − IN − MAY − {1 ≤ φ1, φ1 ≤ N} >
                // < A[φ1] − OUT − MAY − {2 ≤ φ1, φ1 ≤ 2N, φ1 ≤ N} >
for(i = 1; i <= N; i++)
  if (i!=n/2)
    A[i] = foo(A[i]);
//S1
                // < A[φ1] − IN − EXACT − {2 ≤ φ1, φ1 ≤ 2N} >
for(i = 1; i <= n; i += 1)
                // < A[φ1] − IN − EXACT − {i + 1 ≤ φ1, φ1 ≤ i + N} >
  for(j = 1; j <= n; j += 1)
    B[i] = bar(A[i+j]);
//S2
                // < A[φ1] − IN − MAY − {2 ≤ φ1, φ1 ≤ 2N} >
for(j = 1; j <= n; j += 1)
  B[j] = baz(A[j*2]);
```

Figure 7.9: An example of a C code and its in and out array regions analysis for two communications from S_0 to S_1 and to S_2

scheme for such cases might be overly complex and even possibly jeopardize the correctness of the distributed memory code. In this thesis, we propose a compositional static solution that is structured and correct but which may communicate more data than necessary but their coherence is guaranteed.

Communication Scheme

We adopt a hierarchical scheme of communication between multiple levels of nesting of tasks and inside one level of hierarchy. An example of this scheme is illustrated in Figure 7.10, where each box represents a task τ. Each box is seen as a host of its enclosing boxes, and an edge represents a data dependence. This scheme illustrates an example of configuration with two nested levels of hierarchy. The level is specified with the superscript on τ in the figure.

Once structured parallelism is generated (sequential sequences of `barrier` and `spawn` statements)[4], the second step is to add `send` and `recv` SPIRE call statements when assuming a message-passing language model. We detail in this section our specification of communications insertion in SPIRE.

We divide parallelism into hierarchical and equilevel parallelisms and distinguish between two types of communications, namely, inside a sequence, zero hierarchy level (equilevel), and between hierarchy levels (hierarchical). In Figure 7.11, where edges represent possible communications, $\tau_0^{(1)}, \tau_1^{(1)}$

[4]Note that communications can also be constructed from an unstructured code; one has to manipulate unstructured statements rather than sequence ones.

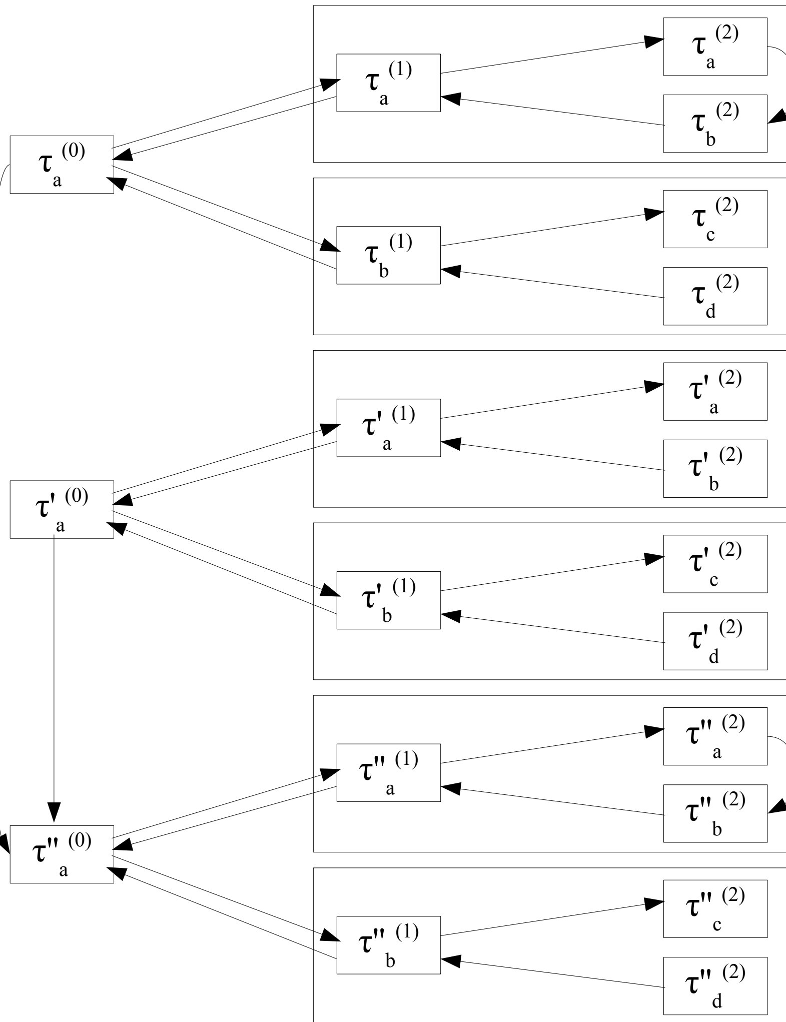


Figure 7.10: Scheme of communications between multiple possible levels of hierarchy in a parallel code; the levels are specified with the superscripts on τ's

and $\tau_2^{(1)}$ are enclosed in the vertex $\tau_1^{(0)}$ that is in the same sequence as $\tau_0^{(0)}$. The dependence $(\tau_0^{(0)}, \tau_1^{(0)})$ for instance denotes an equilevel communication of A; also, the dependence $(\tau_0^{(1)}, \tau_2^{(1)})$ denotes an equilevel communication of B, while $(\tau_1^{(0)}, \tau_0^{(1)})$ is a hierarchical communication of A. We use two main steps to construct communications via `equilevel_com` and `hierarchical_com` functions that are presented below.

The key limitation of our method is that it cannot enforce memory constraints when applying the BDSC-based hierarchical scheduling, which uses an iterative approach on the number of applications of BDSC (see Sec-

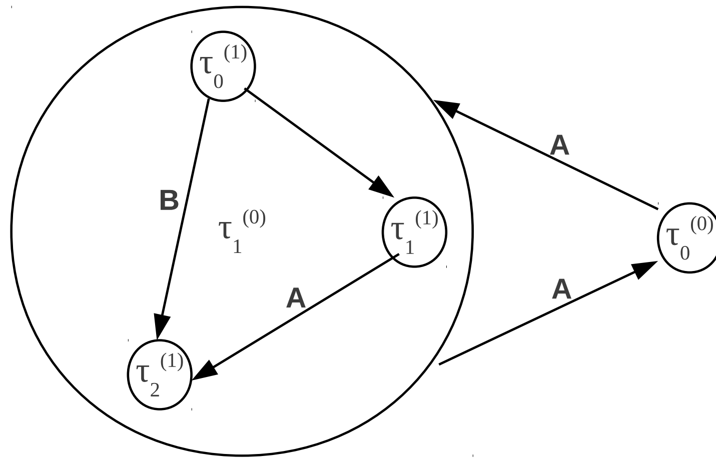


Figure 7.11: Equilevel and hierarchical communications

tion 6.5). Therefore, our proposed solution for the generation of communications cannot be applied when strict limitations on memory are imposed at scheduling time. However, our code generator can easily verify if the data used and written at each level can be maintained in the specific clusters.

As a summary, compared to the works presented in Section 7.3.1, we do not improve upon them (regarding non-affine dependences or efficiency). However, we extend their ideas to our problem of generating communication in task parallelism within a hierarchical code structure, which is more complex than theirs.

Assumptions

For our communication scheme to be valid, the following assumptions must be met for proper communications generation:

- all write regions are exact in order to guarantee the coherence of data communications;
- non-uniform dependences are not handled;
- more data than necessary are communicated but their coherence is guaranteed thanks to the topological sorting of tasks involved by communications. This ordering is also necessary to avoid deadlocks;
- the order in which the send and receive operations are performed is matched in order to avoid deadlock situations;
- HBDSC algorithm is applied without the memory constraint ($M = \infty$);
- the input shared-memory code is a structured parallel code with `spawn` and `barrier` constructs;

- all send operations are non-blocking in order to avoid deadlocks. We presented in SPIRE (see Section 4.2.4 on page 74 how non-blocking communications can be implemented in SPIRE using the blocking primitives within `spawn`ed statements). Therefore, we assume that we use an additional process P_s for performing non-blocking send communications and a macro instruction `asend` for asynchronous send as follows:

```
asend(dest, data) = spawn(Ps, send(dest, data))
```

7.3.4 Equilevel Communications Insertion Algorithm

Function `equilevel_com` presented in Algorithm 23 computes first the equilevel data (array regions) to communicate from (when neighbors = predecessors) or to (when neighbors = successors) S_i inside an SDG G (a code sequence) using the function `equilevel_regions` presented in Algorithm 24. After the call to the `equilevel_regions` function, `equilevel_com` creates a new sequence statement S_i' via the `sequence` function. S_i' contains the receive communication calls, S_i and the asend communication calls. After that, it updates σ with S_i' and returns it. Recall that the `||` symbol is used for the concatenation of two lists.

ALGORITHM 23: Equilevel communications for Task τ_i inside SDG G, updating a schedule σ

```
function equilevel_com(τi, G, σ, neighbors)
  Si = vertex_statement(τi);
  coms = equilevel_regions(Si, G, σ, neighbors);
  L = (neighbors = successors) ? coms : ∅;
  L ||= [Si];
  L ||= (neighbors = predecessors) ? coms : ∅;
  S'i = sequence(L);
  return σ[Si → (S'i, cluster(σ(Si)), nbclusters(σ(Si)))];
end
```

Function `equilevel_regions` traverses all neighbors τ_n (successors or predecessors) of τ_i (task of S_i) in order to generate `asend` or `recv` statements calls for τ_i if they are not in the same cluster. Neighbors are traversed throughout a topological sort-ordered descent in order to impose an order between multiple send or receive communication calls between each pair of clusters.

We thus guarantee the coherence of communicated data and avoid communication deadlocks. We use the function `topsort_ordered` that computes first the topological sort of the transitive closure of G and then accesses the neighbors of τ_i in such an order. For instance, the graph in the right of

ALGORITHM 24: Array regions for Statement S_i to be transferred inside SDG G (equilevel)

```
function equilevel_regions(Sᵢ, G, σ, neighbors)
  κᵢ = cluster(σ(Sᵢ));
  τᵢ = statement_vertex(Sᵢ, G);
  coms = ∅;
  foreach τₙ ∈ topsort_ordered(G, neighbors(τᵢ, G))
    Sₙ = vertex_statement(τₙ);
    κₙ = cluster(σ(Sₙ));
    if(κᵢ ≠ κₙ)
      R = (neighbors = successors) ? transfer_data(τᵢ, τₙ) :
                                       transfer_data(τₙ, τᵢ);
      coms ||= com_calls(neighbors, R, κₙ);
  return coms;
end
```

Figure 7.12 shows the result of the topological sort of the predecessors of τ_4 of the graph in the left. Function `equilevel_regions` returns the list of communications performed inside G by S_i.

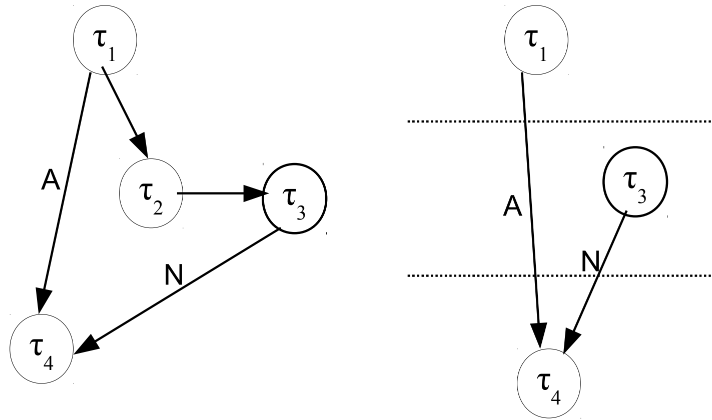


Figure 7.12: A part of an SDG and the topological sort order of predecessors of τ_4

Communication calls are generated in two steps (the two functions presented in Algorithm 25):

First, the `transfer_data` function is used to compute which data elements have to be communicated in form of array regions. It returns the elements that are involved in RAW dependence between two tasks τ and τ_{succ}. It filters the out and in regions of the two tasks in order to get these dependences; it uses `edge_regions` for this filtering. `edge_regions` are computed during the construction of the SDG in order to maintain the

ALGORITHM 25: Two functions to construct a communication call: data to transfer and call generation

```
function transfer_data(τ, τ_succ)
  R'_o = R'_i = ∅;
  R_o = out_regions(vertex_statement(τ));
  foreach r_o ∈ R_o
    if(∃ r ∈ edge_regions(τ, τ_succ) / entity(r) = entity(r_o))
      R'_o ∪= r_o;
  R_i = in_regions(vertex_statement(τ_succ));
  foreach r_i ∈ R_i
    if(∃ r ∈ edge_regions(τ, τ_succ) / entity(r) = entity(r_i))
      R'_i ∪= r_i;
  return regions_intersection(R'_o, R'_i);
end

function com_calls(neighbors, R, κ)
  coms = ∅
  i = cluster_number(κ);
  com = (neighbors = successors) ? "asend" : "recv";
  foreach r ∈ R
    coms ||= region_to_coms(r, com, i);
  return coms;
end
```

dependences that label the edges of the DDG (see Section 6.2.1 on page 113). The intersection between the two results is returned[5]. For instance, in the graph on the right of Figure 7.12, if N is written also in τ_1, filtering out regions of τ_1 avoids to receive N twice (N comes from τ_1 and τ_3).

Second, communication calls are constructed via the function `com_calls` that, from the cluster κ of the statement source (`asend`) or sink (`recv`) of the communication and the set of data in term of regions R, generates a list of communication calls *coms*. These communication calls are the result of the function `region_to_coms` already implemented in PIPS [20]. This function converts a region to a code of send/receive instructions that represent the communications of the integer points contained in a given parameterized polyhedron, from this region. An example is illustrated in Figure 7.13 where the write regions are exact and the other required assumptions are verified. In (b) of the figure, the intersection between in regions of S_2 and out regions of S_1 is converted to a loop of communication calls using `region_to_coms` function. Note that the current version of our implementation prototype of `region_to_coms` is in fact unable to manage this somewhat complicated case. Indeed, the task running on Cluster 0 needs to know the value of the

[5] We assume that R_r is combined with the path transformer between the statements of τ and τ_{succ}.

structure variable M to generate a proper communication with the task on Cluster 1 (see Figure 7.13 (b)). M would need to be sent by Cluster 1, where it is known, before the sending loop is run. Thus, our current implementation assumes that all variables in the code generated by `region_to_coms` are known in each communicating cluster. For instance, in Figure 7.13 with M replaced by N, `region_to_coms` will perform correctly. This constraint was not a limitation in the experimental benchmarks tested in Chapter 8.

```
//S1
                 // < A(φ1) − W − EXACT − {1 ≤ φ1, φ1 ≤ N} >
                 // < A[φ1] − OUT − MAY − {2 ≤ φ1, φ1 ≤ 2 × M} >
for(i = 1; i <= N; i++)
  A[i] = foo();
//S2
                 // < A[φ1] − R − MAY − {2 ≤ φ1, φ1 ≤ 2 × M} >
                 // < A[φ1] − IN − MAY − {2 ≤ φ1, φ1 ≤ 2 × M} >
for(i = 1; i <= M; i++)
  B[i] = bar(A[i*2]);
```

(a) •

```
spawn(0,
  for(i = 1; i <= N; i++)
    A[i] = foo();
  for(i = 2; i <= 2*M; i++)
    asend(1, A[i]);
)
spawn(1,
  for(i = 2; i <= 2*M; i++)
    recv(0, A[i]);
  for(i = 1; i <= M; i++)
    B[i] = bar(A[i*2]);
)
```

(b) •

Figure 7.13: Communications generation from a region using `region_to_coms` function

Also, in the part of the `main` function presented in the left of Figure 7.3, the statement `MultiplY(`Ixy`, `Gx`, `Gy`);` scheduled on Cluster number 2 needs to receive both Gx and Gy from Clusters number 1 and 0 (see Figure 6.16). The receive calls list thus generated for this statement using our algorithm is[6]:

```
recv(1, Gy);
```

[6]Since the whole array is received, the region is converted to a simple instruction, no loop is needed.

```
recv(0, Gx);
```

7.3.5 Hierarchical Communications Insertion Algorithm

Data transfers are also needed between different levels of the code hierarchy because each enclosed task requires data from its enclosing one and vice versa. To construct these hierarchical communications, we introduce Function `hierarchical_com`, presented in Algorithm 26.

ALGORITHM 26: Hierarchical communications for Statement S to its hierarchical parent clustered in κ_p

```
function hierarchical_com(S, neighbors, σ, κp)
  Rh = (neighbors = successors) ? out_regions(S) :
                                  in_regions(S);
  coms = com_calls(neighbors, Rh, κp);
  seq = (neighbors = predecessors) ? coms : ∅
       || [S]
       || (neighbors = successors) ? coms : ∅;
  S′ = sequence(seq);
  σ = σ[S → (S′, cluster(σ(S)), nbclusters(σ(S)))];
  return (σ, Rh);
end
```

This function generates `asend` or `recv` statements in calls for each task S and its hierarchical parent (enclosing vertex) scheduled in Cluster κ_p. First, it constructs the list of transfers R_h that gathers the data to be communicated to the enclosed vertex; R_h can be defined as `in_regions` or `out_regions` of a task S. Then, from this set of transfers R_h, communication calls *coms* (`recv` from the enclosing vertex or `asend` to it) are generated. Finally, σ, updated with the new statement S, and the set of regions R_h that represents the data to be communicated this time from the side of the enclosing vertex, are returned.

For instance, in Figure 7.11, for the task $\tau_0^{(1)}$, the data elements R_h to send from $\tau_0^{(1)}$ to $\tau_1^{(0)}$ are the `out_regions` of the statement labeled $\tau_0^{(1)}$. Thus, the hierarchical `asend` calls from $\tau_0^{(1)}$ to its hierarchical parent $\tau_1^{(1)}$ is constructed using, as argument, R_h.

7.3.6 Communications Insertion Main Algorithm

Function `communications_insertion` presented in Algorithm 27 constructs a new code (`asend` and `recv` primitives are generated). It uses a hierarchical schedule σ, that maps each substatement s of S to $\sigma(s) = (s', \kappa, n)$ (see Section 6.5), to handle new statements. It uses also the function hierarchical

SDG mapping H that we use to recover the SDG of each statement (see Section 6.2).

ALGORITHM 27: Communication primitives `asend` and `recv` insertion in SPIRE

```
function communications_insertion(S, H, σ, κ_p)
  κ = cluster(σ(S));
  n = nbclusters(σ(S));
  switch (S)
    case call:
      return σ;
    case sequence(S_1;...;S_n):
      G = H(S);
      h_recvs = h_sends = ∅;
      foreach τ_i ∈ topsort(G)
        S_i = vertex_statement(τ_i);
        κ_i = cluster(σ(S_i));
        σ = equilevel_com(τ_i, G, σ, predecessors);
        σ = equilevel_com(τ_i, G, σ, successors);
        if(κ_p ≠ NULL ∧ κ_p ≠ κ_i)
          (σ, h_R_recv) =
                hierarchical_com(S_i, predecessors, σ, κ_p);
          h_sends ||= com_calls(successors, h_R_recv, κ_i);
          (σ, h_R_send) =
                hierarchical_com(S_i, successors, σ, κ_p);
          h_recvs ||= com_calls(predecessors, h_R_send, κ_i);
        σ = communications_insertion(S_i, H, σ, κ_i);
      S' = parallel(σ(S));
      S'' = sequence(h_sends || [S'] || h_recvs);
      return σ[S' → (S'', κ, n)];
    case forloop(I, E_lower, E_upper, S_body):
      σ = communications_insertion(S_body, H, σ, κ_p);
      (S'_body, κ_body, nbclusters_body) = σ(S_body);
      return σ[S → (forloop(I, E_lower, E_upper, S'_body), κ, n)];
    case test(E_cond, S_t, S_f):
      σ = communications_insertion(S_t, H, σ, κ_p);
      σ = communications_insertion(S_f, H, σ, κ_p);
      (S'_t, κ_t, nbclusters_t) = σ(S_t);
      (S'_f, κ_f, nbclusters_f) = σ(S_f);
      return σ[S → (test(E_cond, S'_t, S'_f), κ, n)];
end
```

Equilevel

To construct equilevel communications, all substatements S_i in a sequence S are traversed. For each S_i, communications are generated inside each

sequence via two calls to the function `equilevel_com` presented above, one with the function `neighbors` equal to `successors` to compute `asend` calls, the other with the function `predecessors` to compute `recv` calls. This ensures that each send operation is matched to an appropriately-ordered receive operation, avoiding deadlock situations. We use the function H that returns an SDG G for S, while G is essential to access dependences and thus communications between the vertices τ of G, whose statements $s = \texttt{vertex_statement}(\tau)$ are in S.

Hierarchical

We construct also hierarchical communications that represent data transfers between different levels of hierarchy, because each enclosed task requires data from its enclosing one. We use the function `hierarchical_com`, presented above, that generates `asend` or `recv` statements in calls for each task S and its hierarchical parent (enclosing vertex) scheduled in Cluster κ_p. Thus, the argument κ_p is propagated in `communications_insertion` in order to keep correct the information of the cluster of the enclosing vertex. The returned set of regions, h_R_{send} or h_R_{recv}, that represents the data to be communicated at the start and end of the enclosed vertex τ_i, is used to construct, respectively, *send* and *recv* communication calls on the side of the enclosing vertex of S. Note that hierarchical send communications need to be performed before running S', to ensure that subtasks can run. Moreover, no deadlocks can occur, in this scheme, because the enclosing task executes send/receive statements sequentially before spawning its enclosed tasks, which run concurrently.

The top level of a program presents no hierarchy. In order not to generate hierarchical communications that are not necessary, as an optimization, we use the test ($\kappa_p \neq \texttt{NULL}$). Knowing that for the first call to the function `communications_insertion`, κ_p is set to `NULL`. This is necessary to prevent the generation of hierarchical communication for Level 0.

Example

As an example, we apply the phase of communication generation to the code presented in Figure 7.5 to generate SPIRE code with communications; both equilevel (noted with comments *equilevel*) and hierarchical (noted with comments *hierarchical*) communications are illustrated in the right of Figure 7.14. Moreover, a graphical illustration of these communications is given in Figure 7.15.

7.3.7 Conclusion and Future Work

As said before, the work presented in this section is a preliminary and exploratory solution that we believe is useful for suggesting the potentials of

```
for(i = 0;i < N;i++){
  A[i] = 5;
  B[i] = 3;
}
for(i = 0;i < N;i++){
  A[i] = foo(A[i]);
  C[i] = bar();
}
for(i = 0;i < N;i++)
  B[i] = baz(B[i]);
for(i = 0;i < N;i++)
  C[i] += A[i]+ B[i];
```

```
barrier(
  spawn(0, forloop(i, 1, N, 1,
    asend(1, i);          //hierarchical
    asend(2, i));         //hierarchical
    barrier(
      spawn(1,
        recv(0, i);       //hierarchical
        B[i] = 3;
        asend(0, B[i]));//hierarchical
      spawn(2,
        recv(0, i);       //hierarchical
        A[i] = 5;
        asend(0, A[i])); //hierarchical
    ); recv(1, B[i]);     //hierarchical
    recv(2, A[i])         //hierarchical
    , sequential);
    asend(1, B)));        //equilevel
barrier(
  spawn(0, forloop(i, 1, N, 1,
    asend(2, i);          //hierarchical
    asend(2, A[i]);       //hierarchical
    asend(3, i);          //hierarchical
    barrier(
      spawn(2,
        recv(0, i);       //hierarchical
        recv(0, A[i]);    //hierarchical
        A[i] = foo(A[i]);
        asend(0, A[i])); //hierarchical
      spawn(3,
        recv(0, i);       //hierarchical
        C[i] = bar();
        asend(0, C[i])); //hierarchical
    ); recv(2, A[i]);     //hierarchical
    recv(3, C[i])         //hierarchical
    , sequential));
  spawn(1, recv(0, B);    //equilevel
    forloop(i, 1, N, 1,
      B[i] = baz(B[i]), sequential);
    asend(0, B));         //equilevel
barrier(
  spawn(0, recv(1, B);    //equilevel
    forloop(i, 1, N, 1,
      C[i] += A[i]+B[i], sequential)))
```

Figure 7.14: Equilevel and hierarchical communications generation and SPIRE distributed memory representation of a C code

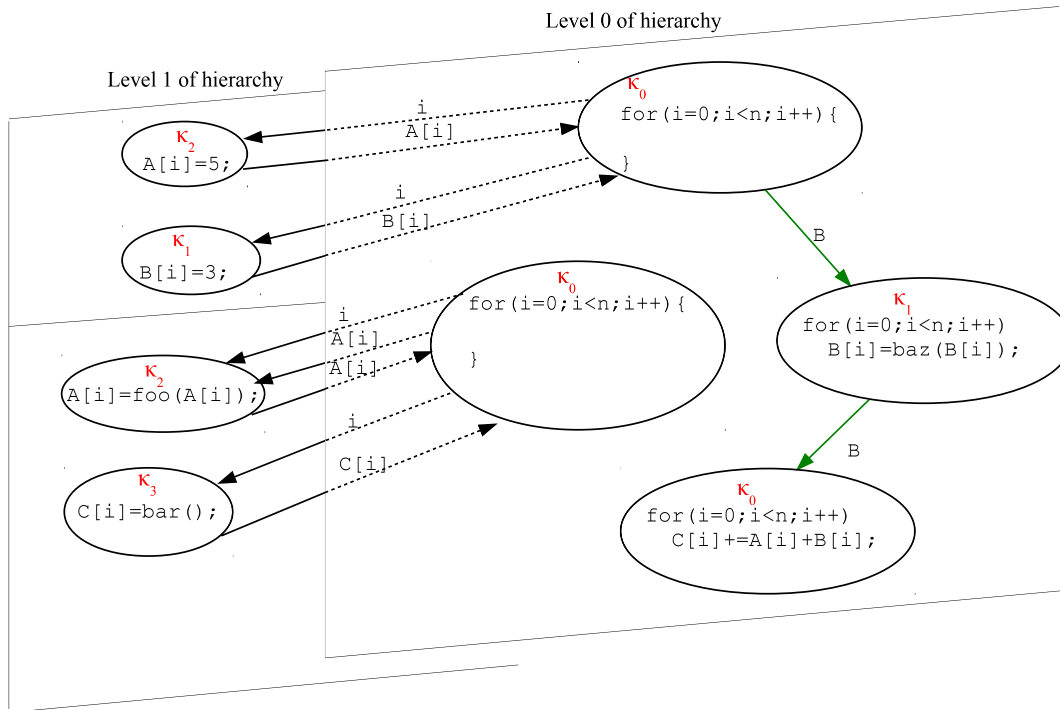


Figure 7.15: Graph illustration of communications made in the C code of Figure 7.14; equilevel in green arcs, hierarchical in black

our task parallelization in the distributed memory domain. It may also serve as a base for future work to generate more efficient code than ours.

In fact, the solution proposed in this section may communicate more data than necessary but their coherence is guaranteed thanks to the topological sorting of tasks involved by communications at each level of generation: equilevel (ordering of neighbors of every task in an SDG) and hierarchical (ordering of tasks of an SDG). Moreover, as a future work, we intend to optimize the generated code by eliminating redundant communications and aggregating small messages to larger ones.

For instance, hierarchical communications transfer all out regions from the enclosed task to its enclosing task. A more efficient scheme than this one would be to not send all out regions of the enclosed task but only regions that will not be modified afterward. This is another track of optimization to eliminate redundant communications that we leave for future work.

Moreover, we optimize the generated code by imposing that the current cluster executes the last nested spawn. The result of application of this optimization on the code presented in the right of Figure 7.14 is illustrated in the left of Figure 7.16. We also eliminate barriers with one spawn statement;

the result is illustrated in the right of Figure 7.16.

```
barrier(
  forloop(i, 1, N, 1,
    asend(1, i);
    barrier(
      spawn(1,
        recv(0, i);
        B[i] = 3;
        asend(0, B[i]));
      A[i] = 5
    );
    recv(1, B[i]);
    , sequential)
);
barrier(
  spawn(1, forloop(i,1,N,1,
    asend(2, i);
    asend(2, A[i]);
    barrier(
      spawn(2,
        recv(0, i);
        recv(0, A[i]);
        A[i] = foo(A[i]);
        asend(0, A[i]));
      C[i] = bar()
    );
    recv(2, A[i]);
    , sequential));
  forloop(i, 1, N, 1,
    B[i] = baz(B[i]),
    sequential)
);
barrier(
    forloop(i, 1, N, 1,
      C[i] += A[i]+B[i],
      sequential)
)
```

```
forloop(i, 1, N, 1,
  asend(1, i);
  barrier(
    spawn(1,
      recv(0, i);
      B[i] = 3;
      asend(0, B[i]));
    A[i] = 5
  );
  recv(1, B[i]);
  , sequential);
barrier(
  spawn(1, forloop(i,1,N,1,
    asend(2, i);
    asend(2, A[i]);
    barrier(
      spawn(2,
        recv(0, i);
        recv(0, A[i]);
        A[i] = foo(A[i]);
        asend(0, A[i]));
      C[i] = bar()
    );
    recv(2, A[i]);
    , sequential));
  forloop(i, 1, N, 1,
    B[i] = baz(B[i]),
    sequential)
);
forloop(i, 1, N, 1,
      C[i] += A[i]+B[i],
      sequential)
```

Figure 7.16: Equilevel and hierarchical communications generation after optimizations: the current cluster executes the last nested spawn (left) and barriers with one spawn statement are removed (right)

7.4 Parallel Code Generation

SPIRE is designed to fit different programming languages, thus facilitating the transformation of codes for different types of parallel systems. Existing parallel language constructs can be mapped into SPIRE. We show in this section how this intermediate representation simplifies the targeting of various languages via a simple mapping approach and use the prettyprinted generations of OpenMP and MPI as an illustration.

7.4.1 Mapping Approach

Our mapping technique to a parallel language is made of three steps, namely: (1) the generation of parallel tasks by translating `spawn` attributes and synchronization statements by translating `barrier` attributes to this language, (2) the insertion of communications by translating `asend` and `recv` primitives to this target language, if this latter uses a distributed memory model, and (3) the generation of hierarchical (nested) parallelism by exploiting the code hierarchy. For efficiency reasons, we assume that the creation of tasks is done once at the beginning of programs; after that, we just synchronize the tasks, which are thus persistent.

Task Parallelism

The parallel code is generated in a fork/join manner; a barrier statement encloses spawned parallel tasks. Mapping SPIRE to an actual parallel language consists in transforming the `barrier` and `spawn` annotations of SPIRE into the equivalent calls or pragmas available in the target language.

Data Distribution

If the target language uses a shared-memory paradigm, no additional changes are needed to handle this language. Otherwise, if it is a message-passing library/language, a pass of insertion of appropriate `asend` and `recv` primitives of SPIRE is necessary (see section 7.3). After that, we generate the equivalent communication calls of the corresponding language from SPIRE by transforming `asend` and `recv` primitives into these equivalent communication calls.

Hierarchical Parallelism

Along this thesis, we take into account the hierarchy structure of code to achieve maximum efficiency. Indeed, exploiting the hierarchy of code when scheduling the code via the `HBDSC` algorithm, within the pass of transformation of SPIRE via the `unstructured_to_structured` algorithm, and also when generating both equilevel and hierarchical communications, made it

possible to handle hierarchical parallelism in order to generate more parallelism. Moreover, we take into account the ability of parallel languages to implement hierarchical parallelism, and thus improve the performance of the generated parallel code. Figure 7.5 on page 157 illustrates zero and one level of hierarchy in the generated SPIRE code (right): we have generated equilevel and hierarchical SPIRE(PIPS IR).

7.4.2 OpenMP Generation

OpenMP 3.0 has been extended with tasks and thus supports both task and hierarchical task parallelism for shared memory systems. For efficiency, since the creation of threads is done once at the beginning of the program, `omp parallel` pragma is used once in the program. This section details the generation of OpenMP task parallelism features using SPIRE.

Task Parallelism

OpenMP supports two types of task parallelism: the dynamic, via `omp task` directive, and static, via `omp section` directive, scheduling models (see Section 3.4.4 on page 50). In our implementation of the OpenMP back-end generation, we choose to generate `omp task` instead of `omp section` tasks for four reasons, namely:

1. the HBDSC scheduling algorithm uses a static approach, and thus we want to combine the static results of HBDSC with the run-time dynamic scheduling of OpenMP provided when using `omp task`, in order to improve scheduling;

2. unlike the `omp section` directive, the `omp task` directive allows the implementation of recursive parallel tasks;

3. `omp section` may only be used in `omp sections` construct, and thus, we cannot implement a parallel loop using `omp sections`;

4. the actual implementations of OpenMP do not provide the possibility of nesting many `omp sections`, and thus, cannot take advantage of the BDSC-based hierarchical parallelization (HBDSC) feature to exploit more parallelism.

As regards the use of the OpenMP task construct, we apply a simple process: a SPIRE statement annotated with the synchronization attribute `spawn` is decorated by the pragma `omp task`. An example is illustrated in Figure 7.17, where the SPIRE code in the left hand side is rewritten in OpenMP on the right hand side.

Synchronization

In OpenMP, the user can specify explicitly synchronization to handle control and data dependences, using the `omp single` pragma for example. There, only one thread (`omp single` pragma) will encounter a task construct (`omp task`), to be scheduled and executed by any thread in the team (the set of threads created inside an `omp parallel` pragma). Therefore, a statement annotated with the SPIRE synchronization attribute `barrier` is decorated by the pragma `omp single`. An example is illustrated in Figure 7.17.

```
barrier(
  spawn(0, Gauss(Sxx, Ixx));
  spawn(1, Gauss(Syy, Iyy));
  spawn(2, Gauss(Sxy, Ixy))
)
```

```
#pragma omp single
{
#pragma omp task
  Gauss(Sxx, Ixx);
#pragma omp task
  Gauss(Syy, Iyy);
#pragma omp task
  Gauss(Sxy, Ixy);
}
```

Figure 7.17: SPIRE representation of a part of Harris, and its OpenMP task generated code

Hierarchical Parallelism

SPIRE handles multiple levels of parallelism providing a trade-off between parallelism and synchronization overhead. This is taken into account at scheduling time, using our BDSC-based hierarchical scheduling algorithm. In order to generate hierarchical parallelism in OpenMP, the same process as above of translating `spawn` and `barrier` SPIRE annotations should be applied. However, synchronizing nested tasks (with synchronization attribute `barrier`) using `omp single` pragma cannot be used because of a restriction of OpenMP-compliant implementations: single regions may not be closely nested inside explicit task regions. Therefore, we use the pragma `omp taskwait` for synchronization when encountering a statement annotated with a barrier synchronization. The second `barrier` in the left of Figure 7.18 implements nested parallel tasks; `omp taskwait` is then used (right of the figure) to synchronize the two tasks spawned inside the loop. Note also that the barrier statement with zero spawn statement (left of the figure) is omitted in the translation (right of the figure) for optimization purposes.

```
barrier(
  spawn(0,
    forloop(i, 1, N, 1,
    barrier(
      spawn(1, B[i] = 3);
      spawn(2, A[i] = 5)
    ),
    sequential))
);
...
barrier(
  spawn(0,
    forloop(i, 1, N, 1,
          C[i] = A[i]+B[i],
          sequential))
)
```

```
#pragma omp single
{
  for(i = 1; i <= N; i += 1) {
#pragma omp task
    B[i] = 3;
#pragma omp task
    A[i] = 5;
#pragma omp taskwait
  }
...
  for(i = 1; i <= N; i += 1)
    C[i] = A[i]+B[i];
}
```

Figure 7.18: SPIRE representation of a C code, and its OpenMP hierarchical task generated code

7.4.3 SPMDization: MPI Generation

MPI is a message-passing library; it is widely used to program distributed-memory systems. We show here the generation of communications between processes. In our technique, we adopt a model, often called "SPMDization", of parallelism where processes are created once at the beginning of the program (static creation), each process executes the same code and the number of processes remains unchanged during execution. This allows to reduce the process creation and synchronization overheads. We state in Section 3.4.5 on page 52 that MPI processes are created when the `MPI_Init` function is called; it defines also the universal intracommunicator *MPI_COMM_WORLD* for all processes to drive various communications. We use rank of process parameter ***rank0*** to manage the relation between process and task; this rank is obtained by calling the function `MPI_Comm_rank(MPI_COMM_WORLD, &rank0)`. This section details the generation of MPI using SPIRE.

Task Parallelism

A simple SPMDization technique is applied, where a statement with the synchronization attribute `spawn` of parameter *entity0* that specifies the cluster number (process in the case of MPI) is filtered by an `if` statement on the condition `rank0 == entity0`. An example is illustrated in Figure 7.19.

Communication and Synchronization

MPI supports distributed memory systems, we have to generate communications between different processes. `MPI_Isend` and `MPI_Recv` functions are called to replace communication primitives `asend` and `recv` of SPIRE. `MPI_Isend` [3] starts a nonblocking send. These communication functions are sufficient to ensure the synchronization between different processes. An example is illustrated in Figure 7.19.

```
...
barrier(
  spawn(0,
    Gauss(Sxx, Ixx)
  );
  spawn(1,
    Gauss(Syy, Iyy);
    asend(0, Syy)
  );
  spawn(2,
    Gauss(Sxy, Ixy);
    asend(0, Sxy)
  )
);
...
```

```
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank0);
  ...
  if (rank0==0)
    Gauss(Sxx, Ixx);
  if (rank0==1) {
    Gauss(Syy, Iyy);
    MPI_Isend(Syy, N*M, MPI_FLOAT, 0, MPI_ANY_TAG,
              MPI_COMM_WORLD, &request);
  }
  if (rank0==2) {
    Gauss(Sxy, Ixy);
    MPI_Isend(Sxy, N*M, MPI_FLOAT, 0, MPI_ANY_TAG,
              MPI_COMM_WORLD, &request);
  }
  ...
  MPI_Finalize();
```

Figure 7.19: SPIRE representation of a part of Harris, and its MPI generated code

Hierarchical Parallelism

Most people use only *MPI_COMM_WORLD* as a communicator between MPI processes. This universe intracommunicator is not enough for handling hierarchical parallelism. MPI implementation offers the possibility of creating subset communicators via the function `MPI_Comm_split` [82] that creates new communicators from an initially defined communicator. Each new communicator can be used in a level of hierarchy in the parallel code. Another scheme consists in gathering the code of each process and then assigning the code to the corresponding process.

In our implementation of the generation of MPI code from SPIRE code, we leave the implementation of hierarchical MPI as future work: we implement only flat MPI code. Another type of hierarchical MPI is obtained by merging MPI and OpenMP together (hybrid programming) [12]; this is also called multi-core-aware message-passing (MPI). We also discuss this in the future work section (see Section 9.2).

7.5 Conclusion

The goal of this chapter is to put the first brick of a framework of parallel code transformations and generation using our BDSC- and SPIRE-based hierarchical task parallelization techniques. We present two parallel transformations: first, from unstructured to structured parallelism for high-level abstraction of parallel constructs and, second, the conversion of shared memory programs to distributed ones. For these two transformations, we juggle with SPIRE constructs for performing them. Moreover, we show the generation of task parallelism and communications at both levels: equilevel and hierarchical. Finally, we generate both OpenMP and MPI from the same parallel IR derived from SPIRE, except for the hierarchical MPI that we leave as future work.

The next chapter provides experimental results using OpenMP and MPI generated programs via the methodology presented in this thesis, based on HBDSC scheduling algorithm and SPIRE-derived parallel languages.

Chapter 8

Experimental Evaluation with PIPS

Experience is a lantern tied behind our backs, which illuminates the path.
Confucius

In this chapter, we describe the integration of SPIRE and HBDSC within the PIPS compiler infrastructure and their application to the parallelization of five significant programs, targeting both shared and distributed memory architectures: the image and signal processing benchmarks Harris and ABF, the SPEC2001 benchmark equake, the NAS parallel benchmark IS and an FFT code. We present the experimental results we obtained for these benchmarks; our experiments suggest that HBDSC's focus on efficient resource management leads to significant parallelization speedups on both shared and distributed memory systems, improving upon DSC results, as shown by the comparison of the sequential and parallelized versions of these five benchmarks running on both OpenMP and MPI frameworks.

We provide also a comparative study between our task parallelization implementation in PIPS and that of the audio signal processing language Faust, using two Faust programs: Karplus32 and Freeverb.

Dans ce chapitre, nous décrivons l'intégration de SPIRE et HBDSC au sein de l'infrastructure de compilation PIPS et leur application à la parallélisation de cinq "programmes test" (benchmarks) importants, ciblant à la fois les architectures à mémoires partagée et distribuée : les programmes de traitement d'image et de traitement du signal Harris et ABF, le programme equake de SPEC2001, le programme NAS IS et un code de transformée de Fourier rapide, FFT. Nous présentons les résultats expérimentaux que nous avons obtenus pour ces programmes. Nos expériences suggèrent que l'accent mis par HBDSC sur la gestion efficace des contraintes de ressources conduit à des accélérations de parallélisation importantes sur les systèmes à mémoire aussi bien partagée que distribuée. Cela surpasse nettement les résultats obtenus en utilisant simplement DSC, comme le montre la comparaison des versions séquentielles et parallélisées de ces cinq programmes, écrites en OpenMP et MPI.

Nous fournissons également une étude comparative entre notre implémentation de parallélisation de tâches dans PIPS et celle utilisée dans le langage audio de traitement du signal Faust, en nous fondant sur deux programmes

écrits au départ en Faust : Karplus32 et Freeverb.

8.1 Introduction

The HBDSC algorithm presented in Chapter 6 has been designed to offer better task parallelism extraction performance for parallelizing compilers than traditional list-scheduling techniques such as DSC. To verify its effectiveness, BDSC has been implemented in the PIPS compiler infrastructure and tested on programs written in C. Both the HBDSC algorithm and SPIRE are implemented in PIPS. Recall that HBDSC is the hierarchical layer that adapts a scheduling algorithm (DSC, BDSC...) which handles a DAG (sequence) for a hierarchical code. The analyses of estimation of time execution of tasks, dependences and communications between tasks for computing tlevel and blevel are also implemented in PIPS.

In this chapter, we provide experimental BDSC-vs-DSC comparison results based on the parallelization of five benchmarks, namely ABF [52], Harris [96], equake [22], IS [87] and FFT [?]. We chose these particular benchmarks since they are well-known benchmarks and exhibit task parallelism that we hope our approach will be able to take advantage of. Note that, since our focus in this chapter is to compare BDSC with prior work, namely DSC, our experiments in this chapter do not address the hierarchical component of HBDSC.

BDSC uses numerical constants to label its input graph; we show in Section 6.3 how we lift this restriction using static and dynamic approaches based on array region and complexity analyses. In this chapter, we study experimentally the input sensitivity issues on the task and communication time estimations in order to assess the scheduling robustness of BDSC since it relies on approximations provided by these analyses.

Several existing systems already automate the parallelization of programs targeting different languages (see Section 6.6) such as the non-open-source compiler OSCAR [63]. In the goal of comparing our approach with others, we choose the audio signal processing language Faust [88] because it is an open-source compiler and also generates automatically parallel tasks in OpenMP.

The remainder of this chapter is structured as follows. We detail the implementation of HBDSC in Section 8.2. Section 8.3 presents the scientific benchmarks under study: Harris, ABF, equake, IS and FFT, the two target machines that we use for our measurements, and also the effective cost model parameters that depend on each target machine. Section 8.4 explains the process we follow to parallelize these benchmarks. Section 8.5 provides performance results when these benchmarks are parallelized on the PIPS platform targeting a shared memory architecture and a distributed memory architecture. We also assess the sensitivity of our parallelization technique

on the accuracy of the static approximations of the code execution time used in task scheduling, in Section 8.6. Section 8.7 provides a comparative study between our task parallelization implementation in PIPS and that of Faust when dealing with audio signal processing benchmarks.

8.2 Implementation of HBDSC- and SPIRE-Based Parallelization Processes in PIPS

We have implemented the HBDSC parallelization (Chapter 6) and some SPIRE-based parallel code transformations and generation (Chapter 7) algorithms in PIPS. PIPS is structured as a collection of independent code analysis and transformation phases that we call passes. This section describes the order and the composition of passes that we have implemented in PIPS to enable the parallelization of sequential C77 language programs. Figure 8.1 summarizes the parallelization processes.

These passes are managed by the Pipsmake library[1]. Pipsmake manages objects that are resources stored in memory or/and on disk. The passes are described by generic rules that indicate used and produced resources. Examples of these rules are illustrated in the following subsections. Pipsmake maintains the global data consistency intra- and inter-procedurally.

The goal of the following passes is to generate the parallel code expressed in SPIRE(PIPS IR) using HBDSC scheduling algorithm, then communications primitives and, finally, OpenMP and MPI codes, in order to automate the task parallelization of sequential programs.

Tpips[2] is the line interface and scripting language of the PIPS system. The execution of the necessary passes is obtained by typing their name in the command line interpreter of PIPS, tpips. Figure 8.2 depicts harris.tpips, the script used to execute our shared and distributed memory parallelization processes on the benchmark Harris. It chains the passes described in the following subsections.

8.2.1 Preliminary Passes

Sequence Dependence Graph Pass

Pass `sequence_dependence_graph` generates the Sequence Dependence Graph (SDG), as explained in Section 6.2. It uses in particular the `dg` resource (data dependence graph) of PIPS and outputs the resulting SDG in the resource `sdg`.

```
sequence_dependence_graph          > MODULE.sdg
```

[1] http://cri.ensmp.fr/PIPS/pipsmake.html

[2] http://www.cri.ensmp.fr/pips/tpips-user-manual.htdoc/tpips-user-manual.pdf

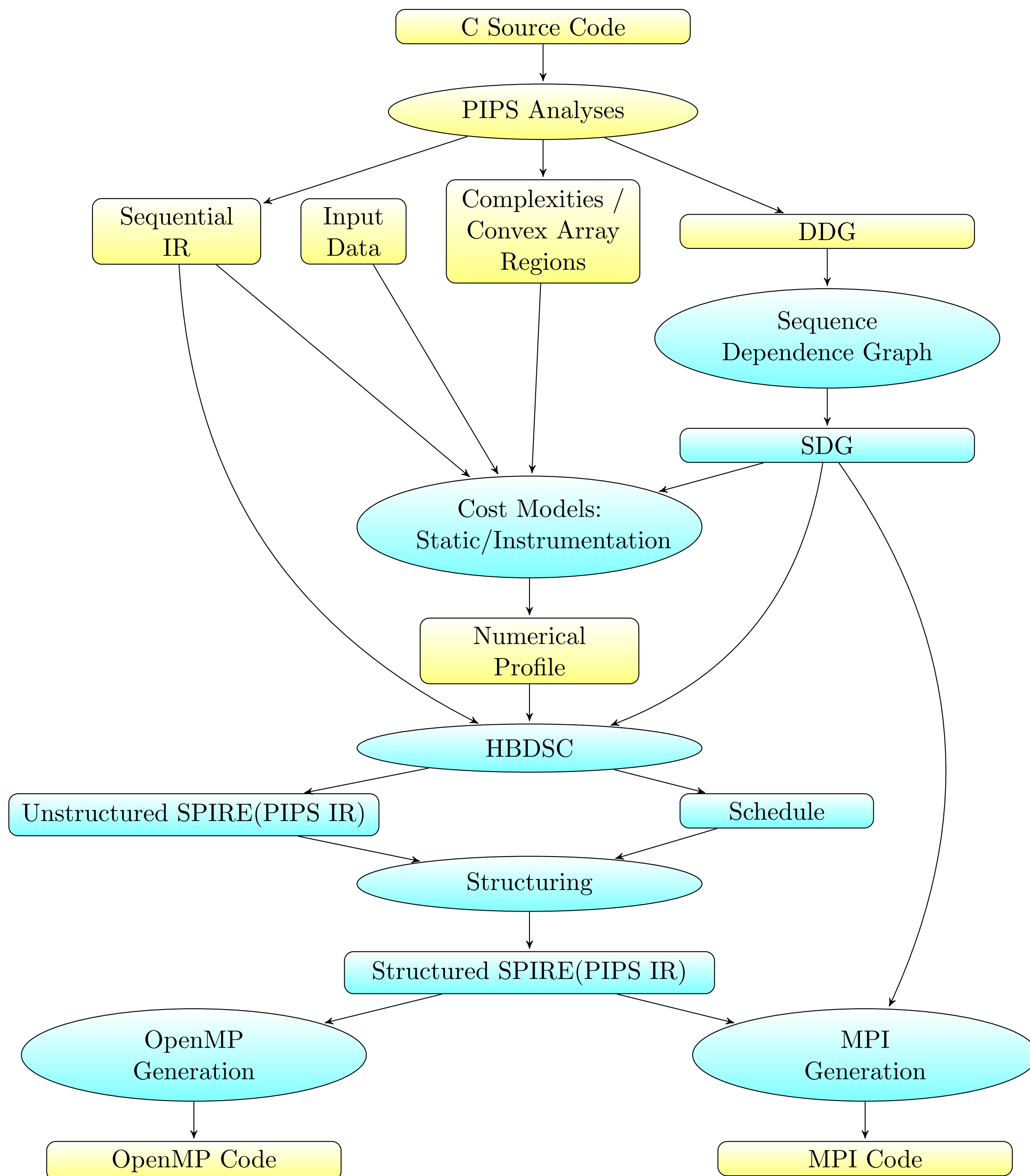


Figure 8.1: Parallelization process: blue indicates thesis contributions; an ellipse, a pass or a set of passes; and a rectangle, results

```
< PROGRAM.entities
< MODULE.code
```

```
#Create a workspace for the program harris.c
create harris harris.c

apply SEQUENCE_DEPENDENCE_GRAPH[main]
shell dot -Tpng harris.database/main/main_sdg.dot > main_sdg.png
shell gqview main_sdg.png

setproperty HBDSC_NB_CLUSTERS 3

apply HBDSC_PARALLELIZATION[main]
apply SPIRE_UNSTRUCTURED_TO_STRUCTURED[main]
#Print the result on stdout
display PRINTED_FILE[main]

echo // OMP style
activate OPENMP_TASK_GENERATION
activate PRINT_PARALLELIZEDOMPTASK_CODE
display PARALLELPRINTED_FILE[main]

echo // MPI style
apply HBDSC_GEN_COMMUNICATIONS[main]
activate MPI_TASK_GENERATION
activate PRINT_PARALLELIZEDMPI_CODE
display PARALLELPRINTED_FILE[main]

#Print the scheduled SDG of harris
shell dot -Tpng harris.database/main/main_ssdg.dot > main_ssdg.png
shell gqview main_scheduled_ssdg.png
close
quit
```

Figure 8.2: Executable (harris.tpips) for harris.c

```
        < MODULE.proper_effects
        < MODULE.dg
        < MODULE.regions
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

Code Instrumentation Pass

We model the communication costs, data sizes and execution times of different tasks with polynomials. These polynomials have to be converted to numerical values in order to be used by BDSC algorithm. We instrument thus, using the pass `hbdsc_code_instrumentation`, the input sequential code and run it once in order to obtain the numerical values of the polynomials. The instrumented code contains the initial user code plus statements that compute the values of the cost polynomials for each statement (as detailed in

Section 6.3.2).

```
hbdsc_code_instrumentation         > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.dg
        < MODULE.regions
        < MODULE.summary_complexity
```

Path Transformer Pass

The path transformer (see Section 6.4) between two statements computes the possible changes performed by a piece of code delimited by two statements S_{begin} and S_{end} enclosed within a statement S. The goal of the following pass is to compute the path transformer between two statements labeled by the labels `sbegin` and `ssend` in the `code` resource. It outputs a resource file that contains the resulting transformer.

```
path_transformer                  > MODULE.path_transformer_file
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.transformers
        < MODULE.preconditions
        < MODULE.cumulated_effects
```

8.2.2 Task Parallelization Passes

HBDSC Task Parallelization Pass

Pass `hbdsc_parallelization` applies HBDSC on the resulting SDG (the input resource dg) and generates a parallel code in an unstructured form using the resource `dg` implementing the scheduled SDG. Resource `schedule` implements our function of hierarchical schedule σ (see Section 6.5); it maps each statement to a given cluster. This pass implements the `HBDSC` algorithm presented in Section 6.5, Algorithm 19. Note that Algorithm 19 uses the `unstructured` construct instead of scheduled SDG to represent unstructured parallelism. Our prototype uses SDG instead for historical reasons. This pass uses the analyses of regions, preconditions, transformers, complexities and effects and outputs scheduled SDG, in the resource `dg`, and a function σ in the resource `schedule`.

```
hbdsc_parallelization                       > MODULE.sdg
        < PROGRAM.entities                  > MODULE.schedule
```

```
< MODULE.code
< MODULE.proper_effects
< MODULE.sdg
< MODULE.regions
< MODULE.summary_complexity
< MODULE.transformers
< MODULE.preconditions
< MODULE.cumulated_effects
```

Parallelism Structuring Pass

Pass `hbdsc_parallelization` generates unstructured parallelism. The goal of the following pass is to structure this parallelism. Pass `spire_unstructured_to_structured` transforms the parallelism generated in form of graphs (SDG) into `spawn` and `barrier` constructs. It uses the scheduled SDG and the schedule function σ (the resource `schedule`) and generates a new code (structured), and updates the `schedule` resource, since it adds to the schedule the newly created statements of the new code. This pass implements Algorithm `unstructured_to_structured` presented in Section 7.2, Algorithm 21.

```
spire_unstructured_to_structured           > MODULE.code
             < PROGRAM.entities            > MODULE.schedule
             < MODULE.code
             < MODULE.sdg
             < MODULE.schedule
```

HBDSC Tuning

In PIPS, one can use different properties to parameterize passes and select the precision required or the algorithm to use. Default properties are defined by PIPS but they can be redefined by the user using the command `setproperty` when the tpips interface is used.

The following properties are used by Pass `hbdsc_parallelization` to allow the user to define the number of clusters and memory size, based on the properties of the target architecture.

The number of clusters is set by default to 4, the memory size to -1 which means that we have infinite memory.

```
HBDSC_NB_CLUSTERS 4
```

```
HBDSC_MEMORY_SIZE -1
```

In order to control the granularity of parallel tasks, we introduce the following property. By default, we generate the maximum parallelism in

codes. Otherwise, i.e. when COSTLY_TASKS_ONLY is TRUE, only loops and function calls are used as parallel tasks. This is important in order to make a trade-off between task creation overhead and task execution time.

```
COSTLY_TASKS_ONLY FALSE
```

The next property specifies if communication, data and time information is to be extracted from the result of the instrumentation and stored into `HBDSC_INSTRUMENTED_FILE` file, which is a dynamic analysis, or if the static cost models have to be used. The default option is an empty string. It means that static results have to be used.

```
HBDSC_INSTRUMENTED_FILE ""
```

8.2.3 OpenMP Related Passes

SPIRE is designed to help generate code for different types of parallel languages, just like different parallel languages can be mapped into SPIRE. The four following passes show how this intermediate representation simplifies the task of producing code for various languages such as OpenMP and MPI (see Section 8.2.4).

OpenMP Task Generation Pass

Pass `openmp_task_generation` generates OpenMP task parallel code from SPIRE(PIPS IR) as detailed in Section 7.4.2. It replaces `spawn` constructs by the OpenMP directives `omp task`, `barrier` constructs by `omp single` or `omp taskwait`. It outputs a new resource, `parallelized_code`, for the OpenMP code.

```
openmp_task_generation            > MODULE.parallelized_code
        < PROGRAM.entities
        < MODULE.code
```

OpenMP Prettyprint Pass

The next pass `print_parallelizedOMPTASK_code` is a pretty printing function. It outputs the code decorated with OpenMP (`omp task`) directives using `parallelized_code` resource.

```
print_parallelizedOMPTASK_code    > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.parallelized_code
```

8.2.4 MPI Related Passes: SPMDization

Automatic Data Distribution Pass

Pass `hbdsc_gen_communications` generates `send` and `recv` primitives. It implements the algorithm `communications_construction` presented in Section 7.3, Algorithm 27. It takes as input the parallel code (`<` symbol) and the precedent resources of the sequential code (`=` symbol). It generates a new parallel code that contains communications calls.

```
hbdsc_gen_communications                       > MODULE.code
         < PROGRAM.entities
         < MODULE.code
         = MODULE.dg
         = MODULE.schedule
         = MODULE.proper_effects
         = MODULE.preconditions
         = MODULE.regions
```

MPI Task Generation Pass

The same process applies for MPI. The pass `mpi_task_generation` generates MPI parallel code from SPIRE(PIPS IR) as detailed in Section 7.4.3. It replaces `spawn` constructs by guards (`if` statements), `barrier` constructs by `MPI_barrier`, `send` functions by `MPI_Isend` and `recv` functions by `MPI_Recv`. Note that we leave the issue of generation of hierarchical MPI as future work. Our implementation generates only flat MPI code (a sequence of non-nested tasks).

```
mpi_task_generation              > MODULE.parallelized_code
        < PROGRAM.entities
        < MODULE.code
```

MPI Prettyprint Pass

Pass `print_parallelizedMPI_code` outputs the code decorated with MPI instructions.

```
print_parallelizedMPI_code        > MODULE.parallelprinted_file
        < PROGRAM.entities
        < MODULE.parallelized_code
```

We generate an MPI code in the format described in Figure 8.3.

```
{
  /* declare variables */
  MPI_Init(&argc, &argv);
  /* parse arguments */
  /* main program */
  MPI_Finalize();
}
```

Figure 8.3: A model of an MPI generated code

8.3 Experimental Setting

We carried out experiments to compare DSC and BDSC algorithms. BDSC provides significant parallelization speedups on both shared and distributed memory systems on the set of benchmarks that we present in this section. We also present the cost model parameters that depend on the target machine.

8.3.1 Benchmarks: ABF, Harris, Equake, IS and FFT

For comparing DSC and BDSC, on both shared and distributed memory systems, we use five benchmarks, ABF, Harris, Equake, IS and FFT, briefly presented below.

ABF

The signal processing benchmark ABF (Adaptive Beam Forming) [52] is a 1,065-line program that performs adaptive spatial radar signal processing for an array of smart radar antennas in order to transmit or receive signals in different directions and enhance these signals by minimizing interference and noise. It has been developed by Thales [106] and is publicly available. Figure 8.4 shows the scheduled SDG of the function `main` of this benchmark on three clusters (κ_0, κ_1 and κ_2). This function contains 7 parallel loops while the degree of task parallelism is only three (there are three parallel tasks at most). In Section 8.4 we show how we use these loops to create more parallel tasks.

Harris

The image processing algorithm Harris [54] is a 105-line image processing corner detector used to identify points of interest. It uses several functions (operators such as auto-correlation) applied to each pixel of an input image. It is used for feature extraction for motion detection and image matching. In Harris [96], at most three tasks can be executed in parallel. Figure 6.16 on

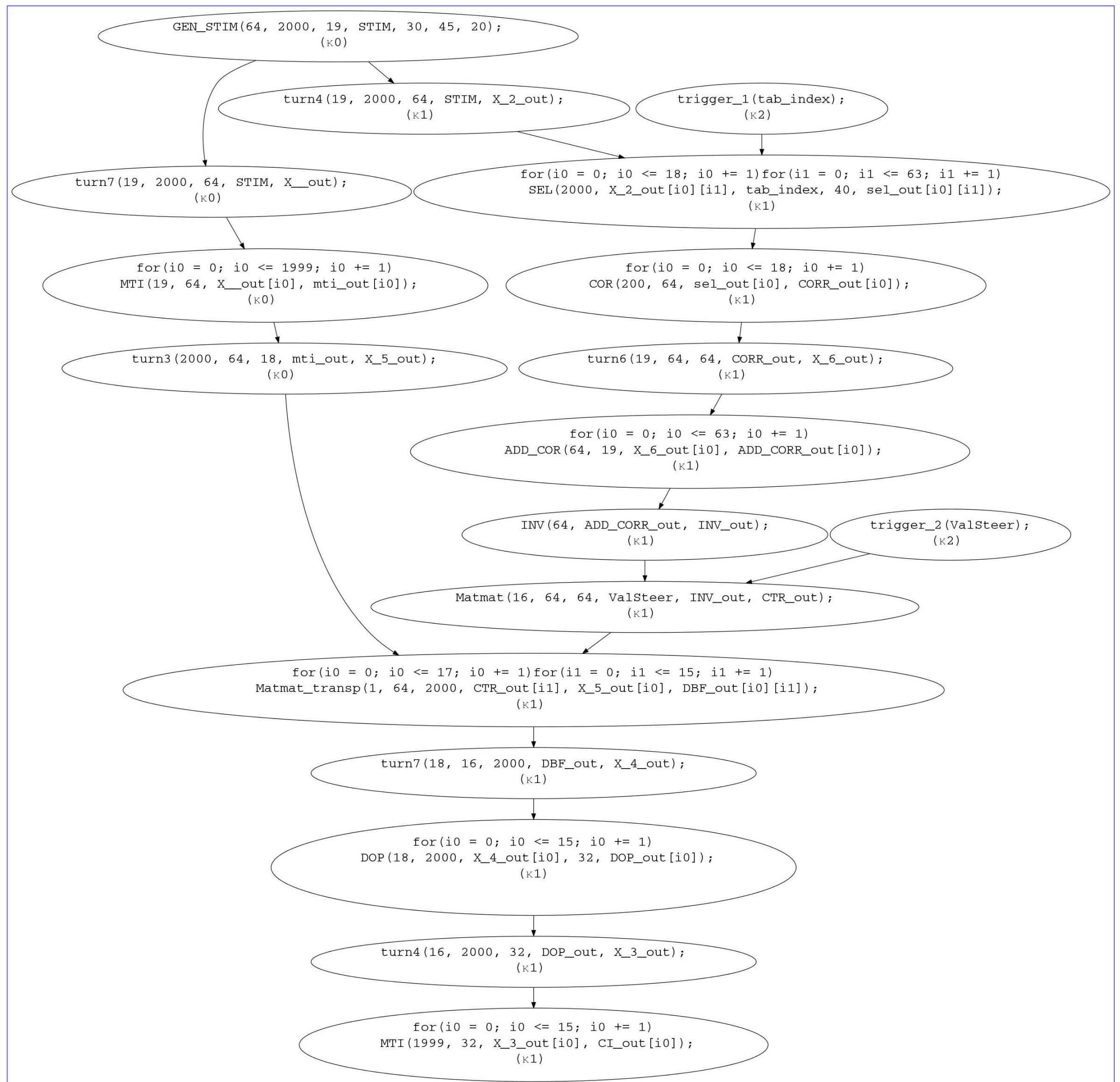


Figure 8.4: Scheduled SDG of the main function of ABF

page 139 shows the scheduled SDG for Harris obtained using three processors (aka clusters).

Equake (SPEC Benchmarks)

The 1,432-line SPEC benchmark equake [22] is used in the simulation of seismic wave propagation in large valleys, based on computations performed on an unstructured mesh that locally resolves wavelengths, using the Archimedes

finite element method. Equake contains four hot loops: three in the function `main` and one in the function `smvp_opt`. A part of the function `main` is illustrated in Figure 6.5 on page 117. We use these four loops for parallelization.

IS (NAS Parallel Benchmarks)

Integer Sort (IS) is one of the eleven benchmarks in the NAS Parallel Benchmarks suite [87]. The serial version contains 1,076 lines. We exploit the hot function `rank` of IS that contains a degree of task parallelism of two and two parallel loops.

FFT

Fast Fourier Transform (FFT) is widely used in many areas of science, mathematics and engineering. We use the implementation of FFT3 of [8]. In FFT, for N complex points, both time domain decomposition and frequency domain synthesis require three loops. The outer loop runs through the log_2N stages. The middle loop (chunks) moves through each of the individual frequency spectra (chunk) in the stage being worked on. The innermost loop uses the Butterfly pattern to compute the points for each frequency of the spectrum. We use the middle loop for parallelization since it is parallel (each chunk can be launched in parallel).

8.3.2 Experimental Platforms: Austin and Cmmcluster

We use the two following experimental platforms for our performance evaluation tests.

Austin

The first target machine is a host Linux (Ubuntu) shared memory machine with a 2-socket AMD quadcore Opteron with 8 cores, with $M = 16$ GB of RAM, running at 2.4 GHz. We call this machine *Austin* in the remainder of this chapter. The version of gcc 4.6.3 is installed where the version of OpenMP 3.0 is supported (we use gcc to compile, using -fopenmp for OpenMP programs and -O3 as optimization flags).

Cmmcluster

The second machine that we target for our performance measurements is a host Linux (RedHat) distributed memory machine with 6 dual-core processors Intel(R) Xeon(R), with $M = 32$ GB of RAM per processor, running at 2.5 GHz. We call this machine *Cmmcluster* in the remainder of this chapter. The versions of gcc 4.4.6 and Open MPI 1.6.2 are installed and used for compilation (we use mpicc to compile MPI programs and -O3 as optimization flags).

8.3.3 Effective Cost Model Parameters

In the default cost model of PIPS, the number of clock cycles per instruction (CPI) is equal to 1 and the number of clock cycles for the transfer of one byte is $\beta = 1$. Instead of using this default cost model, we use an effective model for each target machine, Austin and Cmmcluster. Table 8.1 summarizes the cost model for the integer and floating-point operations of addition, subtraction, multiplication, division and transfer cost of one byte. The parameters of the arithmetic operations of this table are extracted from agner.org[3] that gives the measurements of CPU clock cycles for AMD (Austin) and Intel (Cmmcluster) CPUs. The transfer cost of one byte in number of cycles β has been measured by using a simple MPI program that sends a number of bytes over the Cmmcluster memory network. Since Austin is a shared memory machine that we use to execute OpenMP codes, we do not need to measure β for this machine (NA).

Machine / Operation	Austin		Cmmcluster	
	int	float	int	float
Addition (+)	2.5	2	3	3
Subtraction (-)	2.5	2	3	3
Multiplication (×)	3	2	11	3
Division (/)	40	30	46	39
One byte transfer (β)	NA	NA	2.5	2.5

Table 8.1: CPI for the Austin and Cmmcluster machines, plus the transfer cost of one byte β on the Cmmcluster machine (in #cycles)

8.4 Protocol

We have extended PIPS with our implementation in C of BDSC-based hierarchical scheduling (HBDSC). To compute the static execution time and communication cost estimates needed by HBDSC, we relied upon the PIPS run time complexity analysis and a more realistic, architecture-dependent communication cost matrix (see Table 8.1). For each code S of our test benchmarks, PIPS performed automatic task parallelization, applied our hierarchical scheduling process `HBDSC`(S, P, M, $\perp$) (using either BDSC or DSC) on these sequential programs to yield a schedule $\sigma_{unstructured}$ (unstructured code). Then, PIPS transforms the schedule $\sigma_{unstructured}$ via `unstructured_to_structured`(S, P, $\sigma_{unstructured}$) to yield a new schedule $\sigma_{structured}$. PIPS automatically generated an OpenMP [4] version from the parallel statements encoded in SPIRE(PIPS IR) in $\sigma_{structured}(S)$, using `omp task` directives; another version, in MPI [3], was also generated. We also

[3] `http://www.agner.org/optimize/instruction_tables.pdf`

applied the DSC scheduling process on these benchmarks and generated the corresponding OpenMP and MPI codes. The execution times have been measured using `gettimeofday` time function.

Compilation times for these benchmarks were quite reasonable, the longest (equake) being 84 seconds. In this last instance, most of the time (79 seconds) was spent by PIPS to gather semantic information such as regions, complexities and dependences; our prototype implementation of HBDSC is only responsible for the remaining 5 seconds.

We ran all these parallelized codes on the two shared and distributed memory computing systems presented in Section 8.3.2. To increase available coarse-grain task parallelism in our test suite, we have used both unmodified and modified versions of our benchmarks. We tiled and fully unrolled by hand the most costly loops in ABF (7 loops), FFT (2 loops) and equake (4 loops); the tiling factor for the BDSC version depends on the number of available processors P (tile size = number of loop iterations / P), while we had to find the proper one for DSC, since DSC puts no constraints on the number of needed processors but returns the number of processors its scheduling requires. For Harris and IS, our experiments have looked at both tiled and untiled versions of the benchmarks. The full unrolling that we apply for our experiments creates a sequence of statements from a nested loop. An example of loop tiling and full unrolling of a loop of the ABF benchmark is illustrated in Figure 8.5. The function `COR` estimates covariance based on a part of the received signal `sel_out`, averaged on `Nb_rg` range gates and `nb_pul` pulses. Note that the applied loop tiling is equivalent to loop chunking [84].

For our experiments, we use $M = 16$ GB for Austin and $M = 32$ GB for Cmmcluster. Thus, the schedules used for our experiments satisfy these memory sizes.

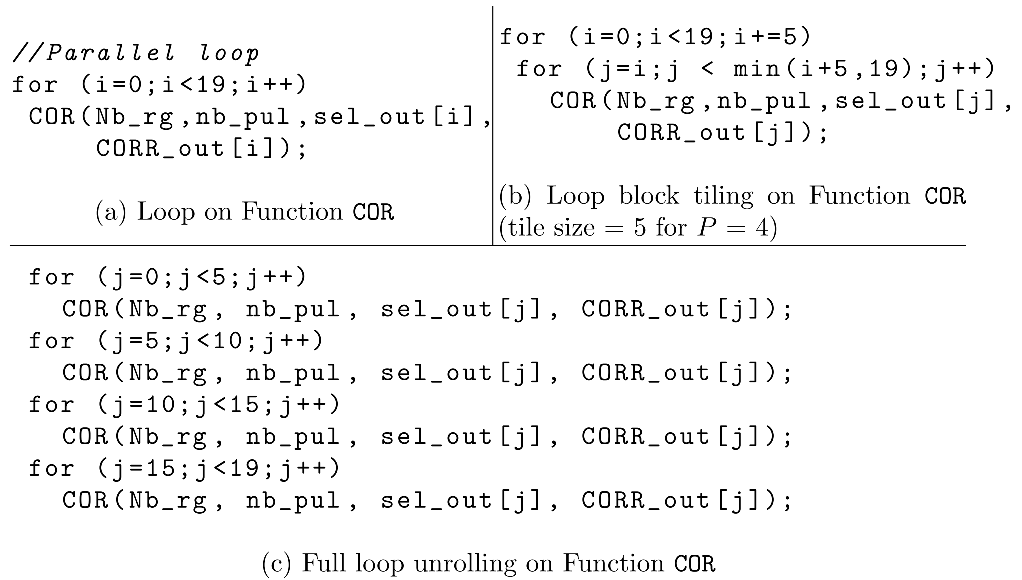
8.5 BDSC vs. DSC

We apply our parallelization process on the benchmarks ABF, Harris, Equake and IS. We execute the parallel versions on both the Austin and Cmmcluster machines. As for FFT, we get an OpenMP version and study how more performance can be extracted using streaming languages.

8.5.1 Experiments on Shared Memory Systems

We measured the execution time of the parallel OpenMP codes on the $P =$ 1, 2, 4, 6 and 8 cores of Austin.

Figure 8.6 shows the performance results of the generated OpenMP code on the two versions scheduled using DSC and BDSC on ABF and equake. The speedup data show that the DSC algorithm is not scalable on these examples, when the number of cores is increased; this is due to the generation



```
//Parallel loop
for (i=0;i<19;i++)
 COR(Nb_rg,nb_pul,sel_out[i],
      CORR_out[i]);
```

(a) Loop on Function COR

```
for (i=0;i<19;i+=5)
 for (j=i;j < min(i+5,19);j++)
   COR(Nb_rg,nb_pul,sel_out[j],
        CORR_out[j]);
```

(b) Loop block tiling on Function COR (tile size = 5 for $P = 4$)

```
for (j=0;j<5;j++)
  COR(Nb_rg, nb_pul, sel_out[j], CORR_out[j]);
for (j=5;j<10;j++)
  COR(Nb_rg, nb_pul, sel_out[j], CORR_out[j]);
for (j=10;j<15;j++)
  COR(Nb_rg, nb_pul, sel_out[j], CORR_out[j]);
for (j=15;j<19;j++)
  COR(Nb_rg, nb_pul, sel_out[j], CORR_out[j]);
```

(c) Full loop unrolling on Function COR

Figure 8.5: Steps for the rewriting of data parallelism to task parallelism using the tiling and unrolling transformations

of more clusters (task creation overhead) with empty slots (poor potential parallelism and bad load balancing) than with BDSC, a costly decision given that, when the number of clusters exceeds P, they have to share the same core as multiple threads.

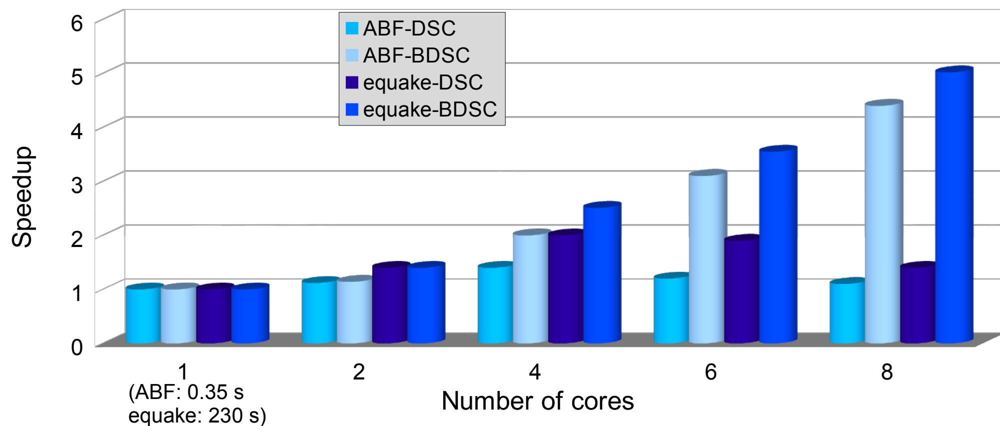


Figure 8.6: ABF and equake speedups with OpenMP

Figure 8.7 presents the speedup obtained using $P = 3$, since the maximum parallelism in Harris is three, assuming no exploitation of data parallelism, for two parallel versions: BDSC with and BDSC without tiling of the

kernel `CoarsitY` (we tiled by 3). The performance is given using three different input image sizes: 1024×1024, 2048×1024 and 2048×2048. The best speedup corresponds to the tiled version with BDSC because, in this case, the three cores are fully loaded. The DSC version (not shown in the figure) yields the same results as our versions because the code can be scheduled using three cores.

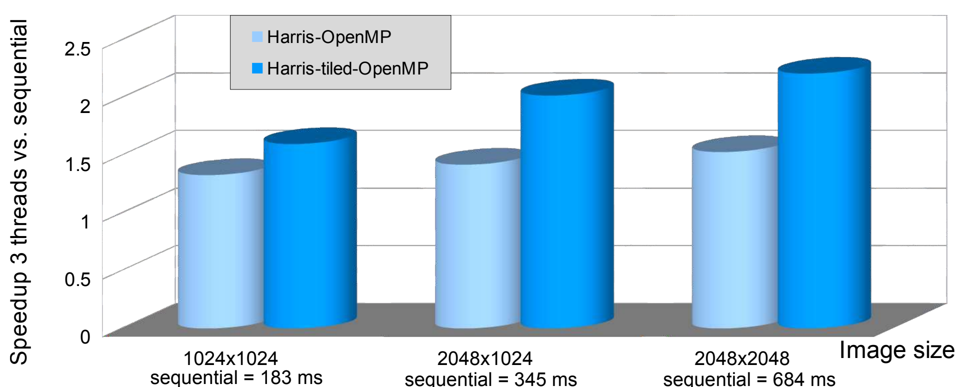


Figure 8.7: Speedups with OpenMP: impact of tiling (P=3)

Figure 8.8 shows the performance results of the generated OpenMP code on the NAS benchmark IS after applying BDSC. The maximum task parallelism without tiling in IS is two, which is shown in the first subchart; the other subcharts are obtained after tiling. The program has been run with three IS input classes (A, B and C [87]). The bad performance of our implementation for Class A programs is due to the large task creation overhead, which dwarfs the potential parallelism gains, even more limited here because of the small size of Class A data.

As for FFT, after tiling and fully unrolling the middle loop of FFT and parallelizing it, an OpenMP version is generated using PIPS. Since we only exploit the parallelism of the middle loop, this is not sufficient to obtain good performance. Indeed, a maximum speed up of 1.5 is obtained for an FFT of size $n = 2^{20}$ on Austin (8 cores). The sequential outer loop of $log_2 n$ iterations makes the performance that bad. However, our technique can be seen as a first step of parallelization for languages such as OpenStream.

OpenStream [94] is a data-flow extension of OpenMP in which dynamic dependent tasks can be defined. In [93], an efficient version of the FFT is provided, where in addition to parallelizing the middle loop (data parallelism), it exploits possible streaming parallelism between the outer loop stages. Indeed, once a chunk is executed, the chunks that get their dependences satisfied in the next stage start execution (pipeline parallelism). This way, a maximum speedup of 6.6 on a 4-socket AMD quadcore Opteron with

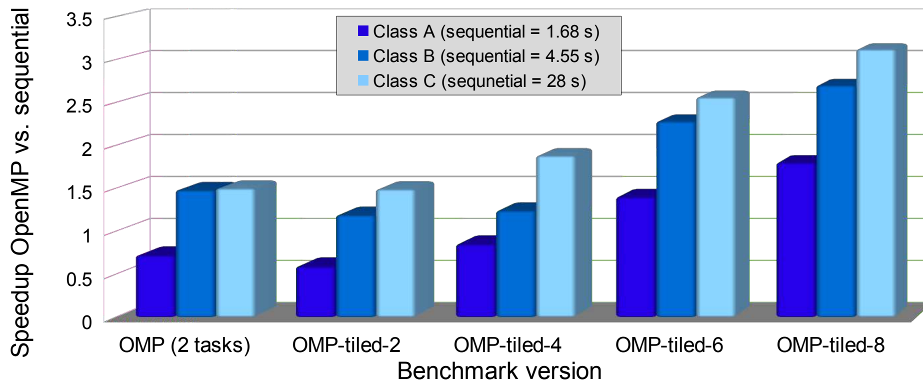


Figure 8.8: Speedups with OpenMP for different class sizes (IS)

16 cores for size of FFT $n = 2^{20}$ [93] is obtained.

Our parallelization of FFT does not generate automatically pipeline parallelism but generates automatically parallel tasks that may also be coupled as streams. In fact, to use OpenStream, the user has to write manually the input streaming code. We can say that, thanks to the code generated using our approach, where the parallel tasks are automatically generated, the user just has to connect these tasks by providing (manually) the data flow necessary to exploit additional potential pipeline parallelism.

8.5.2 Experiments on Distributed Memory Systems

We measured the execution time of the parallel codes on $P = 1$, 2, 4 and 6 processors of Cmmcluster.

Figure 8.9 presents the speedups of the parallel MPI vs. sequential versions of ABF and equake using $P = 2$, 4 and 6 processors. As before, the DSC algorithm is not scalable, when the number of processors is increased, since the generation of more clusters with empty slots leads to higher process scheduling cost on processors and communication volume between them.

Figure 8.10 presents the speedups of the parallel MPI vs. sequential versions of Harris using three processors. The tiled version with BDSC gives the same result as the non-tiled version since the communication overhead is so important when the three tiled loops are scheduled on three different processors that BDSC scheduled them on the same processor; this led thus to a schedule equivalent to the one of the non-tiled version. Compared to OpenMP, the speedups decrease when the image size is increased because the amount of communication between processors increases. The DSC version (not shown on the figure) gives the same results as the BDSC version because the code can be scheduled using three processors.

Figure 8.11 shows the performance results of the generated MPI code

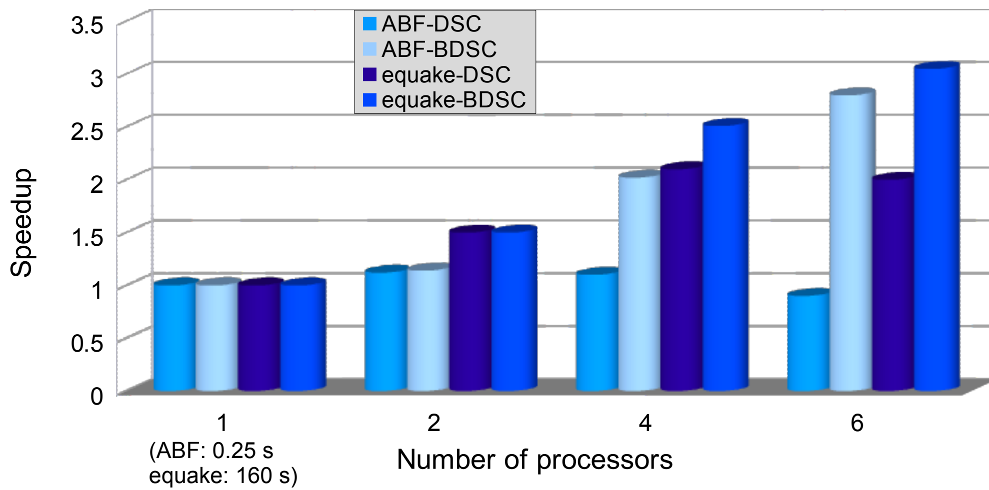


Figure 8.9: ABF and equake speedups with MPI

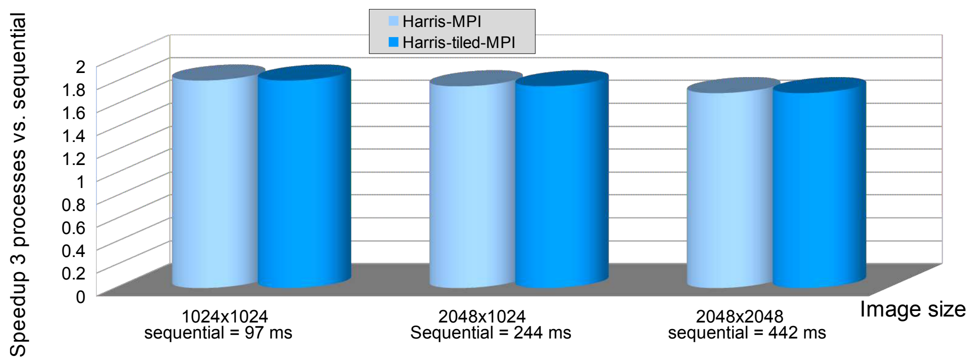


Figure 8.10: Speedups with MPI: impact of tiling (P=3)

on the NAS benchmark IS after application of BDSC. The same analysis as the one for OpenMP applies here, in addition to communication overhead issues.

8.6 Scheduling Robustness

Since our BDSC scheduling heuristic relies on the numerical approximations of the execution and communication times of tasks, one needs to assess its sensitivity to the accuracy of these estimations. Since a mathematical analysis of this issue is made difficult by the heuristic nature of BDSC and, in fact, of scheduling processes in general, we provide below experimental data that show that our approach is rather robust.

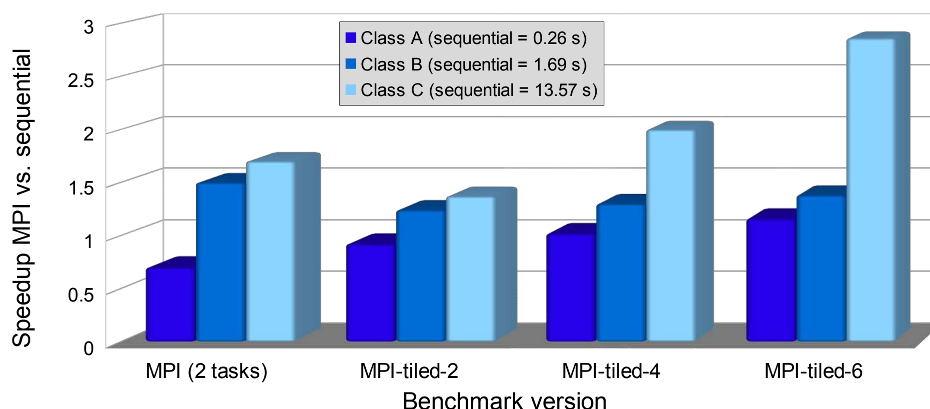


Figure 8.11: Speedups with MPI for different class sizes (IS)

In practice, we ran multiple versions of each benchmark using various static execution and communication cost models:

- the naive variant, in which all execution times and communications costs are supposed constant (only data dependences are enforced during the scheduling process);
- the default BDSC cost model described above;
- a biased BDSC cost model, where we modulated each execution time and communication cost value randomly by at most $\Delta\%$ (the default BDSC cost model would thus correspond to $\Delta = 0$).

Our intent, with introduction of different cost models, is to assess how small to large differences to our estimation of task times and communication costs impact the performance of BDSC-scheduled parallel code. We would expect that parallelization based on the naive variant cost model would yield the worst schedules, thus motivating our use of complexity analysis for parallelization purposes if the schedules that use our default cost model are indeed better. Adding small random biases to task times and communication costs should not modify too much the schedules (to demonstrate stability), while adding larger ones might, showing the quality of the default cost model used for parallelization.

Table 8.2 provides, for four benchmarks (Harris, ABF, equake and IS) and execution environment (OpenMP and MPI), the worst execution time obtained within batches of about 20 runs of programs scheduled using the naive, default and biased cost models. For this last case, we only kept in the table the entries corresponding to significant values of Δ, namely those at which, for at least one benchmark, the running time changed. So, for instance, when running ABF on OpenMP, the naive approach run time is

321 ms, while BDSC clocks at 214; adding random increments to the task communication and execution estimations provided by our cost model (Section 6.3) of up to, but not including, 80% does not change the scheduling, and thus running time. At 80%, running time increases to 230, and reaches 297 when $\Delta = 3,000$. As expected, the naive variant always provides sched-

Benchmark	Language	BDSC	Naive	$\Delta = 50(\%)$	80	100	200	1000	3000
Harris	OpenMP	153	277	153	153	153	153	153	277
	MPI	303	378	303	303	303	303	303	378
ABF	OpenMP	214	321	214	230	230	246	246	297
	MPI	240	310	240	260	260	287	287	310
equake	OpenMP	58	134	58	58	80	80	80	102
	MPI	106	206	106	106	162	162	162	188
IS	OpenMP	16	35	20	20	20	25	25	29
	MPI	25	50	32	32	32	39	39	46

Table 8.2: Run-time sensitivity of BDSC with respect to static cost estimation (in ms for Harris and ABF; in s for equake and IS).

ules that have the worst execution times, thus motivating the introduction of performance estimation in the scheduling process. Even more interestingly, our experiments show that one needs to introduce rather large task time and communication cost estimation errors, i.e., values of Δ, to make the BDSC-based scheduling process switch to less efficient schedules. This set of experimental data thus suggests that BDSC is a rather useful and robust heuristic, well adapted to the efficient parallelization of scientific benchmarks.

8.7 Faust Parallel Scheduling vs. BDSC

In music the passions enjoy themselves. **Friedrich Nietzsche**

Faust (Functional AUdio STream) [88] is a programming language for real-time signal processing and synthesis. FAUST programs (DSP code) are translated into equivalent imperative programs (C, C++, Java, etc.). In this section, we compare our approach in generating `omp task` directives with Faust approach in generating `omp section` directives.

8.7.1 OpenMP Version in Faust

The Faust compiler produces a single sample computation loop using the scalar code generator. Then, the vectorial code generator splits the sample processing loop into several simpler loops that communicate by vectors and produces a DAG. Next, this DAG is topologically sorted in order to detect loops that can be executed in parallel. After that, the parallel scheme generates OpenMP `parallel sections` directives using this DAG in which

each node is a computation loop; the granularity of tasks is only at the loop level. Finally, the compiler generates OpenMP `sections` directives in which a pragma *section* is generated for each loop.

8.7.2 Faust Programs: Karplus32 and Freeverb

In order to compare our approach with Faust's in generating task parallelism, we use Karplus32 and Freeverb as running examples [7].

Karplus32

The 200-line Karplus32 program is used to simulate the sound produced by a string excitation passed through a resonator to modify the final sound. It is an advanced version of a typical Karplus algorithm with 32 resonators in parallel, instead of one resonator only.

Freeverb

The 800-line Freeverb program is free-software that implements artificial reverberation. It filters echoes to generate a signal with reverberation. It uses four Schroeder allpass filters in series and eight parallel Schroeder-Moorer filtered-feedback comb-filters for each audio channel.

8.7.3 Experimental Comparison

Figure 8.12 shows the performance results of the generated OpenMP code on Karplus32 after application of BDSC and Faust, for two sample buffer sizes: `count` equals 1024 and 2048. Note that BDSC is performed on the generated C codes of these two programs using Faust. The hot function in Karplus32 (`compute`, of 90 lines) contains only 2 sections in parallel; sequential code represents about 60% of the whole code of Function `compute`. Since we have only two threads that execute only two tasks, Faust execution is a little better than BDSC. The dynamic scheduling of `omp task` cannot be shown in this case due to the small amount of parallelism in this program.

Figure 8.13 shows the performance results of the generated OpenMP code on Freeverb after application of BDSC and Faust, for two sizes: `count` equals 1024 and 2048. The hot function in Freeverb (`compute`, of 500 lines) contains up to 16 sections in parallel (the degree of task parallelism is equal to 16); sequential code represents about 3% of the whole code of Function `compute`. BDSC generates better schedules since it is based on sophisticated cost models, uses top level and bottom level values for ordering, while Faust proceeds by a topological ordering (only top level) and is based only on dependence analysis. In this case, results show that BDSC generates better

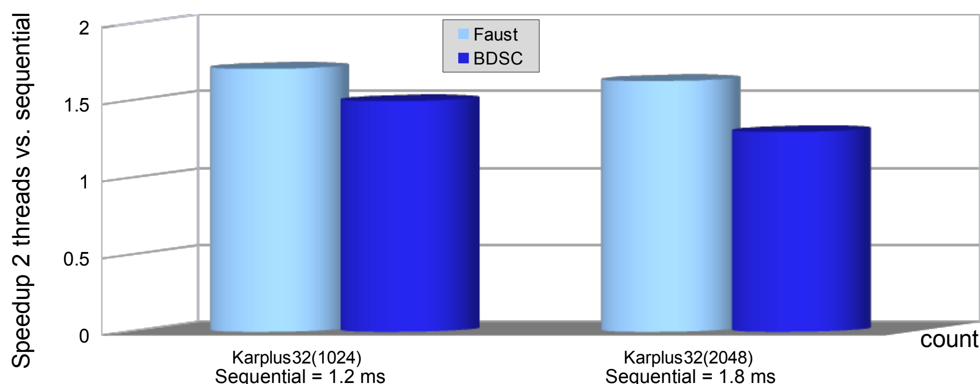


Figure 8.12: Run-time comparison (BDSC vs. Faust for Karplus32)

schedules and the generation of `omp task` instead of `omp section` is beneficial. Furthermore, the dynamic scheduler of OpenMP can make better decisions at run time.

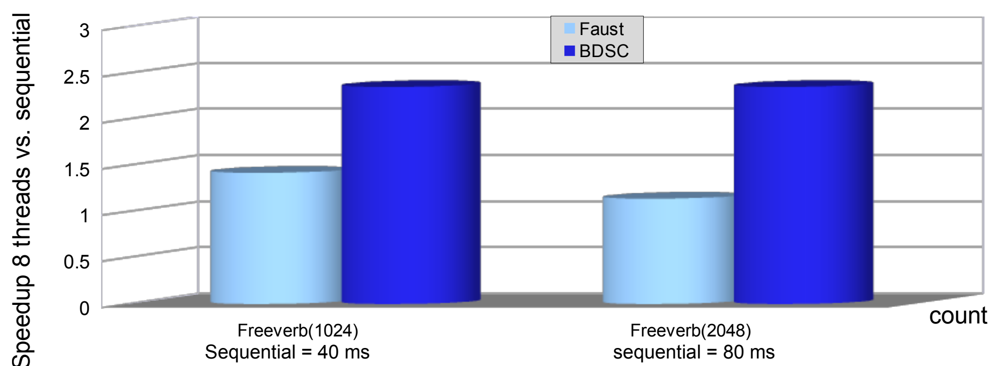


Figure 8.13: Run-time comparison (BDSC vs. Faust for Freeverb)

8.8 Conclusion

This chapter describes the integration of SPIRE and HBDSC within the PIPS compiler infrastructure and their application for the automatic task parallelization of seven sequential programs. We illustrate the impact of this integration using the signal processing benchmark ABF (Adaptive Beam Forming), the image processing benchmark Harris, the SPEC benchmark equake and the NAS parallel benchmark IS on both shared and distributed memory systems. We also provide a comparative study between our task

parallelization implementation in PIPS and that of the audio signal processing language Faust, using two Faust programs: Karplus32 and Freeverb. Experimental results show that BDSC provides more efficient schedules than DSC and Faust scheduler in these cases.

The application of BDSC for the parallelization of the FFT (OpenMP) shows the limits of our task parallelization approach, but we suggest that an interface with streaming languages such as OpenStream could be derived from our work.

Chapter 9

Conclusion

Celui qui passe à côté de la plus belle histoire de sa vie n'aura que l'âge de ses regrets et tous les soupirs du monde ne sauraient bercer son âme... **Yasmina Khadra**

The interest in compiler design for parallel architectures has exploded in the last decade, with the widespread availability of manycores and multiprocessors and the need for efficient parallelism management to profit from these powerful computing resources. There are two main approaches to parallelism management: manual and automatic. For many reasons, practical as well as more fundamental ones, an automatic approach has always been quite appealing. Yet, designing an automatic parallelizing compiler is a big challenge: this is a tricky task, which is still for the most part unsatisfied.

In this thesis, we have proposed a new method to automatically exploit the parallelism present in applications and thus take advantage, with no cost to the programmer, of the performance benefits multiprocessors can provide. At an abstract level, we have thus delegated the "think parallel" mindset to the compiler, which detects parallelism in sequential codes and, automatically, writes equivalent efficient parallel code. To reach that goal, we have developed an automatic task parallelization methodology for compilers: the key characteristics we focus on are resource constraints and static scheduling. It contains all the techniques required to decompose applications into tasks and generate equivalent parallel code, using a generic approach that targets different parallel languages and different architectures. We have applied this extension methodology in an existing software tool, the comprehensive source-to-source compilation platform PIPS.

To conclude this dissertation, we review its main contributions and results, and present future work opportunities.

9.1 Contributions

This thesis contains five main contributions.

HBDSC (Chapter 6)

We have extended the DSC scheduling algorithm to handle simultaneously two resource constraints, namely a bounded amount of memory per processor and a bounded number of processors, which are key parameters when scheduling tasks on actual multiprocessors. We have called this extension

Bounded DSC (BDSC). The practical use of BDSC in a hierarchical manner led to the development of a BDSC-based hierarchical scheduling algorithm (HBDSC).

A standard scheduling algorithm works on a DAG (a sequence of statements). To apply HBDSC on real applications, our approach was to first build a Sequence Dependence DAG (SDG) for each sequence in an application, to build the input of the BDSC algorithm. Then, we used the whole code, presented in form of an AST, to define a hierarchical mapping function, that we call H, to map each sequence statement of the code to its SDG. H is used as input of the HBDSC algorithm.

Since the volume of data used or exchanged by SDG tasks and their execution times are key factors in the BDSC scheduling process, we estimate this information as precisely as possible. We provide new cost models based on execution time complexity estimation and convex polyhedral approximations of array sizes for the labeling of SDG vertices and edges. We look first at the size of array regions to estimate precisely the communication costs and data sizes used by tasks. Then, we compute Ehrhart polynomials that represent the size of the message to communicate and volume of data, in number of bytes. They are then converted into numbers of transfer instruction cycles. Besides, the execution time for each task is computed using a static execution time approach based on a program complexity analysis provided by PIPS. Complexities, in number of cycles, are also represented via polynomials. Finally, we use a static, when possible, or a code instrumentation method to convert these polynomials to the numerical values required by BDSC.

Path Transformer Analysis (Chapter 6)

The cost models that we define in this thesis use affine expressions on program variables. They are used in array regions to estimate the communications and data volumes induced by the execution of statements. We used a set of operations on array regions such as the `regions_intersection` function (see Section 6.3.1). However, one must be careful there, since region operations should be defined with respect to a common memory store. In this thesis, we introduced a new analysis that we call "path transformer" analysis. A path transformer permits to compare array regions of statements originally defined in different memory stores. We present a recursive algorithm for the computation of path transformers between two statements of specific iterations of their enclosing loops (if they exist). This algorithm computes affine relations between program variables at any two points in the program, from the memory store of the first statement to the memory store of the second one, along any sequential execution that links the two memory stores.

SPIRE (Chapter 4)

An automatic parallelization platform should be adapted to different parallel languages. In this thesis, we introduced SPIRE, which is a new and general 3-step extension methodology for mapping any intermediate representation (IR) used in compilation platforms for representing sequential programming constructs to a parallel IR. This extension of an existing IR introduces (1) a parallel execution attribute for each group of statements, (2) a high-level synchronization attribute on each statement and an API for low-level synchronization events and (3) two built-ins for implementing communications in message-passing memory systems.

A formal semantics of SPIRE transformational definitions is specified using a two-tiered approach: a small-step operational semantics for its base parallel concepts and a rewriting mechanism for high-level constructs. The SPIRE methodology is presented via a use case, the intermediate representation of PIPS, a powerful source-to-source compilation infrastructure for Fortran and C. In addition, we apply SPIRE on another IR, namely the one of the widely-used LLVM compilation infrastructure.

Parallel Code Transformations and Generation (Chapter 7)

In order to test the flexibility of the parallelization approach presented in this thesis, we put the first brick of a framework of parallel code transformations and generation using our BDSC- and SPIRE-based hierarchical task parallelization techniques. We provide two parallel transformations of SPIRE-based constructs: first, from unstructured to structured parallelism, for high-level abstraction of parallel constructs, and, second, the conversion of shared memory programs to distributed ones. Moreover, we show the generation of task parallelism and communications at different AST levels: equilevel and hierarchical. Finally, we generate both OpenMP (hierarchical) and MPI (only flat) from the same parallel IR derived from SPIRE.

Implementation and Experimental Results (Chapter 8)

A prototype of the algorithms presented in this thesis has been implemented in the PIPS source-to-source compilation framework. It includes (1) an automatic parallelization technique for programs encoded in SPIRE(PIPS IR) and based on BDSC, HBDSC, SDG, the cost models and the path transformer, plus DSC for comparison purposes; (2) a SPIRE-based parallel code generation algorithm targeting two parallel languages, namely OpenMP and MPI; and (3) two SPIRE-based parallel code transformations, from unstructured to structured code and from shared to distributed memory code.

We apply our parallelization technique on scientific applications. We provide performance measurements for our parallelization approach, based on five significant programs: the image and signal processing benchmarks

Harris and ABF, the SPEC2001 benchmark equake, the NAS parallel benchmark IS, and the FFT, targeting both shared and distributed memory architectures. We use their automatic translations into two parallel languages: OpenMP and MPI. Finally, we provide a comparative study between our task parallelization implementation in PIPS and that of the audio signal processing language Faust, using two Faust applications: Karplus32 and Freeverb. Our results illustrate the ability of SPIRE(PIPS IR) to efficiently express parallelism present in scientific applications. Moreover, our experimental data suggests that our parallelization methodology is able to provide parallel codes that encode a significant amount of the already-present intrinsic parallelism.

9.2 Future Work

The automatic parallelization problem is a set of many tricky subproblems, as illustrated along this thesis or in the summary of contributions presented above. Many new ideas and opportunities for further research have popped up all along this thesis.

SPIRE

SPIRE is a “work in progress” for the extension of sequential IRs to completely fit multiple languages and different models. Future work could address the representation via SPIRE of the PGAS memory model, of other parallel execution paradigms such as speculation and of more programming features such as exceptions. Another interesting question could be to see how many of the existing program transformations performed by compilers using sequential IRs can be preserved when these are extended via SPIRE. Finally, the formal semantics introduced here could possibly be used in the future for proving the legality of transformations of parallel codes.

HBDSC Scheduling

HBDSC assumes that all processors are homogeneous. It does not handle heterogeneous devices efficiently. However, one can probably fit in easily these architectures by adding another dimension to our cost models. We suggest to add the parameter “processor type” to all the functions part of the cost models such as `task_time` (we have to define its time if mapped on such an such processor) or `edge_cost` (we have to define its cost when the two tasks of the edge are mapped on such and such processor). Moreover, each processor κ_i has its own memory M_i. Thus, our parallelization process could target subsequently an adapted language that targets heterogeneous architectures such as StarSs [21]. StarSs introduces an extension to the task mechanism in OpenMP 3.0. It adds many features to the `#pragma omp`

`task` construct, namely (1) dependences between tasks using the `input`, `output` and `inout` clauses and (2) targeting of a task to hardware accelerators using `#pragma omp target device(device-name-list)`, knowing that an accelerator can be an SPE, a GPU...

Moreover, it might be interesting to try to find a more efficient hierarchical processor allocation strategy, which would address load balancing and task granularity issues, in order to yield an even better HBDSC-based parallelization process.

Cost Models

Currently, there are some cases where our cost models cannot provide precise or even approximate information. When the array is a pointer (for example with the declaration `float *A` while `A` is used as an array), the compiler has no information about the size of the array. Currently, for such cases, we have to change manually the programmer's code to help the compiler. Moreover, currently, the complexity estimation of a while loop is not computed, but there is already on-going work on this issue (in particular at MINES ParisTech), based on transformer analysis.

Parallel Code Transformations and Generation

Future work could address more transformations for parallel languages (*à la* [84]) encoded in SPIRE. Moreover, the optimization of communications in the generated code, by eliminating redundant communications and aggregating small messages to larger ones, is an important issue since our current implementation communicates more data than necessary.

In our implementation of MPI code generation from SPIRE code, we left the implementation of hierarchical MPI as future work. Besides, another type of hierarchical MPI can be obtained by merging MPI and OpenMP (hybrid programming). However, since we use in this thesis a hierarchical approach in both HBDSC and SPIRE generation, OpenMP can be seen, in this case, as a sub-hierarchical model of MPI, i.e., OpenMP tasks could be enclosed within MPI ones.

وتعظم في عينِ الصغيرِ صِغارها وتصغر في عينِ العَظيمِ العظائمُ[1]

[1] *In the eye of the small, small things are huge, but, for the great souls, huge things appear small.* Al-Mutanabbi

Conclusion (*en français*)

Celui qui passe à côté de la plus belle histoire de sa vie n'aura que l'âge de ses regrets et tous les soupirs du monde ne sauraient bercer son âme... **Yasmina Khadra**

L'intérêt pour la compilation visant les architectures parallèles a explosé lors de la dernière décennie, du fait de la disponibilité maintenant commune de multiprocesseurs à plusieurs coeurs et de la nécessité d'une prise en compte efficace du parallélisme des tâches afin de profiter de ces ressources de calcul. Cette gestion du parallélisme des programmes peut être envisagée essentiellement de deux façons : manuelle ou automatique. Pour diverses raisons, aussi bien pratiques que plus fondamentales, l'approche automatique a toujours paru particulièrement attrayante. Toutefois, la conception d'un système logiciel permettant une parallélisation automatique de programmes est un défi immense ; cette automatisation est une opération délicate, et reste encore largement insatisfaisante.

Dans cette thèse, nous avons proposé une méthode nouvelle pour permettre d'exploiter automatiquement le parallélisme présent dans les applications et de profiter, en conséquence, de manière transparente pour le programmeur, des capacités offertes par les multiprocesseurs. D'un point de vue abstrait, nous proposons de déléguer le ≪ penser parallèle ≫ que requiert la gestion du parallélisme au seul compilateur, chargé de détecter le parallélisme dans des codes séquentiels et d'écrire, automatiquement, un code parallèle équivalent. Pour ce faire, avons développé une méthodologie de parallélisation automatique des tâches pour les compilateurs. Les principales caractéristiques sur lesquelles nous avons concentré nos efforts sont les contraintes de ressources et l'ordonnancement statique. Cette méthode contient toutes les techniques nécessaires pour décomposer les applications en tâches et générer un code parallèle équivalent, et ce en utilisant une approche générique qui vise différents langages et architectures parallèles. Afin de valider en pratique ces propositions, nous avons incorporé cette méthode comme extension dans un outil logiciel existant, à savoir la plate-forme de compilation source à source complète PIPS.

Pour conclure cette thèse, nous en passons en revue les principales contributions et résultats, et présentons certaines futures opportunités de travail.

Contributions

Cette thèse comprend cinq contributions principales.

HBDSC (Chapitre 6)

Nous avons étendu l'algorithme d'ordonnancement DSC pour gérer simultanément deux contraintes de ressources, à savoir un montant bornée de mémoire par processeur et un nombre borné de processeurs ; ce sont des paramètres clés lors de la planification de tâches sur les multiprocesseurs réels. Nous avons appelé cette extension bornée de DSC (BDSC). L'utilisation pratique de BDSC de manière hiérarchique a conduit à l'élaboration d'un algorithme d'ordonnancement hiérarchique fondé sur BDSC (HBDSC).

Un algorithme standard d'ordonnancement opère traditionnellement sur un DAG, représentant une séquence d'instructions. Pour appliquer HBDSC sur une application donnée, notre approche a été d'utiliser les dépendances de donnée pour construire d'abord un DAG spécifique pour chaque séquence (SDG), qui servira d'entrée pour l'algorithme BDSC. Ensuite, nous avons utilisé le code complet de l'application, présenté sous la forme d'un AST, pour définir une fonction d'association hiérarchique ou *mapping*, que nous appelons H, pour lié chaque séquence du code à son SDG. H est utilisé comme entrée de l'algorithme de HBDSC .

Comme le volume de données utilisées ou échangées par les tâches d'un SDG et leurs temps d'exécution sont des facteurs clés dans le processus de planification BDSC, il convient d'estimer ces informations de manière aussi précise que possible. Nous élaborons de nouveaux modèles de coûts fondés sur l'estimation de la complexité algorithmique en temps du code et des approximations convexes polyédriques de la taille des tableaux ; ces données sont utilisées pour réaliser l'étiquetage des sommets et des arêtes des SDG. Nous tirons tout d'abord partie de la taille des régions de tableau et des formats des données utilisées par les tâches pour estimer précisément les coûts de communication. Puis, on détermine les polynômes de Ehrhart qui représentent la taille du message à transmettre et le volume de données, en nombre d'octets. Ceux-ci sont ensuite convertis en nombres de cycles d'instructions de transfert. Par ailleurs, le temps d'exécution de chaque tâche est calculé en utilisant une approche statique fondée sur une analyse de la complexité du programme fourni par PIPS. Ces complexités, en nombre de cycles, sont également représentées par des polynômes . Enfin, nous utilisons une méthode statique, lorsque cela est possible, ou une méthode d'instrumentation de code pour convertir ces polynômes en les valeurs numériques nécessaires pour faire tourner BDSC.

Analyse de *Path Transformer* (Chapitre 6)

Les modèles de coûts que nous définissons dans cette thèse utilisent des expressions affines sur les variables du programme. Ces relations sont utilisées pour définir les régions de tableau ainsi que pour estimer les communications et les volumes de données induits par l'exécution des instructions.

Nous avons utilisé un ensemble d'opérations sur les régions de tableau telles que la fonction d'intersection des régions (voir la section 6.3.1). Cependant, il faut être prudent lors de ces utilisations, car les opérations sur régions doivent être définies par rapport à un espace mémoire commun. Dans cette thèse, nous avons introduit une nouvelle analyse, que nous appelons "transformeur de chemin". Un transformeur de chemin permet de comparer les régions de tableau définies à l'origine dans des espaces mémoire différents. Nous présentons un algorithme récursif pour le calcul des transformeurs de chemin entre deux instructions, prenant en paramètre des informations supplémentaires concernant les itérations des boucles imbriquantes (si elles existent). Cet algorithme calcule une relation affine sur les variables d'un programme entre deux points d'instructions, dans l'état mémoire du premier vers l'état mémoire du second, et ce le long de toute exécution séquentielle qui relie les deux états mémoire.

SPIRE (Chapitre 4)

Une plate-forme de parallélisation automatique doit être adaptée aux différents langages parallèles. Dans cette thèse, nous avons introduit SPIRE, une nouvelle méthodologie générique permettant d'étendre, en trois étapes, une représentation intermédiaire (RI) de programmation séquentiels utilisée dans une plate-forme de compilation en une RI parallèle. Cette extension d'une RI existante introduit (1) un attribut d'exécution parallèle pour chaque groupe d'instructions, (2) un attribut de synchronisation de haut niveau sur chaque instruction et une API pour les événements de synchronisation bas niveau et (3) deux fonctions prédéfinies pour spécifier les communications dans les systèmes à mémoire distribuée avec envoi de messages.

Une sémantique formelle de SPIRE est fournie, en utilisant une approche à deux niveaux : une sémantique opérationnelle em small-step pour les concepts de base parallèles et un mécanisme de réécriture pour les constructions de plus haut niveau. L'application de la méthodologie SPIRE est illustrée par un cas d'utilisation, la représentation intermédiaire de PIPS, une infrastructure puissante source-à-source de compilation pour Fortran et C. En outre, nous appliquons SPIRE sur une autre RI, à savoir celle de l'infrastructure de compilation LLVM, largement utilisée.

Transformations et génération de code parallèle (Chapitre 7)

Afin de tester la souplesse de l'approche de parallélisation présentée dans cette thèse, nous avons mis la première brique vers la construction d'un cadre général de transformations et génération de codes parallèles en utilisant notre technique de parallélisation de tâchs hiérarchiques fondée sur SPIRE et BDSC. Nous proposons deux transformations parallèles des constructions de base de SPIRE : d'abord, une structuration du parallélisme

sous forme de constructions de haut niveau et, ensuite, une conversion de programmes pour mémoire partagée vers une mémoire distribuée. En outre, nous indiquons comment effectuer une génération avec parallélisme de tâches et communications à différents niveaux d'AST: plat et hiérarchique. Enfin, nous générons du code parallèle en utilisant deux langages, OpenMP (hiérarchique) et MPI (seulement plat), à partir de la même RI parallèle dérivée de SPIRE.

Implémentation et résultats d'expérimentation (Chapitre 8)

Un prototype des algorithmes présentés dans cette thèse a été implanté dans le compilateur source-à-source PIPS. Il contient (1) une méthode de parallélisation automatique pour des programmes encodés en SPIRE(PIPS IR) et fondée sur BDSC, HBDSC, SDG, les modèles de coûts et le transformeur de chemin, plus DSC à des fins de comparaison ; (2) des algorithmes de génération de code parallèle à partir de SPIRE et ciblant deux langages parallèles, OpenMP et MPI ; et (3) deux transformations de code parallèle fondées sur SPIRE, pour transformer un code non structuré en un code structuré et passer de mémoire partagée à mémoire distribuée.

Nous appliquons notre technique de parallélisation sur des applications scientifiques. Nous donnons les mesures de performance de notre approche de parallélisation, ciblant à la fois des architectures à mémoire partagée et distribuée, sur la base de cinq programmes importants : deux benchmarks de traitement d'image et de traitement du signal, Harris et ABF ; SPEC2001 equake ; NAS benchmark IS ; et la FFT. Nous utilisons leur traduction essentiellement automatique en deux langages parallèles : OpenMP et MPI. Enfin, nous proposons une étude comparative entre notre implémentation de parallélisation de tâches dans PIPS et celle du langage de traitement du signal audio Faust, en utilisant deux programmes Faust : Karplus32 et Freeverb. Nos résultats illustrent la capacité de SPIRE(PIPS IR) à exprimer efficacement le parallélisme présent dans les applications scientifiques. En outre, nos données expérimentales suggèrent que notre méthodologie de parallélisation est en mesure de fournir des codes parallèles qui contiennent une part importante du parallélisme intrinsèque déjà présent.

Perspectives

Le problème de la parallélisation automatique est un ensemble de plusieurs sous-problèmes, comme on l'a vu tout au long de cette thèse ou dans le résumé des contributions présenté ci-dessus. De nombreuses idées et de nouvelles opportunités pour de nouvelles recherches ont surgi tout au long de ce travail.

SPIRE

SPIRE est un "travail en cours" pour étendre des RI séquentielles afin de les adapter aussi complètement que possible aux différents langages et modèles. Les travaux à venir pourraient s'intéresser à la représentation, via SPIRE, du modèle de mémoire PGAS ou d'autres paradigmes d'exécution parallèle comme par exemple la spéculation, tout en prenant en compte plus de fonctionnalités de programmation comme les exceptions. Une autre question intéressante serait de déterminer dans quelle mesure les transformations de programmes existantes réalisées par les compilateurs en utilisant une RI séquentielle peuvent être préservées lorsque celles-ci sont étendues par SPIRE. Enfin, la sémantique formelle introduite dans ce travail pourrait être utilisée dans le futur pour tenter de prouver la légalité des transformations effectuées sur des codes parallèles.

Ordonnancement via HBDSC

HBDSC suppose que tous les processeurs sont homogènes. Il ne gère pas efficacement les processeurs hétérogènes. Cependant, il est probablement facilement adaptable à ces architectures en ajoutant une autre dimension à nos modèles de coûts. Nous suggérons d'ajouter le paramètre "type de processeur" à toutes les fonctions essentielles dans la modélisation des coûts tels que `task_time` (il faudrait définir le coût d'une tâche si elle est affectée à tel ou tel processeur) ou `edge_cost` (il faudrait définir le coût de communication lorsque les deux tâches adjacentes sont affectées à tel ou tel processeur). De plus, chaque processeur κ_i a sa propre mémoire M_i. Ainsi , notre processus de parallélisation pourrait cibler par la suite un langage adapté aux architectures hétérogènes tel que StarSs [21]. StarSs introduit une extension du mécanisme de tâche de OpenMP 3.0. Il ajoute de nombreuses fonctionnalités à la construction `#pragme omp task`, à savoir : (1) les dépendances entre tâches, en utilisant les clauses `input`, `output` et `inout`, (2) l'affectation d'une tâche à des accélérateurs, en utilisant `#pragma omp target device(device-name-list)`, sachant qu'un accélérateur peut être un SPE, un GPU ...

En outre, il pourrait être intéressant d'essayer de trouver une stratégie d'allocation hiérarchique plus efficace des processeurs que celle présentée ici ; elle pourrait aborder l'équilibrage de charge et les questions de granularité de tâche, afin de conduire à un meilleur processus de parallélisation fondé sur HBDSC.

Modèles de coût

Actuellement, il existe certains cas pour lesquels nos modèles de coûts ne peuvent pas fournir des informations précises ou même suffisamment approximatives. Lorsqu'un tableau est accédé via un pointeur (par exemple

avec la déclaration `float *A`, alors que `A` est utilisé comme un tableau), le compilateur n'a aucune information pertinente concernant la taille du tableau. Actuellement, dans de tels cas, nous devons modifier manuellement le code du programme pour aider le compilateur. En outre, actuellement, l'estimation de la complexité d'une boucle `while` n'est pas effectuée de manière fine, mais des travaux en cours (notamment à MINES ParisTech) fondés sur l'analyse des transformers laissent espérer des avancées dans ce domaine.

Transformations et génération de code parallèle

Les travaux futurs pourraient aborder plusieurs transformations spécifiques aux langages parallèles(*à la* [84]) encodées en SPIRE. De plus, l'optimisation des communications dans le code généré, en éliminant les communications redondantes et l'agrégation des petits messages en de plus grands, est une question importante, puisque notre prototype actuel communique plus de données que nécessaire.

Dans notre implémentation de génération de MPI à partir du code SPIRE, nous avons laissé l'implémentation de MPI hiérarchique pour les travaux futurs. Par ailleurs, un autre type de MPI hiérarchique peut être obtenu par la fusion de MPI et OpenMP (programmation hybride). Toutefois, étant donné que nous utilisons dans cette thèse une approche hiérarchique dans à la fois HBDSC et la génération de SPIRE, OpenMP peut être vu, dans ce cas, comme un modèle sous-hiérarchique de MPI, c'est à dire que les tâches OpenMP pourraient être incluses dans celles de MPI.

وتعظـم فـي عيـنِ الصغـيرِ صِغارها   وتصغـر فـي عيـنِ العَظيـمِ العظـائمُ[2]

[2] *Aux yeux des petits, les petites choses sont immenses ; pour les grandes âmes, les grandes choses sont si petites.* Al-Mutanabbi

Bibliography

[1] The Mandelbrot Set. http://warp.povusers.org/Mandelbrot/.

[2] *Cilk 5.4.6 Reference Manual.* Supercomputing Technologies Group, MIT Laboratory for Computer Science, http://supertech.lcs.mit.edu/cilk, 1998.

[3] *MPI: A Message-Passing Interface Standard.* http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Sep 2012.

[4] *OpenMP Application Program Interface.* http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf, Mar. 2013.

[5] *X10 Language Specification*, Feb. 2013.

[6] *Chapel Language Specification 0.796.* Cray Inc., 901 Fifth Avenue, Suite 1000, Seattle, WA 98164, Oct 21, 2010.

[7] Faust, an Audio Signal Processing Language. http://faust.grame.fr/.

[8] StreamIt, a Programming Language and a Compilation Infrastructure. http:groups.csail.mit.edu/cag/streamit/.

[9] B. Ackland, A. Anesko, D. Brinthaupt, S. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. Nicol, J. O'Neill, J. Othmer, E. Sackinger, K. Singh, J. Sweet, C. Terman, and J. Williams. A single-chip, 1.6-billion, 16-b mac/s multiprocessor dsp. *Solid-State Circuits, IEEE Journal of*, 35(3):412 –424, Mar 2000.

[10] T. L. Adam, K. M. Chandy, and J. R. Dickson. A Comparison of List Schedules For Parallel Processing Systems. *Commun. ACM*, 17(12):685–690, Dec. 1974.

[11] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1996.

[12] S. Alam, R. Barrett, J. Kuehn, and S. Poole. Performance Characterization of a Hierarchical MPI Implementation on Large-scale Distributed-memory Platforms. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 132–139, 2009.

[13] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, Jr, and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[14] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, Oct. 1987.

[15] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 126–138, New York, NY, USA, 1993. ACM.

[16] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *Solid-State Circuits Newsletter, IEEE*, 12(3):19 –20, summer 2007.

[17] M. Amini, F. Coelho, F. Irigoin, and R. Keryell. Static Compilation Analysis for Host-Accelerator Communication Optimization. In *24th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Fort Collins, Colorado, USA, Sep 2011. Also Technical Report MINES ParisTech A/476/CRI.

[18] C. Ancourt, F. Coelho, and F. Irigoin. A Modular Static Analysis Approach to Affine Loop Invariants Detection. *Electr. Notes Theor. Comput. Sci.*, 267(1):3–16, 2010.

[19] C. Ancourt, B. Creusillet, F. Coelho, F. Irigoin, P. Jouvelot, and R. Keryell. PIPS: a Workbench for Interprocedural Program Analyses and Parallelization. In *Meeting on data parallel languages and compilers for portable parallel computing*, 1994.

[20] C. Ancourt and F. Irigoin. Scanning Polyhedra with DO Loops. In *Proceedings of the third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 39–50, New York, NY, USA, 1991. ACM.

[21] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. González, F. D. Igual, D. Jiménez-González, and J. Labarta. Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *International Journal of Parallel Programming*, pages 440–459, 2010.

[22] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu. Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers. *Computer Methods in Applied Mechanics and Engineering*, 152(1–2):85–102, 1998.

[23] A. Basumallik, S.-J. Min, and R. Eigenmann. Programming Distributed Memory Sytems Using OpenMP. In *Parallel and Distributed*

Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1–8, 2007.

[24] N. Benoit and S. Louise. Kimble: a Hierarchical Intermediate Representation for Multi-Grain Parallelism. In F. Bouchez, S. Hack, and E. Visser, editors, *Proceedings of the Workshop on Intermediate Representations*, pages 21–28, 2011.

[25] G. Blake, R. Dreslinski, and T. Mudge. A Survey of Multicore Processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, Nov. 2009.

[26] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[27] U. Bondhugula. Automatic Distributed Memory Code Generation using the Polyhedral Framework. Technical Report IISc-CSA-TR-2011-3, Indian Institute of Science, Bangalore, 2011.

[28] C. T. Brown, L. S. Liebovitch, and R. Glendon. Lévy Flights in Dobe Ju/'hoansi Foraging Patterns. *Human Ecology*, 35(1):129–138, 2007.

[29] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[30] V. Cavé, J. Zhao, and V. Sarkar. Habanero-Java: the New Adventures of Old X10. In *9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, Aug 2011.

[31] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge. Stream Compilation for Real-Time Embedded Multicore Systems. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 210–220, Washington, DC, USA, 2009.

[32] B. Cirou and E. Jeannot. Triplet: A Clustering Scheduling Algorithm for Heterogeneous Systems. In *Parallel Processing Workshops, 2001. International Conference on*, pages 231–236, 2001.

[33] F. Coelho, P. Jouvelot, C. Ancourt, and F. Irigoin. Data and Process Abstraction in PIPS Internal Representation. In F. Bouchez, S. Hack, and E. Visser, editors, *Proceedings of the Workshop on Intermediate Representations*, pages 77–84, 2011.

[34] T. U. Consortium. `http://upc.gwu.edu/documentation.html`, Jun 2005.

[35] B. Creusillet. *Analyses de Régions de Tableaux et Applications*. PhD thesis, MINES ParisTech, A/295/CRI, Décembre 1996.

[36] B. Creusillet and F. Irigoin. Interprocedural Array Region Analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.

[37] E. Cuevas, A. Garcia, F. J. J.Fernandez, R. J. Gadea, and J. Cordon. Importance of Simulations for Nuclear and Aeronautical Inspections with Ultrasonic and Eddy Current Testing. Simulation in NDT, Online Workshop in `www.ndt.net`, Sep 2010.

[38] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[39] M. I. Daoud and N. N. Kharma. GATS 1.0: a Novel GA-based Scheduling Algorithm for Task Scheduling on Heterogeneous Processor Nets. In *GECCO'05*, pages 2209–2210, 2005.

[40] J. B. Dennis, G. R. Gao, and K. W. Todd. Modeling The Weather With a Data Flow Supercomputer. *IEEE Trans. Computers*, pages 592–603, 1984.

[41] E. W. Dijkstra. Re: "Formal Derivation of Strongly Correct Parallel Programs" by Axel van Lamsweerde and M.Sintzoff. circulated privately, 1977.

[42] E. Ehrhart. Polynômes arithmétiques et méthode de polyèdres en combinatoire. *International Series of Numerical Mathematics*, page 35, 1977.

[43] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine^{TM} Architecture. *IBM Syst. J.*, 45(1):59–84, Jan. 2006.

[44] P. Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20, 1991.

[45] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph And Its Use In Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[46] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, Sep. 1972.

[47] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[48] A. Gerasoulis, S. Venugopal, and T. Yang. Clustering Task Graphs for Message Passing Architectures. *SIGARCH Comput. Archit. News*, 18(3b):447–456, June 1990.

[49] M. Girkar and C. D. Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE Trans. Parallel Distrib. Syst.*, 3:166–178, Mar 1992.

[50] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Message-Passing Code Generation for Non-rectangular Tiling Transformations. *Parallel Computing*, 32, 2006.

[51] Graphviz. Graph Visualization Software. `http://www.graphviz.org`.

[52] L. Griffiths. A Simple Adaptive Algorithm for Real-Time Processing in Antenna Arrays. *Proceedings of the IEEE*, 57:1696 – 1704, 1969.

[53] R. Habel, F. Silber-Chaussumier, and F. Irigoin. Generating Efficient Parallel Programs for Distributed Memory Systems. Technical Report CRI/A-523, MINES ParisTech and Télécom SudParis, 2013.

[54] C. Harris and M. Stephens. A Combined Corner and Edge Detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[55] M. J. Harrold, B. Malloy, and G. Rothermel. Efficient Construction of Program Dependence Graphs. *SIGSOFT Softw. Eng. Notes*, 18(3):160–170, July 1993.

[56] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, Feb.

[57] A. Hurson, J. T. Lim, K. M. Kavi, and B. Lee. Parallelization of DOALL and DOACROSS Loops – a Survey . In M. V. Zelkowitz,

editor, *Emphasizing Parallel Programming Techniques*, volume 45 of *Advances in Computers*, pages 53 – 103. Elsevier, 1997.

[58] Insieme. Insieme - an Optimization System for OpenMP, MPI and OpenCL Programs. `http://www.dps.uibk.ac.at/insieme/architecture.html`, 2011.

[59] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *ICS*, pages 244–251, 1991.

[60] K. Jainandunsing. Optimal Partitioning Scheme for Wavefront/Systolic Array Processors. In *Proceedings of IEEE Symposium on Circuits and Systems*, pages 940–943, 1986.

[61] P. Jouvelot and R. Triolet. Newgen: A Language Independent Program Generator. Technical report, CRI/A-191, MINES ParisTech, Jul 1989.

[62] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 1998.

[63] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. A Multi-Grain Parallelizing Compilation Scheme for OSCAR (Optimally Scheduled Advanced Multiprocessor). In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 283–297, London, UK, 1992. Springer-Verlag.

[64] A. Kayi and T. El-Ghazawi. PGAS Languages. `wsdmhp09.hpcl.gwu.edu/kayi.pdf`, 2009.

[65] P. Kenyon, P. Agrawal, and S. Seth. High-Level Microprogramming: an Optimizing C Compiler for a Processing Element of a CAD Accelerator. In *Microprogramming and Microarchitecture. Micro 23. Proceedings of the 23rd Annual Workshop and Symposium., Workshop on*, pages 97–106, nov 1990.

[66] D. Khaldi, P. Jouvelot, and C. Ancourt. Parallelizing with BDSC, a Resource-Constrained Scheduling Algorithm for Shared and Distributed Memory Systems. Technical Report CRI/A-499 (Submitted to *Parallel Computing*), MINES ParisTech, 2012.

[67] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages. In H. Kasahara and K. Kimura, editors, *LCPC*, volume

7760 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 2012.

[68] D. Khaldi, P. Jouvelot, F. Irigoin, and C. Ancourt. SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension. In *Proceedings of the 17th Workshop on Compilers for Parallel Computing*, CPC'13, 2013.

[69] M. A. Khan. Scheduling for Heterogeneous Systems using Constrained Critical Paths. *Parallel Computing*, 38(4–5):175 – 193, 2012.

[70] Khronos OpenCL Working Group. *The OpenCL Specification*, Nov. 2012.

[71] B. Kruatrachue and T. G. Lewis. Duplication Scheduling Heuristics (DSH): A New Precedence Task Scheduler for Parallel Processor Systems. Oregon State University, Corvallis, OR. 1987.

[72] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen. Atomic Vector Operations on Chip Multiprocessors. *SIGARCH Comput. Archit. News*, 36(3):441–452, June 2008.

[73] Y.-K. Kwok and I. Ahmad. Benchmarking the Task Graph Scheduling Algorithms. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pages 531–537, 1998.

[74] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31:406–471, Dec 1999.

[75] J. Larus and C. Kozyrakis. Transactional Memory. *Commun. ACM*, 51:80–88, Jul 2008.

[76] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[77] The LLVM Development Team, `http://llvm.org/docs/LangRef.html`. *The LLVM Reference Manual (Version 2.6)*, Mar. 2010.

[78] V. Maisonneuve. Convex Invariant Refinement by Control Node Splitting: a Heuristic Approach. *Electron. Notes Theor. Comput. Sci.*, 288:49–59, Dec. 2012.

[79] J. Merrill. GENERIC and GIMPLE: a New Tree Representation for Entire Functions. In *GCC developers summit 2003*, pages 171–180, 2003.

[80] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier. STEP: a Distributed OpenMP for Coarse-Grain Parallelism Tool. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 83–99, Berlin, Heidelberg, 2008. Springer-Verlag.

[81] D. I. Moldovan and J. A. B. Fortes. Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays. *IEEE Trans. Comput.*, 35(1):1–12, Jan. 1986.

[82] A. Moody, D. Ahn, and B. Supinski. Exascale Algorithms for Generalized MPI_Comm_split. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 9–18. Springer Berlin Heidelberg, 2011.

[83] G. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82 –85, jan. 1998.

[84] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 35(1):3:1–3:48, Apr. 2013.

[85] C. J. Newburn and J. P. Shen. Automatic Partitioning of Signal Processing Programs for Symmetric Multiprocessors. In *IEEE PACT*, pages 269–280. IEEE Computer Society, 1996.

[86] D. Novillo. OpenMP and Automatic Parallelization in GCC. In *the Proceedings of the GCC Developers Summit*, Jun 2006.

[87] NPB. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`.

[88] Y. Orlarey, D. Fober, and S. Letz. Adding Automatic Parallelization to Faust. In *Linux Audio Conference*, 2009.

[89] D. A. Padua, editor. *Encyclopedia of Parallel Computing*. Springer, 2011.

[90] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29:1184–1201, Dec 1986.

[91] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil. PLASMA: Portable Programming for SIMD Heterogeneous Accelerators. In

Workshop on Language, Compiler, and Architecture Support for GPGPU, held in conjunction with HPCA/PPoPP 2010, Bangalore, India, Jan 9, 2010.

[92] PolyLib. A Library of Polyhedral Functions. `http://icps.u-strasbg.fr/polylib/`.

[93] A. Pop and A. Cohen. A Stream-Computing Extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 5–14, New York, NY, USA, 2011. ACM.

[94] A. Pop and A. Cohen. OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, Jan. 2013.

[95] T. Saidani. *Optimisation multi-niveau d'une application de traitement d'images sur machines parallèles.* PhD thesis, Université Paris-Sud, Nov. 2012.

[96] T. Saidani, L. Lacassagne, J. Falcou, C. Tadonki, and S. Bouaziz. Parallelization Schemes for Memory Optimization on the Cell Processor: A Case Study on the Harris Corner Detector. In P. Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers III*, volume 6590 of *Lecture Notes in Computer Science*, pages 177–200. Springer Berlin Heidelberg, 2011.

[97] V. Sarkar. Synchronization Using Counting Semaphores. In *ICS'88*, pages 627–637, 1988.

[98] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, USA, 1989.

[99] V. Sarkar. COMP 322: Principles of Parallel Programming. `http://www.cs.rice.edu/~vs3/PDF/comp322-lec9-f09-v4.pdf`, 2009.

[100] V. Sarkar and B. Simons. Parallel Program Graphs and their Classification. In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *LCPC*, volume 768 of *Lecture Notes in Computer Science*, pages 633–655. Springer, 1993.

[101] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *Micro, IEEE*, 29(1):10–21, 2009.

[102] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: A Unified Deadlock-Free Construct for Collective and Point-To-Point

Synchronization. In *ICS'08*, pages 277–288, New York, NY, USA, 2008. ACM.

[103] M. Solar. A Scheduling Algorithm to Optimize Parallel Processes. In *Chilean Computer Science Society, 2008. SCCC '08. International Conference of the*, pages 73–78, 2008.

[104] M. Solar and M. Inostroza. A Scheduling Algorithm to Optimize Real-World Applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04) - Volume 7*, ICDCSW '04, pages 858–862, Washington, DC, USA, 2004. IEEE Computer Society.

[105] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, Nov. 2001.

[106] Thales. THALES Group. `http://www.thalesgroup.com`.

[107] H. Topcuouglu, S. Hariri, and M.-y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, Mar. 2002.

[108] S. Verdoolaege, A. Cohen, and A. Beletska. Transitive Closures of Affine Integer Tuple Relations and Their Overapproximations. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 216–232, Berlin, Heidelberg, 2011. Springer-Verlag.

[109] M.-Y. Wu and D. D. Gajski. Hypertool: A Programming Aid For Message-Passing Systems. *IEEE Trans. on Parallel and Distributed Systems*, 1:330–343, 1990.

[110] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 428–437, New York, NY, USA, 1992. ACM.

[111] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9):951–967, 1994.

[112] X.-S. Yang and S. Deb. Cuckoo Search via Levy Flights. In *Nature Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 210–214, 2009.

[113] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, W. Michael, and T. Wen. Productivity

and Performance Using Partitioned Global Address Space Languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 24–32, New York, NY, USA, 2007. ACM.

[114] J. Zhao and V. Sarkar. Intermediate Language Extensions for Parallelism. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 329–340, New York, NY, USA, 2011. ACM.

Parallélisation automatique et statique de tâches sous contraintes de ressources – une approche générique –

Résumé : Le but de cette thèse est d'exploiter efficacement le parallélisme présent dans les applications informatiques séquentielles afin de bénéficier des performances fournies par les multiprocesseurs, en utilisant une nouvelle méthodologie pour la parallélisation automatique des tâches au sein des compilateurs. Les caractéristiques clés de notre approche sont la prise en compte des contraintes de ressources et le caractère statique de l'ordonnancement des tâches. Notre méthodologie contient les techniques nécessaires pour la décomposition des applications en tâches et la génération de code parallèle équivalent, en utilisant une approche générique qui vise différents langages et architectures parallèles. Nous implémentons cette méthodologie dans le compilateur source-à-source PIPS. Cette thèse répond principalement à trois questions. Primo, comme l'extraction du parallélisme de tâches des codes séquentiels est un problème d'ordonnancement, nous concevons et implémentons un algorithme d'ordonnancement efficace, que nous nommons BDSC, pour la détection du parallélisme ; le résultat est un SDG ordonnancé, qui est une nouvelle structure de données de graphe de tâches. Secondo, nous proposons une nouvelle extension générique des représentations intermédiaires séquentielles en des représentations intermédiaires parallèles que nous nommons SPIRE, pour la représentation des codes parallèles. Enfin, nous développons, en utilisant BDSC et SPIRE, un générateur de code que nous intégrons dans PIPS. Ce générateur de code cible les systèmes à mémoire partagée et à mémoire distribuée via des codes OpenMP et MPI générés automatiquement.
Mots clés : Parallélisation automatique, représentation intermédiaire parallèle, ordonnancement, PIPS

Automatic Resource-Constrained Static Task Parallelization – A Generic Approach –

Abstract: This thesis intends to show how to efficiently exploit the parallelism present in applications in order to enjoy the performance benefits that multiprocessors can provide, using a new automatic task parallelization methodology for compilers. The key characteristics we focus on are resource constraints and static scheduling. This methodology includes the techniques required to decompose applications into tasks and generate equivalent parallel code, using a generic approach that targets both different parallel languages and architectures. We apply this methodology in the existing tool PIPS, a comprehensive source-to-source compilation platform.

This thesis mainly focuses on three issues. First, since extracting task parallelism from sequential codes is a scheduling problem, we design and implement an efficient, automatic scheduling algorithm called BDSC for parallelism detection; the result is a scheduled SDG, a new task graph data structure. In a second step, we design a new generic parallel intermediate representation extension called SPIRE, in which parallelized code may be expressed. Finally, we wrap up our goal of automatic parallelization in a new BDSC- and SPIRE-based parallel code generator, which is integrated within the PIPS compiler framework. It targets both shared and distributed memory systems using automatically generated OpenMP and MPI code.
Keywords: Automatic parallelization, parallel intermediate representation, scheduling, PIPS

MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY