© 2015 by Adam R. Smith. All rights reserved.

THE PARALLEL INTERMEDIATE LANGUAGE

BY

ADAM RANDALL SMITH

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor David Padua, Chair, Director of Research
Professor Laxmikant Kale
Professor Wen-Mei Hwu
Doctor Wilfred Pinfold, Concurrent Systems LLC

Abstract

The next challenge in the evolution of supercomputers will be the transition to exascale systems. However, while the move from terascale to petascale processing was considered evolutionary, it is widely believed that the leap to exascale supercomputers will require revolutionary advances. Simply scaling up current technology will not work. The projections for the exascale systems indicate that applications may have to support up to a billion separate threads to efficiently use the hardware, while the amount of memory per arithmetic functional unit will drop significantly. This implies the need for exploiting fine-grain parallelism with a programming model other than the currently used message passing or coarse-grain threads. As a response, the programming community is exploring data-driven runtimes. However, in order to utilize the new runtime systems, users will either need to rewrite all of their applications by hand in the new languages, or be provided with tools to help them move to the new languages. Requiring users to rewrite applications is very costly, time consuming, and error prone. We believe a better approach is to help ease users into new programming paradigms by providing them with both a way to utilize existing programming paradigms and applications, as well as providing them a way to write applications directly in the new programming notations.

There is a disconnect between the high level languages such as HTAs that provide high levels of expressibility and programmability, and new data-driven runtimes, such as SCALE and OCR that provide high levels of control on supercomputers of the future. We want to bridge the gap between these notations with a Parallel Intermediate Language (PIL). As new

runtimes are being developed to run on future supercomputers, we believe that a framework to help programmers target these new runtime systems is necessary. Thus, PIL should be retargetable, efficient, and should accept many high level languages as input. Such a framework can provide portability across many different machines and runtimes. Furthermore, we believe that when targeting a new runtime systems programmers can achieve increased productivity and performance through the utilization of multiresolution programming in their applications, while allowing a framework to ease the transition to new notations.

For my loving wife, Miranda.

Acknowledgments

The work contained in this thesis would not have been possible without the support and encouragement of many people. I would like to thank my advisor, Professor David Padua. His guidance throughout this work has been invaluable. I would like to thank my committee members Professor Laxmikant Kale, Professor Wen-Mei Hwu, and Doctor Wilfred Pinfold for their helpful suggestions and discussions on the ideas in my work.

I would also like to thank my family for their unending love and support in all of my endeavors. My parents have always encouraged me to follow my dreams, and my wife has helped me to achieve them.

Table of Contents

List of Figures . . . ix

List of Tables . . . xi

Chapter 1 Introduction . . . 1
- 1.1 Data-Driven Runtimes . . . 3
  - 1.1.1 The Swift Adaptive Runtime Machine . . . 5
  - 1.1.2 The Open Community Runtime . . . 7
- 1.2 Comparison of Data-Driven Runtimes to Dataflow Programming . . . 10
- 1.3 High Level Programming Notations and Compilers . . . 10
  - 1.3.1 Hierarchically Tiled Arrays . . . 11
  - 1.3.2 The R-Stream Compiler . . . 12
- 1.4 Focus for this Thesis . . . 12
- 1.5 Goals of Thesis . . . 13
- 1.6 Organization of Thesis . . . 13

Chapter 2 Universal Translation of Parallel Languages . . . 15
- 2.1 Parallelism . . . 19
  - 2.1.1 Classifying Parallelism . . . 21
  - 2.1.2 Composing Parallelism . . . 24
  - 2.1.3 Examples of Parallelism in Popular Languages . . . 25
  - 2.1.4 Universal Parallelism . . . 27
- 2.2 Memory Models . . . 27
  - 2.2.1 Shared Memory . . . 27
  - 2.2.2 Distributed Memory . . . 28
  - 2.2.3 Data and Its Placement . . . 28
  - 2.2.4 Communication and Synchronization . . . 29
- 2.3 Discussion of Parallelism and Memory Models . . . 31
  - 2.3.1 Data Parallelism . . . 31
  - 2.3.2 Task Parallelism . . . 33
  - 2.3.3 SPMD Parallelism . . . 36
- 2.4 Conclusions on Universal Translation . . . 39

Chapter 3 The Parallel Intermediate Language . . . 41

3.1 Syntax and Semantics of PIL 42
3.1.1 Declaration of Data in PIL 43
3.1.2 Body Functions of PIL Nodes 44
3.1.3 PIL Nodes 45
3.1.4 Library Functions Provided with PIL 48
3.1.5 Support for the Implementation of Libraries 54
3.2 Parallelism in PIL 56
3.2.1 Data Parallelism 58
3.2.2 Task Parallelism 58
3.2.3 SPMD Parallelism 58
3.2.4 Composing Parallelism 58
3.3 Memory Models 59
3.3.1 Communication and Synchronization 60
3.4 Discussion of Parallelism and Memory Models 60
3.4.1 Memory Models in PIL 60
3.4.2 Parallelism 61
3.5 Examples of Popular Languages in PIL 69
3.5.1 Acceptance as Input 69
3.5.2 Generation as Output 71
3.6 Portability 72
3.7 Multiresolution Programming 72
3.8 Optimizations in PIL 75
3.9 Conclusions of the Language 78

Chapter 4 The PIL Compiler 79
4.1 Parsing a PIL Program 79
4.2 Generating OpenMP code from PIL 80
4.2.1 Encapsulation of Data 80
4.2.2 Body Functions 80
4.2.3 PIL Nodes 80
4.2.4 Memory Management Library 81
4.2.5 Communication Library 81
4.2.6 Support for the Implementation of Libraries 82
4.3 Generating SCALE code from PIL 82
4.3.1 Encapsulation of Data 82
4.3.2 Body Functions 83
4.3.3 PIL Nodes 83
4.3.4 Memory Management Library 84
4.3.5 Communication Library 84
4.3.6 Support for the Implementation of Libraries 85
4.4 Generating OCR code from PIL 85
4.4.1 Encapsulation of Data 86
4.4.2 Body Functions 86

4.4.3 PIL Nodes . . . 86
4.4.4 Memory Management Library . . . 86
4.4.5 Communication Library . . . 87
4.4.6 Support for the Implementation of Libraries . . . 87
4.5 Summary of the PIL Compiler Implementation . . . 88

Chapter 5 Experimental Framework . . . 89
5.1 I2PC3 Machine . . . 89
5.2 X-Stack Cluster . . . 90

Chapter 6 Cholesky Factorization: A Top to Bottom Case Study . . . 91
6.1 Tiled Cholesky Factorization . . . 91
6.2 Fork/Join Cholesky . . . 96
6.2.1 HTA Implementation . . . 98
6.2.2 PIL Implementation . . . 98
6.3 SPMD Cholesky with HTAs . . . 99
6.4 Task Parallelism with Hand Coded PIL . . . 101
6.5 Experimental Results . . . 102
6.5.1 What We Will Show . . . 104
6.5.2 Experimental Data . . . 104
6.5.3 HTA Has Little Overhead . . . 110
6.5.4 Efficiency of the Backends . . . 111
6.5.5 Efficiency of Parallel Versions of the Code . . . 113
6.6 Conclusions from Cholesky Factorization . . . 118

Chapter 7 The NAS Benchmarks with HTAs . . . 119
7.1 The NAS Parallel Benchmarks . . . 120
7.2 HTA NAS Benchmarks Fork/Join Results . . . 122
7.3 HTA NAS Benchmarks SPMD Results . . . 129
7.4 Conclusions from the NAS Benchmarks . . . 135

Chapter 8 Related Work . . . 136

Chapter 9 Future Work . . . 138
9.1 Limitations of the Language . . . 139

Chapter 10 Conclusion . . . 141

Appendix A Supplemental Results . . . 143
A.1 NAS Benchmarks . . . 143
A.2 Cholesky Factorization . . . 146

Bibliography . . . 151

List of Figures

1.1 Implementations focused on in this thesis. 12

2.1 Existing Languages to Data-Driven Runtimes 16
2.2 Existing Languages to PIL to Data-Driven Runtimes 17
2.3 Any to PIL to Any 17
2.4 High level view of our compilation flow. 19
2.5 Shared and distributed memory data parallelism. 32
2.6 Shared and distributed memory task parallelism. 34
2.7 Shared Memory SPMD Parallelism 37
2.8 Distributed Memory SPMD Parallelism 37

3.1 An example PIL Hello World program. 42
3.2 The implementation of the gpp_t type. 44
3.3 The syntax for the declaration of a PIL node. 45
3.4 Declaration of a `pil_barrier` node. 51
3.5 Declaration of a `pil_send` node. 51
3.6 Task graph for communication example. 53
3.7 Declaration of a `pil_recv` node. 54
3.8 PIL library example. 54
3.9 Pseudo code and task graph for an example PIL program 57
3.10 PIL code for the example PIL program 57
3.11 Shared Memory Data Parallel Hello World 62
3.12 Distributed Memory Data Parallel Hello World 63
3.13 An example PIL task parallel Hello World program. 65
3.14 Shared Memory Task Parallel Hello World 66
3.15 Distributed Memory Task Parallel Hello World 66
3.16 An example SPMD PIL Hello World program. 67
3.17 Shared Memory SPMD Hello World 68
3.18 Distributed memory SPMD Hello World 69
3.19 PIL compiler flow. 73
3.20 Node coarsening optimization 76
3.21 Loop interchange optimization 77

4.1 The composition of a PIL node for data-driven runtimes. 83

6.1 Pseudo code for sequential tiled Cholesky factorization. 92
6.2 Progression of a single iteration of the tiled Cholesky factorization algorithm. 93
6.3 Progress of Cholesky factorization on a 4×4 tiled matrix. 94
6.4 Task graph of Cholesky factorization. 95
6.5 Pseudo code for fork/join parallel tiled Cholesky factorization. 96
6.6 Task graph of tiled Cholesky factorization fork/join algorithm. Horizontal lines are global barriers. 97
6.7 Pseudo code for HTA tiled Cholesky factorization. 98
6.8 PIL code for nodes of data parallel tiled Cholesky factorization. 99
6.9 Graph of PIL nodes for data parallel tiled Cholesky factorization. 100
6.10 PIL code for nodes of task parallel tiled Cholesky factorization. 101
6.11 Graph of PIL nodes for task parallel tiled Cholesky factorization. 102
6.12 Speedup on the I2PC machine for Cholesky factorization with 1×1 tiling. . 106
6.13 Speedup on the I2PC machine for Cholesky factorization with 100×100 tiling. 107
6.14 Speedup on the I2PC machine for Cholesky factorization with 200×200 tiling. 108
6.15 Speedup on the I2PC machine for Cholesky factorization with random tiling. 109
6.16 SPMD asynchrony scheduling. 115
6.17 SPMD Cholesky task graph scheduling. 116
6.18 Speedup on the I2PC machine for Cholesky factorization with 100×100 elements per tile using reflected cyclic tile distribution. 117

7.1 EP and IS fork/join performance on the I2PC machine. 123
7.2 CG and MG fork/join performance on the I2PC machine. 125
7.3 FT and LU fork/join performance on the I2PC machine. 127
7.4 EP and IS SPMD performance on the I2PC machine. 130
7.5 CG and MG SPMD performance on the I2PC machine. 132
7.6 FT and LU SPMD performance on the I2PC machine. 133

A.1 All fork/join performance on X-Stack. 144
A.2 All fork/join performance on X-Stack. 145
A.3 Cholesky factorization speedup on the X-Stack machine with 1×1 tiling. . . 147
A.4 Cholesky factorization speedup on the X-Stack machine with 100×100 tiling. 148
A.5 Cholesky factorization speedup on the X-Stack machine with 200×200 tiling. 149
A.6 Cholesky factorization speedup on the X-Stack machine with random tiling. 150

List of Tables

2.1 Flynn's taxonomy 21
2.2 Parallel Constructs in Popular Languages 25

6.1 Frontend implementations 103
6.2 Parallel Algorithms. 103
6.3 Generated backend codes. 103

Chapter 1

Introduction

The next challenge in the evolution of supercomputers will be the transition to exascale systems. However, while the move from terascale to petascale processing was considered evolutionary, it is widely believed that the leap to exascale supercomputers will require revolutionary advances. Simply scaling up current technology will not work. The projections for the exascale systems indicate that applications may have to support up to a billion separate threads to efficiently use the hardware, while the amount of memory per arithmetic functional unit will drop significantly [2], [5]. This implies the need for exploiting fine-grain parallelism with a programming model other than the currently used message passing or coarse-grain threads. As a response, the programming community is exploring data-driven runtimes [33]. However, in order to utilize the new runtime systems, users will either need to rewrite all of their applications by hand in the new languages, or be provided with tools to help them move to the new languages. Requiring users to rewrite applications is very costly, time consuming, and error prone. We believe a better approach is to help ease users into new programming paradigms by providing them with both a way to utilize existing programming paradigms and applications, as well as providing them a way to write applications directly in the new programming notations.

The task parallelism required by data-flow runtimes can be highly unstructured. Unstructured applications and algorithms, while capable of providing exceptional performance, can be extremely difficult to reason about, and can therefore cause increased programming effort. In an effort to reduce programming effort while maintaining high performance, several high-level languages have been designed including Chapel [11], Intel Concurrent Collections (CnC) [10], and Hierarchically Tiled Arrays [6]. Furthermore, many applications have been developed in more traditional languages such as MPI [22] and OpenMP [25].

A programming framework that accepts code in high-level parallel notations, and targets parallel runtimes, provides the users the ability to leverage multiresolution programming. Multiresolution programming techniques allow for users to implement code in two or more languages simultaneously. This allows the implementation of separate portions of an application in the notation that is most suited to the expression of the algorithm. Furthermore, multiresolution programming can allow programmers an environment to facilitate a natural transition from an existing programming notation to a new runtime system.

We propose a parallel intermediate language to try and bridge the gap between a variety of popular programming models and the runtimes of future exascale machines. Our original goal was to target the previously mentioned data-driven runtimes (macro dataflow), but we also want to explore other more traditional runtimes, since these may coexist with the data-driven runtimes on future machines. Additionally, we believe that providing users a multiresolution programming environment enables the user to more easily transition to the new runtime systems. Thus, the intermediate language will need to provide the multiple classes of parallelism necessary for exascale computing as well as target a variety of parallel runtimes while providing the mechanisms necessary for multiresolution programming.

1.1 Data-Driven Runtimes

The data-driven runtimes proposed to run on future exascale supercomputers center around the idea of having many, small, tasks that are created with dependences that are scheduled and coordinated by a dynamic runtime. The runtime can make decisions about where data should be placed, where tasks should be executed, and other issues such as communication patterns that can be based on the state of the machine. Below are some examples of these data-driven runtimes. The goals of these runtimes center around giving the programmer a high-level of control when managing their fine grained tasks. This control can come with a high cost of development in the application due to a lack of expressiveness.

There are many similarities between the data-driven runtimes. All of them have a way of defining a task. Tasks are described in a similar notion to the way a procedure is described in most programming languages. Each task can have multiple instances executing simultaneously as well.

Computational Units. The computational units available in the data-driven runtimes we have studied are not the physical processors available on the machine, but instead *worker threads* running on the machine. The units of work, tasks, are managed by the runtime system, and the runtime must choose a worker thread on which to schedule a task that is ready to run. By managing the worker threads and scheduling units of work on them, the runtime systems can enforce the non-preemption required by the systems. Once a task has been scheduled on the worker thread, it cannot be unscheduled from the thread. It *must* run to completion.

While the runtime system manages the scheduling of tasks onto worker threads, the operating system manages the scheduling of worker threads to processors. The operating system that schedules the worker thread to a processor can preempt the worker thread, or chose to move it to another processor, but the task cannot be removed from the worker

thread without running to completion. Usually, the runtime systems pin the worker threads to a core so the operating system doesn't interfere with their scheduling too much.

Task Creation. In all of the data-driven runtimes, there is a syntax for creating an instance of a task. During task creation, the dependence information is specified for the task. Tasks that are created are placed on a queue of tasks we shall call the *created queue*. This is a list of tasks with none or some, but not all, of their dependences satisfied.

Dependence Satisfaction. Tasks can be created with zero or more dependences. All dependences must be satisfied before a task may be scheduled to run. A task created with zero dependences is immediately schedulable. Tasks that have a subset of their dependences satisfied are called *partially satisfied*. Partially satisfied tasks remain on the created queue until they are fully satisfied.

Schedulable Tasks. Tasks that have all of their dependences satisfied are moved from the created queue to the *schedulable queue*. This is the work queue that the runtime can pull from to schedule tasks to worker threads.

Scheduling a Task. When there is an idle worker thread, the runtime system will remove one task from the schedulable queue and schedule the task on the worker thread. The runtime system must try and keep all worker threads busy by load balancing the schedulable tasks to worker threads, while paying attention to the data that the tasks might access. It may or may not make good locality decisions based on the access patterns of the task.

Running Tasks. In all of the data-driven runtimes we have studied, all of the tasks are non-preemptable. This means that once a task is moved from the schedulable queue and onto a worker thread, it cannot be put back into the schedulable queue. It must run to

completion. This leads to some interesting design issues about how to break tasks up into smaller tasks. A task that needs to create a subtask to calculate some result, cannot wait for the result to be computed. It must yield the worker thread to allow other tasks to make progress. The task must schedule a continuation task with a dependence on the desired results, so that the continuation task is scheduled when the results are available.

1.1.1 The Swift Adaptive Runtime Machine

The Swift Adaptive Runtime Machine (SWARM) [19] is a runtime system being developed by E.T. International. SWARM is a data-driven runtime that uses codelets as a unit of computation that are triggered by dependences. SWARM is an extension to the C language. SWARM has a higher level programming language called the SWARM Codelet Association Language (SCALE) that provides syntactic sugar for SWARM code while utilizing the SWARM runtime. In this thesis, we deal exclusively with SCALE code.

Data in SCALE. There is nothing special about data management in SCALE. It just uses the system `malloc`, `free`, and associated routines. However, codelets are associated into SCALE *procedures*. A SCALE procedure is a collection of codelets, and its entry point is always the `entry` codelet. Each SCALE procedure has a context that contains static variables associated with the procedure. The lifetime of these variables extends across the entire run of the application, but are only accessible by the codelets in the procedure. This provides a mechanism within a procedure for a codelet, if it needs to wait on an event to happen, to initialize a procedure variable, and create a new codelet instance to run when the event has been satisfied. The new EDT can continue using data that is still available in the procedure's context.

Dependences in SCALE. SCALE utilizes counting dependences. A codelet is specified to be runnable after N satisfactions on the dependence have been called. Once the satisfactions are all made, the codelet is placed on the schedulable queue, and when it begins execution, has access to any procedure variables that may be set.

Program Startup in SCALE. On program start, the program begins in the C `main` function, just like a C program. It is the user's responsibility to call routines to initialize and start the SCALE runtime. Once the SWARM runtime is initialized and started, the user may make calls into the SWARM runtime through SCALE procedures.

Memory Models of SCALE. SCALE has two modes of operation. There is a shared memory mode where a program begins in the C main function, and all codelets that are created will be scheduled to processors available on the local machine. All codelets share the same address space and can share global data through pointers if desired.

There is also a distributed memory mode for SCALE. A program is started with the `swarmrun` command, and the command is given arguments that specify the number of processes to begin, the name of the executable to run, and a host file. The host file specifies on which machines to run the SWARM runtimes. Each process contains its own SCALE runtime in its own address space. Each process may register codelets as remotely runnable, and these special codelets may be instantiated and satisfied by remote processes.

Communication in SCALE. As previously mentioned, a process in SCALE can register codelets as remotely schedulable. These codelets can take parameters, just like any other SCALE codelet. If a remotely schedulable codelet takes parameters, they are sent from the caller, A, to the callee, B, in a buffer, and can be unpacked by the callee when the codelet runs. Thus, A sends a message to B. Notice that the communication is one way, and B does not synchronize with A when the send is initiated. Once the remote codelet begins

execution, it behaves just like a normal codelet and has access to its procedures context.

Computational Units. The SWARM runtime allows for the specification of two levels of logical computational units. The SWARM runtime can work within a shared memory environment, but SWARM is able to handle distributed memory as well. Within a single instance of the SWARM runtime (on a shared memory environment) the environment variable `SWARM_NR_THREADS` specifies how many worker threads to create at program startup. The host file specifies, for distributed computing, the number of distinct SWARM runtimes to create at program start, as well as the machines on which to run those runtimes. It also specifies how to distribute the processes among the machines.

1.1.2 The Open Community Runtime

The Open Community Runtime (OCR) [24] is a runtime system being developed as part of the Intel X-Stack project. OCR is a data-driven runtime that uses Event Driven Tasks (EDTs) as a unit of computation that are triggered by dependences such as data-blocks (DBs) and events. OCR is built on top of the C programming language. One point of note about OCR is that in order to conform to the OCR model, all dynamically allocated data must be packaged into DBs, and a DB must be set as a dependence to an EDT in order for an EDT to be able to have access to the data. This provides a unique challenge in code generation, since a DB cannot be acquired mid execution of an EDT. Instead, if an EDT needs to acquire a DB, a new EDT must be created with the DB as a dependence.

Data in OCR. All dynamically allocated data in OCR must be packaged into data-blocks. Data-blocks are referred to by a Global Unique Identifier (GUID). In order to receive a pointer to the data in the block, the data-block must be acquired. This acquisition can only be done through dependence satisfactions before the EDT is scheduled. An EDT releases a

data-block on completion. OCR requires that data be packaged into data-blocks because it is designed to run on an architecture being developed with a single physical address space (no virtual memory) [24]. Lack of virtual memory implies that if a piece of data is to be moved from one physical location to another, the pointer to that data must change. OCR reserves the right to move a data-block in memory (hopefully to improve the locality of the EDTs that need access to the data-block), thus changing the pointer that accesses the data. Data-blocks can only be moved when no EDT currently has acquisition of the data block. It may happen that an EDT, A, acquires a data-block, and runs to completion using a pointer acquired to the data. Then, OCR might move the data block in memory, changing the address of the data-block. If a second EDT, B, is then scheduled, it must acquire the data-block to receive a valid pointer to the data, since the pointer that B receives is not different than that of A. If EDT A passes a pointer to the data-block to EDT B, the pointer would be to an invalid address.

Dependences in OCR. When an EDT is created, the number of input dependences is specified, and each dependence gets a *slot*. Each slot is assigned a GUID. To satisfy a dependence, the GUID of the slot must be known. When a data dependence is satisfied, a data-block can be specified as an input (data dependence) or the data-block field can be left empty (control dependence). Once all of the dependences have been satisfied, and the EDT is running, it may access the DB (or `NULL` pointer for a control dependence) that is in each of its slots.

Program Startup in OCR. Program execution begins within the `mainEdt`, similar to the way a C program begins in the `main` function. At program start, the OCR runtime initializes itself and then creates and schedules the `mainEdt`. The `mainEdt` contains user implemented code.

Memory Models of OCR. One interesting facet of OCR is that since it requires containing all dynamically allocated data within data-blocks, runtime manageable pieces of memory, there is only one memory model for OCR. Any program that follows the required conventions regarding data-blocks can be a shared memory or a distributed memory program, depending on how the runtime system manages the data. Thus, in OCR, there is no distinction between a shared memory and a distributed memory application. Currently, OCR only works on shared memory, but the distributed memory implementation is under development. The central theme of the distributed memory OCR runtime is that it will manage placement of EDTs and data-blocks across memory spaces automatically based on runtime information about access patterns of EDTs on their data-blocks. Furthermore, it is expected that the user can provide hints to the runtime about these access patterns and can suggest placement options.

Communication in OCR. Since there is no distinction between the shared memory version of OCR and the distributed memory version, a communication in OCR is as simple as one EDT, A, satisfying a dependence to another EDT, B, with a data-block, D. This is the same as a send of the message D from A to B.

Computational Units. OCR uses a configuration file to specify the organization of the machine on which it is running. A user can specify in the configuration file the number of processors and the amount of memory available to the runtime system to manage. The version of OCR that we use in this thesis is built for x86 using POSIX Threads (Pthreads). There is one Pthread created for each of the processing units specified in the configuration file. Furthermore, the runtime system does one large memory allocation at program start to allocate a single large memory pool that it manages. The runtime system is free to manage the memory within this large block of data any way it sees fit to place and move data-blocks.

1.2 Comparison of Data-Driven Runtimes to Dataflow Programming

The data-driven runtimes like SCALE and OCR have some of the ideas of dataflow programming. Each EDT or codelet has a series of dependences that must be satisfied before the EDT or codelet can execute. Once all of the dependences are met, the codelet may be made ready for execution by being placed on the list of schedulable codelets. Like dataflow, the execution order of codelets is nondeterminate and is decided by the runtime scheduler. Furthermore, each worker thread has its own work queue, but the runtime system will work steal from large queues to make sure to keep work in smaller work queues. This dynamic scheduling and work stealing can give performance boosts to unbalanced work loads, while hopefully keeping good locality.

However, in the dataflow model [14, 28] the unit of computation is very fine-grained (often a single operation) and a program expressed in this model requires hardware support for efficient execution. In contrast, in the codelet model, the units of computation, codelets, can be fine-grained computations but not as fine-grained as a single operation. A codelet program contains the definition of codelets and the dependences between codelets. A software runtime system, required for program execution, is responsible for keeping a record of dependences and scheduling codelets for execution when their dependences are satisfied.

1.3 High Level Programming Notations and Compilers

Contrary to the direction that the data-driven runtimes are headed are popular high-level programming notations. These notations center on a theme of high expressivity, fewer lines of code, and high programmer productivity.

1.3.1 Hierarchically Tiled Arrays

Hierarchically Tiled Arrays (HTAs) [6] is a class of objects and operations that encapsulate tiled arrays and data parallel operations on them. HTAs and their operations provide a natural expression for most parallel applications using tiling ubiquitously to control locality and parallelism. The HTA programming paradigm could be a solution to the programmability difficulties of codelets discussed previously discussed. HTAs have been successfully implemented in Matlab and C++, and have been studied for both distributed and shared memory environments [9, 16].

An HTA program can be seen as a sequential program containing operations on tiled arrays. It has been demonstrated that HTA code is expressive, concise, and easy to reason about. It is also simple to start from a baseline sequential program and parallelize it with HTA notations. The model provides a global view of data which lets tiles and scalar elements be easily accessed through chains of index tuples without explicitly specifying communication functions to fetch the data required. These features improve programmability and minimizes application development time.

Application codes written in the HTA notation are portable across different classes of machines since low-level details are not exposed to the users; the programmer only needs to be concerned with expressing their application. For example, a map operation can be implemented using a parallel for loop, or it can be implemented as SPMD computations. Users write code using the high-level constructs provided by the HTA paradigm and they need not know the details of the underlying machines. Compared with completely rewriting existing applications using codelets, it can be preferable for application developers to use the more familiar programming paradigm provided by HTA while still enjoying the benefits of executing applications on codelet runtime systems.

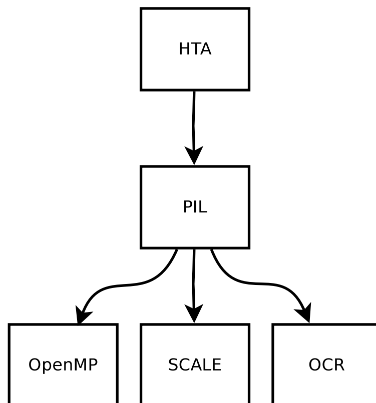


Figure 1.1: Implementations focused on in this thesis.

1.3.2 The R-Stream Compiler

R-Stream is a compiler being developed by Reservoir Labs [29]. It is a source-to-source compiler that takes sequential (restricted) C programs as input and automatically generates parallel code for various parallel runtimes and architectures, including SWARM and OCR. R-Stream has knowledge of the architecture being targeted and optimizes for locality in the memory hierarchy during the compilation process. We believe that leveraging the automatic parallelization and low-level cache hierarchy optimizations that R-Stream excels at provides a unique opportunity for nested parallelism within our compilation framework.

1.4 Focus for this Thesis

In this thesis we shall restrict the focus of performance evaluation to one high-level frontend, Hierarchically Tiled Arrays, and three backends: OpenMP, SCALE, and OCR, as depicted in Figure 1.1. The HTA notation is implemented using the library mode available in PIL described in Section 3.1.4, and [31], and the backend code implementation is described in Chapter 4.

We have implemented several benchmarks in the HTA notation that will be discussed in chapters 6 and 7. Where necessary, we have implemented some of the benchmarks directly

in hand-coded PIL, to help explain what is happening within the HTA constructs.

1.5 Goals of Thesis

There is a disconnect between the high-level languages such as HTAs that provide high-levels of expressibility and programmability, and new data-driven runtimes, such as SCALE and OCR that provide high-levels of control on supercomputers of the future. We want to bridge the gap between these notations with a Parallel Intermediate Language (PIL). As new runtimes are being developed to run on future supercomputers, we believe that a framework to help programmers target these new runtime systems is necessary. Thus, PIL should be retargetable, efficient, and accept many high-level languages as input. Such a framework can provide portability across many different machines and runtimes. Furthermore, we believe that when targeting a new runtime, systems programmers can achieve increased productivity and performance through the utilization of multiresolution programming in their applications, while allowing a framework to ease the transition to new notations.

In this thesis, we will describe such a language and its implementation with various high-level languages and low-level parallel runtimes. Furthermore, we will demonstrate that the implementations are efficient by providing an evaluation with multiple motivating examples.

1.6 Organization of Thesis

The remainder of this document is organized as follows. Chapter 2 describes the notion of any-to-any parallel translation. Chapter 3 describes the Parallel Intermediate Language and its design considerations. Chapter 4 details the code generation for the implemented backends. Chapter 5 describes the experimental framework for the performance evaluations. Chapter 6 provides a performance evaluation of a tiled Cholesky Factorization benchmark.

Chapter 7 provides a performance evaluation of the NAS benchmarks. Chapter 8 describes related work on intermediate languages. Chapter 9 describes the proposed direction of future work. Finally, Chapter 10 concludes.

Chapter 2

Universal Translation of Parallel Languages

The original goal of the work reported in this thesis was to ease the burden of writing code for data-driven runtimes, such as OCR and SWARM. We believe the best way to drive adoption of these new runtimes is to provide the programmer a familiar programming paradigm, while simultaneously providing the ability to write code in the new notation in the form of multiresolution programming [12]. We chose to generate data-driven code based on existing programming paradigms, as shown in Figure 2.1. This approach allows users to focus on algorithm development without needing to be concerned with all of the low level details required by the new runtime systems, such as dependences and task interactions. However, we also want to provide users with a way to supply code written in the new programming paradigms, as well as being able to utilize the expressiveness of high level programming languages.

We want to provide the user with not just a single high level programming language to utilize, but to facilitate programming in multiple frontend languages and running on top of multiple data-driven runtimes. To facilitate this, we chose to standardize on a single

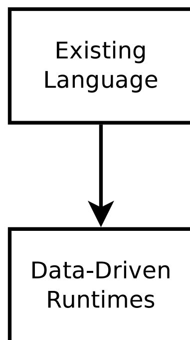


Figure 2.1: Existing Languages to Data-Driven Runtimes

intermediate language, as most compilers do, to ease the implementation of code generation with several frontend languages and several backend languages in mind. This leads to a natural paradigm of existing languages generating the Parallel Intermediate Language (PIL) code, and then the PIL compiler generating backend code for the supported data-driven languages, as in Figure 2.2.

Naturally, the paradigm in Figure 2.2 sparks some interesting questions. Can any notation be translated into PIL code? Can the PIL notation generate any arbitrary backend representation, even non-data-driven languages like MPI and OpenMP? What techniques are needed? Do the translations generate efficient code? These questions encapsulate the idea of any-to-any parallel programming, represented in Figure 2.3. In this work we explore the idea of creating a single intermediate language that retains the parallel semantics of the high level languages generating it. We want to explore if we can create such an intermediate language that can be generated from *any* parallel language, and can simultaneously generate *any* other parallel language.

Although the original design for PIL was to target data-driven runtimes (OCR and SWARM), we wanted to make sure that it was designed well enough to do more. We wanted to design PIL in such a way that it can easily compile from a wide range of parallel source language into a wide range of parallel backend language.

Frontends we considered for this work include OpenMP [25], MPI [22], Pthreads [23],

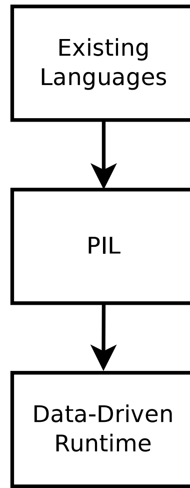


Figure 2.2: Existing Languages to PIL to Data-Driven Runtimes

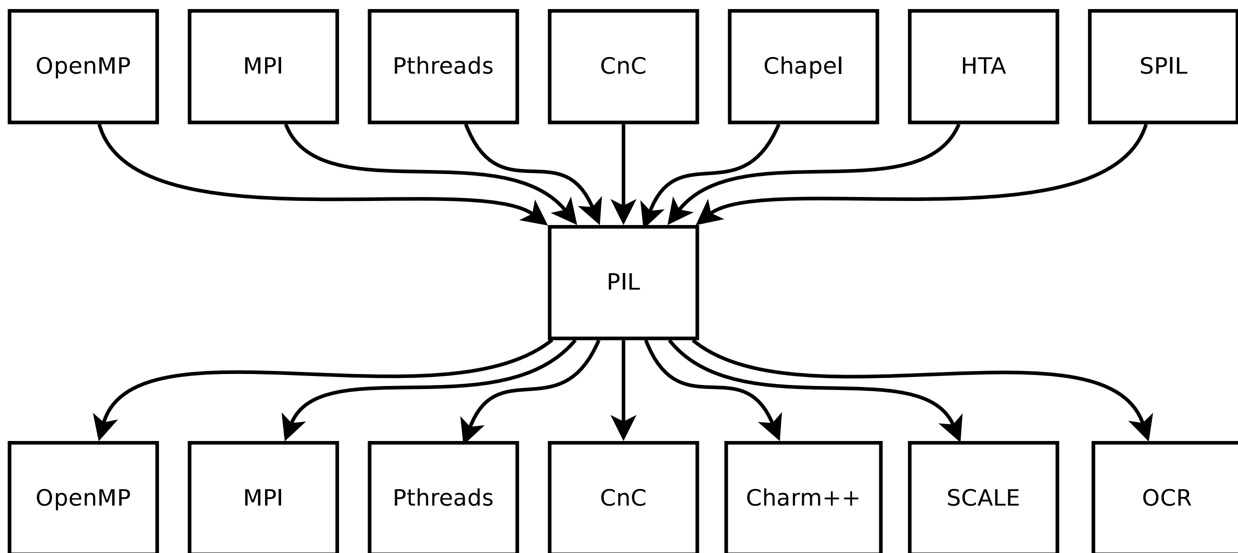


Figure 2.3: Any to PIL to Any

Intel Concurrent Collections (CnC) [10], Chapel [11], Hierarchically Tiled Arrays (HTA) [6], and Structured PIL (SPIL) [26]. Backends that were considered include OpenMP, MPI, Pthreads, CnC, Charm++ [13], SCALE [19], and OCR [24] as seen in Figure 2.3. In this thesis, we focus on the implementation of the HTA frontend, and the OpenMP, SCALE, and OCR backends.

Since PIL will generate parallel runtime code, that is itself a programming language, we added a new compilation phase into the traditional compilation process. This compilation flow can be seen in Figure 2.4. Thus, we are augmenting and complementing existing compilers with our new translation phase. We believe that a major problem traditional compilers face with compiling parallel programs is that the compilers lower the program into a very low level intermediate language at an early phase. Later, when trying to reason about parallel operations, they must try and raise the parallel operations back up to a higher level that is easier to reason in. This process is very difficult, and as a result, traditional compilers are usually overly cautious about the operations they perform across parallel operations. PIL can provide a framework for optimizations at a higher level, where parallelism is a first-class object and easy to reason about.

During the design of PIL, it became clear that it was not necessary to re-engineer solutions to already solved problems. Thus, we made a conscious decision that PIL itself should not describe the sequential computation steps to be done, but rather describe the parallel structure in the code. PIL relies on the fact that a parallel program is made up of multiple sequential threads of execution working simultaneously. Accordingly, a PIL code is made up of two key pieces. First, the description of the parallelism, and second, the sequential instructions that each worker instance will execute. For simplicity we have chosen to generate all of our backend code based in the C programming language (C+OpenMP, C+SCALE, C+OCR, C+MPI, etc). Thus, a program in PIL is a description of the parallelism in a series of PIL nodes, and the nodes' associated body functions written in C which represents the

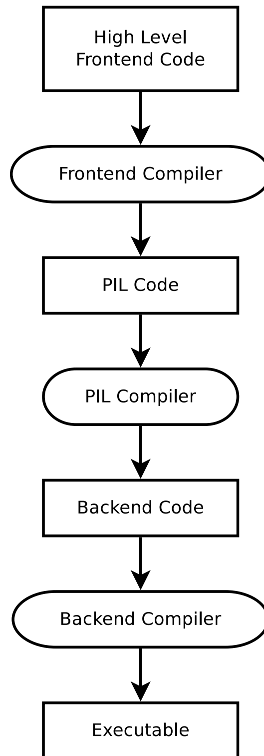


Figure 2.4: High level view of our compilation flow.

sequential computation steps.

2.1 Parallelism

When we analyze a program for parallelism, we are working at a higher level than traditional compilers. Accordingly, we are also working at a coarser granularity than traditional compilers. Most conventional compilers have intermediate languages that operate at or very near the target instruction level. This approach makes instruction-level optimizations easy to reason about, but the coarse-grained parallel operations become obfuscated since they must be raised up from this notation into something easier to reason about. In contrast, PIL operates at a level of granularity where we consider an entire task as a single unit of parallelism. The parallel application is thus concerned with the coordination of these parallel

tasks.

Serializable Parallelism. Because we are not only targeting traditional notations, but also data-driven runtime systems, PIL programs must be serializable. This means that every parallel program represented in the PIL notation must be able to be serialized and run on a single processor. This includes the communication and synchronization operations. The primary reason for this requirement is that some runtime systems (OCR and SCALE) require that a task be non-blocking and tasks are non-preemptable. This means that once a task is scheduled to a worker thread, it cannot be swapped out for another task. Thus, in order to insure that all programs can run to completion, they must be serializable. A non-serializable program is a program that can only correctly execute in parallel. Furthermore, a non-serializable program might result in deadlock in some runtime systems.

Let us take a barrier as a motivating example. There are many ways to implement a barrier in a language. One way is a polling barrier. A polling barrier is one in which the tasks enter the barrier and increment the barrier counter. Once the counter reaches the number of active tasks in the system, then all tasks have entered the barrier, and it is safe for the tasks to be released from the barrier. The tasks that have entered the barrier can poll the counter to monitor its value to check for all tasks to have arrived at the barrier.

The polling barrier approach is not a problem if the number of computational units is larger than the number of active tasks. However, consider the situation when we have four computational units (worker threads), and eight active tasks. In a data-driven runtime like OCR or SCALE, the runtime will schedule the first four tasks that arrive to the barrier to the four worker threads. Then, the active tasks will continue polling the counter waiting for it to be incremented by the other tasks. Remember, the tasks in OCR and SCALE are non-preemptable, and the runtime will not remove them from the worker threads until the tasks run to completion. However, no threads can leave the barrier until all have participated.

Table 2.1: Flynn's taxonomy

	Single Instruction	**Multiple Instructions**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

The system is now in deadlock because this polling barrier is not serializable.

A solution to make the barrier serializable is to use a two stage barrier. When a task enters the barrier, it can schedule a task with a dependence on the condition that all tasks have entered the barrier. The task that is entering the barrier cannot release the worker thread, so other tasks can enter the barrier. Once all tasks have entered the barrier, the continuation tasks will be fully satisfied and can continue from the barrier.

This algorithm is one example of serializable. The general notion of serializable is that if you have T tasks and P processing elements, an algorithm is serializable if it can run correctly to completion when $P = 1$. Many algorithms, such as the polling barrier, can encounter deadlock or errors if $P < T$, and thus are not serializable.

2.1.1 Classifying Parallelism

We use Flynn's taxonomy, as shown in Table 2.1, to describe the classes of parallelism available in an application. Flynn's taxonomy is usually referred to within the context of low-level instructions. For example, the first class of parallelism is usually referred to as Single Instruction Single Data (SISD). However, since we are concerned with a coarser grain of parallelism, the task, we do not think of a single instruction, but rather a series of instructions, the task. Nonetheless, we believe that there is a strong corollary between the classifications of Flynn's taxonomy and the classes of parallelism available in any application. In this work we will replace instructions in Flynn's taxonomy with tasks and discuss STSD, STMD, MTSD, MTMD parallelism.

Single Task Single Data (STSD). STSD is simply the case of sequential programming. There is no parallelism available here. In the traditional sense each instruction executes one after the other, each operating on its own data. In contrast to SISD, we do not consider a *single* instruction with that instruction's *single* data, but rather an entire single task with that task's data.

Multiple Task Multiple Data (MTMD). MIMD performs independent sets of instructions on separate data items. This type of parallelism is called **task parallelism**. From a course-grained perspective, MTMD, this is distinct tasks with each task executing its own series of instructions, and operating on its own private data. Single Program Multiple Data (SPMD) is a sub-category of MTMD, and we treat it as a special case. SPMD parallelism is discussed later.

For the purposes of this thesis, we consider parallel computing to be the management of multiple, simultaneous tasks. Any parallel computation can be achieved by the proper coordination of independent tasks. In its most pure form, multiple tasks will execute asynchronously operating on independent data. Synchronization and communication mechanisms, discussed later, will need to be provided in order to successfully manage tasks working on shared data.

Single Task Multiple Data (STMD). SIMD performs the same set of instructions on separate data items. This type of parallelism is called **data parallelism**. In our course-grained context, STMD, we consider a single task with multiple instances each executing on separate data. Data parallelism is a very common way to parallelize operations on arrays. The data to be operated on is separated into logical chunks, for example tiling an array, and each task instance is assigned one of the pieces of work to operate on. All of the tasks work in parallel on their own data. These tasks are coordinated to perform the same instructions on their own data. Technically data parallelism is actually a subset of task parallelism,

since multiple tasks are executing simultaneously on data. This, data parallelism can be constructed from task parallelism. However, it can be extremely convenient to provide syntax for data parallelism, rather than to require the user to construct data parallelism through tasking constructs.

Multiple Task Single Data (MTSD). MISD is another unique case of parallelism, that doesn't merit much study within the scope of this thesis. MISD uses multiple instruction streams on the same data to try and arrive at the same solution. This mode of parallelism is typically used for fault tolerance. The extra tasks all working to reach the same solution consume extra power, and require more execution resources, without increasing the performance of the application. Furthermore, since the tasks are all executing different code to compute the solution, they will all arrive at the solution at (possibly greatly) different times.

One of the primary considerations of our parallel intermediate language is high performance. In keeping in the mindset of high performance we do not discuss MTSD execution models in this thesis. That is not to say that MTSD cannot be implemented within the PIL framework. On the contrary, MTSD computation can easily be achieved by using task parallelism to set multiple tasks off executing on the same data. Since MTMD parallelism can be composed from task parallelism, a user that needs MTMD parallelism can construct it.

SPMD Parallelism Single Program Multiple Data (SPMD) programming is a special case of the composition of task parallelism to achieve data parallelism. SPMD parallelism uses a special terminology for the threads of execution within the model, the **rank**. At program start, multiple SPMD ranks are created and each assigned a specific set of data to operate on. Ranks are differentiated by an identifier that can be queried within the runtime. An SPMD rank may perform specific operations in accordance with its rank identifier. The defining characteristic of SPMD parallelism is that the independent ranks are created at

program start and live throughout the program's entire execution. This stems from the fact that the P in SPMD stands for an entire program. Traditionally, multiple programs are started to process privately-owned data. Since our model focuses on smaller tasks, we can compose an SPMD rank from multiple consecutive tasks if necessary.

Summary of Parallelism Categorization. As can be derived from our previous discussion about the classification of different types of parallelism, we believe that it is sufficient to support only two types of parallelism when compiling from one parallel notation to another: task parallelism and data parallelism. These two forms of parallelism can be combined to represent any type of parallel computation that may be desired. In the formulation of our parallel intermediate language we took this fact into heavy consideration.

2.1.2 Composing Parallelism

Composing parallelism into multiple levels can be beneficial for several reasons. It might be easiest to describe your algorithm if you have a few asynchronous tasks nested within a data parallel operation, or a data parallel operation nested within an asynchronous task. One ubiquitous form of nested parallelism is data parallelism within SPMD, for example, OpenMP nested under MPI. This form of nested parallelism is usually done to more efficiently leverage heterogeneous architectures. That is, to most efficiently utilize hardware at different levels in the hardware hierarchy. The SPMD part of the computation can be used, for example, across a cluster of distributed nodes, and the data parallelism can be used to leverage the multiprocessing capabilities of a single node.

In the context of this thesis, we consider nested parallelism to be the nesting of subtasks under other tasks. In other words, task parallelism within data parallelism, data parallelism within task parallelism, task parallelism within task parallelism, or data parallelism within data parallelism. Another form of nested parallelism can easily be achieved by using instruc-

Table 2.2: Parallel Constructs in Popular Languages

	Operation	**OpenMP**	**Pthreads**	**MPI**	**Chapel**	**SCALE**	**OCR**
Data Parallel	Parallel Loop	yes	no	no	yes	no	no
Task Parallel	Spawn	yes	yes	no	yes	yes	yes
	Confluence	yes	yes	no	yes	yes	yes
SPMD Parallel	SPMD Parallelism	yes	no	yes	no	no	no

tion level parallelism (ILP) within a single task. In fact, the performance of our parallel intermediate language hinges on the lower level compiler's ability to take advantage of ILP when available on the hardware. However, PIL operates at a higher level, and does not work at the instruction level.

2.1.3 Examples of Parallelism in Popular Languages

The Parallel constructs from popular programming notations can be seen in Table 2.2. We will now discuss these constructs, and how they fit into our parallel model in a general way. Concrete discussions relating to our intermediate language will be discussed in the next chapter.

Data Parallel Constructs. The basic data parallel operation is the parallel loop. It is available in many parallel programming languages. For example, the `parallel for` loop in OpenMP and the `forall` and `coforall` loops in Chapel. The data parallel notions discussed in Section 2.1.1 encompass these data parallel notations. Chapel's `forall` and OpenMP's `parallel for` loop behave in the same way. The loop iterations are chunked using a distribution either specified by the user, or implicitly, and one task is created to operate on the iterations composing a chunk. The tasks must all join before continuing execution, unless modified with a construct such as the OpenMP `nowait`. The Chapel `coforall` behaves the same as the `forall` loop except that one task is always created for each iteration of the loop. The blocking of the `forall` loop is used as an optimization to keep task creation overhead low.

Task Parallel Constructs. The basic task parallel operation to create a new task is the spawn operation. All of the parallel notations in Table 2.2 support this basic operation except MPI. The reason that MPI does not support this construct is that its parallelism is restricted to SPMD parallelism, and all of the parallel tasks that are to be created are specified on the command line at program start. The basic task parallel operation for two tasks to merge is confluence. OpenMP, Pthreads, SCALE, and OCR all support explicit confluence.

In OpenMP, there are two operations for spawn. First, is the `parallel sections` operation. This is structured task parallelism in that each task that will be spawned is specified by each section. There is an implicit confluence at the end of the sections, and all tasks must merge before execution can continue. The second form of task parallelism in OpenMP is using the `task` and `taskwait` operations for spawn and confluence respectively.

The Pthread language centers around the idea of task parallelism. It provides the spawn operation with `pthread_create`, and the confluence operation as the `pthread_join` call.

The Chapel language supports task parallelism with the `begin`, `sync`, and `cobegin` statements. The `begin` statement is the spawn operation. The `sync` statement is the confluence operation, and is wrapped around multiple `begin` statements that will all merge before the sequential execution can proceed. The `cobegin` statement provides functional equivalence of wrapping `begin` statements with a `sync` statement, but provides a slight optimization in its implementation. The `cobegin` statement is semantically equivalent to OpenMP's `parallel sections`.

SCALE and OCR focus on task parallelism. Both allow the spawn of a task with a create syntax. The confluence operation is not explicit in the languages since it is created through dependences. However, confluence is allowed. A compiler is required to detect confluence when generating parallel intermediate language code. However, since confluence is explicit in the intermediate language, constructing a confluence operation is trivial.

SPMD Parallel Constructs. MPI is the only programming notation that is build entirely around SPMD parallelism. With an MPI program, the number of ranks is specified at program start, and those are all of the MPI ranks that will exist through the program execution. However, since all of the other languages in Table 2.2 support task parallelism, the functionality of SPMD parallelism can be created by the user is desired. Furthermore, the OpenMP language provides the `parallel` construct. The `parallel` construct creates one task per worker thread and all begin executing the instructions inclosed by construct, creating SPMD parallelism.

2.1.4 Universal Parallelism

Any programming notation that wants to be able to translate from any notation to any other notation must be able to represent data parallel as well as task parallel notations. Any successful parallel intermediate language must be able to compose parallelism into as many nested levels as the user requires.

2.2 Memory Models

We classify computer memory models into two broad categories: shared memory and distributed memory. Any programming notation that wants to be able to translate from any notation to any other notation must be able to represent and reason about shared and distributed memory spaces.

2.2.1 Shared Memory

A shared memory machine is one in which a set of processors all have access to the same memory space, be it physical or virtual memory. All forms of parallelism are viable within this machine. Data sharing can be achieved simply by sharing pointers to data within the

memory. Synchronization on the data can be performed with low level atomic operations, with semaphores or mutexes, or with more coarse grain operations such as barriers or point-to-point synchronization.

2.2.2 Distributed Memory

In a distributed memory machine, each distributed task is considered to have its own private memory space. Since no task can access the data contained within other tasks, the tasks must coordinate access to data by some communication method. Tasks cannot simply send a pointer of their data to another task, like in shared memory, since the data may not exist in both tasks address spaces, and the communicated pointer cannot be assumed to be valid. Communicated data must be copied from one memory space to another to be available in the new memory space.

The distributed memory model is more restrictive than shared memory. Because of this, distributed memory tasks can be pooled and composed on a shared memory machine. They can use the same communication model as the pure distributed memory model, but live within the same address space. This is analogous to running multiple MPI ranks on a single, shared-memory machine. There are optimizations that can be achieved if multiple tasks do share a memory space, such as sending as a message a pointer to the data instead of copying the data if it is known that the data will indeed be in scope of the receiving task.

2.2.3 Data and Its Placement

When data is allocated, there are three levels of visibility. The lowest level of data ownership and visibility is task private data. This data can only be read or modified by the task that owns it. This type of data is very common in programming paradigms, and provides a mechanism for data hiding. A parallel intermediate language will need to provide a container

for task private data.

The second level of data visibility is memory space visible. This type of data is visible to all tasks executing in the same memory space. This is a very common form of data sharing. A common usage case is to allocate an array within the shared memory space, and partition it logically to tasks for computation.

The final level of data visibility is globally visible. All tasks, even those in distributed memory spaces, can read or modify this data. This type of memory visibility is uncommon in programming models, since the bookkeeping required for access to the data results in high overhead, and can lead to very poor locality and memory performance if not handled very delicately.

2.2.4 Communication and Synchronization

To coordinate access between multiple executing tasks and shared data, it is necessary to support synchronization of the threads. This can be achieved through the use of communication and synchronization mechanism.

Communication. The two basic communication mechanisms required are send and receive. Send and receive are point-to-point operations, requiring the sender and the receiver to meet to communicate data. Data is communicated in buffers. There are two main considerations when dealing with communication: synchronous or asynchronous, and blocking or non-blocking. The most restrictive form of communication is is a blocking synchronous communication. Synchronous means that both the send and the receive must complete before either task is allowed to continue. Blocking means the the buffer provided to the communication operation is immediately useable. A synchronous operation implies blocking, but asynchronous operations can either be blocking or non-blocking. For example, a blocking send will copy the data in the buffer immediately so the sender can reuse the buffer im-

mediately after the send completes. A non-blocking send will not make the copy, and the communication of the data will happen at some unknown time after the send is initiated. A synchronization call is necessary after a non-blocking operation to ensure that communication has completed, and the buffer can be reused or freed. The communication library we implemented with our intermediate language utilizes blocking asynchronous sends, with blocking synchronous receives.

Complex collective operations can be composed of point-to-point communications. Common collective operations include broadcast, all-to-all, reduction, scan, and more. These collective operations can be provided with special, highly-optimized routines. However, they can be composed from point-to-point communications and are not strictly necessary in a language. A compiler could be used to detect these patters in a language the only provides point-to-point operations and generate the most optimized code. Alternatively, a compiler could expand a collective operation into its basic parts if a language only provides point-to-point constructs.

Synchronization. Since the sends and receives in communication provide a form of synchronization, they can compose all of the synchronization one may need. The communication mechanisms send and receive provide point-to-point synchronization. Point-to-point synchronization provides the foundation of any synchronization suite, and can be composed to form any other form of synchronization.

Take, for example, the barrier as a common form of collective synchronization. A barrier can be formed in many ways using point-to-point synchronization. One simple way is a ring barrier composed of two stages. Upon entering the barrier, each task will post a receive to the task on its left to check when the task has entered the barrier. The root task has no prerequisites and can immediately notify the task on his right with a message that it has entered the barrier. The final task will notify the root task that it has entered the barrier,

and this serves as the first release task. Once the root task receives this notification, it knows that all tasks have arrived at the barrier and can be released from the barrier. The root sends a message to the task on his right to notify the task that it may be release, and so on around the ring.

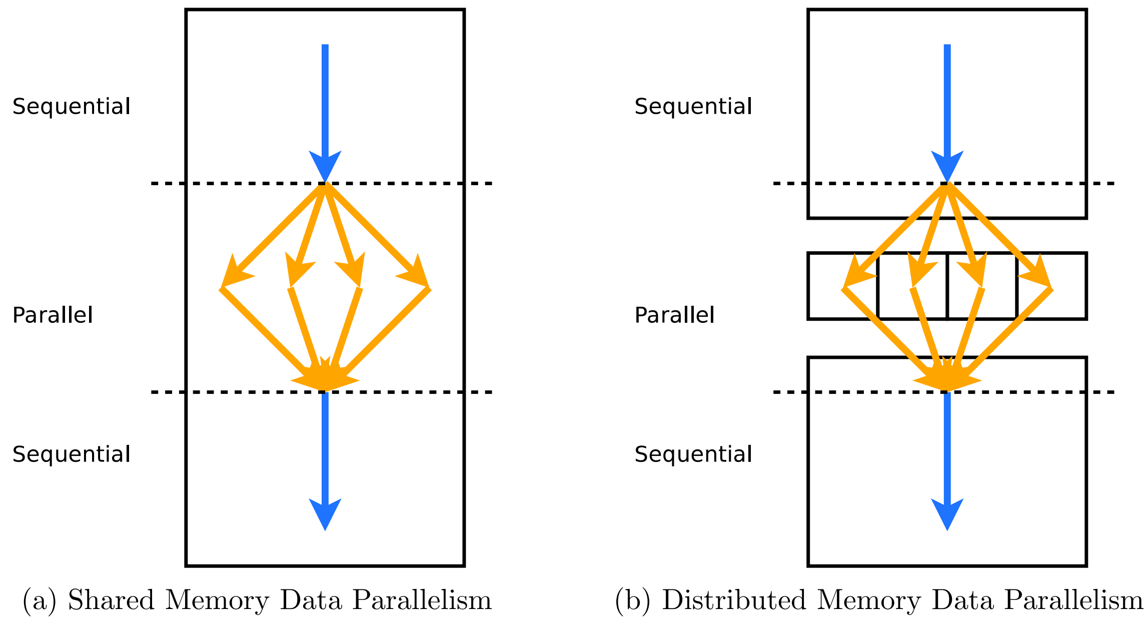
More efficient forms of barriers can be composed using these mechanisms, for example, using tree structures instead of rings. We believe that it can be useful to provide common synchronization mechanisms with a direct syntax, and a more efficient implementation than what can be built from the point-to-point synchronization primitives. However, strictly speaking, these mechanisms are not required in a language, since they can be derived from the point-to-point primitives.

2.3 Discussion of Parallelism and Memory Models

In this section we will discuss each of the classes of parallelism and how they are handled within each memory model. For the sake of clarity we use the word **fork** when creating tasks within data parallelism and **spawn** when creating tasks within task parallelism. Additionally, we use the term **join** when data parallel tasks merge, and the term **confluence** when task parallel tasks merge.

2.3.1 Data Parallelism

A diagram of how data parallelism works can be seen in Figure 2.5 The basic construct in many languages for data parallelism is a *parallel for* loop. Data parallelism allows for easy understanding of what is happening in a parallel algorithm, since it provides a strict structure. There is a single master task that is operating for the sequential parts that forks multiple worker tasks for the parallel operations. There is a global barrier where all worker tasks must join before the sequential master task can continue operation. The strict



(a) Shared Memory Data Parallelism (b) Distributed Memory Data Parallelism

Figure 2.5: Shared and distributed memory data parallelism.

structure of this data parallelism provides a clean algorithmic representation, however, the global barriers can have high overhead when there are many worker tasks.

Shared Memory Data Parallelism. Data parallelism operating in a shared memory environment can be seen in Figure 2.5a. In the diagram an arrow represents a task. Blue tasks represent the sequential portion of the code, while orange tasks represent the parallel portion. The single box encompassing all of the tasks represents the memory space of the shared memory machine. Since no task arrow crosses the memory space box, they are all contained within the same memory space.

Distributed Memory Data Parallelism. Data parallelism operating in a distributed memory environment can be seen in Figure 2.5b. Notice now we are in distributed memory that the tasks are crossing memory space boundaries. Each parallel task in the diagram operates in its own memory space. Since the tasks are crossing memory space boundaries, communication must be involved. The communication introduces a new form of overhead

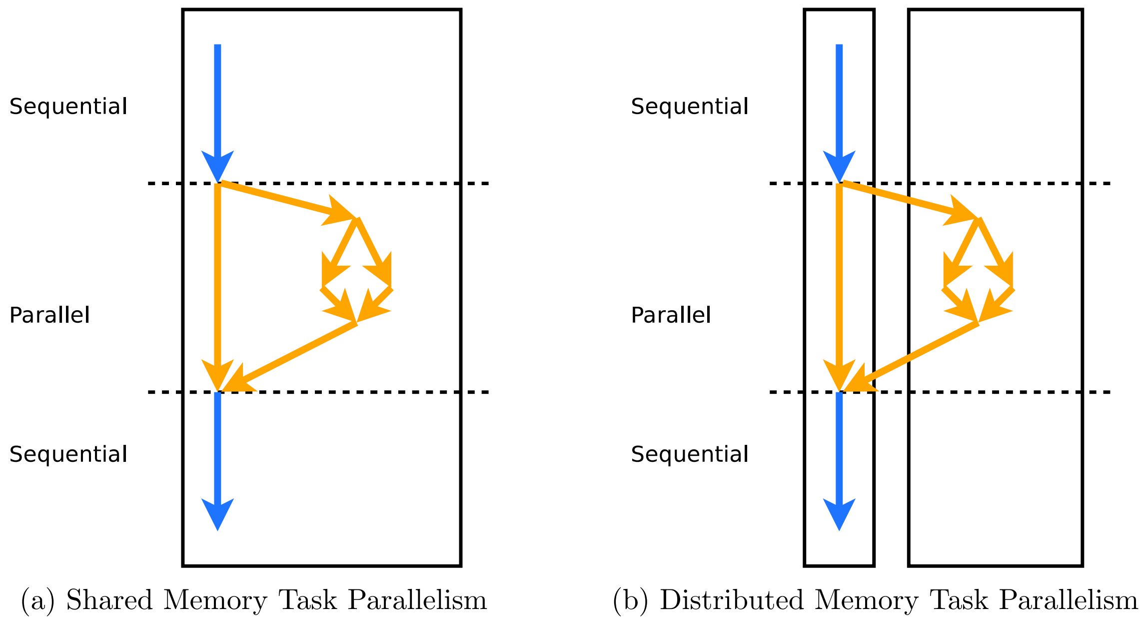
into the system, however, it might be worthwhile since distributed machines allow for larger numbers of processors than are available on a single shared memory machine.

A naive implementation of distributed memory data parallelism might ship all data required for a task to complete with the task at creation time. Once the task completes, the resultant data must be sent back to the master thread to be read. However, more efficient communication can be achieved by saving data on the memory spaces of all processors. If possible, the remote tasks can reuse the data and it need not be communicated to that location. Prudent bookkeeping is required for this optimization and must be maintained by the master task.

Universal Data Parallelism Compilation. In order to compile from any parallel notation to any other notation, an intermediate language and compiler must be able to work with distributed and shared memory data parallelism. However, we believe there is no difference from the user's view point of the two memory models when concerned with data parallelism. This means when users write parallel codes they need only be concerned with the expression of the parallelism in the algorithm and not with communication of data among the tasks. With this view of data parallelism, a single algorithm can be expressed and written once, while executing on shared or distributed memory. The change from shared memory to distributed memory, or vice versa, can be achieved by supplying a flag to the compiler during compilation to choose the shared or distributed memory model. It is the compiler's responsibility to ensure that the communication of data between the parallel tasks for distributed memory data parallelism is efficient and does not cause unnecessary overhead.

2.3.2 Task Parallelism

Task parallelism is, in our view, the most generic form of parallelism, since any other form of parallelism can be constructed from task parallelism. When a new task is created it can



(a) Shared Memory Task Parallelism (b) Distributed Memory Task Parallelism

Figure 2.6: Shared and distributed memory task parallelism.

execute any instructions or routine it needs to. This is in contrast to data parallelism where all of the created tasks execute the same operation on the data they are assigned. Tasks created can execute asynchronously on independent data or synchronously on shared data.

A diagram of how task parallelism operates can be seen in Figure 2.6. Once again, blue arrows represent sequential tasks and orange arrows represent parallel tasks, and the boxes represent memory spaces. In the example, there are two spawn sites and two confluence sites. The example uses the same algorithm for both the shared and distributed memory execution, and can be seen in one of two different ways. The first sequential task is executing and encounters a spawn point. Now, the choice is to create a single task to execute the spawn while the original task continues its execution, or, alternatively, the original task can end and two new tasks can be created. Either of the two interpretations of the execution is correct in our view and independent of how the users would write their code.

Task parallelism provides a unique opportunity for synchronization that is not present in data parallelism. The structure of data parallelism provides for frequent synchronization

through global barriers. However, by its very nature each task in task parallelism is executing independently, and they must be synchronized. There are a wide variety of synchronization mechanisms: point-to-point, collectives, global barriers or barriers with subsections of tasks.

Distributed Memory Task Parallelism. Distributed memory task parallelism is a little more complicated than shared memory task parallelism. An example can be seen in Figure 2.6b. The first consideration required for distributed memory is the placement of the spawned task. This can be decided in one of two ways. First, the user could supply the placement when specifying the information for spawning a task. Alternatively, the runtime system could place the task automatically. Either way, the effect is the same, since the same code can generate either shared or distributed memory task parallelism backend code. In distributed memory, when a task is placed, it must have access to the data it needs to execute. The compiler is responsible for making sure that the data is available to the task by what means may be available through the runtime system.

Since synchronization and communication is a central part of coordinating tasks in task parallelism, the implementation of these operations is essential to distributed task parallelism. These operations need to be efficient, and exposed to a programmer.

Shared Memory Task Parallelism. In shared memory any spawned task will execute in the same memory space as the task that creates it. Since a spawned task will need to access data once it is create, new tasks can simply access data already available to them in the memory space. An example of task parallelism on shared memory can be seen in Figure 2.6a. This mode of execution is less restrictive than distributed memory task parallelism, but still has the same placement requirements of distributed memory task parallelism. The spawned tasks will need to be placed on worker threads. Once again, this information can either be decided at run time by the scheduler, or a priori by the user specifying placement. It is most common for the runtime system to make dynamic decisions for shared memory. However,

since the same code can be used to generate either shared memory or distributed memory task parallelism, the placement information can be specified by the user. If specified, it can either be interpreted as worker task placement, and assume the task is pinned to a core, or alternatively as a suggestion, or hint, and can be taken to have a loose meaning by the runtime system. These runtime hints, if acknowledged by the underlying runtime system, can provide a way for the user to explicitly inform the runtime system about placement of data and tasks for locality.

Universal Task Parallelism Compilation. As discussed in this section, the implementation of a language that provides universal translation of parallel codes needs to ensure that it can handle task parallelism. Furthermore, the implementation needs to ensure that there is no difference from a user's point of view between the implementation of an algorithm for shared memory or distributed memory task parallelism, so that it can generate either at compile time, as specified by compiler flags.

2.3.3 SPMD Parallelism

We believe it is prudent to provide a notation for the SPMD parallelism model because it is so prevalent in the world of parallel computing. The SPMD model provides data parallelism via structured task parallelism. In other words, the coordination of computation on data with a set number of tasks. The tasks for SPMD parallelism are all created at program start, as specified on the command line.

An example of SPMD parallelism can be seen in Figures 2.7, and 2.8. In these diagrams it is clear that the sequential portion of the code is replicated to all SPMD ranks. Any data computed and stored on the stack in these tasks is recomputed by all ranks so they store the same value. This trades communication for computation, instead of having a single rank compute the values and then broadcast them to the other ranks.

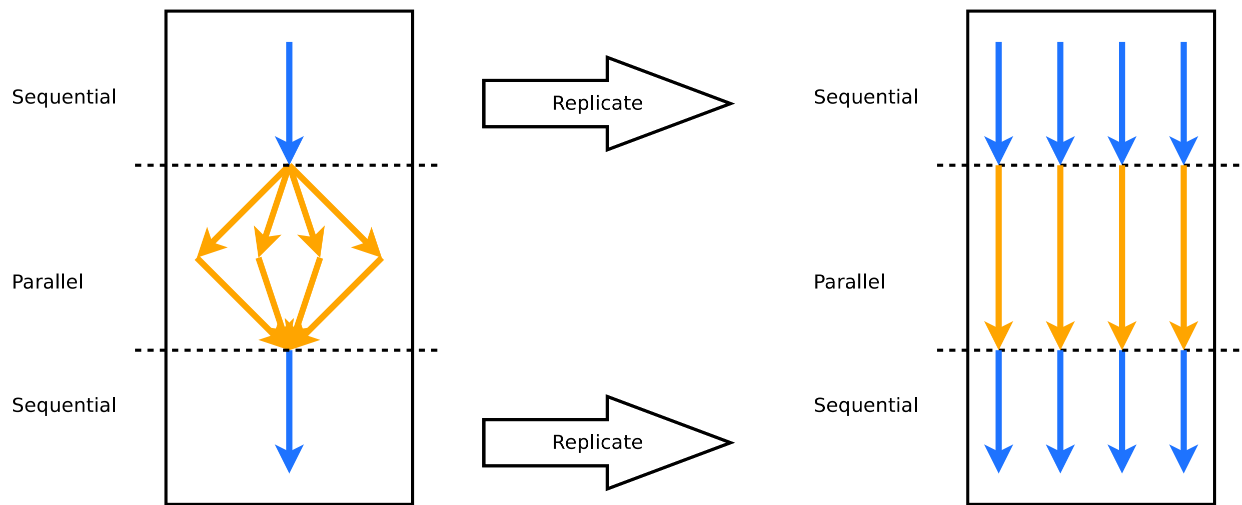


Figure 2.7: Shared Memory SPMD Parallelism

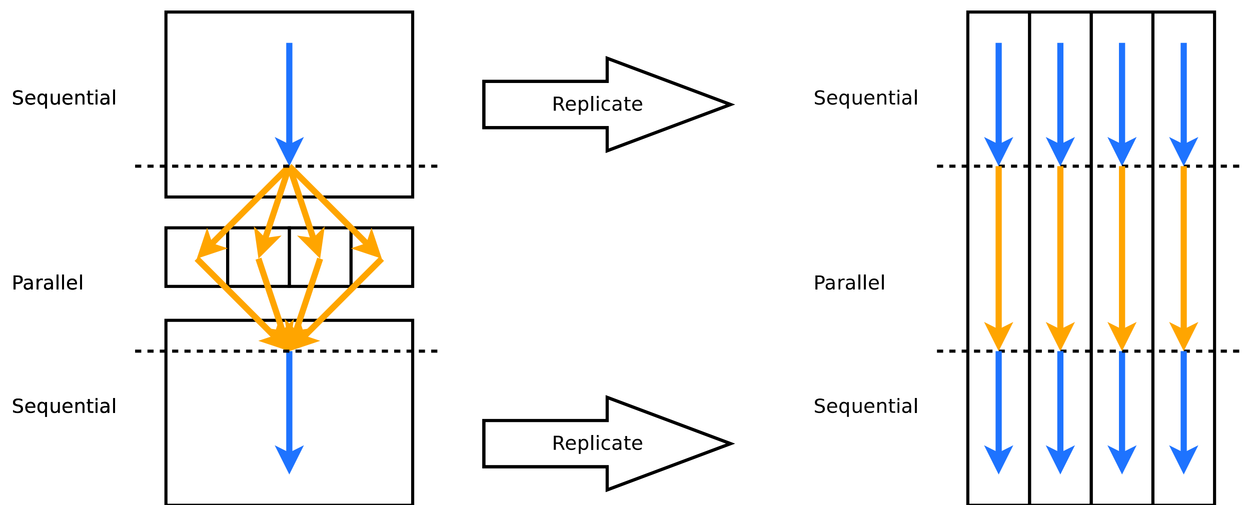


Figure 2.8: Distributed Memory SPMD Parallelism

SPMD parallelism provides an interesting mix of task parallelism and data parallelism. It provides some of the structure of data parallelism while retaining some of the asynchrony available in task parallelism. All of the communication and synchronization mechanisms that are available in task parallelism are available in SPMD parallelism. However, the structure of limiting the number of tasks to those created at program start makes reasoning about the application easier than task parallel algorithms with arbitrarily executing tasks.

Distributed Memory SPMD Parallelism. Distributed memory parallelism is slightly unique compared to distributed memory task parallelism and distributed memory data parallelism. Notice how in distributed memory data parallelism in Figure 2.5b and distributed memory task parallelism in Figure 2.6b that the tasks cross memory space boundaries. However, in distributed memory SPMD parallelism in Figure 2.8 no tasks cross the memory space boundaries. This lack of crossing memory boundaries in distributed memory SPMD parallelism is due to the fact that all tasks are created at program start and persistent until program stop. The tasks in distributed data parallelism and distributed task parallelism are created by the master task which resides in a memory space. The newly created tasks must cross memory space boundaries when they are being created in other memory spaces. However, in SPMD parallel applications, the tasks still need to meet to synchronize or share data. These interactions are not represented in the figures, but do cross memory space boundaries.

Shared Memory SPMD Parallelism. SPMD parallelism executing in shared memory is essentially the same as SPMD parallelism executing in distributed memory, just without crossing memory space boundaries. An example for shared memory SPMD parallelism can be seen in Figure 2.7. The SPMD tasks execute independently and need to be synchronized through synchronization and communication methods in just the same way as the distributed memory SPMD. The major difference is that the underlying mechanisms of the

communication library can take advantage of the shared memory structure to provide more optimized communication.

Universal SPMD Parallelism Compilation. Similarly to data and task parallelism, in an intermediate language that can translate from any notation to any other notation, there is no difference between the representation of a shared memory or a distributed memory SPMD algorithm from and implementation standpoint. Once the algorithm is written with SPMD parallelism, it can generate shared or distributed memory backend code by a simple change of a flag at compile time. Furthermore, the system can leverage the same synchronization and communication mechanisms available within task parallelism.

2.4 Conclusions on Universal Translation

For a parallel intermediate language to be able to translate from any parallel notation into any other parallel notation it needs to be able to represent data parallelism as well as task parallelism. Additionally, it will need to be able to implement these forms of parallelism in shared memory as well as distributed memory environments. Furthermore, we believe that in order to perform these tasks successfully, the intermediate language needs to be able to make no distinction between an algorithm implemented in shared memory or distributed memory. The distinction between the two memory models should be made by the compiler during code generation.

We believe we can do many of the most common transformations, like the ones listed in the table. Furthermore, any new cases not listed can be handled by compiler transformations in the future to generate code for existing parallel mechanisms. We have carefully considered the ideas presented in this chapter and the notations of popular parallel programming languages and present our parallel intermediate language that incorporates these notions in

the next chapter.

Chapter 3

The Parallel Intermediate Language

In this chapter we will take the ideas discussed in Chapter 2 and describe how we implemented the Parallel Intermediate Language (PIL) from those ideas.

A PIL program consists of three parts. First, there is a description of the parallelism with PIL nodes. Second, the user defines the computation to be done inside of the parallel nodes. In our implementation, we use C as the sequential computation language. Finally, there is a description of the data that will be passed between nodes. This gives PIL enough information about the data to be able to make sure it is available to the node before the node begins executing.

A PIL node declaration contains a description of the nodes and arcs in a PIL program. With this information we can statically create a task graph with all possible edges in the program at compile time. Each node declares one of the user body functions to be executed at that node.

Data in PIL can either be a primitive of the implementation language, or a data-block. The data-blocks are arrays of primitives in the implementation language. Data-blocks are allocated and deallocated by routines provided by PIL. Packaging arrays into data-blocks allows PIL to manage the data-blocks as necessary in the generated runtime code to make

```
// data
int i;
int rank;
int target;
gpp_t index_array;
gpp_t data_array;
int rank;

// body functions
void before(int *target, gpp_t index_array, gpp_t data_array) {
  printf("In node before.\n");
  *target = 2;
}

void hello(int *target, gpp_t index_array, gpp_t data_array, int i) {
  printf("Hello from %d!\n", i);
  *target = 3;
}

void after(int *target, gpp_t index_array, gpp_t data_array) {
  printf("In node after.\n");
  *target = 0;
}

// pil nodes
node(1, rank, i, [1:1:1], target, [0], [2], before(&target, index_array, data_array))
node(2, rank, i, [0:1:3], target, [1], [3], hello(&target, index_array, data_array, i))
node(3, rank, i, [1:1:1], target, [2], [0], after(&target, index_array, data_array))
```

Figure 3.1: An example PIL Hello World program.

sure the data is available before the user code begins executing.

3.1 Syntax and Semantics of PIL

PIL programs are composed of three main sections. First, there is the declaration of all data that will be shared amongst all of the parallel tasks. Second, the body functions that are the sequential tasks that are coordinated into parallel computation. Finally, the declaration of the PIL nodes that specifies the parallelism in the application.

An example PIL program can be seen in Figure 3.1. This is a simple Hello World program where the execution of the second node that prints the string is executed in the data parallel style. This is a running code example we will discuss in detail later in this chapter.

3.1.1 Declaration of Data in PIL

The programming language C is used as the underlying base language for the sequential parts of the code. Thus, the data that is declared to be used and passed between PIL nodes are declared using C syntax. Each variable has a type and an identifying name. They are declared the same way that globals are declared in the C language. When the compiler parses the program, the compiler finds all of these declared variables and encapsulates them in a method suitable for the chosen backend. The specifics of the encapsulation and code generation is discussed in Chapter 4.

In Figure 3.1, there are five variables that are encapsulated and passed between PIL nodes: `i`, `rank`, `target`, `index_array`, and `data_array`. More about the specific use of these variables will be discussed in Section 3.1.3.

Notice that we have provided a special built in structure called `gpp_t`. The implementation of this special type is included in Figure 3.2. This encapsulation is needed for pointer types in PIL. Some backends, specifically OCR, require pointers to be encapsulated as a Global Unique Identifier, GUID. We provide a special GUID Pointer Pair type, GPP, to pair the GUID with its pointer value. In OCR, dynamically allocated memory must be referenced not by a pointer, but by a GUID. This is done because OCR is designed for a machine with a single physical address space for memory. There is no virtual memory. Additionally, the dynamic runtime is free to move data in between EDT invocations. For example, dynamically allocated data may be referenced by a pointer with address A in one EDT, and by pointer with address B in the next EDT, where $A \neq B$. The GUID provides a non changing identifier for the data. PIL encapsulates all dynamically allocated memory into the `gpp_t` type. PIL ensures that if the pointer changes between PIL node invocations that the pointer field is set correctly before the user code can access it.

The `index_array` and `data_array` items are required for every PIL node. They are used like an adjacency list. The `data_array` is an array of `gpp_t` items. Each iteration of a parallel

```
#ifdef PIL2OCR
#include "ocr.h"
typedef ocrGuid_t guid_t; // Defines NULL_GUID
#else
#define NULL_GUID NULL
typedef void* guid_t;
#endif // PIL2OCR

typedef struct {
  guid_t guid;
  void *ptr;
} gpp_t;
```

Figure 3.2: The implementation of the gpp_t type.

node has a certain portion of the `data_array` belong to it. The `index_array` contains for each iteration, i, of a data parallel node, n, the elements in `data_array` for that iteration. In other words, `index_array[`i`]` contains the starting element in the `data_array` for iteration i, and iteration i owns all elements of the `data_array` up to, but not including, those in the `data_array` at index `index_array[`$i+1$`]`.

Every PIL node also takes a `target` variable, or in the case of task parallelism a `targets` array. The `target` variable is set during the execution of the body function to indicate the label of the next PIL node to be fired. In the case of task parallelism, the user can set the contents of the `targets` array to contain a list of all of the PIL nodes to be fired in parallel, as discussed in detail in Section 3.1.3.

3.1.2 Body Functions of PIL Nodes

Body functions in PIL are declared using normal sequential C syntax, as seen in Figure 3.1. As discussed in the previous section, each PIL body function must include the `target`, `index_array`, and `data_array` variables. This particular example does not make use of the `data_array` and `index_array` variables, but they are still necessary to appear in the body function prototypes. Each PIL node can only have a single body function. Every body function must be reenterable and serializable, or the behavior of the function is indeterminate.

```
node(label, rank, index, [lower:step:upper], target, [<preds>], [<succs>], func())
```

Figure 3.3: The syntax for the declaration of a PIL node.

3.1.3 PIL Nodes

In this section we will describe the syntax of a PIL node, and describe each part of its declaration. A PIL node has the form seen in Figure 3.3.

Label. The label is a unique identifier for each node that is a positive integer. The label 0 is a special case that means `entry` when used as a predecessor or `exit` when used as a successor. The user selects his or her own labels for each node, but must make sure that each node has a unique label. When PIL is generated from a frontend compiler, the compiler can simply keep a count of the PIL nodes generated, and automatically label each node.

Rank. The rank variable is used for task parallelism. This variable can be used when creating multiple instances of the same PIL node so the user can differentiate between the instances. Furthermore, the rank variable is important in SPMD mode. For SPMD mode the specified number of tasks, N, are created at program startup with ranks 0 to $N-1$. This variable must be passed in to the body function to be accessed. When used in task parallelism, the rank variable can be set to any value the user chooses, so that the task might know what work it is supposed to do. For example, in the Cholesky factorization discussed in Chapter 6, we make extensive use of the rank variable to distinguish tasks. Each tile of a tiled array can be assigned a tile number, and a created task can check its rank variable to know which tile number to update. The rank variable need not be named rank, but, at the declaration of the PIL node, the name of the rank variable is specified. The variable must still exist for data parallelism, but since there is a single master task, it doesn't make much sense to access this variable.

The value of the rank variable is forwarded on from one PIL node instance to the next, so that a sequence of PIL nodes can be seen as a single rank. For example, with SPMD parallelism, P tasks are started up at program start, and their ranks are set to the values 0 through $P-1$. All ranks start execution as separate instances of the same PIL node. If there is no further spawning of tasks (i.e., nested parallelism), then as a rank moves from one PIL node to another, it can still be identified as a single rank since the value of the rank variable does not change. This allows the user to reason about their codes by breaking tasks into ranks, with each rank made up of a succession of PIL node instances.

Index. The index variable is used for data parallelism. This is the loop index variable for the parallel loop that represents which iteration of a parallel loop the body function is currently processing. The variable name is chosen by the user, or generated by the frontend compiler, and can have any legal identifier name. This variable must be passed into the body function to be accessed. The value of the variable is set by PIL before the body function is called, and thus it is a read-only variable.

The Iteration Space [lower:step:upper]. This is the iteration space of the parallel loop over the computation. The range is from `lower` to `upper`, inclusive, by `step`. These values must be integers, and can be constant values or variables. If they are variables, they must be passed in to the body function to be accessed.

Target. This target variable must always be passed into the body function. The target variable may have any legal identifier name. Upon completion of the body function, it specifies the label of the PIL node to fire next, and must be set within the body function. If the PIL node will spawn multiple successors, this variable must be an array, and the contents of the array at the termination of the body function will contain the labels of all of the PIL nodes to fire next. If the target variable is an array, upon successful completion of the body

function, PIL will create and spawn one new task for each target specified within the array. The same target label can be specified more than once in an array variable, and PIL will create one instance for each enumeration in the array.

The Predecessor List. This list contains all of the possible predecessors of this PIL node. The predecessor list and successor list are used to create a task graph during compilation. The list is a comma separated list of PIL node labels. For example, the predecessor list $[2, 3, 5]$ says that the node can have as a predecessor node 2, 3, *or* 5. Each node will have as its predecessor precisely one of the nodes in its predecessor list. We have a special syntax for confluence nodes.

Confluence Nodes. If a node will have more than one predecessor it is a confluence node. Multiple predecessors are grouped with (). For example, the predecessor list $[2, (3, 5)]$ says that the node can have as its predecessors either node 2, or nodes 3 *and* 5. Once again, each node can have as a predecessor precisely one of the nodes in its predecessor list. Now, however, the items in the list are the node 2 and the grouped nodes $(3, 5)$. The predecessor list allows for precisely one level of grouping. This means that the lists $[4, 3, 8]$ and $[(1, 4), 5]$ are valid predecessor lists, but the list $[1, (2, (3, 4))]$ has no meaning, is invalid, and will result in a compiler error.

The Successor List. This list contains all of the possible successors to this PIL node. The predecessor list and successor list are used to create a task graph during compilation. The list is a comma separated list of PIL node labels. For example, the successor list $[4, 7, 9]$ says that the node will set its target variable to fire a successor node with label 4, 7, *or* 9. Each node will have as its successor precisely one of the nodes in its successor list. We have a special syntax for spawn nodes.

Spawn Nodes. If a node will have more than one successor it is a spawn node. Multiple successors are grouped with (), in just the same way as for confluence nodes. For example, the successor list $[4, (7, 9)]$ says that the node can have as its successors node 4, or nodes 7 *and* 9.

Body Functions. The body function is a C function, and it is written here exactly as it will be called, including how the variables will be passed to the function, for example, by value or by reference. A concrete example can be seen in Figure 3.1. In that example, the `target` variable is passed by reference to the body functions while all of the other variables are passed by value. Following C conventions, any variable that will change within the body function must be passed by reference, or any changes to its state will not be saved when the function returns.

3.1.4 Library Functions Provided with PIL

PIL includes some library functions built into the language. These library functions are provided as a way to interact with the underlying runtime system without having to have any knowledge about how the actual backend code is generated. The library functions encapsulate common resource management operations that need to be exposed to the user, but are written to handle any of the available backend languages. Most notably are the communication library and memory management library.

The Memory Management Library

Essential to any program is the ability to allocate and manage memory. We have found it sufficient to provide two functions to implement a full memory management system for PIL. These functions encapsulate the corresponding memory management systems of the underlying generated runtime systems.

Allocation of Memory. The allocation of memory is handled by the `pil_alloc(gpp_t *g, size_t size)` routine. The routine takes two arguments: a pointer to a `gpp_t`, `g`, and the size of the memory to be allocated, `size`. Upon completion of the routine, the contents of `g` will be set as a valid data block GUID and a pointer to the contents of the data block. For the most part this function behaves similarly to the way that the C `malloc` function behaves. However, we have found it necessary to encapsulate data within data blocks for certain backends, notably OCR. Once the memory is allocated and the GPP set, the user must use the GPP to access memory. We cannot simply rely on pointers as the limitations of the OCR runtime have prohibited this. However, the user can safely pass this GPP around. PIL is responsible for making sure that the pointer field within the GPP is set to the valid pointer that corresponds to where the data is located, should the runtime decide to move the data to a new location.

We have limited the implementation of memory allocation to the single `pil_alloc` routine. Other common functions such as `realloc` or `calloc` are considered by us to be syntactic sugar for memory management and thus are not strictly necessary. Their operations can be implemented by the user if required by building on the supplied library functions.

Deallocation of Memory. The deallocation of memory is handled by the `pil_free` routine. This routine takes as its one argument a GPP, `g`, which upon successful completion will be deallocated. This function is analogous to the C function `free`. Once a GPP has been deallocated, an access to the contents of the memory location returns an undefined value.

The Communication Library

The communication library is used to synchronize and share data amongst PIL tasks. This is useful for task parallelism and SPMD parallelism, as well as shared memory and distributed memory. The operations of the communication library work similarly to those of MPI. The

operations must be treated as if they are executing on a distributed memory machine with each task in a separate memory space. However, if two tasks happen to be running on the same shared memory machine, the library will take advantage, and use more optimized methods for communication. Thus, there is only one mode of writing programs in PIL assuming distributed memory, but the program may actually be run as a shared memory program or a distributed memory program, simply by performing a recompile to generate the appropriate code.

Each operation of the communication library is represented within PIL as a special node. These special nodes have a slightly different syntax than the previously discussed general PIL node. Communication and synchronization operations must be contained within special nodes for two reasons. First, as previously discussed, PIL programs need to be serializable. This can only be achieved if the communication and synchronization operations are in their own nodes, and cannot be called by body functions. Second, some backends (e.g., SCALE and OCR) require that the tasks are non-blocking. However, most synchronization and communication operations are by nature synchronous, which causes blocking. By providing a special PIL node for these blocking operations, they can be broken into two separate tasks: a pre-synchronization task that initiates the synchronization operation, and a post-synchronization task that is executed when the synchronization operation has completed. The two staged synchronization operation prevents either of the tasks from having to block while waiting for the operation to complete.

We have distilled the communication library down to three operations: `pil_barrier`, `pil_send`, and `pil_recv`. These three operations are necessary to implement any communication pattern that we have come across. Other more complex operations such as broadcast, reduce, or scan can be implemented by composing these three operations. It is possible to provide these compound operations in a more efficient form than what a user can compose from the basic operations, but we have chosen not to at this point.

```
1 pil_barrier(label, rank, [<preds>], [<succs>])
```

Figure 3.4: Declaration of a `pil_barrier` node.

```
1 pil_send(label, rank, [<preds>], [<succs>], dest, size, offset, buf)
```

Figure 3.5: Declaration of a `pil_send` node.

Global Barriers. The first operation that we have provided is `pil_barrier`, and its syntactical declaration can be seen in Figure 3.4. The components in the declaration of a `pil_barrier` node are the same as those previously described for a general PIL node in Section 3.1.3. This is a global barrier operation that requires all existing tasks to participate. While not a communication operation, it is a synchronization operation. The barrier operation is currently implemented as a two stage counting barrier. Each time a new task is created or ends, a count of the total currently active tasks is updated. When a task enters the barrier node, the barrier counter is incremented, and a continuation task is created that will be executed when the final task has entered the barrier. Once the final task enters the barrier, all of the continuation tasks are executed to exit the barrier.

Sending of Data. The operation that implements the sending of data is `pil_send`. Figure 3.5 shows the syntax to declare a `pil_send` node. The first four components `label`, `rank`, the predecessor list, and successor list are the same as those described in Section 3.1.3. The `dest` variable contains the rank of the destination task. The `size` variable contains the size of the message to be communicated. The `buf` variable is a GPP that contains the message to be sent. The `offset` variable allows the user to send a portion of the buffer, starting at the location `buf->ptr+offset` and ending at the location `buf->ptr+offset+size`, if they so choose. Normally the offset is 0, and the `size` variable specifies the entire size of the buffer.

Sends in PIL are asynchronous and blocking. Asynchronous means that they need not

wait for the corresponding receive to complete before returning. Blocking means that when the send returns, the buffer supplied to the send routing is free to be reused. The blocking call is completed by copying the contents of the buffer into a communication buffer. When the corresponding receive is initiated, the communication buffer will be drained and made available for the next send.

A blocking send is different than blocking a task. The task performing the send is still making progress by copying the send buffer to the reserved communication buffer, unless the buffer is full. In which case we need to create a continuation task to complete the copy once the buffer is empty. In PIL's implementation we have a reserved communication buffer for each pair of ranks. In other words, if rank A wants to send a message to rank B, the buffer AB is used. If rank B wants to send a message to rank A, a different buffer BA is used.

An example program using `pil_send` can be seen in Figure 3.6. This example illustrates the case that a send operation might need to block. In this example rank A sends a message to rank B to populate the buffer AB, and then initiates a second send to B. If it happens that the second send to B occurs before B has posted its first receive, the buffer AB will still be full from the previous send. Now, this second send operation will need to block. However, blocking operations are prohibited in PIL, and we have to utilize a two stage send operation. The current send operation initiates a continuation task that will complete the send once the buffer has been drained by B's receive.

Our experiments, discussed later, in trying to make the implementations of the algorithms as asynchronous as possible, has lead to some cases in which the exact scenario just described occurs frequently, and causes unnecessary overhead in sending operations. We found that we can reduce the overheads and can alleviate the contention for the buffers by utilizing circular buffers. Circular buffers allow for multiple buffers for each AB pair so that multiple sends can be in flight from A to B before all the buffers fill up and cause the send to block. The number of the buffers needed for a single pair is application dependent and can be set

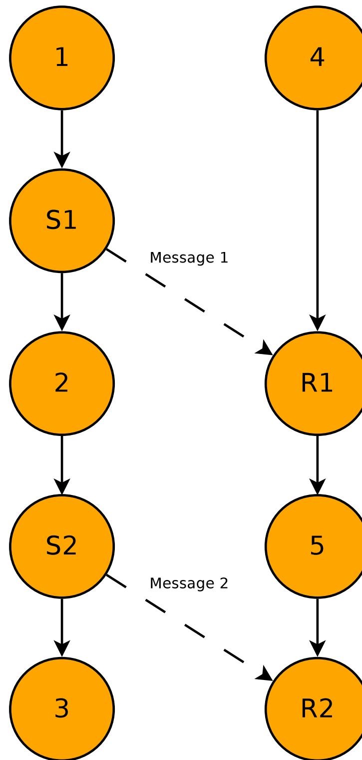


Figure 3.6: Task graph for communication example.

at runtime by the user to tune the applications performance.

Receiving of Data. The operation that implements the receiving of data is `pil_recv`. Figure 3.7 shows the syntax to declare a `pil_recv` node. The first four components `label`, `rank`, the predecessor list, and successor list are the same as those described in Section 3.1.3. The `src` variable contains the rank of the source task. The `size` variable contains the size of the message to be communicated. The `buf` variable is a GPP that will contain the message to be received. The `offset` variable allows the user to send a portion of the buffer, starting at the location `buf->ptr+offset` and ending at the location `buf->ptr+offset+size`, if they so choose. Normally the offset is 0, and the `size` variable specifies the entire size of the buffer.

Receive operations, by definition, are synchronous operations. This means that the re-

```
1 pil_recv(label, rank, [<preds>], [<succs>], src, size, offset, buf)
```

Figure 3.7: Declaration of a `pil_recv` node.

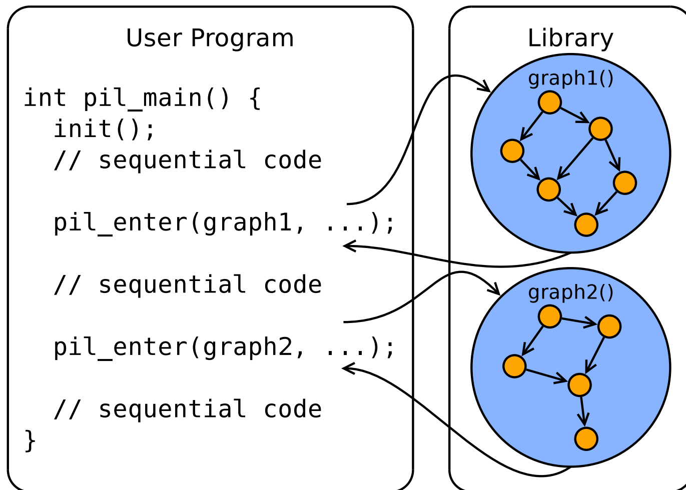


Figure 3.8: PIL library example.

ceive operation cannot complete until the corresponding send operation has completed. Thus, receive operations in PIL are two stage nodes. If, when the receive node begins, the communication buffer is already full, it may empty the buffer and continue without blocking. However, if the buffer is not full, the data to be received is not ready, and the node must create a continuation task to execute when the buffer has been populated by the corresponding send operation.

3.1.5 Support for the Implementation of Libraries

We have provided support for programming libraries directly in PIL. A unique challenge of implementing a library in a task graph based language is providing a programming interface that the user can use and not have to break the code into multiple nodes. We accomplish this by providing two new constructs into the PIL language: the `pil_main` function and the `pil_enter` call. The interactions of these functions can be seen in Figure 3.8.

The `pil_main` function. The usage of this function is a signal to the PIL compiler to process the program in library mode. The `pil_main` function provides a single context to contain user code. Inside of the function, the program has sequential semantics. There is no parallelism within this function. Parallelism is achieved by calls to the `pil_enter` function. Normally program execution will begin at the root of the task graph specified by the user's program. However, when the `pil_main` function is used, program execution begins with this function. The user can make calls to the root of multiple task graphs through the use of the `pil_enter` function. If the program is compiled with the SPMD flag, the multiple SPMD tasks are created and the first place the user can execute code is within the `pil_main` function. In other words, multiple SPMD tasks are created with `pil_main` as the entry point.

The `pil_enter` function. The user can supply a collection of library PIL graphs to execute in parallel. The `pil_enter` function allows calls to parallel library PIL graphs from within the `pil_main` function. All parallelism is contained within these graphs. The parallelism may be of any form supported by PIL, and the graph may have any shape.

If the program is compiled with the SPMD flag, each SPMD task can enter its own library functions. Each task may enter its own instance of the library. Usually, in SPMD mode each task will call a sequential graph. However, nested parallelism, as discussed in Sections 2.1.2 and 3.2.4, can be achieved by calling a graph with parallelism within it.

Programming with Libraries. Programming directly in PIL with the usage of libraries provides a unique user experience in PIL. Since the `pil_main` function has sequential semantics, and all of the parallelism is contained within the graphs that are entered, the programs can have a very clean implementation that is easy to reason about. If enough library graphs are provided with a library implementation, the user need not even know how to use or mange PIL nodes! For example, a library implementation could be provided for the general multiplication of two matrices, `GEMM`. The library could provide the graph of PIL nodes that

implements the operation, as well as a wrapper function for the GEMM operation. The wrapper function could make the call to pil_enter, and the user need not know the implementation of the GEMM function. This allows the user to have an efficient implementation of the GEMM function, while removing the specifics of the function's implementation from the user. The user may then focus on programming their algorithm, and not on the management of tasks, nodes, or parallelism.

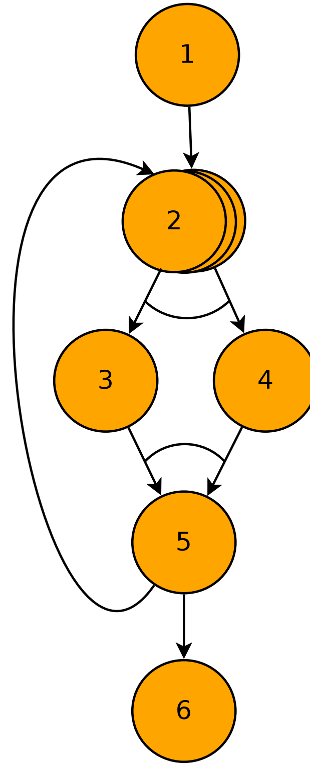
3.2 Parallelism in PIL

In order to facilitate the compilation of any notation to any other notation, PIL supports all of the forms of parallelism discussed in Section 2.1. A PIL program is made up of a task graph of PIL nodes, and is generated by a frontend compiler. An example application in pseudocode can be seen in Figure 3.9a. This example uses task parallelism with a cobegin statement, as well as data parallelism with the parallel forall loop. In this example, all sequential statements have been encapsulated into sequential functions, labeled func1 through func6. This encapsulation into functions needs to be performed by the frontend compiler, and they will be used as body functions of the generated PIL nodes, one for each PIL node.

The generated PIL code from the pseudo code example can be seen in Figure 3.10, and the corresponding task graph can be seen in Figure 3.9b. In the task graph, each circle in the figure represents a single PIL node, and the graph is made up of six PIL nodes. The arc between the arrows after node 2 represent a spawn operation, and the arc between the arrows before node 5 represent a confluence operation. We will be referencing this example PIL program as we talk about the different types of parallelism available in PIL.

```
 1 func1();
 2 do {
 3   forall i in [0 to N] {
 4     func2(i);
 5   }
 6   cobegin {
 7     func3(&not_done);
 8     func4(&not_done);
 9   }
10 } while(not_done);
11 func6();
```

(a) Pseudo code



(b) Task graph

Figure 3.9: Pseudo code and task graph for an example PIL program

```
 1 // data
 2 int i;
 3 int target;
 4 gpp_t index_array;
 5 gpp_t data_array;
 6 bool not_done;
 7
 8 // body functions
 9 void func1(int *target, gpp_t index_array, gpp_t data_array) {}
10 void func2(int *target, gpp_t index_array, gpp_t data_array, int i) {}
11 void func3(int *target, gpp_t index_array, gpp_t data_array, bool *not_done) {}
12 void func4(int *target, gpp_t index_array, gpp_t data_array, bool *not_done) {}
13 void func5(int *target, gpp_t index_array, gpp_t data_array) {}
14 void func6(int *target, gpp_t index_array, gpp_t data_array) {}
15
16 // pil nodes
17 node(1, rank, i, [1:1:1], target, [0], [2], func1(&target,index_array,data_array))
18 node(2, rank, i, [0:1:N], target, [1,5], [(3,4)], func2(&target,index_array,data_array,i))
19 node(3, rank, i, [1:1:1], target, [2], [5], func3(&target,index_array,data_array,&not_done))
20 node(4, rank, i, [1:1:1], target, [2], [5], func4(&target,index_array,data_array,&not_done))
21 node(5, rank, i, [1:1:1], target, [(3,4)], [2,6], func5(&target,index_array,data_array))
22 node(6, rank, i, [1:1:1], target, [5], [0], func6(&target,index_array,data_array))
```

Figure 3.10: PIL code for the example PIL program

3.2.1 Data Parallelism

A single PIL node can have multiple instances executing simultaneously on different data items, thus allowing data parallelism in a single node. We represent a parallel loop as a single PIL node with multiple simultaneously executing instances. For example, in Figure 3.9b, Node 2 may have multiple instances concurrently executing. Thus, node 2 is a data parallel node. The remaining nodes only have a single instance and can be considered sequential nodes.

3.2.2 Task Parallelism

Multiple different PIL nodes can be executed simultaneously to facilitate task parallelism. For example, in Figure 3.9b, node 3 and node 4 will execute in parallel before joining at Node 5. The spawn of node 3 and 4 will happen after node 2 completes. The confluence of nodes 3 and 4 will happen before node 5 can start.

3.2.3 SPMD Parallelism

SPMD parallelism is achieved by specifying at compile time, that the PIL program will execute in SPMD mode. If the program is specified to execute in SPMD mode, the entry of the PIL program will be started with multiple tasks executing different instances. The number of ranks to be created is specified with the environment variable `PIL_NUM_THREADS`. This variable is read at program start when creating the multiple tasks for the ranks.

3.2.4 Composing Parallelism

All types of parallelism in PIL can be combined into one program. Figure 3.9b is a good example. We use both data parallelism in node 2, as well as task parallelism in nodes 3 and

4. Parallelism can also be nested in PIL. If, for example, node 3 contained a parallel loop, multiple instances of this node could all be executed in parallel.

In SPMD parallel mode, each rank begins executing with a single task composing the rank. Typically, the tasks that make up a logical SPMD rank are sequential tasks. However, if one of the tasks were to be a data parallel task with multiple instances, this would nest data parallelism within the SPMD parallelism. Furthermore, the task for a logical rank could create multiple successor tasks for task parallelism within SPMD parallelism.

3.3 Memory Models

PIL supports both shared memory and distributed memory as discussed in Section 2.2. The memory management library discussed in Section 3.1.4 employs the memory space visibility discussed in Section 2.2.3. When data is allocated, it is placed at the level of visibility that any task within the same memory space (shared memory) can access it. However, when the data is allocated, and the GPP is set, only the allocating thread will have the GPP to access the memory. The allocating thread will have to share the GPP with other threads that need access to the data through communication.

PIL requires that once a piece of memory is allocated, it remains visible in the allocating tasks memory space. A runtime, such as OCR, might be able to move the block of memory, but the GPP will always be set such that the pointer of the GPP points to the memory block. Furthermore, a user cannot move a block of memory. The only way to have a previously allocated block of memory to be visible in a new memory space (distributed memory) is to copy the block through communication. Once the data has been communicated, there will be two copies of the data block: one in the sending task's memory space, and one in the receiving task's memory space. It is up to the user to consolidate changes made to both blocks, or to free the copy from the sender's memory space, so that it can no longer be

accessed.

3.3.1 Communication and Synchronization

The communication library discussed in Section 3.1.4 implements the ideas of communication and synchronization discussed in Section 2.2.4. In this section we relate the communication and synchronization operations available in PIL to the ideas of universal translation from Section 2.2.4.

Communication. We provide the point-to-point communications send and receive in PIL with the `pil_send` and `pil_recv` operations. These operations can be used to construct any collective communication operation the user may require. The receive operation is synchronous, but is serializable since it can be implemented as a two stage operation. The send operation is asynchronous, and thus is serializable.

Synchronization. We provide the barrier operation for synchronization with the operation `pil_barrier`. If the user requires more control and more synchronization operations than just the barrier, any synchronization operation can be implemented using the point-to-point synchronization mechanisms available with the send and receive operations.

3.4 Discussion of Parallelism and Memory Models

3.4.1 Memory Models in PIL

PIL supports shared and distributed memory. As discussed in Section 3.3, we have chosen not to specify the placement of data when allocated, but rather place the data within the allocating task's memory address space where it is sharable with other tasks in the address space. Similarly, in PIL we do not specify were tasks should be placed when they are created.

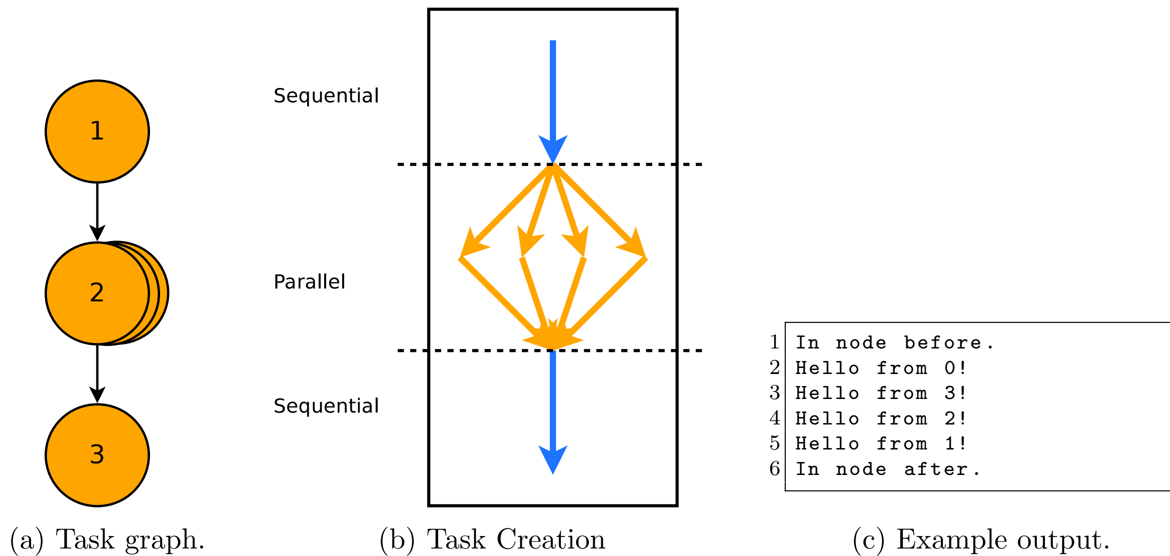
Some runtimes are only supported in shared memory, like OpenMP, and cannot handle distributed tasks. Also, some of the dynamic runtimes, like OCR, will support the automatic placement of tasks on distributed nodes. Data parallelism uses an *implicit* communication model while task and SPMD parallelism use an *explicit* communication model. Accordingly, when creating tasks for data and task parallelism we assume that the runtime will automatically place them. In SPMD parallelism, the initial tasks for each rank are placed at program startup. In SPMD parallelism we allow the specification of where to place the initial tasks for the ranks at program start. Subsequent tasks from those rank tasks might be placed by the runtime in the same memory space as the creating task or placed elsewhere if the runtime decides that is more efficient. This representation of how tasks are created allows PIL to maintain a single algorithmic description for an application that operates in both shared and distributed memory environments.

3.4.2 Parallelism

In this section we will describe how to use PIL to create the different forms of parallelism available. As an illustrative example we will refer to several PIL examples for data parallelism, task parallelism, and SPMD parallelism, including the previously presented data parallel Hello World program in Figure 3.1.

Data Parallelism

For the data parallel example, we will refer to the Hello World program in Figure 3.1. This program has three PIL nodes, with node 2 executing in data parallel mode. Since the iteration space in PIL is inclusive, there will be four instances of node 2 created with indexes $0 - 3$.

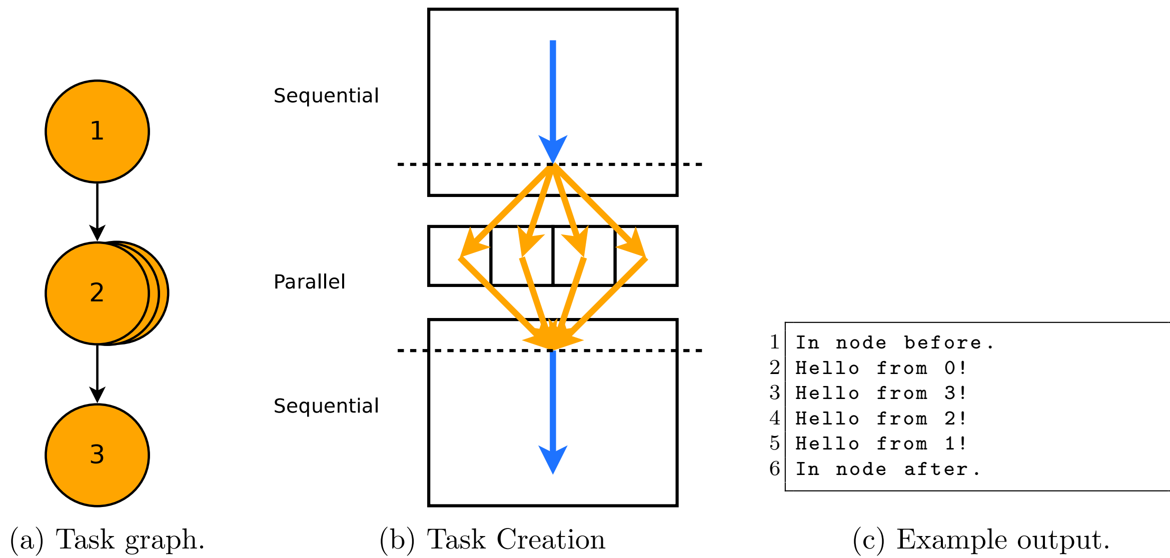


(a) Task graph. (b) Task Creation (c) Example output.

Figure 3.11: Shared Memory Data Parallel Hello World

Shared Memory Data Parallelism. An example execution of the shared memory data parallelism for Hello World can be seen in Figure 3.11. The PIL nodes graph for the program can be seen in Figure 3.11a. In Figure 3.11b each arrow represents a single instance of one PIL node executing. In this model a single thread of execution executes the sequential parts of the code. Then, when a parallel node is to be fired, the sequential node creates all of the parallel workers (fork). When the workers have completed, they all join together and begin another sequential thread of execution. Since all nodes are executing in the same shared memory space, represented by the box, no data need be moved for the computation to proceed. An example output of the program can be seen in Figure 3.11c.

Distributed Memory Data Parallelism. The data parallel model can also be applied to a distributed memory scheme. This can be seen in Figure 3.12. This example uses the same source code as the one seen in Figure 3.11. The same source code produces the same graph of PIL nodes in Figure 3.12a. The primary difference is that the instances of the PIL nodes are executed in a different memory space. The different memory spaces are represented by the multiple boxes in Figure 3.12b. In the beginning all data needed for the program is



(a) Task graph. (b) Task Creation (c) Example output.

Figure 3.12: Distributed Memory Data Parallel Hello World

contained in the first memory space, where the single sequential node is executing. In order for the new parallel instances to execute, they must have data shipped from their parents address space to their own address space. Thus, data movement is represented by an arrow leaving one address space and entering a new address space. When the parallel nodes finish execution, their results must be shipped back to the original address space for collection. An example output can be seen in Figure 3.12c. Notice that the output will be the same as for the shared memory example except for the scheduling of the parallel instances.

Task Parallelism

An example task parallel Hello World program can be seen in Figure 3.13. This program has seven PIL nodes, and since each node will only have a single instance executed, there is no data parallelism. However, at the conclusion of node 1, two parallel nodes will be scheduled: nodes 2 and 3. When node 3 completes, it will also fire two nodes to execute in parallel: nodes 4 and 5. Both nodes 4 and 5 will need to complete before the confluence at node 6 can be satisfied, and node 6 will execute. There is a second confluence at node 7, that has

to wait for nodes 2 and 6 to complete before it can be scheduled.

Shared Memory Task Parallelism. The graph of PIL nodes for the shared memory task parallel Hello World example can be seen in Figure 3.14a. A line from one node to another represents a descendent, and an arc between the lines represents a spawn or a confluence site, as discussed in Section 3.2. The execution of the created tasks for the PIL nodes can be seen in Figure 3.14b. Note that in this shared memory environment, no arrows cross the memory space boundaries. There is no communication required in this example. The output from an example run can be seen in Figure 3.14c. In the example it is possible to have up to three-way parallelism if nodes 2, 4, and 5 are scheduled and executing at the same time.

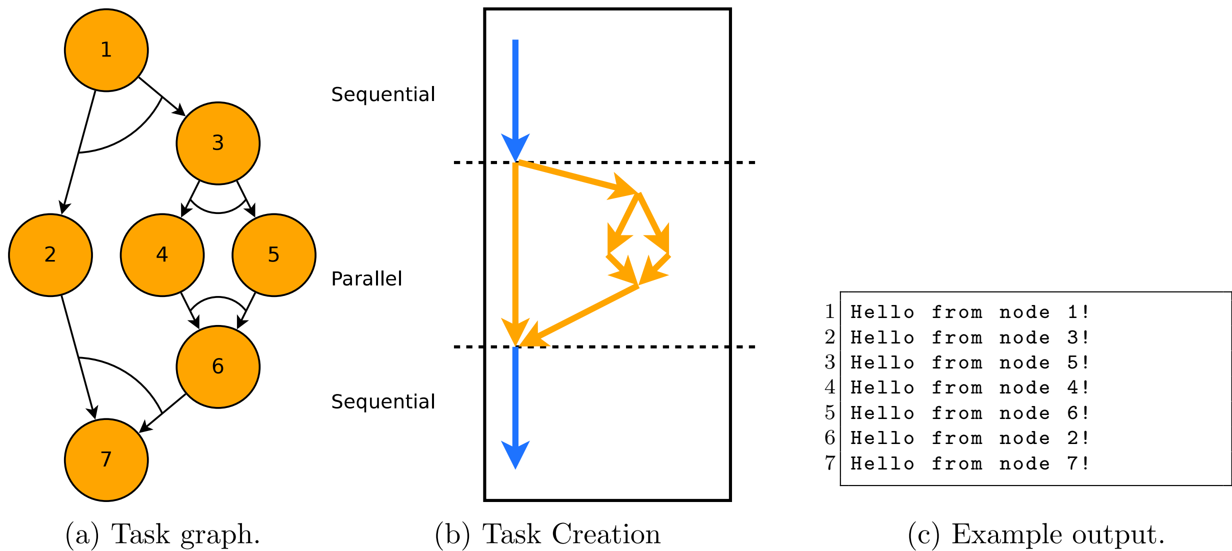
Distributed Memory Task Parallelism. The distributed memory example of the task parallel Hello World program is very similar to the shared memory example. The source code is the same, and so the graph of PIL nodes in Figure 3.15a is the same as for the shared memory version. The placement of the tasks can be different, however. In the graphic in Figure 3.15b, it is illustrated that the task for node 3 is scheduled in a new memory space, but the successors of node 3, nodes 4 and 5, are also scheduled in the same address space as node 3. The an example output of the program can be seen in Figure 3.15c, and it is expected to be the same as the output for the shared memory version of the application, if accounting for possible scheduling race conditions. For example, it is possible for node 2 to execute any time after node 1, and before node 7.

SPMD Parallelism

We will use a new example for SPMD parallelism. This example is another simple Hello World program, and its PIL code can be seen in Figure 3.16. As we discussed in Section 2.3.3, the sequential parts of the code are duplicated, and all ranks begin execution with node 1. Each rank will compute their own, identical, value for the variable `n`, that each rank will

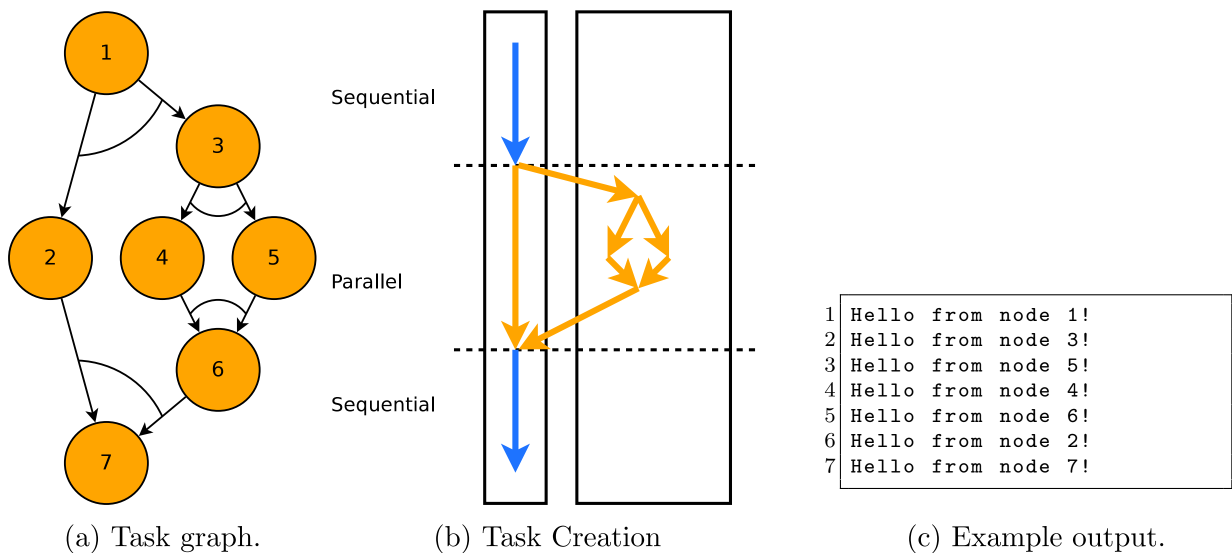
```
1  int i;
2  int rank;
3  int target;
4  int targets[10];
5  int _pil_num_targets;
6  int _pil_task_names[10];
7  gpp_t index_array;
8  gpp_t data_array;
9
10 void f1(int *targets, gpp_t index_array, gpp_t data_array, int *_pil_num_targets, int *
       _pil_task_names) {
11   printf("Hello from node 1!\n");
12   *_pil_num_targets = 2;
13   targets[0] = 2;
14   targets[1] = 3;
15   _pil_task_names[0] = 0;
16   _pil_task_names[1] = 1;
17 }
18 void f2(int *target, gpp_t index_array, gpp_t data_array) {
19   printf("Hello from node 2!\n");
20   *target = 7;
21 }
22 void f3(int *targets, gpp_t index_array, gpp_t data_array, int *_pil_num_targets, int *
       _pil_task_names) {
23   printf("Hello from node 3!\n");
24   *_pil_num_targets = 2;
25   targets[0] = 4;
26   targets[1] = 5;
27   _pil_task_names[0] = 1;
28   _pil_task_names[1] = 2;
29 }
30 void f4(int *target, gpp_t index_array, gpp_t data_array) {
31   printf("Hello from node 4!\n");
32   *target = 6;
33 }
34 void f5(int *target, gpp_t index_array, gpp_t data_array) {
35   printf("Hello from node 5!\n");
36   *target = 6;
37 }
38 void f6(int *target, gpp_t index_array, gpp_t data_array) {
39   printf("Hello from node 6!\n");
40   *target = 7;
41 }
42 void f7(int *target, gpp_t index_array, gpp_t data_array) {
43   printf("Hello from node 7!\n");
44   *target = 0;
45 }
46
47 node(1, rank, i, [1:1:1], targets, [0], [(2,3)], f1(targets, index_array, data_array, &
       _pil_num_targets, _pil_task_names))
48 node(2, rank, i, [1:1:1], target, [1], [7], f2(&target, index_array, data_array))
49 node(3, rank, i, [1:1:1], targets, [1], [(4,5)], f3(targets, index_array, data_array, &
       _pil_num_targets, _pil_task_names))
50 node(4, rank, i, [1:1:1], target, [3], [6], f4(&target, index_array, data_array))
51 node(5, rank, i, [1:1:1], target, [3], [6], f5(&target, index_array, data_array))
52 node(6, rank, i, [1:1:1], target, [(4,5)], [7], f6(&target, index_array, data_array))
53 node(7, rank, i, [1:1:1], target, [(2,6)], [0], f7(&target, index_array, data_array))
```

Figure 3.13: An example PIL task parallel Hello World program.



(a) Task graph. (b) Task Creation (c) Example output.

Figure 3.14: Shared Memory Task Parallel Hello World



(a) Task graph. (b) Task Creation (c) Example output.

Figure 3.15: Distributed Memory Task Parallel Hello World

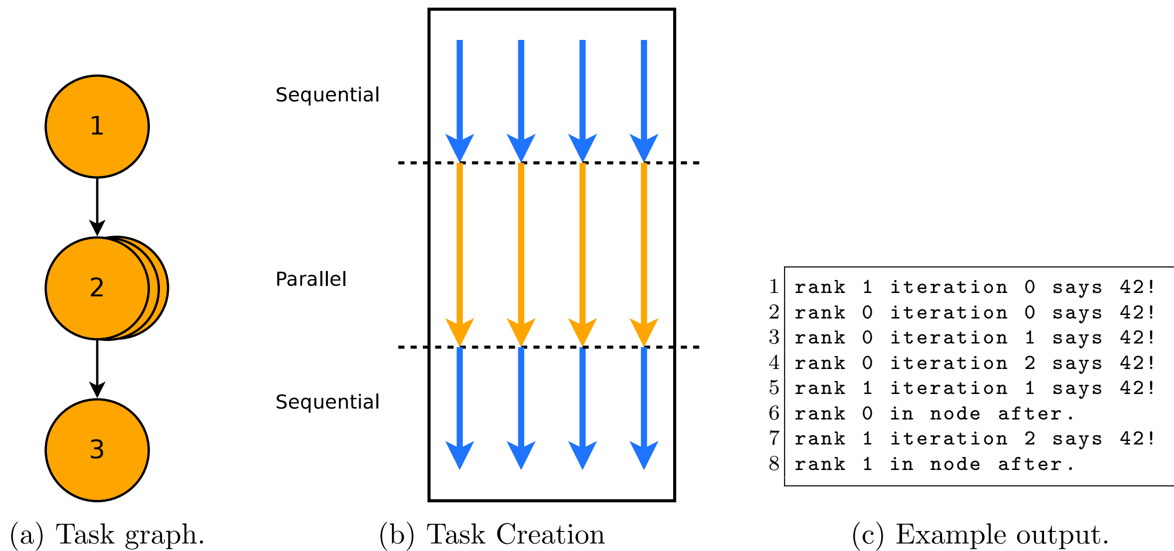
```
1 // data
2 int i;
3 int n;
4 int rank;
5 int target;
6 gpp_t index_array;
7 gpp_t data_array;
8
9 // body functions
10 void before(int *target, gpp_t index_array, gpp_t data_array, int *n) {
11   *n = 42;
12   *target = 2;
13 }
14
15 void hello(int *target, gpp_t index_array, gpp_t data_array, int rank, int i, int n) {
16   printf("rank %d iteration %d says %d!\n", rank, i, n);
17   *target = 3;
18 }
19
20 void after(int *target, gpp_t index_array, gpp_t data_array, int rank) {
21   printf("rank %d in node after.\n", rank);
22   *target = 0;
23 }
24
25 // pil nodes
26 node(1, rank, i, [1:1:1], target, [0], [2], before(&target,index_array,data_array,&n))
27 node(2, rank, i, [0:1:2], target, [1], [3], hello(&target,index_array,data_array,rank,i,n))
28 node(3, rank, i, [1:1:1], target, [2], [0], after(&target,index_array,data_array,rank))
```

Figure 3.16: An example SPMD PIL Hello World program.

store in task private data and forward on to their rank's task for node 2. This program has nested parallelism, since node 2 is a parallel node, and each rank will execute multiple instances of the node.

Shared Memory SPMD Parallelism. The SPMD shared memory model is slightly different, as seen in Figure 3.17. The graph of PIL nodes for the shared memory version of the code can be seen in Figure 3.17a. In this shared memory SPMD, the sequential parts of the code are replicated across all processing elements so that they compute identical values for all variables, as seen in Figure 3.17b. When a parallel portion of code is reached, each processing element creates its own successor to continue the computation. Once again, since all nodes reside in the same shared memory space, there is no need for data movement when a node needs data.

The output for an example run of the application can be seen in Figure 3.17c. Notice



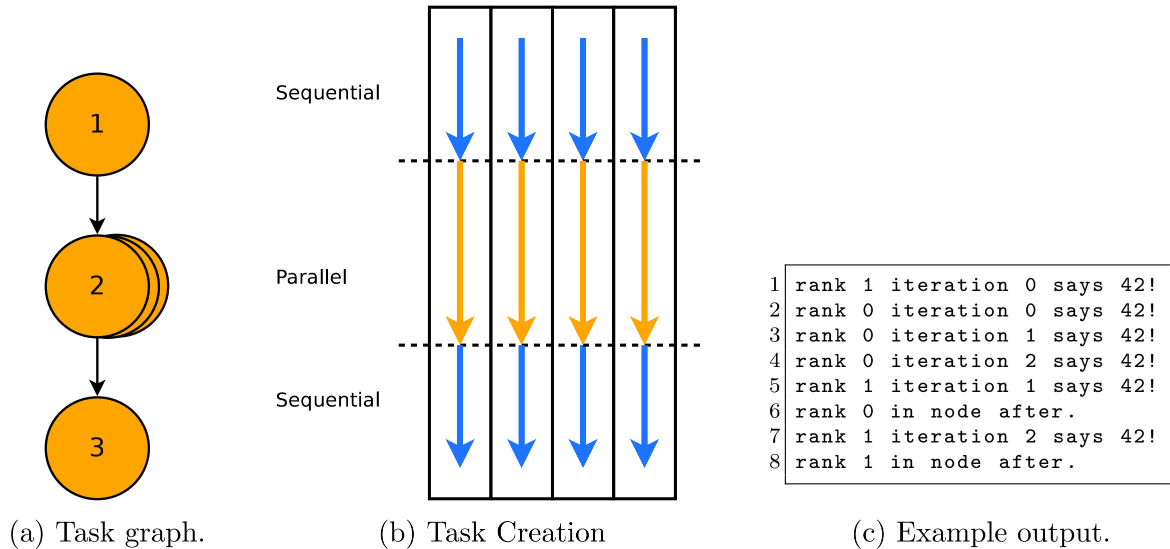
(a) Task graph. (b) Task Creation (c) Example output.

Figure 3.17: Shared Memory SPMD Hello World

how even though there are multiple instance for each rank executing node 2, that the rank variable for all of the instances remains the same. The execution of the node is part of the rank, and all instances belong to the rank. Each instance of the nested loop can differentiate between themselves by access the loop index variable, i, within the node 2 body function.

Distributed Memory SPMD Parallelism. The SPMD version of the Hello World program can also run in distributed memory, as seen in Figure 3.18. Any computation done in the sequential parts is computed exactly the same on all nodes. This replication of the computation of data prevents the need for any communication when moving from a sequential node to a parallel node or when moving from a parallel node to a sequential node. Instead, all communication has to be explicit in the algorithm and only takes place at these specific points, and will only be between the parallel nodes.

The distributed memory version of the SPMD Hello World program has the same source code as the shared memory version, and so the graph of PIL nodes in Figure 3.18a is the same as the shared memory version. At program start, all of the ranks are placed in their own memory space, as seen in Figure 3.18b. The example program has no communication, but



(a) Task graph. (b) Task Creation (c) Example output.

Figure 3.18: Distributed memory SPMD Hello World

if the tasks need to share data, it would have to happen with the communication operations discussed in Section 3.1.4. The output for an example run can be seen in Figure 3.18c, and it should be expected to be the same as for the shared memory version.

3.5 Examples of Popular Languages in PIL

In Section 2.1.3 we described how to represent the parallel constructs from several popular languages in our model of parallelism. In this section we will describe how to generate PIL code from the parallel constructs in these languages, as well as how to generate those parallel constructs in the backend languages from PIL. The languages and constructs are represented in Table 2.2.

3.5.1 Acceptance as Input

We will describe how to generate PIL code from the parallel constructs in popular programming languages from Table 2.2.

Data Parallel Constructs. The OpenMP and Chapel language support parallel loops for data parallelism. To generate PIL code for a parallel loop, the statements that make up the body of the loop must be encapsulated into a function. A data parallel PIL node can be created for the loop, and the iteration space and index variables for the loop will be the iteration space and index variables for the PIL node. The encapsulated statements in the function will become the body function of the PIL node.

Task Parallel Constructs. The OpenMP, Pthreads, Chapel, SCALE, and OCR languages all support constructs for the creation of asynchronous tasks. To create a new asynchronous task in PIL, a new PIL node must be constructed to encapsulate the task. The body function of the PIL node will need to be a function that encapsulates all of the statements that are executed by the task in its original source language. If the language supports the creation of multiple tasks simultaneously, like the OpenMP `parallel section` construct or Chapel's `cobegin` statement, all of the tasks enumerated in the source language can be spawned as successors to the PIL node that is finishing execution. In OCR and SCALE each EDT or codelet can be encapsulated by a PIL node.

SPMD Parallel Constructs. The OpenMP and MPI languages explicitly support SPMD parallelism. In SPMD parallelism, the series of tasks is specified once for all ranks, and at program start multiple instances of the ranks are created. This when generating PIL code from a language that supports SPMD parallelism, the code in the originating language can be considered as sequential, and can be broken up into subtasks for nested parallel constructs, or for communication library calls.

3.5.2 Generation as Output

Once the computation is represented in PIL, the generation of code will utilize the constructs of the language being targeted as effectively as possible. We will discuss how to generate code at a high level here. We will focus on the generation of code for specific backends from PIL in Chapter 4.

Sequential PIL Nodes. Moving from one node to another in PIL with sequential nodes can be generated into a backend rather trivially. Since the PIL nodes each have a body function the generation of backend code to execute the node is as simple as calling the body function.

Data Parallel PIL Nodes. If a node has multiple parallel instances that will execute, a data parallel operation will need to be constructed. When generating OpenMP or Chapel, that each support data parallelism with parallel loops, those parallel constructs can be generated directly. However, if code for a data parallel PIL node is to be generated for a language like Pthreads, SCALE, or OCR that have no data parallel constructs, the data parallelism must be constructed from tasks. As we have previously discussed, a parallel loop can be constructed as fork-join parallelism with tasks. In languages like OCR and SCALE, a parallel loop can be constructed from three tasks. The first task can create the third join task, as well as fork the parallel tasks to execute the body function. The body tasks can each satisfy a dependence on the join task, and the join task will execute when all of the body tasks have completed.

Task Parallel PIL Nodes. The languages Chapel, OpenMP, Pthreads, SCALE, and OCR all support the creation of tasks. When a PIL node will have multiple successors, the PIL node can call the task creation constructs within these languages directly. If a backend

language has no constructs for task creation, the successors of the PIL node will have to be arbitrarily serialized.

3.6 Portability

Portability of PIL. The generation of PIL code from the frontend language is agnostic of the underlying machine. The PIL code itself is perfectly portable from one machine to another. Depending on the capabilities of the backend runtime, PIL may need to generate different code for shared memory and distributed memory backends, so compiler flags are provided that the user can specify the desired output.

Portability of Backend Code. Once PIL has generated code in a backend runtime, the code is as portable as any other code written in the backend language. OpenMP, MPI, and Pthreads all have a history of providing portability to applications. Similarly, SCALE and OCR are portable.

3.7 Multiresolution Programming

PIL facilitates the use of multiresolution programming [12] in various ways. Some of these techniques can even be combined. Multiresolution programming is analogous to programming in a language like C and dropping into assembly when the absolute highest of performance and control is demanded. This gives the programmer the best combination of programmability and performance when representing their algorithm. However, in this analogy, the programmer loses portability when programming in assembly. This is not usually the case with multiresolution programming in PIL.

Multiresolution programming can be achieved in the following ways in PIL:

1. Programming in the frontend language and PIL.

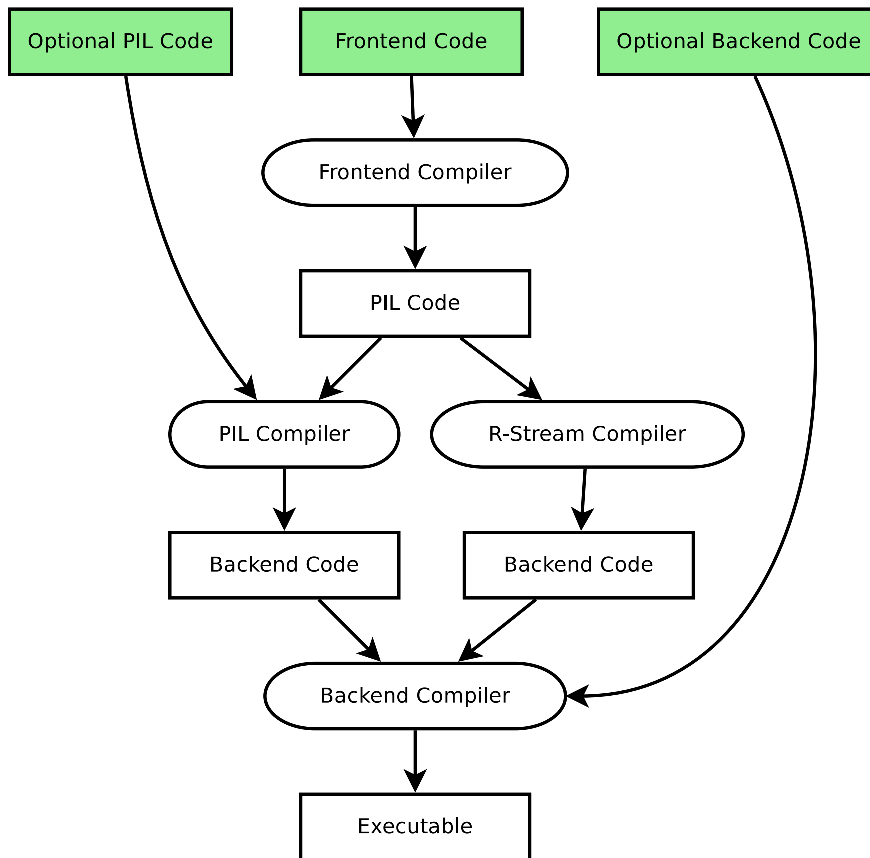


Figure 3.19: PIL compiler flow.

2. Programming in PIL and the backend language.

3. Programming in the frontend language and the backend language.

4. Auto parallelization with the R-Stream compiler.

The PIL compiler flow accepts files in three different levels, as depicted in Figure 3.19. In order to utilize multiresolution programming, the user will need to provide code at two or more of the levels.

Programming in the Frontend Language and PIL. You can use the high level language that is targeting PIL to describe the computation in a clean and concise way. However, if for some reason, the programmer needs more control, he or she can program directly in

PIL. This is directly analogous to programming in assembly with some important differences. First, while the programmer may be able to achieve a higher level of performance by writing PIL code by hand, the programmer does not lose any portability of his or her algorithm. The frontend code the programmer writes will be compiled into PIL code. Then that generated code and the hand written PIL code will be compiled together into a single representation of the backend code, thus preserving the portability of the algorithm. Second, PIL is a high level language that lends itself to efficient representation of certain algorithms. This leads to multiresolution, yet highly readable code.

Programming in PIL and the Backend Language. It is possible to write a portion of an algorithm directly in the backend runtime when programming in PIL. This lends itself to a slight loss of portability in that the PIL code may only be run on machines that support the hand coded backend.[1] This can be alleviated in several ways. It may not even be an issue if the selected backend is highly available on many machines.

It is possible to program directly in the target runtime if the programmer doesn't want his or her code to be portable. You could do this in the body function that we currently assume is sequential. However, it could be possible for the programmer to program in, for example, OpenMP when targeting the OpenMP runtime. It could also be possible to, for example, use OpenMP in the body function when targeting any of the runtimes, like SCALE or OCR.

Programming in the Frontend Language and the Backend Language. A simple extension to the two previous methods is to allow the user to write code in the frontend language as well as the backend language, and not write PIL code at all. This will greatly enhance the portability of legacy code to new runtimes. Performance sensitive algorithms,

[1]This may or may not be a problem since the programmer chose that backend for a reason. This may actually be desirable. See **Programmability with multiresolution programming** below.

or algorithms that highly benefit from the new notation can be coded by hand, while the remaining code is unchanged. This technique has the same limitations as mentioned in the previous section in that the backend code may not be as portable as PIL code.

Auto Parallelization with the R-Stream Compiler. We have integration with the R-Stream compiler [20]. The body function of a PIL node is in the sequential notation of the chosen implementation language (C). R-Stream takes as input this code, and generates code that is automatically parallelized in the target backend. The R-Stream compiler also optimizes for locality. The flow of the PIL compiler with the R-Stream compiler can be seen in Figure 3.19.

Programmability with multiresolution programming. Multiresolution programming in PIL increases programmability by lessening the burden of targeting new languages from the programmers view. If the application is already represented in one of the supported frontends of PIL, but the programmer wants to leverage the power of a new backend without rewriting the code from scratch, PIL can help. Your application can be compiled by the PIL compiler to generate the desired backend code automatically. Then, if desired, the programmer can convert one method at a time, beginning with the most performance critical ones. It may not even be necessary, or desirable to rewrite the entire application.

3.8 Optimizations in PIL

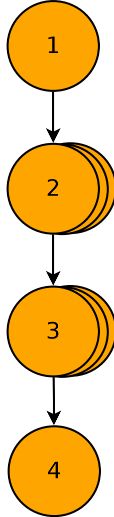
As in any intermediate language, there are opportunities for optimizations. PIL code retains the semantics of parallel code in a high level form that is easy to reason about, unlike traditional approaches that require thorough analysis to try and discover the parallelism in a low level language.

```
1 array A, B;
2 forall i in M {
3   func2(A, i);
4 }
5 forall i in M {
6   func3(B, i);
7 }
```

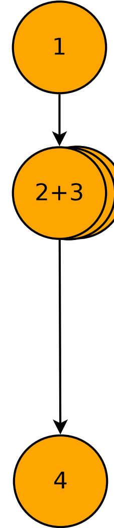
(a) Pseudocode before node coarsening

```
1 array A, B;
2 forall i in M {
3   func2(A, i);
4   func3(B, i);
5 }
```

(b) Pseudocode after node coarsening



(c) Graph before node coarsening



(d) Graph after node coarsening

Figure 3.20: Node coarsening optimization

Node Coarsening. Node coarsening in PIL is analogous to a loop fusion [1] optimization. Node coarsening is the combination of two PIL nodes that have a data parallel loop that have the same iteration space. Node coarsening is allowed when the two nodes do not operate on the same data. An example where node coarsening can result in lower overhead can be seen in Figure 3.20. In this example the parallel nodes 2 and 3 are fused into a single larger node. Node fusion is important in PIL because it reduces the number of parallel tasks that need to be started up, and task creation can have a large impact on performance.

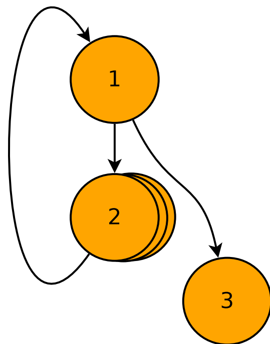
Loop Interchange. The loop interchange [1] optimization exchanges the order of two loops. Traditionally, this optimization is used to ensure that the elements accessed within the loop are accessed in the order that they are stored in memory. However, the impact in

```
1 while(i < N) {
2   forall j in M {
3     foo(i,j);
4   }
5 }
```

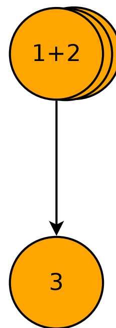
(a) Pseudocode before loop interchange

```
1 forall j in M {
2   while(i < N) {
3     foo(i,j);
4   }
5 }
```

(b) Pseudocode after loop interchange



(c) Graph before loop interchanging



(d) Graph after loop interchange

Figure 3.21: Loop interchange optimization

PIL will reduce task creation overhead. An example where loop interchange can result in lowered task creation overhead can be seen in Figure 3.21. In PIL, when a parallel loop is nested within a sequential loop, the outer loop control must be contained within its own PIL node. There will be very few instructions executed within the body function for the loop control node. If it is allowed to interchange the parallel loop with the sequential loop, the two nodes will be fused, since the parallel loop can execute all of the sequential statements in its body function.

Further Optimizations. We have discussed just a few of the optimizations that are available within PIL, however there are many more optimizations that could be studied within the PIL framework. The exploration and implementation of these optimizations is its own vast body of research, and is unfortunately beyond the scope of this thesis.

3.9 Conclusions of the Language

The parallel intermediate Language described in this chapter supports both data parallelism and task parallelism, as well as provides a mechanism for SPMD parallelism. PIL supports the composition of these parallelism in any way desired by the user. Furthermore, the language provides generation of code for both shared memory and distributed memory environments from a single algorithmic implementation while remaining portable. PIL provides mechanisms for multiresolution programming for performance as well as programmability, and the facilitation of transitioning to new programming models. In the next chapter we will describe how we implemented the code generation for the languages we focus on in this thesis.

Chapter 4

The PIL Compiler

In this chapter we describe how we implemented the PIL compiler to generate code for three parallel backend languages: OpenMP, OCR, and SCALE. For the sake of simplicity, we have limited the backend code generation to shared memory architectures. This is due to the fact that OpenMP has no distributed memory capabilities, and the distributed memory version of OCR is still under development.

4.1 Parsing a PIL Program

The parsing of the program is straightforward. The compiler records the three primary portions of the PIL code: the data, the body functions, and the PIL nodes. The information for these constructs is recorded so that the PIL compiler can generate appropriate code for the specified backend. We will discuss each implemented backend in the next sections.

Once the program has been parsed, and all relevant information is recorded, the task graph of the PIL nodes can be constructed. This task graph is used throughout the rest of the compilation process. The graph of PIL nodes is constructed by traversing the predecessor and successor lists within the PIL nodes. The spawn and confluence sites are recorded in

this graph.

Once the graph of PIL nodes is constructed, any backend independent transformations can be performed, such as the optimizations discussed in Section 3.8. In the work for this thesis, no such transformations are performed.

After the backend independent transformations are performed, the backend code generation begins. The implementations for the backends implemented in this work are discussed in the following sections.

4.2 Generating OpenMP code from PIL

4.2.1 Encapsulation of Data

The data passed between PIL nodes is encapsulated within a C structure, the GPP. It is necessary to encapsulate the data in a structure so that each created task can have its own private data. When a new task is created, the current values from the creating task's structure are copied into the new task's structure. In this way the newly created task can have access to any shared data previously created. The tasks created for data parallelism all share access to the same data structure to work on the shared data.

4.2.2 Body Functions

The body functions declared by the user, and associated with each PIL node are emitted as regular functions so they can be called from the PIL nodes.

4.2.3 PIL Nodes

The OpenMP backend has the most straightforward handling of PIL nodes of the implemented backends. Each PIL node becomes a function. From the data parallel nodes, the

compiler generates an OpenMP `parallel for` loop to create the parallel iterations over the declared iteration space. Task parallelism is handled with OpenMP's `task` construct. If there is a node with multiple parallel successor nodes, one task is created for each successor that calls the corresponding node function.

4.2.4 Memory Management Library

The data management library for the OpenMP backend is implemented as simple wrappers around the standard C functions. The `pil_alloc` call is a wrapper to a `malloc` call, and the `pil_free` call is a wrapper to a `free` call. The GUIDs are constructed by performing a simple bit manipulation on the value of the pointer returned for the memory storage. Since each allocated block of memory will have a unique pointer address, the GUIDs will be unique. The bit manipulation is performed as a simple safety measure to aid in the programming of PIL. If the bit manipulation is not performed, the user might try and dereference the GUID as a pointer, which should not be done. If the user dereferences the GUID after the bit manipulation is performed, the user will not receive the data at the address desired. The bit manipulation is performed by setting the least significant bit to a 1, which cannot be a valid allocated memory address since all of the allocated memory addresses are word aligned.

4.2.5 Communication Library

As with the PIL nodes, the communication nodes in the OpenMP backend are each implemented as a function.

Global Barriers. The global barriers are implemented as a counting barrier as discussed in Section 3.1.4. A count of the total number of tasks is maintained, and all tasks are required to enter the barrier. The last task to enter the barrier will reset the counter to the number of tasks that have entered the barrier, and begin releasing the tasks in the barrier.

Sending of Data. The sends in PIL are asynchronous and blocking. Asynchronous means that the send function can be implemented as a single function, and blocking means that the buffer must be copied to an internal communication buffer before the function can complete.

Receiving of Data. Receiving of data is a synchronous blocking operation. We take advantage of the fact that in the OpenMP runtime, the tasks are load balanced. If a task is scheduled to the worker thread for too long, it will be swapped out for another task. We implement the receive operations as a polling wait on a full/empty bit on the communication buffer. The task will poll the bit until the buffer is filled, and then copy the data from the communication buffer to the supplied buffer for the receive operation. Once the copy is complete, the full/empty but is set to the empty state, and the next PIL node is called.

4.2.6 Support for the Implementation of Libraries

The `pil_main` function has to be called as the body of the first PIL node created. Since all PIL nodes in the OpenMP backend are just function calls, the compiler simply calls the supplied `pil_main` function from the `main` function.

The `pil_enter` function is provided to enter a graph of PIL nodes. In the OpenMP backend, the function simply calls the function generated from the PIL node specified.

4.3 Generating SCALE code from PIL

4.3.1 Encapsulation of Data

The SCALE language provides its own built in type system for the creation of SWARM types. Any parameter passed to a codelet or stored within a SWARM procedure context must be a SWARM type. SWARM types are declared in the same way as C structures. The

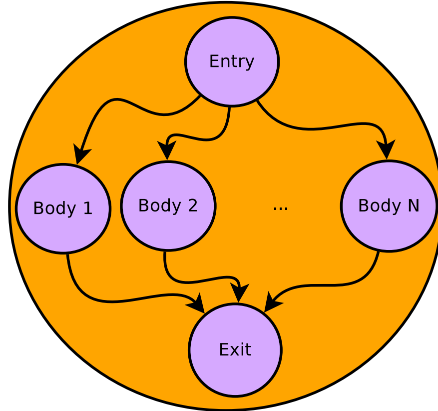


Figure 4.1: The composition of a PIL node for data-driven runtimes.

task private data for the SCALE runtime is appropriately encapsulated within a SWARM type.

4.3.2 Body Functions

The body functions declared by the user, and associated with each PIL node are emitted as regular functions so they can be called from the PIL nodes.

4.3.3 PIL Nodes

Each PIL node must be represented as a sequence of codelets. Sequential nodes can be implemented as a single codelet. However, data-driven runtimes have code generated from a PIL data parallel node as a sequence of three tasks, as depicted in Figure 4.1. For the SCALE backend there is an entry codelet, a body codelet, and an exit codelet. The body codelet can have multiple instances over the data parallel iteration space for the node. The exit codelet, and all body codelets are created by the entry codelet. The exit codelet is created with one dependence per body codelet, and each body codelet satisfies one dependence after the body function runs to completion.

Many of the operations described in Chapters 2 and 3 rely on the use of a continuation

codelet. We have collaborated with the ETI team on some of our needs for generating code for their runtime. They have plans to implement and create the continuation codelet we need. However, the implementation is not as yet complete. As such, some of our operations that rely on the continuation codelet have had to take liberties with the notion of non-blocking codelets. These variations from the ideal plan described in Chapters 2 and 3 will be described as necessary in the following sections.

4.3.4 Memory Management Library

The data management in the SCALE runtime is handled by the built in functions for `malloc` and `free`. The PIL memory management routines are provided for SCALE in the same way as for OpenMP as wrappers around the built in functions.

4.3.5 Communication Library

Each communication operation is contained within its own PIL node. As such, the implementation of the communication operations each use one or more codelets.

Global Barriers. Global barriers adherent to the codelet philosophy can be implemented as described in Section 3.1.4. However, for data-driven runtimes, such as SCALE, this requires a continuation codelet. Since we do not yet have a continuation codelet in SCALE, we have used the implementation of the barrier described for the OpenMP backend. With a data-driven runtime, this barrier requires that the number of tasks created not exceed the number of worker threads available, or the program will deadlock. This is not an issue for the programs we have implemented, since the number of tasks created is almost always equal to the number of available worker threads.

Sending of Data. The sending of data in the SCALE backend uses the same implementation as the OpenMP backend. Since the sends are asynchronous, the implementation does not violate the non-blocking codelet philosophy.

Receiving of Data. The receiving of data described in Section 3.1.4 relies on the use of a continuation codelet. As such, we have used the same polling receive that was implemented for the OpenMP backend. Since the receiving codelets are blocked waiting for the buffer to be filled by the send, the number of tasks that can be participating in a receive cannot exceed the number of worker threads, or the application will deadlock.

4.3.6 Support for the Implementation of Libraries

The implementation of the `pil_main` function for libraries expects the usage of a continuation codelet. Since the continuation codelet is not yet ready, we have to block the main task when it enters a graph with the `pil_enter` call. Within the `pil_enter` call, the calling thread blocks until the library graph has completed. Because of this blocking, a program written in library mode requires $P + 1$ worker threads to be created to get P way parallelism. The additional worker thread is consumed by the `pil_main` routine. When using SPMD parallelism, PIL has each created rank begin execution by calling the `pil_main` function. Thus, for SPMD, $2P$ worker threads need to be created. Each rank has a master task executing the `pil_main` function, and the master task schedules a slave task to execute the library graph.

4.4 Generating OCR code from PIL

The code generation for OCR follows a very similar path to that of SCALE, as discussed in Section 4.3. We will describe how the code generation for OCR differs from SCALE code

generation here.

4.4.1 Encapsulation of Data

All data within OCR must be encapsulated in a data block. As such, we have a structure that holds all of the declared data, and store it within a data block. This data block structure is allocated per task, just like in SCALE.

4.4.2 Body Functions

The body functions declared by the user, and associated with each PIL node are emitted as regular functions so they can be called from the PIL nodes.

4.4.3 PIL Nodes

The PIL nodes are composed of EDTs in OCR in the same way that PIL nodes are composed from codelets in SCALE, as discussed in Section 4.3 and Figure 4.1. The only difference is that the dependences for the EDTs are not counting dependences, and each receives an event and a slot. The body EDTs satisfy the events passed to them by the entry EDT to satisfy the exit EDT.

Similarly to SCALE, many of the operations rely on the use of a continuation EDT. There are plans for the implementation of a continuation EDT within OCR, but it is not ready for use yet. Therefore, some of the operations that rely on the continuation EDT have to use blocking in the same way as our SCALE implementation.

4.4.4 Memory Management Library

OCR provides functions for the management of data. They provide the `ocrDbCreate` routine for the allocation of a data block, and the `ocrDbDestroy` routine to free data. The `pil_alloc`

and `pil_free` routines are wrappers around these functions, respectively. The `ocrDbCreate` routine provides a GUID and a pointer to the allocated data block, and we set the fields of the GPP passed in accordingly.

4.4.5 Communication Library

Global Barriers. Global barriers can be implemented as described in Section 3.1.4 with the use of the continuation EDT. However, the lack of continuation EDT forces us to use the same implementation for the barrier as in the SCALE and OpenMP backends.

Sending of Data. The sending of data uses the same implementation as for SCALE and OpenMP, and is adherent to the OCR non-blocking EDT philosophy.

Receiving of Data. Without the continuation EDT, we are forced to use the same polling receive as in the SCALE and OpenMP implementation.

4.4.6 Support for the Implementation of Libraries

Once again, the lack of continuation EDT forces us to follow the same algorithmic implementation as we used for the SCALE backend. When the `pil_enter` call is made, the calling thread blocks and waits for the graph to complete. The task that is created to begin the execution of the graph is scheduled to a slave worker thread. One extra thread is required for the master task to execute the `pil_main` function, and $2P$ worker threads are required for P way SPMD parallelism.

4.5 Summary of the PIL Compiler Implementation

All of the source code for the PIL compiler has been open sourced, and is available as part of the OCR project. The source code for PIL and OCR can be checked out from the GIT repository at [24].

Chapter 5

Experimental Framework

We have available to us two machines on which to run our experiments. The details of these machines are described here. We have found the larger I2PC machine to clearly illustrate the behaviors of the codes that we evaluate, while the X-Stack cluster results mirror the I2PC results. As such, we have limited our discussions of the performance evaluation in Chapters 6 and 7 to the I2PC machine for brevity and cleanness. The results for the X-Stack cluster are provided in Appendix A for completeness.

5.1 I2PC3 Machine

We have access to a few of the machines of the Illinois-Intel Parallelism Center (I2PC) machines. I2PC3 is a four socket, 10 core Intel Xeon E7-4860 Westmere-EX machine with 128 GB of main memory. Available on the machine is GCC 4.8.1 and Intel MKL version 11.1.1. This machine has Intel's Turbo Boost technology enabled. This technology provides frequency scaling as a measure of the current temperature of the cores. This means that the single core performance numbers are running on the processors with a different frequency than when using many threads. The minimum frequency of these processors is 2.26 GHz,

and the maximum frequency is 2.666 GHz. This machine also has Intel's Hyper-Threading enabled, that allows two threads per core.

5.2 X-Stack Cluster

The X-Stack project has a cluster on which to run experiments. While this cluster is made up of many nodes, we stick to shared memory within a single node. Each node of the cluster is a two socket, eight core Intel Sandy Bridge Xeon E5-2690 with 128 GB of main memory. Available on the machine is GCC 4.8.3 with MKL version 11.2.0. This machine has Intel's Turbo Boost technology enabled. The minimum frequency of the cores is 2.9 GHz and the maximum frequency of the cores is 3.8 GHz. This machine also has Intel's Hyper-Threading enabled, which allows two threads per core.

Chapter 6

Cholesky Factorization: A Top to Bottom Case Study

Cholesky factorization is a linear algebraic decomposition of a Hermitian, positive-definite matrix A into the product of a lower triangular matrix L and its conjugate transpose L^*, as seen in in Formula 6.1.

$$A = LL^* \tag{6.1}$$

Cholesky factorization is commonly used to solve systems of linear equations, since it is more efficient than LU factorization. We have chosen to implement a tiled Cholesky factorization, as it is easily expressible in terms of operations on tiles, and the tile accesses can provide a locality optimization versus a naive implementation.

6.1 Tiled Cholesky Factorization

The tiled algorithm for Cholesky factorization consists of four operations: `POTRF`, `TRSM`, `SYRK`, and `GEMM`. The pseudo code for the sequential tiled Cholesky factorization is shown

```
for (k = 0; k < N; k++) {
  /* Factorize the diagonal tile */
  POTRF(A(k,k));
  /* Update kth column tiles */
  for (i = k+1; i < N; i++) {
    TRSM(A(i, k), A(k,k));
  }
  /* Update submatrix */
  for (x = 0; x < numSubTiles(k); x++) {
    i,j = getij(x);
    if(i == j) {
      SYRK(A(i,k), A(i,i));
    } else {
      GEMM(A(i,k), A(j,k), A(i,j));
    }
  }
}
```

Figure 6.1: Pseudo code for sequential tiled Cholesky factorization.

in Figure 6.1. The `POTRF` operation performs a Cholesky factorization on a diagonal tile $A(k,k)$. The `TRSM` operation performs a triangular system solve down a column of tiles using the diagonal component computed in `POTRF`. The `SYRK` operation performs a symmetric rank-k update onto a diagonal tile. The `GEMM` operation performs a matrix-matrix multiplication of the off-diagonal tiles.

The three main steps of the tile Cholesky factorization algorithm are shown for a single iteration in Figure 6.2. For this illustration, we will use the matrix A, a 4×4 tiled matrix. The first step of the algorithm is to do a block-level Cholesky factorization on the first diagonal element $A(k,k)$, as seen in Figure 6.2a. Second, using the newly computed $A(k,k)$ tile, a triangular system solve is performed down the column, as seen in Figure 6.2b. Finally, a sub-matrix update is performed on the remaining sub-matrix using the newly computed results of the triangular solve from step two, as seen in Figure 6.2c. On the next iteration of k, the process begins again beginning with the Cholesky factorization on the next diagonal tile $A(k,k)$.

The entire progression of the tiled Cholesky factorization can be seen in Figure 6.3. The factorization is on a 4×4 tiled matrix, and concludes in four iterations. Each iteration has the operations performed as previously described. Colors are assigned to each operation as

```
 1 for (k = 0; k < N; k++) {
 2   /* Factorize the diagonal tile */
 3   A(k,k) = POTRF(A(k,k));
 4   /* Update kth column tiles */
 5   forall (i = k+1; i < N; i++) {
 6     A(i,k) = TRSM(A(i, k), A(k,k));
 7   }
 8   /* Update submatrix */
 9   forall (x = 0; x < numSubTiles(k); x++) {
10     i,j = getij(x);
11     if(i == j) {
12       SYRK(A(i,k), A(i,i));
13     } else {
14       GEMM(A(i,k), A(j,k), A(i,j));
15     }
16   }
17 }
```

P			
T	S		
T	G	S	
T	G	G	S

(a) Block factorization of diagonal element using `POTRF`.

```
 1 for (k = 0; k < N; k++) {
 2   /* Factorize the diagonal tile */
 3   A(k,k) = POTRF(A(k,k));
 4   /* Update kth column tiles */
 5   for (i = k+1; i < N; i++) {
 6     A(i,k) = TRSM(A(i, k), A(k,k));
 7   }
 8   /* Update submatrix */
 9   forall (x = 0; x < numSubTiles(k); x++) {
10     i,j = getij(x);
11     if(i == j) {
12       SYRK(A(i,k), A(i,i));
13     } else {
14       GEMM(A(i,k), A(j,k), A(i,j));
15     }
16   }
17 }
```

P			
T	S		
T	G	S	
T	G	G	S

(b) Triangular solve on column using `TRSM` with input from previous `POTRF`.

```
 1 for (k = 0; k < N; k++) {
 2   /* Factorize the diagonal tile */
 3   A(k,k) = POTRF(A(k,k));
 4   /* Update kth column tiles */
 5   for (i = k+1; i < N; i++) {
 6     A(i,k) = TRSM(A(i, k), A(k,k));
 7   }
 8   /* Update submatrix */
 9   forall (x = 0; x < numSubTiles(k); x++) {
10     i,j = getij(x);
11     if(i == j) {
12       SYRK(A(i,k), A(i,i));
13     } else {
14       GEMM(A(i,k), A(j,k), A(i,j));
15     }
16   }
17 }
```

P			
T	S		
T	G	S	
T	G	G	S

(c) Sub-matrix update using `SYRK` and `GEMM` with input from previous `TRSM`.

Figure 6.2: Progression of a single iteration of the tiled Cholesky factorization algorithm.

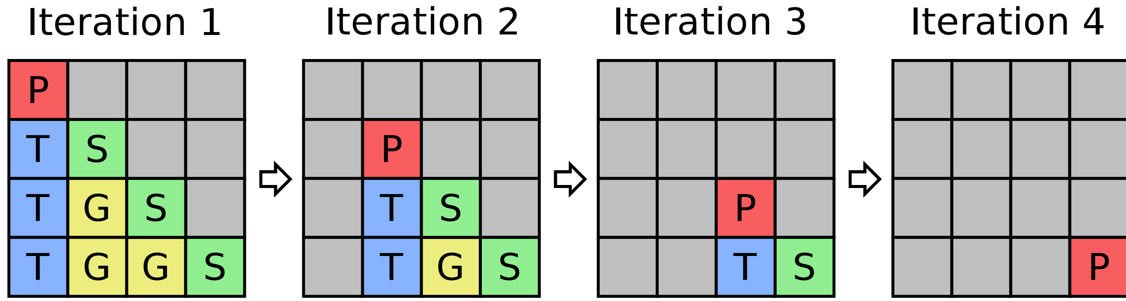


Figure 6.3: Progress of Cholesky factorization on a 4×4 tiled matrix.

follows: Red tiles denote a `POTRF` operation, blue tiles a `TRSM` operation, green tiles a `SYRK` operation, and yellow tiles a `GEMM` operation. Gray tiles denote a tile that is in its final configuration, and needs no further updates.

The task graph for the discussed 4×4 tiled Cholesky factorization can be seen in Figure 6.4. The task graph is based off of the input dependences of each computation to update a single tile. In this discussion A is a two dimensional array, $A(i, j, k)$ denotes the tile in row i, column j for iteration k. For the first iteration, the input dependence of a tile $A(i, j, k-1)$ is not shown, since there is no iteration -1, and it is assumed that those inputs are presatisfied. Each `POTRF` operation on tile $A(i, j, k)$, where $i = j = k$, has a single input dependence of the tile with the same (i, j) location from the previous iteration $k - 1$. Each `TRSM` operation on tile $A(i, j, k)$ has two dependences. One for the update $A(i, j, k - 1)$ on the same tile from the previous iteration, and one for the `POTRF` update $A(k, k, k)$. Each `SYRK` operation on tile (i, j, k) has two dependences. One for $A(i, j, k - 1)$ on the same tile from the previous iteration, and one for the `TRSM` operation on tile $A(i, k, k)$. Each operation `GEMM` on tile $A(i, j, k)$ has three dependences. One for the update $A(i, j, k - 1)$ on the same tile from the previous iteration, and one each for the `TRSM` update on tiles $A(i, k, k)$ and (j, k, k).

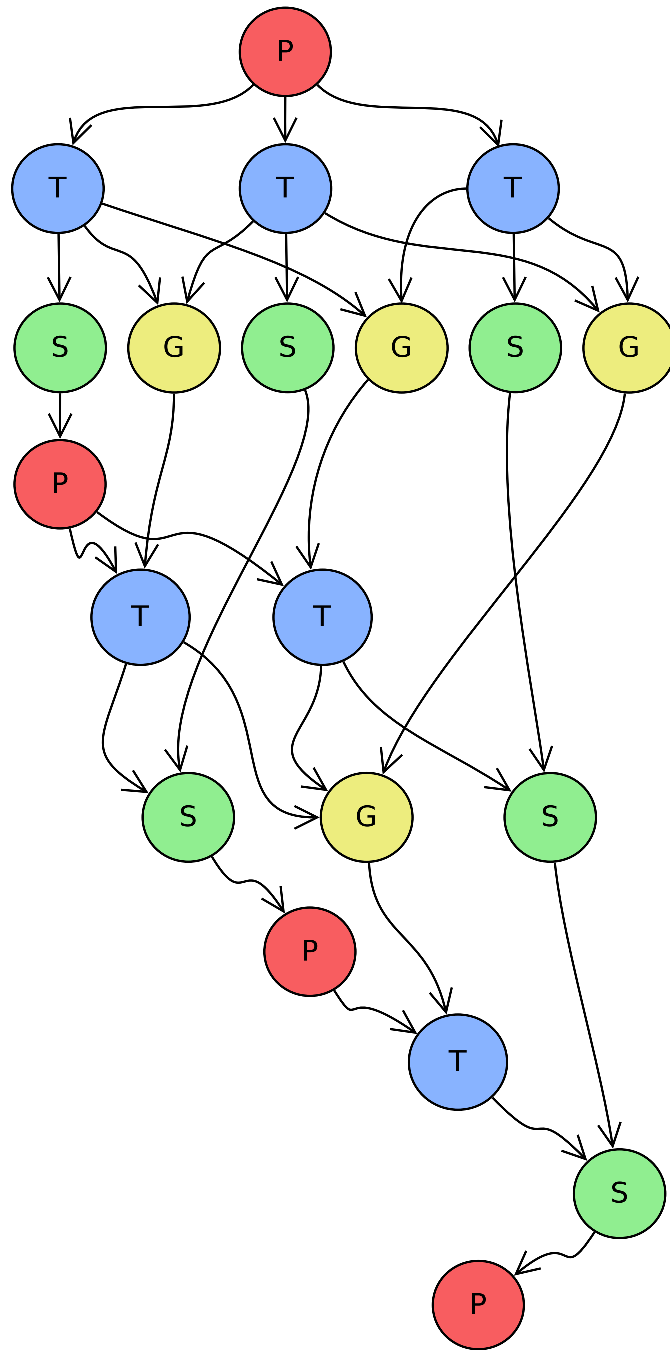


Figure 6.4: Task graph of Cholesky factorization.

```
for (k = 0; k < N; k++) {
  /* Factorize the diagonal tile */
  POTRF(A(k,k));
  /* Update kth column tiles */
  forall (i = k+1; i < N; i++) {
    TRSM(A(i, k), A(k,k));
  }
  /* Update submatrix */
  forall (x = 0; x < numSubTiles(k); x++) {
    i,j = getij(x);
    if(i == j) {
      SYRK(A(i,k), A(i,i));
    } else {
      GEMM(A(i,k), A(j,k), A(i,j));
    }
  }
}
```

Figure 6.5: Pseudo code for fork/join parallel tiled Cholesky factorization.

6.2 Fork/Join Cholesky

The fork/join version of Cholesky factorization is based off the the sequential code discussed in the previous section. The algorithm used for the fork/join version of Cholesky factorization is shown in Figure 6.5. Barriers are used at the end of each parallel loop to guarantee that all input dependences of the next statement are completed before the next statement can begin.

The task graph for the discussed 4×4 tiled Cholesky factorization can be seen in Figure 6.6. The task graph is based off of the input dependences of each computation to update a single tile. However, each horizontal line represents a barrier operation as part of the fork/join algorithm. All operations above the barrier must complete before the next operation after the barrier is allowed to begin.

Both the HTA and hand coded PIL versions of the benchmark are based off of the same algorithm described in Figure 6.5. This is, to our knowledge, the best version of fork/join code for the algorithm. Since the algorithm is identical in both implementations, we can directly compare the PIL and HTA versions of the code.

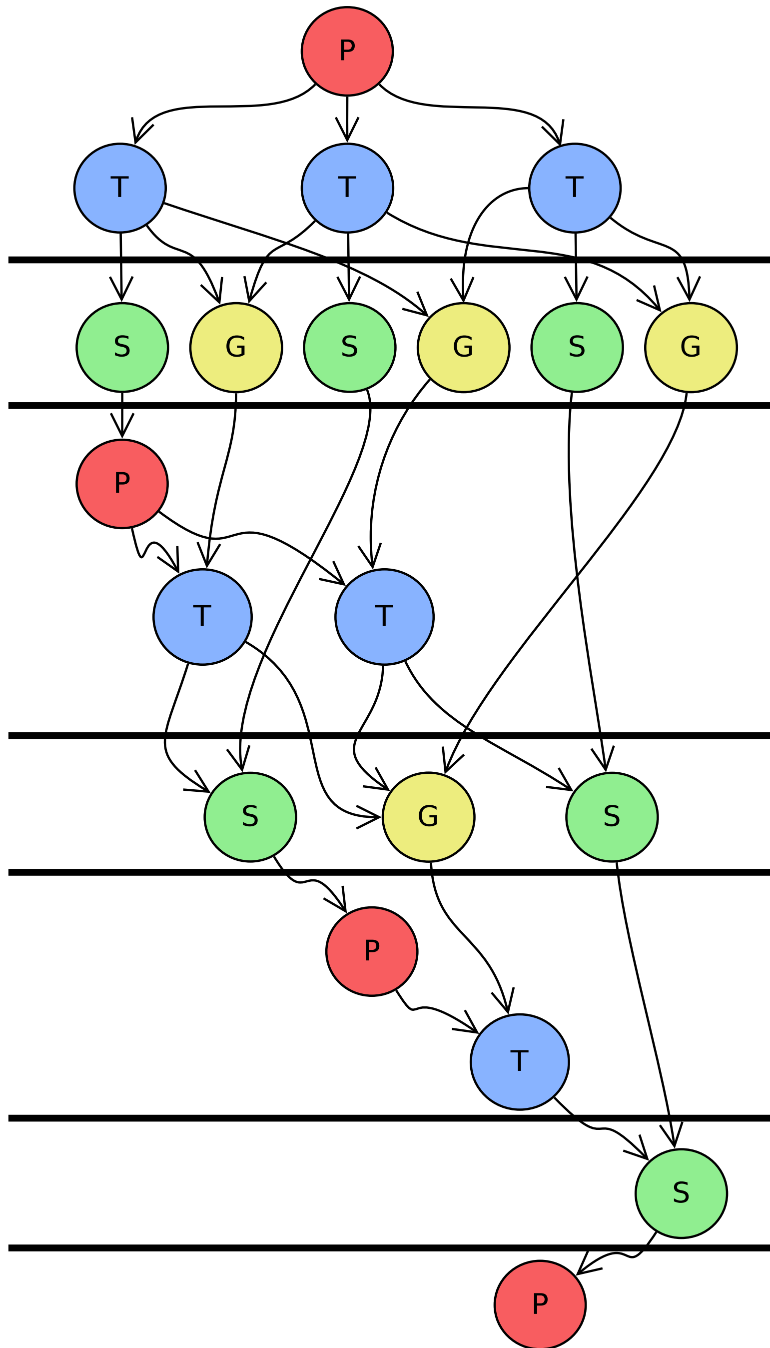


Figure 6.6: Task graph of tiled Cholesky factorization fork/join algorithm. Horizontal lines are global barriers.

```
1  for (k = 0; k < N; k++) {
2    /* Factorize the diagonal tile */
3    POTRF(A(k,k));
4    /* Update kth column tiles */
5    TRSM(A(k+1:N-1, k), A(k,k));
6    /* Update submatrix */
7    forall (i=k+1:n-1, j=k+1:N=1; j <= i) {
8      if(i == j) {
9        SYRK(A(i,k), A(i,i));
10     } else {
11       GEMM(A(i,k), A(j,k), A(i,j));
12     }
13   }
14 }
```

Figure 6.7: Pseudo code for HTA tiled Cholesky factorization.

6.2.1 HTA Implementation

The algorithm for the HTA version of the fork/join algorithm for tiled Cholesky factorization is directly based off of the algorithm from Figure 6.5. HTA pseudocode for the tiled Cholesky factorization is represented in Figure 6.7. The `POTRF` operation is done sequentially in the HTA main thread. The `TRSM` parallel for loop is performed by collecting all input and output dependences for each tile that needs updated sequentially in the main loop, and then forking all instances of the body of the loop in a single PIL fork/join node. Since all instances of the parallel for loop are created at the same time, it is left up to the runtime of the generated code to schedule each instance onto its worker threads. The parallel for loop of the sub-matrix update is done in exactly the same way by collecting all of the input and output dependences of each instance of the parallel for loop and scheduling them all at once. The parallelism is constrained for scaling measurements simply by changing the number of available worker threads in the underlying runtime system.

6.2.2 PIL Implementation

The hand coded PIL fork/join implementation is based directly off of the pseudo code shown from Figure 6.5. The code for the PIL nodes is shown in Figure 6.8, and the corresponding

```
1  node(1, rank, i, [1:1:1], target, [0], [2],
2    init(&target, index_array, data_array, &argc, &argv, &A, &nDGEMMS, &k))
3  node(2, rank, i, [1:1:1], target, [1,4], [3,5],
4    cholesky(&target, index_array, data_array, &k, &A, &nDGEMMS, &nTRSMS, &TRSMstart))
5  node(3, rank, t, [TRSMstart:1:nTRSMS], target, [2], [4],
6    dtrsm(&target, index_array, data_array, &t, &k, &A, nDGEMMS, nTRSMS))
7  node(4, rank, t, [0:1:nDGEMMS], target, [3], [2],
8    dsyrk_dgemm(&target, index_array, data_array, &t, &k, &A, nDGEMMS))
9  node(5, rank, i, [1:1:1], target, [2], [0],
10   verify(&target, index_array, data_array, &A))
```

Figure 6.8: PIL code for nodes of data parallel tiled Cholesky factorization.

graph of PIL nodes is shown in Figure 6.9. The PIL code that makes up the Cholesky factorization is made up of three PIL nodes: 2, 3, and 4. Node 1 is used to initialize the data, and node 5 is to verify the results. Node 2 is a sequential PIL node for the `POTRF` operation. Node 3 is a fork/join parallel node for the `TRSM` loop. Node 4 is a fork/join parallel node for the sub-matrix update loop. The sequential outer k loop is performed by having a back-edge in the PIL task graph from the sub-matrix update node, 4, to the `POTRF` node, 2. Each of the parallel nodes generates one parallel instance for each tile to be updated during the operation giving maximal parallelism for the loop. The underlying runtime is responsible for scheduling and load balancing all parallel instances. Parallelism is constrained for scaling measurements by changing the number of available worker threads for the underlying runtime system.

6.3 SPMD Cholesky with HTAs

The SPMD algorithm uses the same pseudo code as the fork/join version, and can be seen in Figure 6.7. It employs the replication of sequential parts described in Section 3.4.2. For each operation the following steps are performed. First each rank computes the list of tiles involved in the current operation, `current`. Then each rank computes a list of incoming dependences, `dependences`, for each tile in `current`. If a rank owns a tile in current, a receive is posted for each tile the rank does not own in `dependences`. For each tile a rank

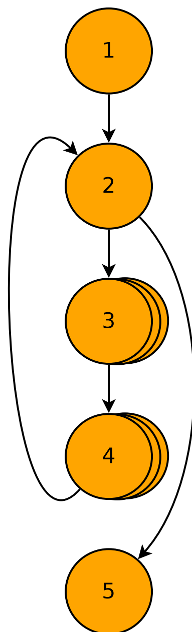


Figure 6.9: Graph of PIL nodes for data parallel tiled Cholesky factorization.

owns in `dependences`, it posts a send to the corresponding rank that needs the tile. This follows the owner computes rule adopted by HTAs. That is, the owner of the tile on the left-hand side of an assignment computes the operation on the right-hand side of the assignment, and receives any inputs necessary from other ranks.

For the sequential `POTRF` portion, each rank checks to see if they are the owner of the tile that needs to be updated with the `POTRF` operation. If so, the rank performs the operation. If not, the rank continues on to the next operation. Similarly, for the operations of the column update and the sub-matrix update, the input dependences are computed for each tile that needs the update. If a rank owns a tile that needs to be updated, it will communicate with the owners of the input dependences and then perform the update.

The SPMD algorithm is only implemented in the HTA notation. The HTA notation provides a succinct representation of the algorithm, while programming directly in PIL is much more verbose. The algorithm employed would be the same, and we feel no need to duplicate the implementation effort. We will show that the HTA library has little overhead

```
node(1, rank, j, [1:1:1], target,  [0], [2],
  init(&target, index_array, data_array, &argc, &argv, &A, &k))
node(2, rank, j, [1:1:1], targets, [1,4], [(3),6],
  potrf(targets, index_array, data_array, &_pil_num_targets, _pil_task_names, &k, &A))
node(3,    i, j, [1:1:1], targets, [2], [(4,5)],
  trsm(targets, index_array, data_array, &_pil_num_targets, _pil_task_names, i, &k, &A))
node(4,    i, j, [1:1:1], target,  [3], [0,2],
  syrk(&target, index_array, data_array, i, &k, &A))
node(5,    x, j, [1:1:1], target, [3], [0],
  gemm(&target, index_array, data_array, x, &k, &A))
node(6, rank, j, [1:1:1], target,  [2], [0],
  verify(&target, index_array, data_array, &A))
```

Figure 6.10: PIL code for nodes of task parallel tiled Cholesky factorization.

on top of PIL by means of the fork/join algorithm.

6.4 Task Parallelism with Hand Coded PIL

Due to the nature of the HTA notation and its heavy reliance on data parallelism, creating a more dynamic version of Cholesky factorization is difficult. However, PIL has no such limitations in its representation. This version of Cholesky factorization was implemented directly in PIL to leverage the benefits of asynchrony available when using task parallelism.

The task parallel version of the Cholesky algorithm relies on building the task graph from Figure 6.4. The code for the PIL nodes is shown in Figure 6.10, and the corresponding graph of PIL nodes is shown in Figure 6.11. Each node of the graph has one or more dependences that are managed by the user in the implementation of the algorithm. The dependences for each node are tracked, and each time a dependence is satisfied, the node is tested to see if all of its dependences are satisfied. Once the final dependence for a node is satisfied, the node is triggered. Each node in the task graph is represented by a single PIL node. Each PIL node created is sequential, and when generating OCR and SCALE can generate only a single codelet. This algorithm provides maximal parallelism in the algorithm by strictly following all of the data dependences present in the task graph. Furthermore, maximal asynchrony is achieved through the same method.

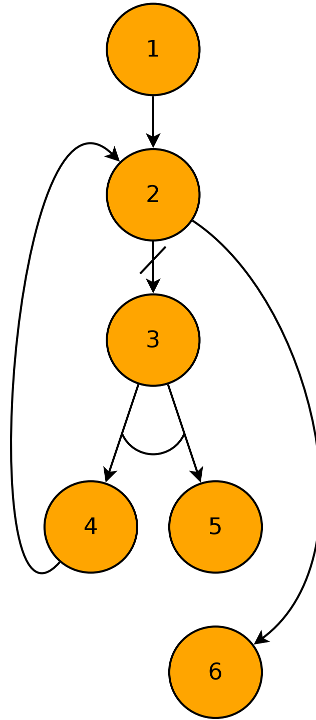


Figure 6.11: Graph of PIL nodes for task parallel tiled Cholesky factorization.

6.5 Experimental Results

In this section we discuss the performance of the four previously discussed implementations of the tiled Cholesky factorization algorithm.

Versions of the code. Each version of the benchmark is referred in the graphs as a triple `<frontend> <algorithm> <backend>`. The following is a discussion of each version of the code.

Frontends. The first part of the triple used to describe a particular benchmark version is its frontend language, `<frontend>`. We have, as discussed, used two languages to implement the various versions of the Cholesky factorization benchmark, as seen in Table 6.1. PIL refers to the benchmarks hand written in PIL code, and HTA refers to the benchmarks written in HTA code.

Table 6.1: Frontend implementations

PIL	The hand coded PIL implementation.
HTA	The HTA implementation.

Table 6.2: Parallel Algorithms.

Fork/Join	The parallel fork/join algorithm.
SPMD	The SPMD parallel algorithm.
Task	The task parallel algorithm.

Parallel algorithms. The second part of the triple used to describe a particular benchmark version is its parallel algorithm, `<algorithm>`. We have, as discussed, used three different parallel algorithms to implement the various versions of the Cholesky factorization benchmark, as seen in Table 6.2. The fork/join version of the code is implemented in both the HTA and PIL frontends. The SPMD version is only implemented in HTAs. The task parallel version of the code is only implemented in PIL, as it would not be possible to implement in HTAs.

Backend generated code. The third part of the triple used to describe a particular benchmark version is its generated backend code, `<backend>`. We have generated code from PIL into three different languages seen in Table 6.3. The three versions of the generated code are OpenMP (OMP), SCALE, and OCR.

Tiling for the I2PC Machine. All results discussed in this section are run on the I2PC machine described in Section 5.1 with 40 processing cores and 80 hardware threads. We have chosen to run with a tiling of 80×80 tiles for this machine, and at all available numbers of

Table 6.3: Generated backend codes.

OMP	OpenMP generated code.
SCALE	SCALE generated code.
OCR	OCR generated code.

cores $1-80$. All experiments discussed in this section use this 80×80 tiling, varying the number of elements per tile.

OCR Limitations. Due to a technical limitation in the current version of the runtime, OCR can only run up to 66 threads. All of the fork/join and task parallel versions will use up to the maximum allowed processors. As discussed previously in Section 3.2, the SPMD version of OCR requires two threads per SPMD rank, and thus can only run up to 33 ranks.

6.5.1 What We Will Show

In these experimental results we shall show

1. The HTA implementation has very little overhead compared to hand coded PIL.
2. We are able to leverage asynchrony in the Cholesky algorithm with task parallelism, compared to fork/join parallelism.
3. We want to leverage asynchrony using SPMD, but we are unable due to the dependence graph and static distribution of tiles.
4. We emit efficient code for the backends. We do the best we can, but OpenMP is more efficient than SCALE, and SCALE is more efficient than OCR. OCR never does as well as the others.

6.5.2 Experimental Data

We have composed four experiments in order to study the performance of the Cholesky factorization implementations and backed. We have chosen to standardize on a single tiling of the original input matrix, and form experiments by changing the size of the tiles. The I2PC machine we are running our experiments on has 40 processor cores and is capable

of handling 80 threads in hardware. Thus, we have chosen to use an 80×80 tiling for all experiments. The four tile sizes we have chosen are 1×1 elements per tile, 100×100 elements per tile, 200×200 elements per tile, and an experiment with random elements per tile. The results of these experiments can be seen in Figures 6.12, 6.13, 6.14, and 6.15

The experimental results from the Cholesky factorization benchmarks on the I2PC machine scale from one to 80 processors, and have 12 versions of the code represented in them. The blue lines denote the implementation of the hand coded PIL fork/join algorithm. The red lines denote the implementation of the hand coded PIL task parallel algorithm. The orange lines denote the HTA fork/join algorithm. The green lines denote the HTA SPMD algorithm. Each version of the algorithm generates three different backends codes: $+$ denote the generated OpenMP code, $\times$ denote the generated SCALE code, and $\circ$ denote the generated OCR code. Speedups are calculated relative to the hand coded PIL fork/join algorithm generating OpenMP code for a single processor.

The 1×1 elements per tile experiment is designed to measure the overhead in the creation and scheduling of tasks by the runtime systems. The result of this experiment can be seen in Figure 6.12. The plot shows the speedup for the experiment. Note that the y-axis is in log-scale.

The 100×100 elements per tile experiment is designed to measure performance of the benchmarks with a medium amount of work per tile. The result of this experiment can be seen in Figure 6.13. The plot shows the speedup for the experiment.

The 200×200 elements per tile experiment is designed to measure performance of the benchmarks with a large amount of work per tile. The result of this experiment can be seen in Figure 6.13. The plot shows the speedup for the experiment.

The final experiment we performed was to simulate sparse data. We ran the 1×1 experiment, but added a random delay after each tile was updated. The delay was precomputed and assigned to each tile in the array, and the same delay was always used for a tile after

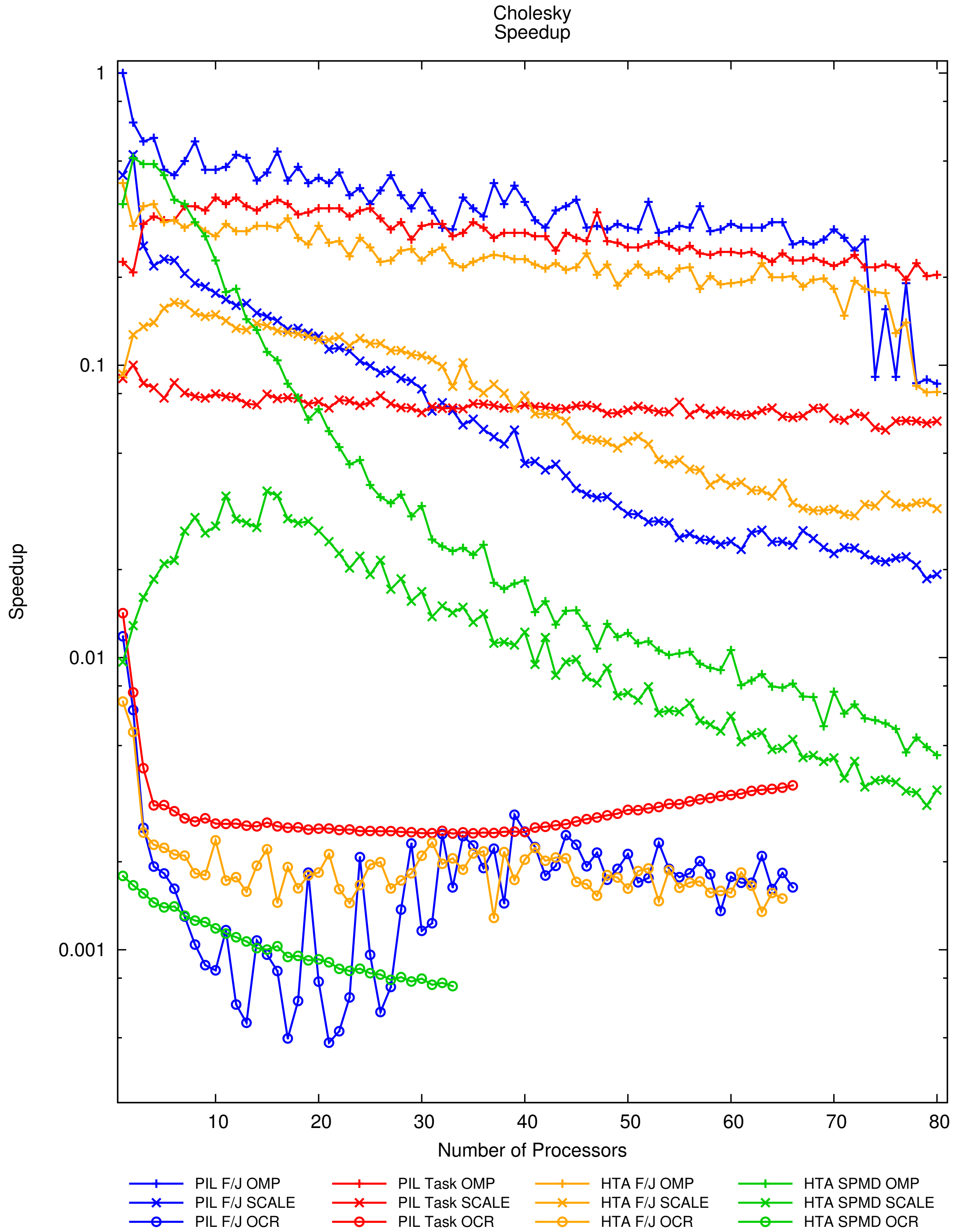


Figure 6.12: Speedup on the I2PC machine for Cholesky factorization with 1×1 tiling.

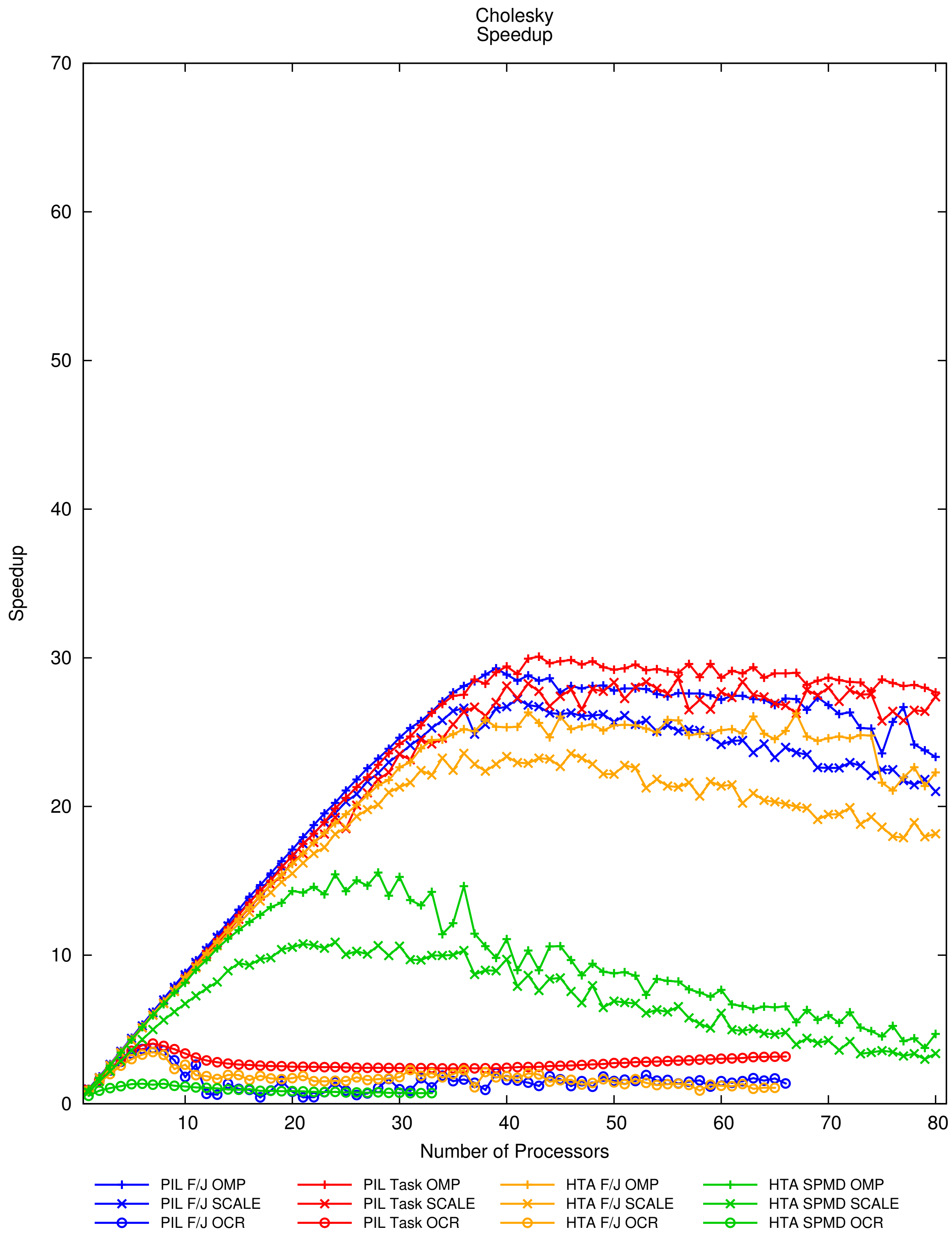


Figure 6.13: Speedup on the I2PC machine for Cholesky factorization with 100×100 tiling.

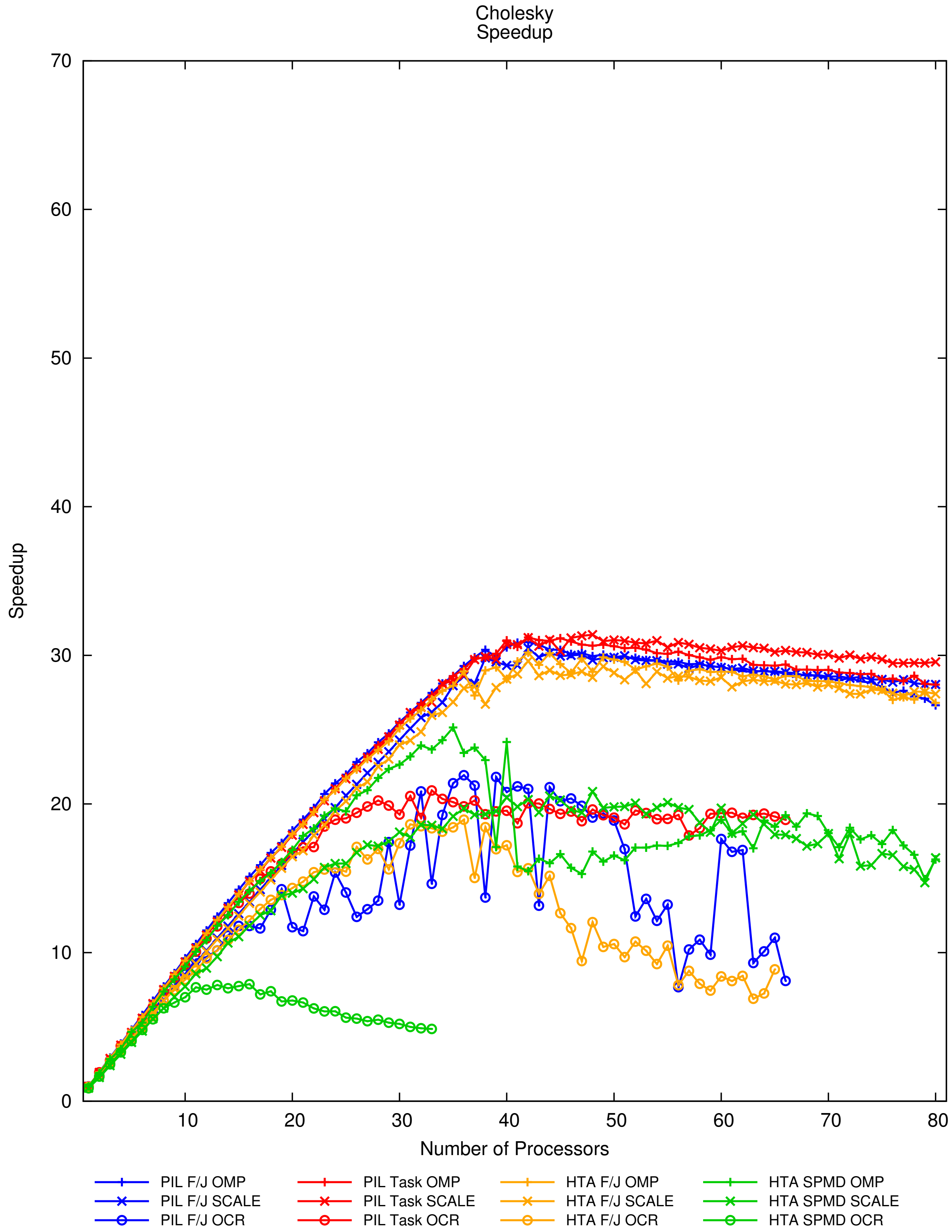


Figure 6.14: Speedup on the I2PC machine for Cholesky factorization with 200×200 tiling.

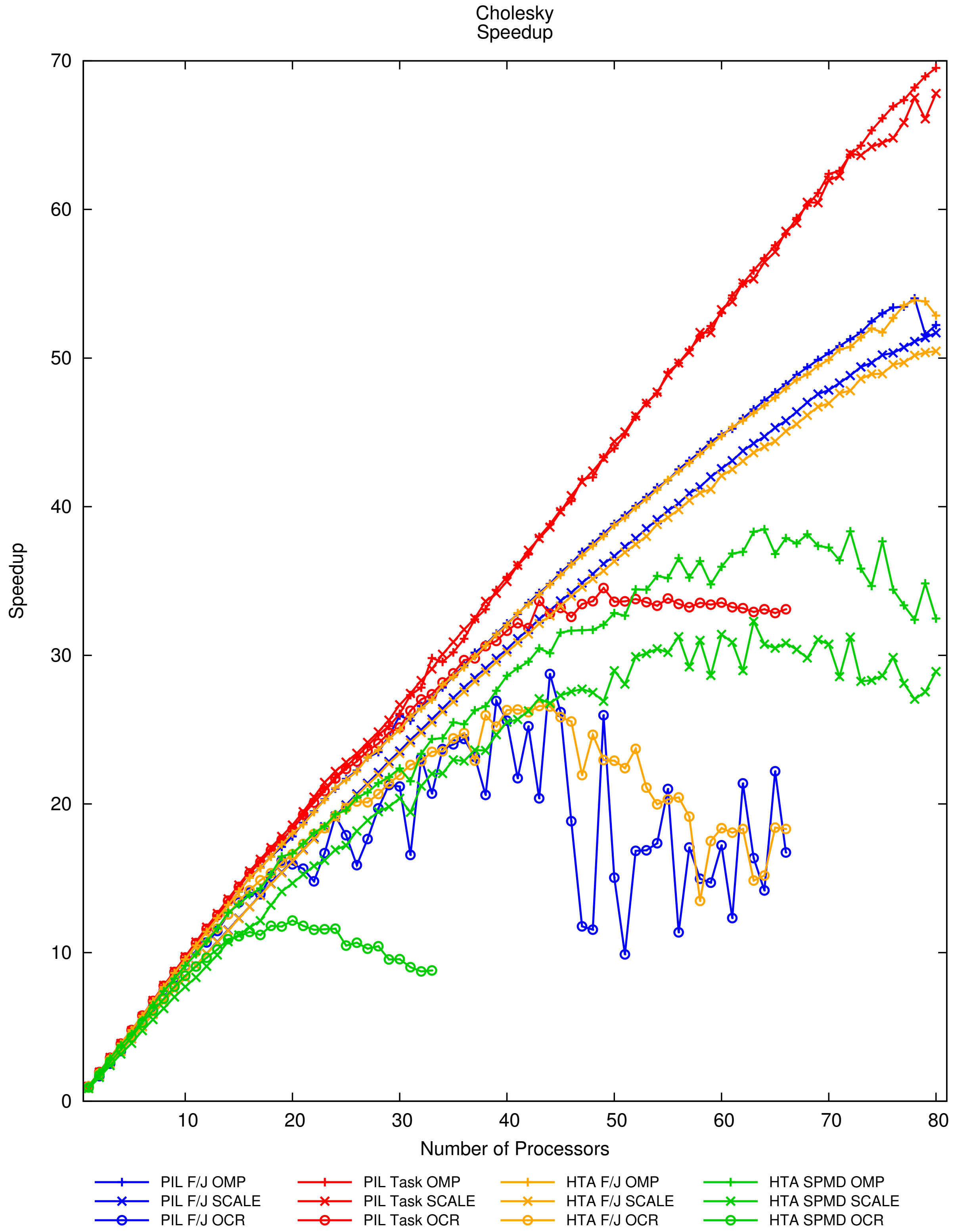


Figure 6.15: Speedup on the I2PC machine for Cholesky factorization with random tiling.

the computation was performed. The delay was implemented as a sequential `for` loop from $0 - N$, where N is the random delay for the tile, and the body of the loop was empty. The numbers were generated as a uniform distribution across the range $0 - 2,000,000$. The upper limit of $2,000,000$ was chosen since it gave the benchmark a sequential execution time slightly larger than the sequential execution time for the 200×200 tiling experiment. We shall refer to this experiment as random tiling.

We are aware that the random tiling experiment is not necessarily a good approximation of how a sparse implementation of the Cholesky benchmark would be implemented, since we are using the dense tiling code to perform the work. Good sparse Cholesky solvers use a different algorithm. However, the goal of the experiment is not to do a sparse tiling experiment, but rather to use the task graph of the dense Cholesky factorization, while providing a load imbalance within the tiles.

6.5.3 HTA Has Little Overhead

As we have previously discussed, the algorithmic implementation for the HTA and PIL versions of the tiled Cholesky fork/join factorization are the same. We have observed that both versions have very little difference in their performance, as seen in the performance difference between the PIL (blue line) and HTA (orange line) in Figures 6.12, 6.13, 6.14, and especially 6.15. It is observed that there is very little overhead since for each backend, the plots are nearly on top of each other.

The lack of overhead discovered in HTAs has lead us to only implement the algorithm twice for the fork/join version of the code. We have only implemented the SPMD version of the algorithm in HTAs, since reimplementing the algorithm in PIL would only lead to similar results. Similarly for task parallelism, we have only implemented the algorithm in PIL.

6.5.4 Efficiency of the Backends

While plotting the results for all of the different versions of the code one trend became very clear. It was observed that the OpenMP generated code was always faster than the SCALE generated code, and that the SCALE generated code was always faster than the OCR generated code.

Interestingly, there is an inverse relationship between the amount of time and effort we have put into the optimization of the generated backend code. We have put the most effort in to trying to get better performance from the OCR runtime, and we have put significantly more effort into the optimization of the SCALE code than we have for the generated OpenMP code.

We believe that our experience with the efficiency of the generated codes can be summarized by the two following points. First, OpenMP is the most mature of the runtimes we are using. Many man-hours from many organizations have gone into the optimization of the OpenMP runtime to guarantee its good performance. SCALE is the next most mature runtime. Luckily we have a close relationship with ETI, the company that develops SCALE and the SWARM runtime. We have generated code that has exposed some previously unknown usage cases and showed a few bugs that were resolved by ETI. We have not been able to do the same with the OpenMP runtime.

The OCR runtime is the newest and least mature of the runtimes discussed here. We have been able to find numerous bugs in the runtime and means of improving the runtime. We have worked closely with collaborates on OCR development, and have been able to influence its implementation. However, OCR *is* a research runtime, and we have had to put in many hours to understand its performance and how to optimize OCR code generation.

The second point is that the SCALE and OCR runtimes are work stealing runtimes. If ever a worker thread becomes idle, it tries to steal work from other threads. While trying to work steal, the worker thread is using maximum compute cycles to try and steal work. This

is in contrast to OpenMP's runtime that leaves threads idle if there is no work for it to do.

The 1×1 experiment shown in Figure 6.12 measures the speedup of task creation and scheduling for the runtimes. There are three main takeaways from this plot. First, is the clustering of the data. If you look at the two fork/join versions as well as the task parallel version, the OpenMP executions times are clustered together, and have the lowest execution time. The SCALE execution times are also clustered, but have increased execution time. The OCR results are also clustered and have the highest execution time. Thus the OpenMP backend has less overhead than the SCALE backend, which has less overhead than the OCR backend. The SPMD version of the code shows the same rankings of the backends from OpenMP to SCALE to OCR, but we will discuss the intricacies of that version of the code in the next section. Take note that the y-axis is log-scale, and that the clusters of lines are orders of magnitude apart. These large overheads provide significant performance degradation for small workloads.

Next, the 100×100 experiment in Figure 6.13 has a medium amount of work to perform per tile. The large overheads of the OCR runtime are clearly evident in the fact that the OCR backend code only scales to 7 processors, and then the overheads prevent scaling. There is also a noticeable gap between the scale backends and their corresponding OpenMP counterparts. This shows that the OCR version does not yet have enough work per tile to ameliorate the overheads associated with task creation and scheduling.

In the 200×200 experiment in Figure 6.14, there is enough work within a tile that the OCR runtime is able to 39 cores. However, there is very large variability in the results for the OCR runtime. Remember that we said that for each number of processors, the benchmark was run 10 times to try and remove variability from the results. There is just too much variability within the OCR runtime to get a good result for each number of processors. The OpenMP and SCALE results for the fork/join and task parallel version of the code are all nearly on top of each other, showing the overheads in SCALE are able to be amortized with

this tile size.

The random tile experiment in Figure 6.15 shows the efficiency of the backends most clearly. For both the HTA and PIL fork/join versions of the code, as well as the HTA SPMD version, the OpenMP generated code clearly outperforms the SCALE code, which out performs the OCR code. For the task parallel version of the code, the OpenMP and SCALE backends have nearly identical performance, while the OCR version of the code lags behind.

6.5.5 Efficiency of Parallel Versions of the Code

We shall begin the discussion of the comparison between the fork/join, task parallel, and SPMD versions of the Cholesky algorithm by looking at the random tiling experiment in Figure 6.15, as this graph most clearly distinguishes between the algorithms. This tiling was created to provide load imbalance amongst the tiles. The task parallel algorithm is able to take the most advantage of the load imbalance. In this graph we can clearly see that the HTA and PIL fork/join versions of the algorithm achieve nearly identical performance. Furthermore, we can see that the task parallel version of the code achieves the best performance. Finally, SPMD is the least efficient of the algorithms.

The 1×1 experiment in Figure 6.12 shows some more interesting results. The fastest OpenMP backend was generated by the hand coded PIL fork/join algorithm. Next is the PIL task parallel algorithm, followed by the HTA fork/join algorithm. This shows the little overhead present in the HTA library. Finally is the SPMD algorithm. This algorithm's OpenMP backend starts with similar performance to the other three for a single processor, but does not scale nearly as well as the number of processors is increased. This is due to the load imbalance present in the SPMD algorithm discussed below.

Another interesting trend from the 1×1 experiment is a comparison between the SCALE and OpenMP backends for the fork/join versions of the code. The OpenMP fork/join have

increased execution times as the number of processors is increased; however, the SCALE fork/join have dramatically increased execution time as the number of processors is increased. The 1×1 experiment has very little work to be done, and when increasing the number of cores, will leave some of the cores without enough work. However, leaving idle cores within the SCALE runtime can be detrimental to the performance of the algorithm.

Limitations with Static Distributions in SPMD. The SPMD implementation of the algorithm has some interesting performance characteristics. In the 1×1 experiment in Figure 6.12 the SPMD has the largest slowdown of any of the parallel versions. Furthermore, in each of the 100×100, 200×200, and random tiling experiments in Figures 6.13, 6.14, and 6.15, respectively, the SPMD implementation has the lowest performance. While it may seem that this means the SPMD algorithm has high overhead, the reason for the lowered performance is actually due to load imbalance.

Figure 6.16 shows how performance can be improved by the asynchrony in SPMD algorithms when compared to a fork/join algorithm. Performance improvements like those shown in the figure are rarely achieved in practical algorithms, however. The amount of improvement that is able to be achieved is dependent on the data dependences as well as the initial distribution of the data. The improvements shown in Figure 6.16 assume no data dependences between iterations.

As we know from Figure 6.4, the Cholesky factorization algorithm has a complicated task graph formed from the data dependences within the array. Let us take a look at two iterations of the tiled Cholesky factorization algorithm with a suboptimal data distribution. The data distribution we shall be using is a row cyclic distribution, with each row of the array assigned to the processors in a cyclic ordering, as seen in the left side of Figure 6.17 with four processors and a 6×6 tiled array. In the row cyclic distribution the first row is assigned to processor 0, row 1 to processor 1, row 2 to processor 2, and row 3 to processor 3.

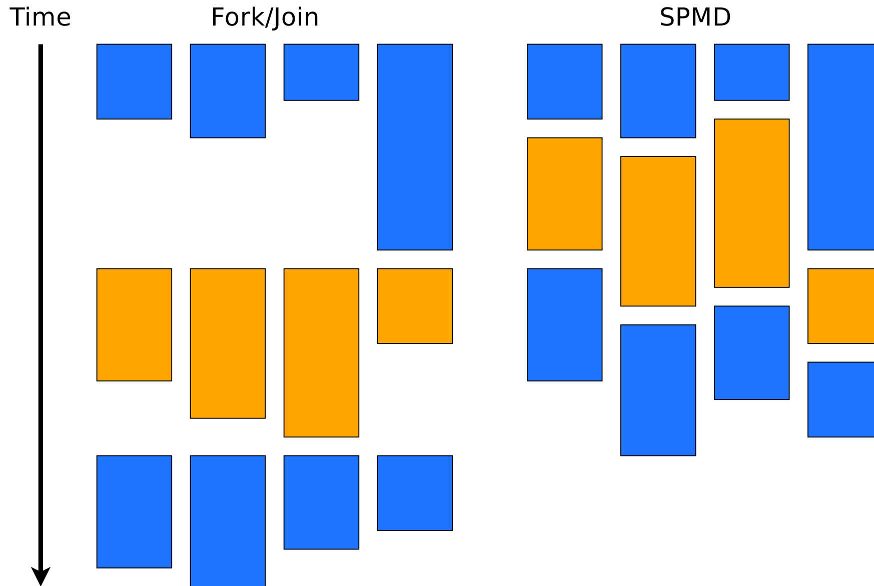


Figure 6.16: SPMD asynchrony scheduling.

Then we cycle through the processors again and row 4 is assigned to processor 0, and row 5 to processor 1. Already we can see there is a load imbalance since processors 0 and 1 receive two rows while processors 2 and 3 only receive one row each.

In the fork/join version of the code, the execution of the tile updates are not preassigned to any processors, and are dynamically scheduled to processors as the finish any work in the current iteration. As we see in the figure, the fork/join algorithm takes seven units of time to complete the first iteration. Since the tiles are load balanced as evenly as possible, each iteration there are at most $P-1$ processors that sit idle for *exactly* one time unit.

In the SPMD algorithm, each processor must process the tiles it owns one at a time. In the first iteration, processor 1 owns the most tiles, and thus executes them while the other processors are idle. This algorithm takes nine units of time to complete the first iteration. The worst case distribution distributes all of the T tiles to one processor, and $P-1$ processors sit idle while the iteration takes T time units to complete.

The process is repeated in the second iteration, but once again, there is a load imbalance, and the asynchronous SPMD implementation takes longer with 6 time units than the

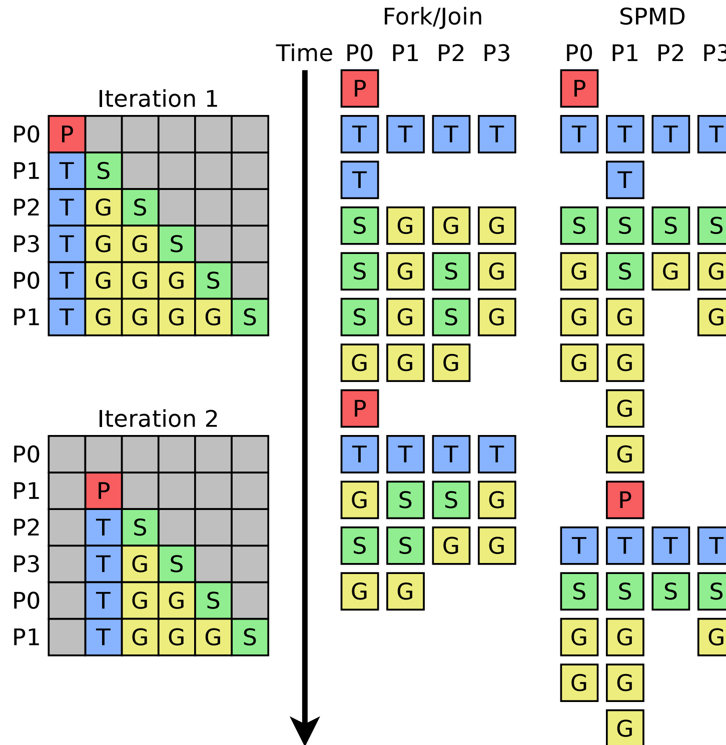


Figure 6.17: SPMD Cholesky task graph scheduling.

fork/join algorithm's 5 time units! As we can see, a row-cyclic distribution is not a good distribution for the tiled Cholesky algorithm.

Reflected Row Distribution. A slightly more load balanced distribution than row cyclic is reflected row. We tried an experiment using a reflected row distribution. In reflected row cyclic distribution, using four processors for example, rows 0-3 are assigned to processors 0-3. The next rows, 4-7, are assigned to the reflected processors 3-0. Then the cycle repeats for the remaining rows. This gives a more even distribution of the tiles as rows with higher numbers have more tiles. The results for a 100×100 elements per tile experiment can be seen in Figure 6.18. Both lines on the plot are generating OpenMP code and use the same 80×80 tiling. However, the line with the $\square$ symbols uses the reflected row cyclic distribution. Notice how the lines have a very pronounced jagged edge. This is due to the way that the load is distributed to the processors. For some numbers of processors, the load is more evenly

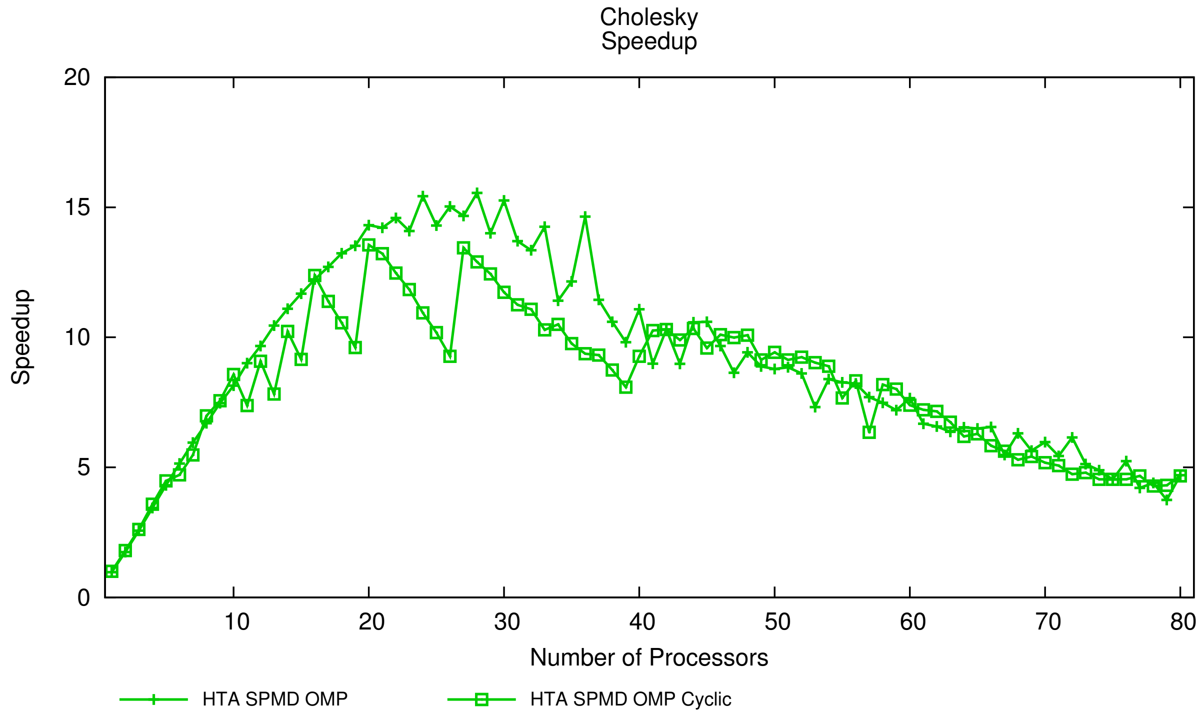


Figure 6.18: Speedup on the I2PC machine for Cholesky factorization with 100×100 elements per tile using reflected cyclic tile distribution.

distributed than for others. For example, the algorithm achieves its best speedup number with 20 processors (a peak), 13.55x. However, using 19 processors (a valley), the algorithm does much worse with a speedup of only 9.61x. This means that the load is much more evenly distributed when using 20 processors than 19.

From our experiments with the reflected row cyclic distribution, we tried to come up with a more efficient distribution. We gave each tile a weight based on the number of updates that is applied to the tile. Tiles in column one are updated once, tiles in column two are updated twice, and so on. Using these weights for the tiles, we distributed tiles to the ranks such that the sum of the weights for the tiles was as balanced as possible. This balanced weights distribution is the one used for all of the SPMD experiments, unless otherwise noted.

Load imbalance is a limitation inherent to static distributions within any SPMD algorithm. Since the task graph for Cholesky factorization (Figure 6.4) changes for each iteration

being executed, the load balance changes for each iteration. The jagged performance shown in Figure 6.18 illustrates this point. Efficient SPMD algorithms require either a good distribution for the data dependences within the data, or to redistribute the data so that the distribution is efficient for the current dependences. However, redistributing data within HTAs is expensive. While SPMD algorithms with static data distributions can achieve performance advantages over fork/join algorithms, we have found that we are unable to achieve improved performance by utilizing SPMD with the Cholesky factorization algorithm.

6.6 Conclusions from Cholesky Factorization

Through the experiments described in this chapter we have shown first that the HTA library has very little overhead compared to hand coded PIL. Second, we have shown that the OpenMP backend is more efficient than the SCALE backend, which is more efficient than the OCR backend. Third, we have shown that task parallel algorithm provides more performance than fork/join algorithm, which has higher performance than the SPMD algorithm. We have also demonstrated that SPMD programming within PIL is not inherently inefficient, but rather that the data dependences within the Cholesky factorization algorithm provides limited asynchrony when statically assigning tiles to processors.

Chapter 7

The NAS Benchmarks with HTAs

When deciding how to evaluate the implementation of the PIL language and compiler, we chose not to focus on the performance of a single benchmark, but rather to showcase its flexibility and robustness by implementing an entire suite of benchmarks designed to evaluate a range of features in the language. We have chosen to implement and evaluate the performance of PIL using the NAS Parallel Benchmarks (NPB). The NAS Parallel Benchmarks are a set of programs designed to evaluate parallel supercomputers [3, 4]. Each of the benchmarks within the NAS suite are designed to stress a particular portion of the machine. This will give us a good foundation to show that PIL can support all of the operations contained within the complex set of benchmarks.

In the evaluation of the PIL notation we wanted to show the following:

1. We will show that the PIL notation is suitable to implement the HTA library
2. The current implementation of the PIL compiler can generate OpenMP, SCALE, and OCR code.
3. The PIL compiler generates efficient code such that it does not interfere with the HTA library's performance.

4. The PIL notation provides the user (the HTA library) flexibility enough to implement the HTA library using global barriers for synchronization in the HTA libraries fork/join version as well as more fine grained point-to-point synchronization in the HTA libraries SPMD version.

5. The PIL compiler is robust enough to provide a framework for a suite of benchmarks designed to test supercomputers.

7.1 The NAS Parallel Benchmarks

The NAS Parallel Benchmarks consist of five kernels and three short applications. We have implemented the five kernels including the Embarrassingly Parallel (EP) benchmark, designed to provide maximum flops; the Integer Sort (IS) benchmark, designed to stress random access memory; the Conjugate Gradient (CG) benchmark, designed to test irregular memory access and communication; the Multi-Grid (MG) benchmark on a sequence of meshes, designed to stress the communication network; and the discrete 3D fast Fourier Transform (FT) benchmark, designed to test all-to-all communication. We have also implemented the Lower-Upper Gauss-Seidel solver (LU) pseudo application.

The following is a description of the benchmarks we have implemented:

EP. The embarrassingly parallel kernel. Designed to provide an estimate of the maximum achievable floating point performance of a machine with minimal communication required.

IS. A large integer sort. This kernel is often important to particle method codes. The kernel is designed to test both integer computational performance as well as communication performance.

CG. A conjugate gradient method kernel used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations and tests irregular long distance communication, and uses unstructured matrix vector multiplication. This is the only kernel we tested that operates on sparse matrix data.

MG. A simplified multi-grid kernel. This benchmark requires highly structured long distance communication and tests both short and long distance data communication.

FT. A 3D partial differential equation solution using FFTs. This kernel is paramount in many signal processing and spectral codes. It rigorously tests the long distance communication performance of a machine.

LU. A lower-upper symmetric Gauss-Seidel solver. This benchmarks solves a synthetic system of nonlinear partial differential equations (PDEs) using a symmetric successive over-relaxation (SSOR) solver.

The LU benchmark is one of the three pseudo applications included in the NAS benchmarks. We have chosen to only implement one of the three pseudo applications, because the applications are significantly more complicated in their implementation than the five kernels. Furthermore, all three of the pseudo applications solves the same system of PDEs, each using a different algorithm.

Seoul National University has a reference implementation of the NAS Parallel Benchmarks called SNU NPB [30]. All of these benchmarks are provided with a reference implementation that includes a highly tuned OpenMP version of the benchmark as well as a highly tuned serial version of the benchmark. We compare our generated code against this highly tuned OpenMP version and reference serial implementations.

The NPB suite come with provided data sets of varying size. From smallest to largest,

they are sizes S, W, A, B, C, and D. We have run all experiments with the size C, as it executes in a reasonable amount of time on a single modern shared memory machine.

For each benchmark, we have collected data for four versions of the code. The line labeled OMP is the hand coded, highly tuned, OpenMP version of the code provided by the SNU NPB. The lines labeled PIL2OMP, PIL2SCALE, and PIL2OCR are the HTA implementations compiled through PIL generating OpenMP, SCALE, and OCR code respectively. All of the speedup numbers are calculated against the performance of the best known sequential implementation of the benchmark from the SNU NPB implementation. Most of the NAS benchmarks require a power of two number of processors. We only run the benchmarks with power of two number of processors for all of the benchmarks. For each number of processors, the benchmarks were run ten times. We selected the minimum time for the ten runs as the maximum performance achievable for that number of processors.

7.2 HTA NAS Benchmarks Fork/Join Results

In this section we compare the obtained results using HTA fork/join implementation of the NAS benchmarks with the highly tuned hand coded version of the NAS benchmarks distributed as a reference implementation. Our goal with this implementation of the NAS benchmarks was to meet or exceed the performance available in the hand coded OpenMP implementation. The coding strategies and shared memory optimizations available to the programmers in the OpenMP version of the benchmark are all available to the programmer of the HTA codes. Furthermore, the parallel constructs used by the programmer of the hand coded OpenMP benchmarks are the same ones used and generated by the PIL compiler.

The following is a description of our experiences and observations with each of the implemented fork/join NAS benchmarks.

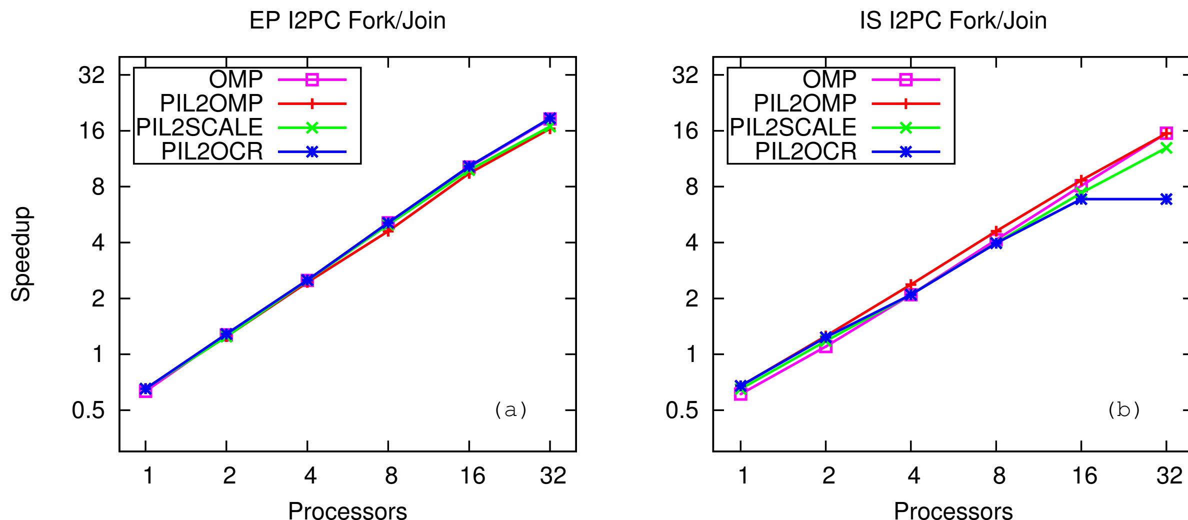


Figure 7.1: EP and IS fork/join performance on the I2PC machine.

EP. The EP benchmark results are rather straightforward. Figure 7.1 (a) shows the speedups achieved for the fork/join implementation of the EP benchmark on the I2PC machine. As described previously, the purpose of this benchmark is to provide as near maximal floating point performance available on the machine. Furthermore, the benchmark has perfect load balancing. Thus, we expect to see no real difference between the implementations, with each implementation achieving near linear speedup. Indeed, all of the different versions of the code have similar performance. The reason we see sub-linear scaling is because the machine has Intel's Turbo Boost technology turned on, which means the runs with a small number of threads running use a higher clock frequency than with a large number of threads running. Also note that the sequential version of the code is very efficient, and all of the parallel implementations for all of the benchmarks have a slowdown for their single threaded version.

IS. The Integer Sort benchmark sorts a large one dimensional array of integers. The array is initialized using a pseudo random number generator. In its initial distribution, each processor receives the same number of integers to perform a local bucket sort. After the

local sort is completed, the buckets are assigned to each processor. Each processor receives an equal number of buckets, but due to the random distribution of keys, there is a load imbalance because of the varying size of the buckets.

The benchmark runs an outer loop that performs the sorting multiple times to increase the running time of the benchmark. The sorting is broken into three phases. First there is the local sorting. Second there is a sharing of the data using a circular shift operation. Finally there is a parallel scan operation to compute the final location of each of the keys. The local sort in HTAs requires three fork/join operations, each requiring a global barrier. The communication step that performs the data swap is a global communication, which involves all processors.

Figure 7.1 (b) shows the speedups achieved for the fork/join implementation of the IS benchmark on the I2PC machine. It is observed that all of the PIL generated backend codes are able to exceed the performance of the hand coded OpenMP implementation for one and two processors. As the number of processors is increased the overhead of the load imbalance and the global communication operation begin to be apparent. However, the PIL generated OpenMP code is able to exceed the performance of the hand coded OpenMP for all number of processors. In OCR for 32 threads, the overhead of the barriers and global communication becomes overwhelming and we are unable to achieve an increase in performance moving from 16 to 32 threads.

CG. CG uses a sparse matrix and thus sparse tiles in HTA. CG is the only benchmark that uses sparse matrices. All of the other benchmarks use dense matrices. As is usual with sparse data, the sparse data in CG provides a load imbalance. As HTAs tiles the data into sparse tiles, each tile contains a different number of nonzero elements. The load imbalance could cause performance problems with the global barriers required in the fork/join mode.

CG has a nested loop. The outer loop has many iterations. Within the inner loop

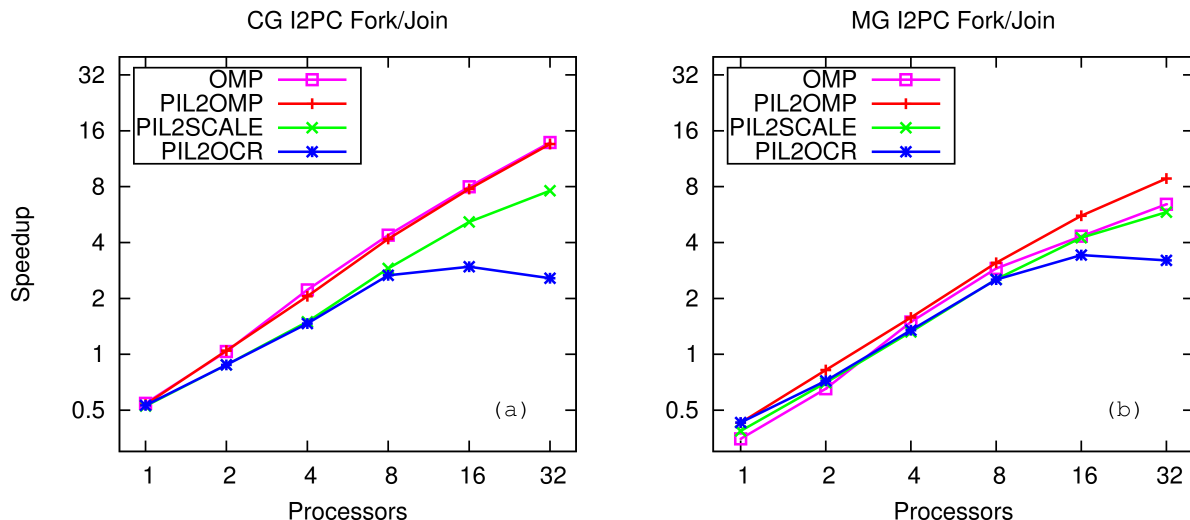


Figure 7.2: CG and MG fork/join performance on the I2PC machine.

there are four map operations and two global reductions. The global reductions are global communication that have a high cost, even though only one scalar value per tile is exchanged within the reduction.

Figure 7.2 (a) shows the speedups achieved for the fork/join implementation of the CG benchmark on the I2PC machine. Once again, we were able to match the performance of the hand coded OpenMP implementation with our generated OpenMP code. However, the cost of the global communication is significantly higher in SCALE and OCR than it is in OpenMP. The SCALE generated code was able to scale nearly as well as the generated OpenMP code, albeit with an added overhead. In OCR, as the number of threads increased, the overhead of the global communication becomes too great, and the code stops scaling.

MG. The multi-grid benchmark is very communication heavy. It requires a lot of short and long distance communications. The kernel operates on a 3D array that requires updates to halo cells at 3D tile boundaries every iteration of the main computation loop. The values of the halo cells are able to be read directly in the fork/join version from neighboring tiles. There is a fork/join operation with an implicit barrier just before the reading of the halo cell

values, so that we know the data is computed and ready to be read.

Figure 7.2 (b) shows the speedups achieved for the fork/join implementation of the MG benchmark on the I2PC machine. The data locality achieved with the tiling of the HTAs allows the OpenMP code generated from PIL to slightly outperform the hand coded OpenMP version. The SCALE version has increased overheads, but is able to match the performance of the hand coded OpenMP. However, the overheads of the communication and barriers in OCR cause its performance to drop for large numbers of threads.

FT. The biggest source of overhead is a communication operation that must occur each iteration of the outer kernel loop. The benchmark requires a transpose of the array being operated on. The highly tuned hand coded OpenMP version of the code does not need to do the transpose because the array is laid out contiguously in memory. Using this information they do an optimization to just swap the index variables when accessing the data in the array.

In the HTA fork/join code we are not able to completely avoid the overhead of the transpose like they can in the hand coded OpenMP version of the code. In HTAs each tile of an array is allocated separately leading to contiguous memory accesses *within* a tile, but not across tiles. Thus, we can swap the accesses to the data within a tile; however, we must swap owners of tiles. The meta-data for the entire HTA must be changed to update who owns which tile. This is analogous to swapping pointers to the tiles. The HTA code uses a single thread to perform the update to the meta-data.

Figure 7.3 (a) shows the speedups achieved for the fork/join implementation of the FT benchmark on the I2PC machine. With the fork/join version of the FT benchmark we are able to achieve almost identical performance to the hand coded OpenMP version of the benchmark despite the extra work that has to be done within the HTA library.

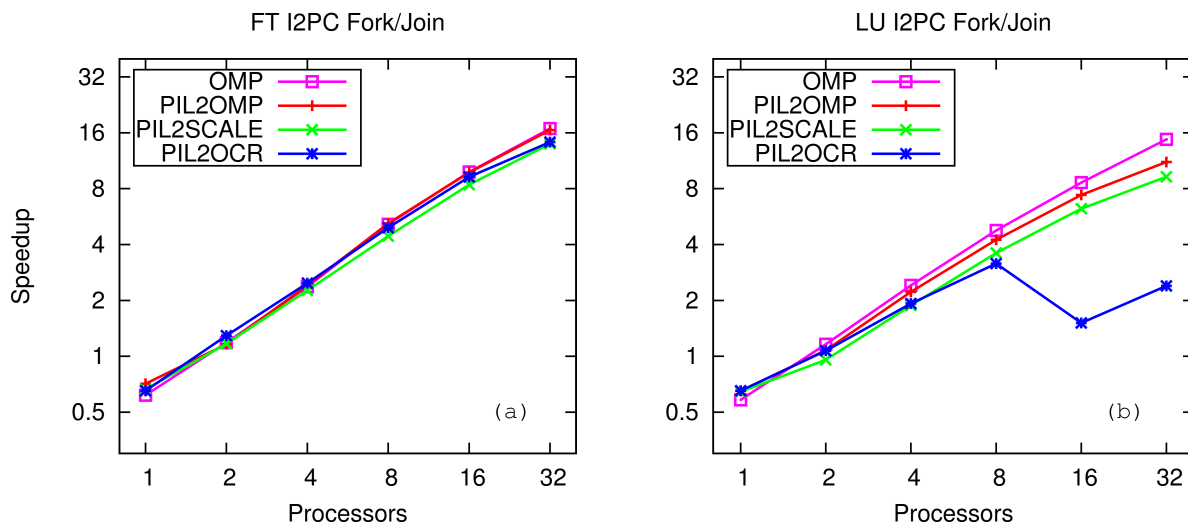


Figure 7.3: FT and LU fork/join performance on the I2PC machine.

LU. The LU benchmark is not a kernel benchmark as the previous five benchmarks have. It is significantly more complicated as a pseudo application. The LU benchmark operates on a 3D mesh in a wavefront computation. There are two main sources of overhead required in the benchmark. The first is the load imbalance in the wavefront computations. The second is the updates of the halo cells in the mesh.

The main computation proceeds as two wave front operations. The first wavefronting from tile $(0, 0, 0)$ to the opposite corner tile $(N-1, N-1, N-1)$. The second wavefronting does an update from the corner tile $(N-1, N-1, N-1)$ back to the corner tile $(0, 0, 0)$. Obviously, this wavefronting starts with only a single thread able to update its tile, followed by three threads updating tiles, as the wavefront proceeds to maximal parallelism. Then the parallelism is decreases back to a single tile being able to update. This waxing and waning phases cause a load imbalance, especially when using a large number of processors as they sit idle waiting to be able to update their tiles.

In order to improve the amount of parallelism available in the wave fronting computation, we have to over-decompose the tiles. We had to do experimentation to determine the best

decomposition of the tiles and for example, in the 32 thread case, we decompose the x,y,z dimensions into 8x8x8 (512 total) tiles. We have to increase the tiling as much as possible to keep the parallelism up, however, as we increase the number of tiles, the overhead of the exchange of boundary elements becomes large, and overwhelming.

In the fork/join there is the global barrier after each iteration for the computation. The computation is completed on a tile, then the owner of the data copies data into the neighbors for the halo cell updates. The copy is to all neighbors in all dimensions. Then there is a the global barrier for the iteration that asserts that all updates for this iteration are completed before the next iteration can begin.

Figure 7.3 (b) shows the speedups achieved for the fork/join implementation of the LU benchmark on the I2PC machine. If you look at the performance graph, our HTA version has overhead compared to the base OpenMP implementation. However, in a separate experiment we have performed in which we remove the updates of the boundary conditions, we compute wrong results, but the experiment shows that our OpenMP implementation then outperforms their OpenMP version, and our SCALE version meets the performance of their OpenMP. In this experiment, OCR performance also greatly improves. Thus, the overhead of tiling introduces communication overhead that is not in the untiled OpenMP version, and causes performance degradation compared to the untiled OpenMP implementation.

Fork/Join Conclusions. We are able to draw several conclusions from the plots in Figures 7.1 to 7.3. First, we can see that for all of the benchmarks, the PIL generated OpenMP code has very similar performance to the hand written OpenMP code. This means that the overheads, if any of HTAs and the PIL generated code are very small. Second, in general, we can see that the PIL generated SCALE code does not quite perform as well as the PIL generated OpenMP code. Third, we can see that the PIL generated OCR code typically has the lowest performance, and can even stop scaling with a large number of processors. In

Chapter 6, we took a deeper look into the causes of this performance gap between the generated code and their runtimes. The results discussed here support the observations made in Chapter 6.

Several of the benchmarks have load imbalances in them. We want to improve the performance of the benchmarks by removing the barriers that are necessary in the fork/join versions of the code. The next section takes a look at our experiences when allowing more asynchrony through point-to-point synchronizations instead of global barriers.

7.3 HTA NAS Benchmarks SPMD Results

As previously mentioned, the fork/join version of these benchmarks rely on the global barrier for synchronization. Global barriers are expensive, especially for a large number of processes. When implementing applications with the SPMD mode, the programmer is able to leverage the communication mechanisms in PIL to do point-to-point synchronizations rather than global synchronizations. This means that two processes only meet when they *need* to share data, in contrast to the fork/join model where *all* threads *always* participate in the *global* barrier for synchronization. We expect the SPMD implementation of the NAS benchmarks to be able to exploit the asynchrony available in the parallel runtimes by allowing the codes to be more asynchronous.

SPMD implementations of the benchmarks require the addition of communication to achieve this asynchrony, as opposed to the fork/join versions of the code that can usually leverage shared memory optimizations to read data directly. The available asynchrony in the algorithm and improvements provided by using point-to-point synchronization will have to outweigh the added overhead of this communication in order for the new implementations to achieve greater speedups than the fork/join versions.

The following is a discussion on the performance of each of the NAS benchmark's SPMD

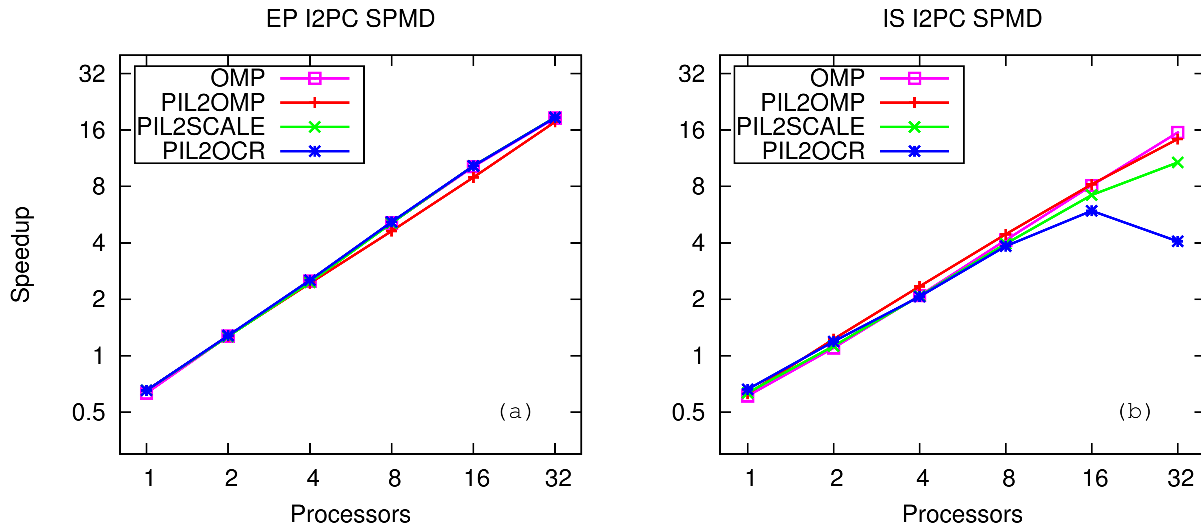


Figure 7.4: EP and IS SPMD performance on the I2PC machine.

implementation.

EP. The EP benchmark does not change much in its SPMD implementation. Figure 7.4 (a) shows the speedups achieved for the SPMD implementation of the EP benchmark on the I2PC machine. The embarrassingly parallel nature of the benchmark means that the threads interact very little in fork/join or this SPMD implementation. As you can see in the results, all of the codes perform very similarly with near linear speedup. The only real difference between the SPMD and fork/join implementations are the way that the three reductions at the end of the benchmark are performed. Reductions in the SPMD framework require a global communication. However, the three reductions are a very small fraction of the overall execution time and have no real impact vs the fork/join version.

IS. The IS benchmark performance does not change much in its SPMD implementation. Figure 7.4 (b) shows the speedups achieved for the SPMD implementation of the IS benchmark on the I2PC machine. In fork/join there is a shared memory optimization step in that each thread can count the number of items each other thread will be putting into its

local bucket. Each thread looks at how many keys are in each bucket in each other thread to know its final values. In the SPMD version the communication has to be done with a circular shift in order to communicate the data from local sorts to the final position. In the IS benchmark we are unable to achieve useful asynchrony since the circular shift operation is a global communication. The OCR performance suffers at 32 threads with this global communication, and its performance is decreased compared to the fork/join version.

CG. As discussed earlier, CG uses a sparse matrix that provides a load imbalance amongst the processors. However, there is a global reduction that needs to be performed each iteration of the main computational loop. The global reduction has no benefit from point-to-point communication because it is a global operation. In SPMD, the global reduction has higher overhead than the fork/join version. This means that even though there is a load imbalance in the computation, the SPMD implementation has no benefit over the fork/join version because *both* require global synchronizations. In fact, the global communication has a higher overhead in the SPMD version of the code.

Figure 7.5 (a) shows the speedups achieved for the SPMD implementation of the CG benchmark on the I2PC machine. It can be observed that the PIL generated OpenMP code no longer matches the performance of the hand coded OpenMP version, but is slightly below it. Furthermore, at large numbers of threads, the SCALE and OCR generated code stop scaling. Once again OCR is the hardest hit with the overheads required for global communication with the OCR runtime.

MG. The multi-grid benchmark is very communication heavy. It requires a lot of short and long distance communications. The kernel operates on a 3D array that requires updates to halo cells at 3D tile boundaries every iteration of the main computation loop. In SPMD the halo regions have to be communicated with point-to-point communications each iteration. However, the implementation of the halo cell exchange is a global communication. Because

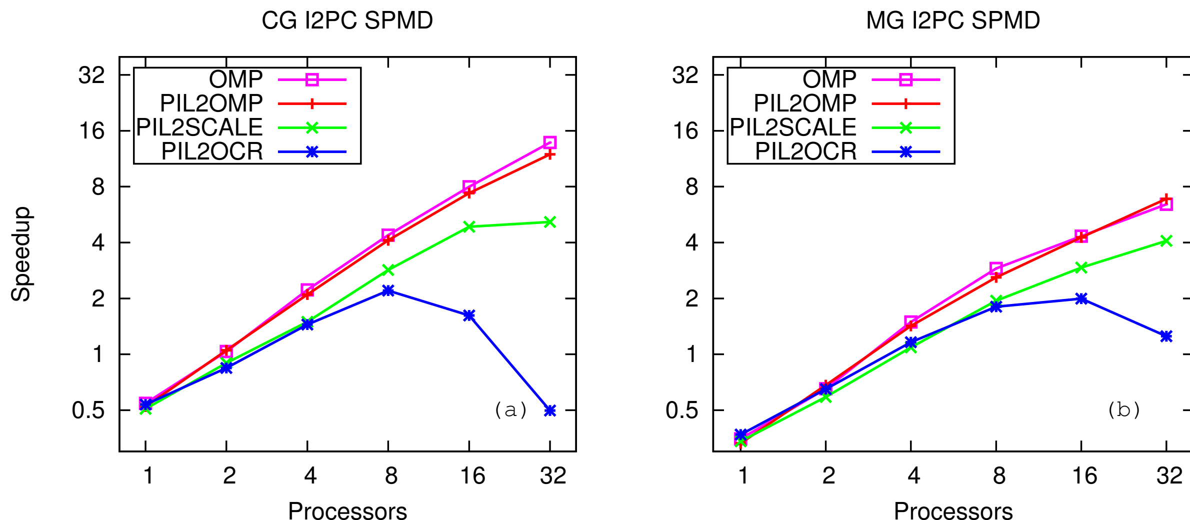


Figure 7.5: CG and MG SPMD performance on the I2PC machine.

of this, and the added overhead of the communication involved, the SPMD version of the code actually performs worse for each backend than in the fork/join version.

Figure 7.5 (b) shows the speedups achieved for the SPMD implementation of the MG benchmark on the I2PC machine. The performance of the PIL generated OpenMP code is now below that of the hand coded OpenMP implementation. There is a steady overhead incurred because of the communications in the SCALE version, pushing the performance below the PIL generated OpenMP code. The OCR runtime has a hard time with all of the communications and incurs the most overhead. In fact, it slows down when using 32 processors versus using 16. This once again highlights the very high communication costs in OCR.

FT. The algorithm is the same as fork/join except that communication has to be involved in order to achieve the transpose. In the fork/join version of the code there are great opportunities. The hand coded OpenMP code doesn't even perform the transpose at all! They just swap the indices used to access the array to achieve the same effect. In the HTA version of the benchmark we have to update the HTA meta-data for who owns the tiles

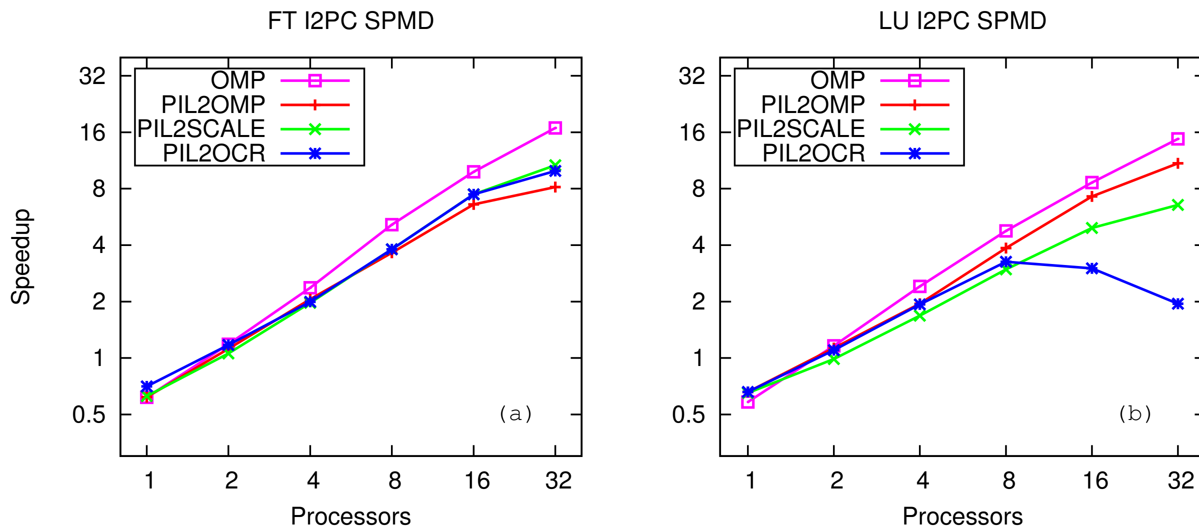


Figure 7.6: FT and LU SPMD performance on the I2PC machine.

(analogous to pointer swapping) to achieve the transpose of the tiles, and then within a tile we can interchange the indices used to access the data. However, things are not so simple in the SPMD version. In the fork/join version of HTAs there is a single master thread that can perform the meta-data update. However, in the SPMD version, each rank has to participate in the update of the meta-data. There is all-to-all communication done for each rank to swap tiles with other ranks. This all-to-all communication has large overhead. The large overhead is because it is a *global communication*, and has even higher overhead than the global barriers used in the fork/join version. Furthermore, in the SPMD version of the code, we cannot do the optimization to only swap pointers, the entire tile has to be communicated.

Figure 7.6 (a) shows the speedups achieved for the SPMD implementation of the FT benchmark on the I2PC machine. In this benchmark, the overhead of the global communication affects all of the PIL backends. They each achieve approximately the same performance as each other, and less than the hand coded OpenMP version.

LU. As discussed in the previous section, the computation in LU follows a wavefronting pattern. After each update during the wavefronting computations, it is necessary to synchro-

nize the halo cells of each tile from its neighboring tiles. This can be done point-to-point, but each rank that owned a tile that was updated has to participate. Since we perform over decomposition to keep as many processors active as possible, during most iterations all ranks participate, usually with multiple tile updates per rank.

In the SPMD version of the code the updates are applied point-to-point to any of your neighbors that needs the halo data. We leverage the ability of the asynchronous sends to post all of the updates to all of our neighbors so we can immediately send the updates and proceed to the next iteration. At the beginning of each iteration, we have to post all of the necessary receives from our neighbors to drain the buffers and obtain the halo region updates before we can perform the computation for the current iteration. This added overhead is a slight overhead vs the fork/join version.

The major source of overhead in the SPMD version of the code is the static distribution of data, as discussed in detail in Section 6.5.5. We have to decide a priori, how the tiles will be distributed to the processors. However, as the wave front computation proceeds, variations of the tiles are being operated on. It is very difficult to define a static distribution that provides an optimal scheduling to keep all of the processors load balanced. The imbalance of the load causes during some iterations some ranks to complete early, and others to have more work. The ranks that complete their iteration early can do nothing but wait for their slow neighbors to complete and post their updates before they can begin the computations of the next iteration.

Figure 7.6 (b) shows the speedups achieved for the SPMD implementation of the LU benchmark on the I2PC machine. It can be observed that each of the generated codes from PIL have slightly decreased performance compared to the fork/join versions of the code.

SPMD Conclusions. The HTA SPMD versions of the NAS benchmarks are a good illustration of PIL's competency. It has been shown in this section that PIL does indeed provide

a good framework on which to implement SPMD codes in HTAs. While the performance of the NAS benchmarks in SPMD HTA was not an improvement as we expected over the HTA fork/join versions of the codes, we have learned some very important lessons. One of the most important lessons came when studying the performance of the LU benchmark.

It was determined while studying the LU benchmark that one of the primary sources of inefficiency in the benchmark had nothing to do with overheads caused by the PIL compiler, the implementation of the HTA library, or the implementation of the benchmark code. The inefficiency was in fact caused by an underlying assumption of the HTA model. This assumption is that the tiles will be statically distributed to each of the ranks. Once a tile is assigned to a rank, that rank must perform the updates to the tile. If we want to perform a load balancing later in the code, we must perform a redistribution of the tiles. The redistribution of the tiles is a very expensive all-to-all communication. However, during some instances, like the LU benchmark, the load imbalance from the static distribution of data also causes overhead.

7.4 Conclusions from the NAS Benchmarks

In this section we have shown that it is possible for the HTA library implementation in PIL to implement the NAS benchmark suite. We are able to generate code for the three backends OpenMP, SCALE, and OCR, and the code that is generated is able to perform as well as the SNU NPB highly tuned OpenMP implementation of the benchmarks. We show that PIL is able to facilitate the programming of the HTA library with both fork/join and SPMD parallelism.

Chapter 8

Related Work

LLVM [18] is an intermediate language and framework for compilation. The goal of LLVM is to provide opportunities for optimization at all levels of the compilation process. However, LLVM is a low level language, without knowledge of parallel semantics. Any optimizations must be done between parallel operations. PIL is designed to work at a higher level than LLVM. PIL retains the semantic knowledge of the parallelism to allow for optimizations *across* parallel operations. After PIL has generated code for the target runtime, LLVM can optimize the code at a lower level.

NESL [8] is a portable nested data parallel language. It is build on the VCODE intermediate language [7]. VCODE is a very simple language designed around data-parallel operations on a vector stack machine. Thus, VCODE is limited to data-parallel operations on vectors on a shared memory machine, while PIL uses task and data-parallel constructs on shared and distributed memory machines on arbitrary data structures.

The ROSE compiler [27] is a source-to-source compilation framework. The focus for the project has been on sequential low-level transformations and optimizations with input from the sequential languages C, C++, and Fortran. However, they have only looked into a few parallel operations like the OpenMP accelerator directives [21].

Some compilation frameworks such as those for Habanero-Java [32] and SPIRE [17] provide extensions to sequential intermediate representations for parallelism. However, the parallel extensions are to traditional intermediate languages that operate at a level much lower than PIL.

While not an intermediate language, MPI is a successful parallel language. In the paper *Learning from the Success of MPI* [15] the author outlines the six keys to MPI's success: portability, performance, simplicity, modularity, composability, and completeness. We believe these are keys to success for any language, and have kept them in mind while designing PIL, and will build on them in the future.

Chapter 9

Future Work

In the future, we would like to experiment with more languages. We have put great thought into frontends for languages such as OpenMP, and MPI generating PIL code. As well as more backends, such as MPI, Pthreads, and Charm++. Support for some of the more popular languages would help cement the foundation for the utility of PIL.

As more languages are supported, we would like to do a recursive code generation study. Something that we have always wanted to do was an experiment of generation from one language to the same language. For example, start with an OpenMP program, compile through PIL to generate OpenMP backend code. Then, we can see how the generated code compares to the original code. The generated code can be compiled through PIL again to do another comparison. It would be very interesting to see the effects of the compilation process of PIL on this recursively generated code.

We have provided a foundation for the implementation of an optimization framework. In this thesis, we have described a few useful optimizations, but have not yet implemented them. The study of the application of traditional compiler optimizations, as well as the discovery of new techniques, at the level of granularity and expressiveness available in PIL is an entire body of research that has yet to be explored.

The current implementation of the HTA notation relies on the implementation of the library of HTA operations in PIL. However, it is possible to implement an HTA compiler to generate PIL code. We believe that such a compiler might provide more efficient HTA applications.

The current communication and synchronization library relies on some very simple implementations of the communication operations. Highly optimized libraries, like MPI, provide much more robust communication operations. We believe that we can build off of the body of work put into the optimization of these operations to provide a more efficient communication library in PIL. Furthermore, the communication library currently relies on messages to be sent and received in order. There is no mechanism yet to provide messages to be communicated in arbitrary order. If a receive operation is executed out of the order of the sent messages, the receive operation will not ever complete. We have found the current communication model sufficient for the work in this thesis, but a more robust model could provide opportunities for algorithmic improvements in the PIL code.

Several of the operations in the backends for SCALE and OCR rely on the use of a continuation task. We are awaiting the implementation of these continuation tasks in OCR and SCALE, and believe that their use could provide some performance advantages in their respective runtimes.

9.1 Limitations of the Language

PIL provides a framework to support task parallelism, and provides syntactic sugar for data parallelism and SPMD parallelism. Since the syntactic sugar for data parallelism provides a parallel loop operation within a single node, nesting task parallelism within a data parallel operation is not allowed. However, data parallelism can be constructed from the task parallelism that PIL provides, to nest a task parallel operation within a data parallel operation.

Task parallelism frequently takes on a form of performing some sequential work within a task, spawning a new task to perform some new work, and the original task continuing on to do some more sequential work. This model provides creation of new tasks in the middle of an executing task. However, in PIL, new tasks can only be created at the end of a PIL node. This means, that to complete the previously mentioned example, a task must be executing, end its PIL node, and create two new tasks to continue on. New parallelism cannot be created within a node. Only between PIL node invocations.

Additionally, PIL does not support communication within a task. The currently executing PIL node must end, and call a new communication node. This breaks the task that wants to perform communication into multiple nodes. A compiler that can generate PIL code, must be able to break larger tasks into smaller tasks to perform communication operations.

Data in PIL is currently shared amongst all of the tasks in the same memory space. This implies that there is no task private data. When necessary, we have solved this problem by allocating an array of variables, and having each task only access the variable in the array assigned to it. This workaround is a bit clumsy, and language support for task private data would be beneficial.

One of the biggest problems with generating PIL code is that the graph of PIL nodes, as declared through the predecessor and successor lists, must be declared statically. This means that any successors to a node must be determined before generating PIL code. This could pose problems to a compiler if a precise graph cannot be built. However, if the precise successors to a node cannot be determined, all nodes can be declared as successors. This ensures a correct graph, as all nodes are made to be possible successors, but could make reasoning about the graph difficult for the optimization framework.

Chapter 10

Conclusion

This thesis describes an intermediate language for parallel compilation, discusses design issues with the language, and shows performance results for various benchmarks and three backends. PIL can support both data parallelism and task parallelism on shared memory or distributed memory machines, as well as a notation for providing SPMD programming. We introduce the notion of any-to-any compilation with the goal that PIL can be the target of any parallel language and can generate any parallel runtime. We discuss the ideas of what is required to provide such an intermediate language, and provide an implementation with three example backends: OpenMP, SCALE, and OCR. Results from a tiled Cholesky decomposition example, as well as the NAS benchmark suite discussed in this thesis, show that the approach is promising.

In addition, we discuss the idea of multiresolution programming, and describe how its inclusion in our language can benefit user in many ways including programmability, portability, and performance.

In summary, we believe that as new and more complex runtime systems for supercomputers are developed, the development of programming, or reprogramming, an algorithm for the new runtime system will become increasingly complex. We strongly believe that users

will begin to rely more on tools, like PIL, to help them target these new runtime systems.

Appendix A

Supplemental Results

This appendix contains the performance results obtained on the X-Stack machine described in Section 5.2. For brevity, these numbers were not discussed in the chapters on performance results, but were studied heavily during our evaluation. We decided to include the results in this appendix for completeness. One of the primary reasons we did not include these results in the primary descriptions on performance is because the X-Stack machine is only a 16 core machine. The 32 core results rely on the SMT technology to handle two threads per core. We believe the numbers are more clear when using as many cores as possible, but not using SMT, so we reported results from the I2PC machine during our discussion.

A.1 NAS Benchmarks

Figure A.1 shows the performance achieved from all six of the NAS benchmarks represented as speedup numbers on the X-Stack machine.

Figure A.2 shows the performance achieved from all six of the NAS benchmarks represented as speedup numbers on the X-Stack machine.

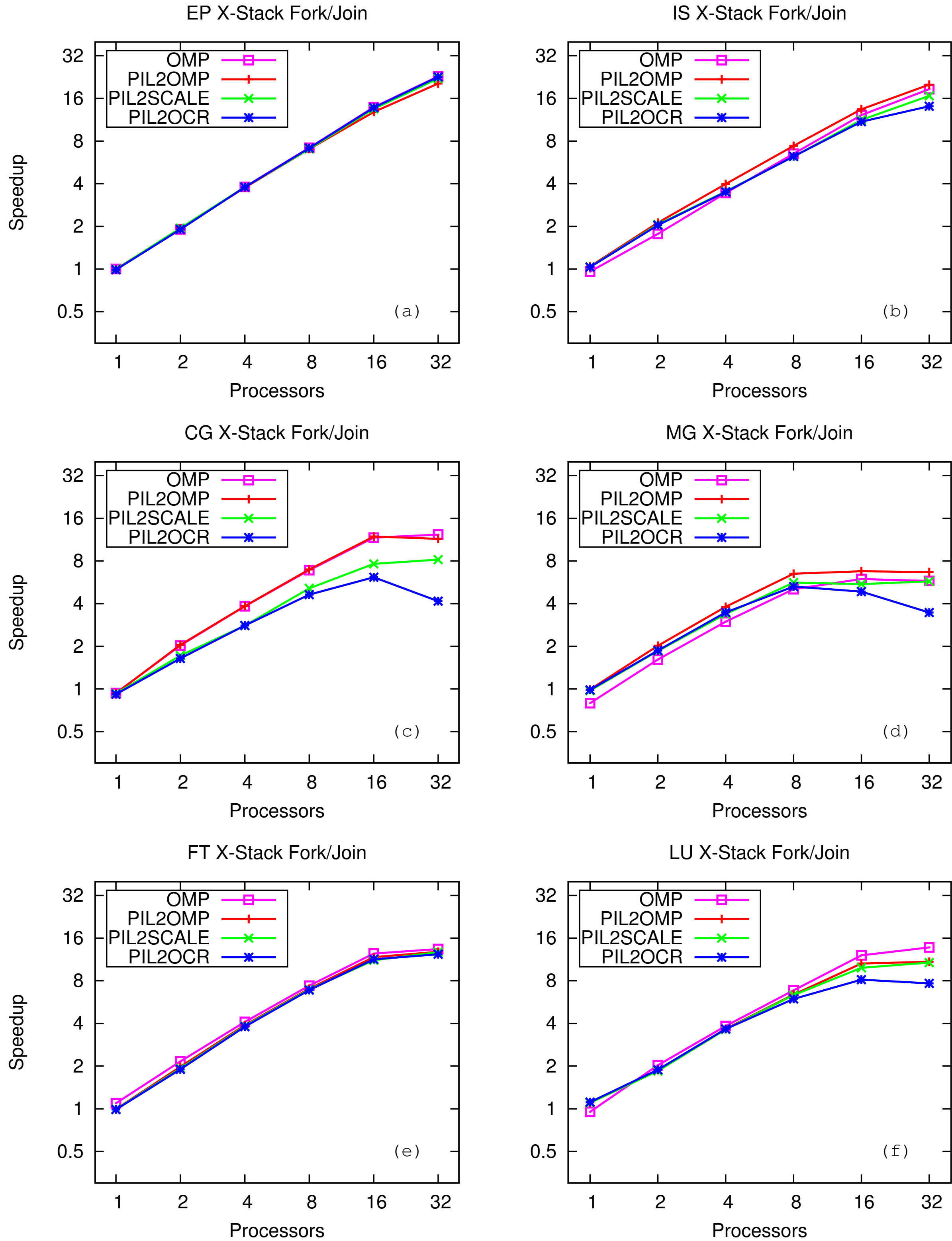


Figure A.1: All fork/join performance on X-Stack.

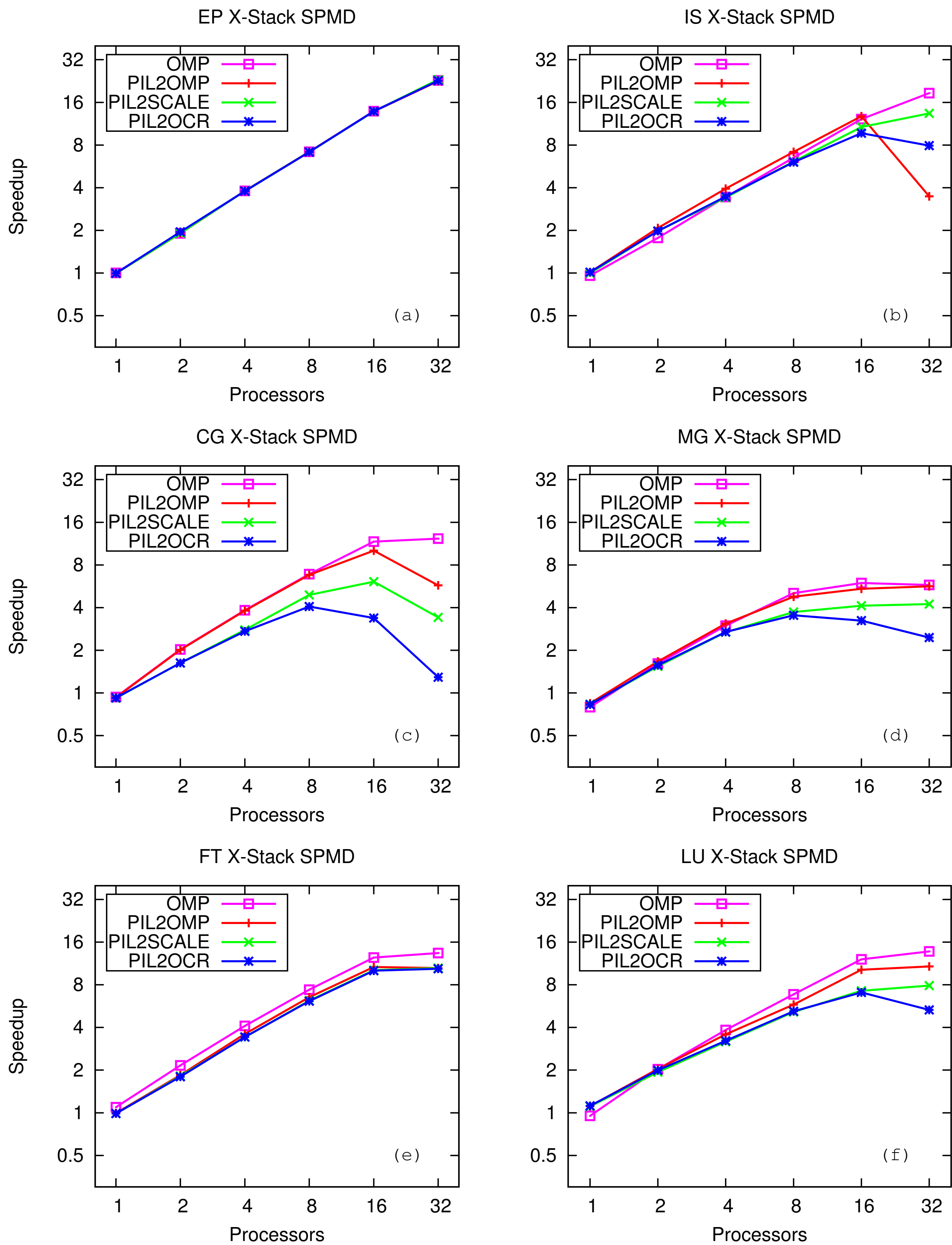


Figure A.2: All fork/join performance on X-Stack.

A.2 Cholesky Factorization

The performance results for the Cholesky factorization experiments on the X-Stack machine are included here. All results use the same 80×80 tiling as discussed in Section 6.5. Figure A.3 shows the results for the 1×1 tiling experiment. Figure A.4 shows the results for the 100×100 tiling experiment. Figure A.5 shows the results for the 200×200 tiling experiment. Figure A.6 shows the results for the random tiling experiment.

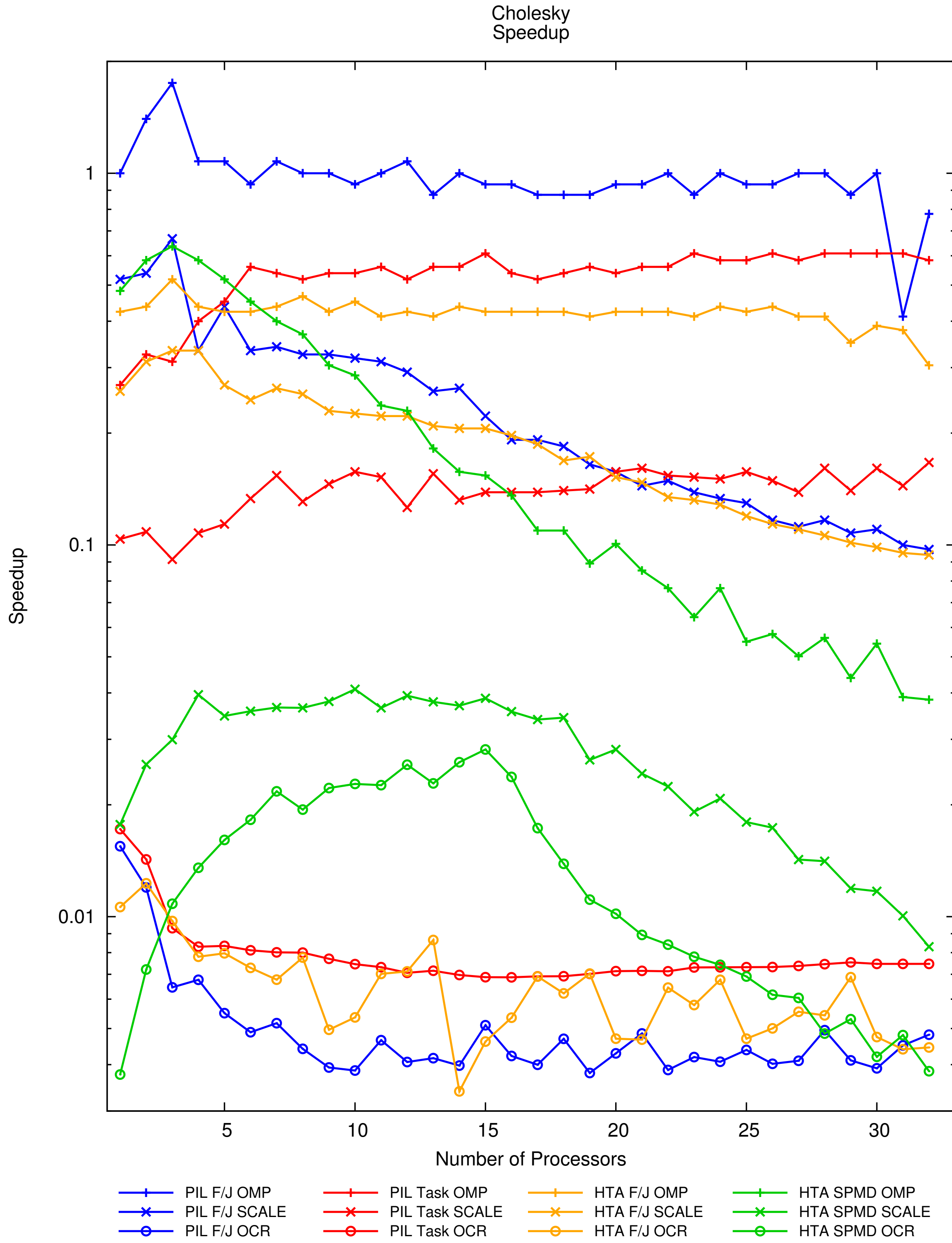


Figure A.3: Cholesky factorization speedup on the X-Stack machine with 1×1 tiling.

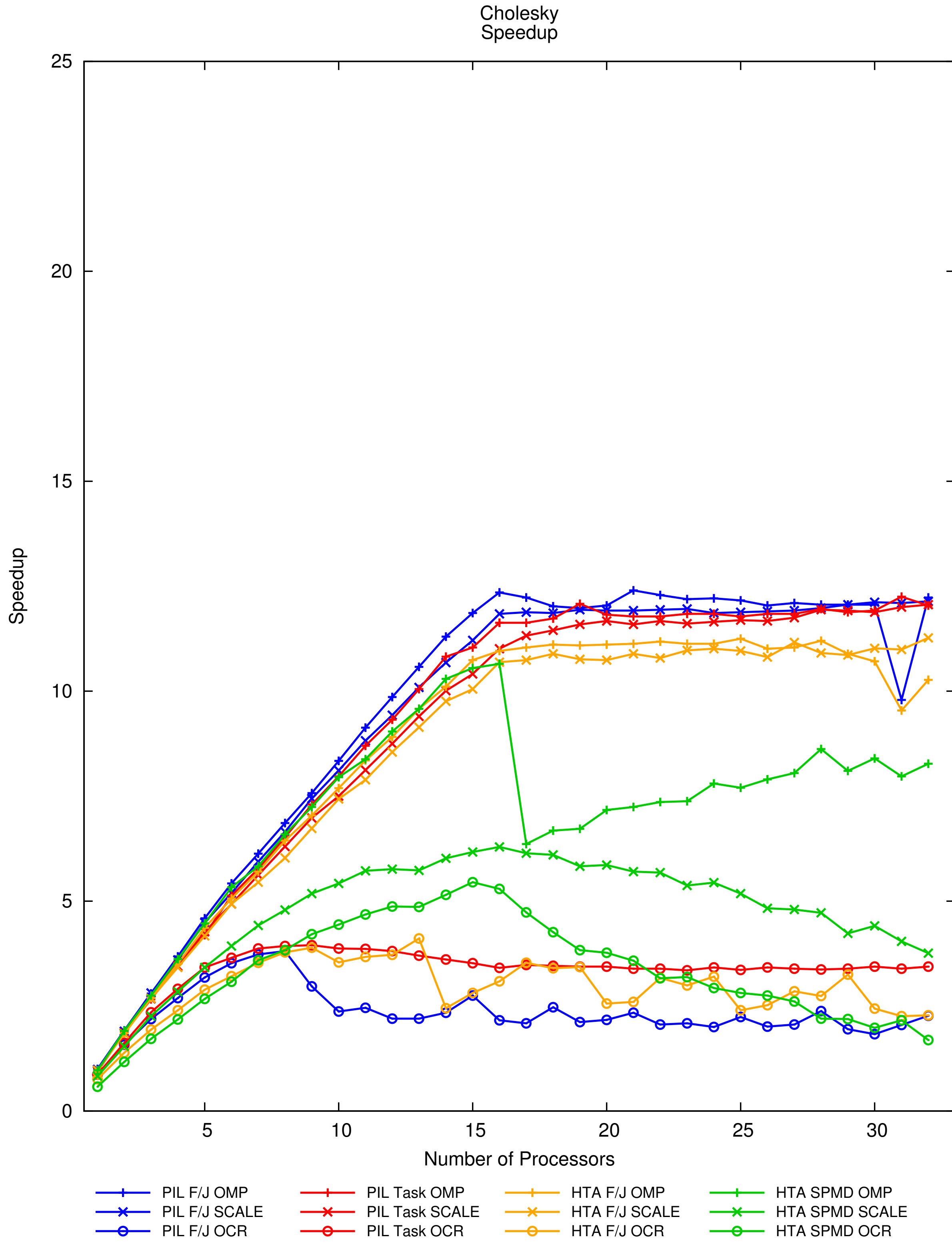


Figure A.4: Cholesky factorization speedup on the X-Stack machine with 100×100 tiling.

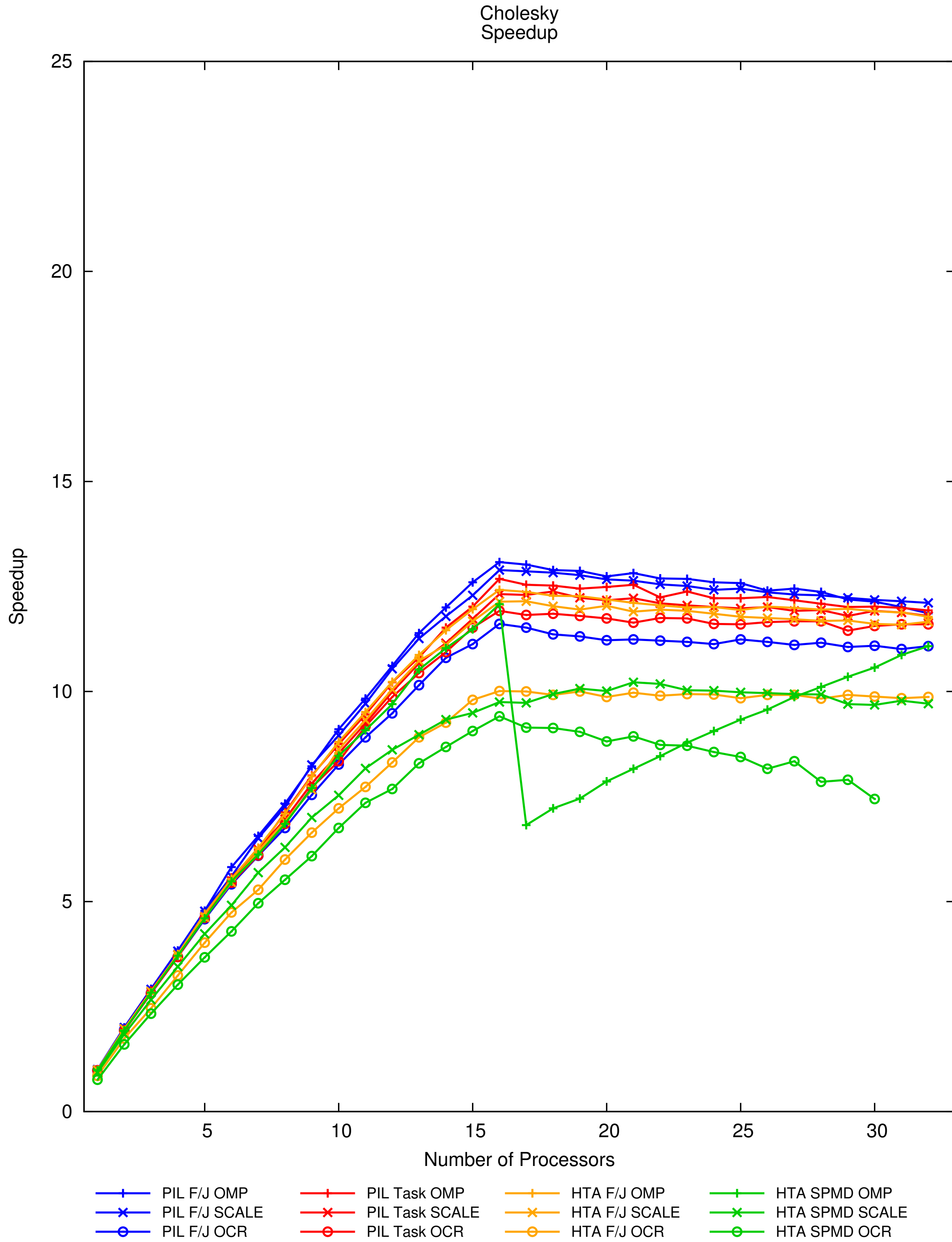


Figure A.5: Cholesky factorization speedup on the X-Stack machine with 200×200 tiling.

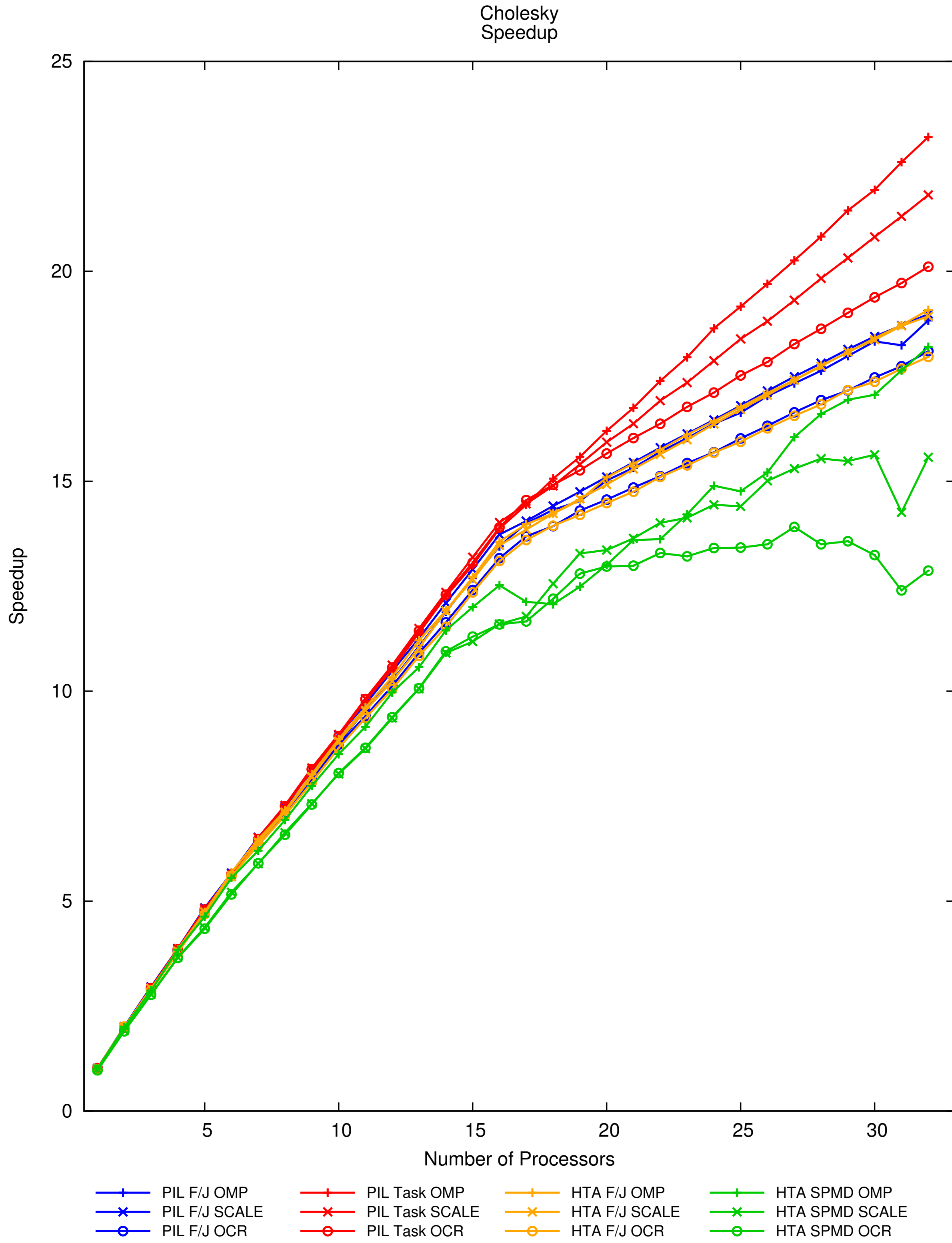


Figure A.6: Cholesky factorization speedup on the X-Stack machine with random tiling.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] S. Ashby, P. B. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright. The opportunities and challenges of exascale computing. Summary report of the Advanced Scientific Computing Advisory Committee (ASCAC) subcommittee at the US Department of Energy Office of Science, 2010.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical Report RNR-94-007, The National Aeronautics and Space Administration (NASA), March 1994.

[5] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[6] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM.

[7] G. Blelloch and S. Chatterjee. Vcode: A data-parallel intermediate language. In *In Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480, 1990.

[8] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, Apr. 1994.

[9] J. Brodman, B. B. Fraguela, M. J. Garzarán, and D. Padua. Design issues in parallel array languages for shared memory. In *Proceedings of the 8th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '08, pages 208–217, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, Aug. 2010.

[11] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.

[12] B. L. Chamberlain and C. Inc. Multiresolution languages for portable yet efficient parallel programming, 2007.

[13] Charm++. http://charm.cs.uiuc.edu/.

[14] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, Dec. 1974.

[15] W. Gropp. Learning from the success of mpi. In *Proceedings of the 8th International Conference on High Performance Computing*, HiPC '01, pages 81–94, London, UK, UK, 2001. Springer-Verlag.

[16] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua. Programming with tiles. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 111–122, New York, NY, USA, 2008. ACM.

[17] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. Spire: A sequential to parallel intermediate representation extension. Technical report, 2012.

[18] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[19] C. Lauderdale and R. Khan. Towards a codelet-based runtime for exascale computing: Position paper. In *Proceedings of the 2Nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM.

[20] R. Lethin, P. Mattson, E. Shweitz, A. Leung, V. Litvinov, M. Engle, and C. Garrett. R-stream 3.0: Technologies for high level embedded application mapping. In *Proceedings of the 8th Annual High Performance Embedded Computing (HPEC) Workshops*, 09 2004.

[21] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman. Early experiences with the openmp accelerator model. In *9th International Workshop on OpenMP*, IWOMP 2013, 2013.

[22] The message passing interface. http://www.mpi-forum.org.

[23] F. Mueller. Pthreads library interface, 1995.

[24] The open community runtime. https://01.org/open-community-runtime.

[25] Openmp. http://www.openmp.org.

[26] The parallel intermediate language. https://polaris.cs.uiuc.edu/~smith195/pil/pil-api.pdf, 2013.

[27] D. J. Quinlan. Rose: Compiler support for object-oriented frameworks. In *Proceedings of Conference on Parallel Compilers*, CPC2000, 2000.

[28] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 202–211, New York, NY, USA, 1986. ACM.

[29] E. Schweitz, R. Lethin, A. Leung, and B. Meister. A parametric high level compiler. In *Proceedings of the High Performance Embedded Computing Workshop (HPEC), Lexington, MA, USA, September*, Sep 2006.

[30] Seoul national university nas parallel benchmarks. http://aces.snu.ac.kr/software/snu-npb.

[31] C.-C. Yang, J. C. Pichel, A. R. Smith, and D. A. Padua. Hierarchically tiled array as a high-level abstraction for codelets. In *Fouth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, DFM, 2014.

[32] 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! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 329–340, New York, NY, USA, 2011. ACM.

[33] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.