UNIVERSITY OF CALIFORNIA,
IRVINE

Incremental Parallelization with Migration

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Wenhui Zhang

Dissertation Committee:
Professor Lubomir F. Bic, Co-Chair
Professor Michael B. Dillencourt, Co-Chair
Professor Amelia C. Regan

2014

DEDICATION

This dissertation is dedicated to my husband, Chuanwen, who has persistently supported me and understood the challenges and sacrifices; our lovely son, who is a smart and handsome boy and who wished his mom could have spent more time with him; my father who has been always encouraged me to pursue higher and higher and my mother who has offered her love without expecting anything back.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vii**

ACKNOWLEDGMENTS **viii**

CURRICULUM VITAE **ix**

ABSTRACT OF THE DISSERTATION **xi**

1 Introduction **1**
- 1.1 Research Motivation and Target Problem 1
- 1.2 Dissertation Overview and Organization 4

2 Navigational Programming **5**
- 2.1 Overview 5
- 2.2 Characteristics 9

3 Incremental Parallelization **15**
- 3.1 Methodology Overview 15
- 3.2 Two Examples 21
  - 3.2.1 An Example of Matrix Multiplication 21
  - 3.2.2 An Example of ADI 22
- 3.3 Code Transformation and Placement 27
  - 3.3.1 Partitioning phase 1: Adapt to the data distribution 27
  - 3.3.2 Partitioning phase 1: Example of ADI 28
  - 3.3.3 Partitioning phase 1: Example of Matrix Multiplication in one dimension 30
  - 3.3.4 Partitioning phase 1: Example of Matrix Multiplication in second dimension 32
  - 3.3.5 Partitioning phase 2: Increase opportunities for parallelism 35
  - 3.3.6 Partitioning phase 2: Example of ADI 35
  - 3.3.7 Partitioning phase 2: Example of Matrix Multiplication in second dimension 36
  - 3.3.8 Assigning a location to each tile 38
- 3.4 Parallelization Generation 38

3.4.1 Building the Weighted Dependence Graph 39
3.4.2 Building the Weighted Dependence Graph: Example of ADI 40
3.4.3 Building the Weighted Dependence Graph: Example of Matrix Multiplication 42
3.4.4 Combining tiles into threads 42
3.4.5 Generating threads for ADI 42
3.4.6 Generating threads for Matrix Multiplication in one dimension 44
3.4.7 Generating threads for Matrix Multiplication in second dimension . . 44
3.4.8 Generating the Code 49
3.4.9 Generating the Code: Example of ADI 49
3.4.10 Generating the Code: Example of Matrix Multiplication in one dimension 51
3.4.11 Generating the Code: Example of Matrix Multiplication in second dimension 53
3.5 Feedback Mechanism 56
3.5.1 Case 1: Feedback to Code Transformer 57
3.5.2 Case 2: Feedback to Data distributor 58

4 Experiments **62**
4.1 Experiment of ADI 62
4.2 Experiment of Matrix Multiplication 66
4.2.1 Gentlemans Algorithm of Matrix Multiplication 66
4.2.2 Performance data 67

5 Preliminary Extension **73**
5.1 Navigational Programming on Cloud 73
5.2 Investigation and Future Improvement 76

6 Related Work/Literature Review **79**
6.1 Distributed Parallel Programming 79
6.2 Parallelizing Compilers 82
6.3 Cloud Computing 83

7 Conclusion **86**
7.1 Contribution 86
7.2 Future Works 88

Bibliography **89**

LIST OF FIGURES

Page

2.1 Four Traveler's Traces 7
2.2 SPMD MPI View:Arrivals Departures at stations 7
2.3 NavP View:Itineraries of trains 8
2.4 Example: Three lines of code computing on distributed data (a) Sequential . 8
2.5 Example: Three lines of code computing on distributed data (b) MP 9
2.6 Example: Three lines of code computing on distributed data (c) NavP 9
2.7 Dependency Graph 10
2.8 SPMD MPI View 11
2.9 NavP View 12

3.1 The code transformations in NavP. (a) Sequential. (b) DSC 17
3.2 The code transformations in NavP. (c) Pipelining (d)Phase shifting 18
3.3 Methodology 20
3.4 Matrix Multiplication Sequential Pseudocode 21
3.5 Matrix multiplication 22
3.6 ADI pseudocode 24
3.7 Two data distribution patterns. (a) unstructured data distribution (b) twisted data distribution 25
3.8 ADI Code and Execution Sequence 26
3.9 Computing sequences of different number of tiles 29
3.10 Computing sequences of different number of tiles. (a) Original computation sequence of loop4 (b) Computing sequence of loop4 after tiling 29
3.11 Computing sequences of different number of tiles. (c) Computing sequence of loop4 with sub-tiles 29
3.12 DSC 31
3.13 Pseudocode for DSC 31
3.14 Pseudocode for DSC in the 2nd dimension 33
3.15 DSC in the second dimension 34
3.16 Loop4 pseudocode after assigning locations for tiles 36
3.17 Sub DSC in the second dimension 37
3.18 Weighted Dependence Graph 41
3.19 Simplified Weighted Dependence Graph 43
3.20 Pipelining in one dimensions 45
3.21 Phase shifting in one dimension 46

3.22 Pipelining in both dimensions 47
3.23 Phase shifting in both dimensions 48
3.24 DPC Code 50
3.25 DPC Code Loop8 51
3.26 Pseudocode for pipelined DSC 52
3.27 Pseudocode for DPC in one dimension 52
3.28 Pseudocode for pipelining in second dimensions. 53
3.29 Pseudocode for full DPC in second dimensions. 55
3.30 Different options of tiles for the same data distribution 57
3.31 (a) Speedup for different tile options for the unstructured data distribution (b) Speedup for different tile options for the twisted data distribution 58
3.32 Speedup achieved for the three cases of Fig.10 59
3.33 6 partitions (2 partitions per node) 60
3.34 9 partitions (3 per node) 61
3.35 12 partitions (4 per node) 61

4.1 Speedup with unstructured data distribution 63
4.2 Speedup with twisted data distribution 65
4.3 Pseudocode for Gentlemans Alg. 67
4.4 Performance on 3 PEs 69
4.5 Performance on 8 PEs 69
4.6 Performance on 2*2 PEs 70
4.7 Performance on 3*3 PEs 70

5.1 Navigational Programming Architecture 76

LIST OF TABLES

Page

4.1 Speedup given unstructured data distribution. 62
4.2 Speedup given twisted data distribution A 63
4.3 Speedup given twisted data distribution B 63
4.4 Speedup given twisted data distribution C 64
4.5 Speedup given twisted data distribution D 64

5.1 ADI Performance on Cloud 77

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to Professors Lubomir F. Bic and Michael B. Dillencourt for their encouragement, advices, supports along the way of my pursuing Ph.D research. They create an inspiring, exciting, creative and free research environment in MESSENGERS group. They have been always there to have constructive discussion whenever I have had new ideas or questions. They have been encouraged me to overcome all of the challenges and steadily achieve my goals. And, I feel privileged and honored to have Professor Amelia C. Regan on my dissertation committee. Without them, the dissertation could not been finished, without them, I could not reach this point. I would like to thanks for their supports from bottom of my hearts.

Many thanks to Professor Isaac D. Scherson, Tony Givargis and Michael Dessen who also in my advancement and topic defense committees. I thank each of them for their time and helpful comments.

During my studies at University of California, I have privilege to work with members' MESSENGERS. Specifically thanks to Lei Pan, who discussed, encouraged and gave suggestions and comments, worked with me on few papers. Really enjoyed the time we spent together with MESSENGERS, during weekday and weekend. Appreciate helps from Koji Noguchi and Ming Kin Lai for their discussion regarding running MESSENGERS system and their support of implementation. Also, I feel pleasure to work with Qinghong Shang on implementation and many thanks for her to cooperate together on papers.

CURRICULUM VITAE

Wenhui Zhang

EDUCATION

Doctor of Philosophy in Computer Science **2014**
University of California, Irvine *Irvine, CA*

Master of Computer Science **2006**
University of California *Irvine, CA*

Master of Computer Engineering **1999**
Chinese Academy of Sciences Institute of Computing Technology *Beijing, China*

EXPERIENCE

Senior Staff Software Engineer **2012–2014**
Broadcom Corporation *Irvine, California*

Software/Firmware Engineer **2008–2012**
Newport Electronics Inc.(Omega Engineering Inc.) *Santa Ana, California*

Software Engineer **1998–2002**
China Internet Network and Information Center *Beijing, P.R.China*

PUBLICATIONS

Incremental Parallelization with Migration **2012**

Zhang, Wenhui, Lei Pan, Qinghong Shang, Lubomir F. Bic, and Michael B. Dillencourt, In Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on

Incremental parallelization using Navigational Programming: A case study **2005**

Pan, Lei, Wenhui Zhang, Arthur Asuncion, Ming Kin Lai, Michael B. Dillencourt, and Lubomir F. Bic. , In Parallel Processing, 2005. ICPP 2005. International Conference on

JaMes: A Java-based System for Navigational Programming. **2011**

Qinghong Shang, Munehiro Fukuda, Wendy Zhang, Lubomir Bic, and Michael B. Dillencourt., In Computational Problem-Solving (ICCP), 2011 International Conference on, pp. 444-449. IEEE, 2011

JOURNAL PUBLICATION

Toward incremental parallelization using navigational programming **2006**

Lei Pan , Wenhui Zhang , Arthur Asuncion , Ming Kin Lai , Michael B. Dillencourt , Lubomir F. Bic, IEICE Trans. Inf. and Syst., Vol. E89-D, No. 2, pp. 390-398,

TECHNICAL REPORTS

Incremental Parallelization Using Navigational Programming: A Case Study **2005**

Lei Pan , Wenhui Zhang, Arthur Asuncion, Ming Kin Lai, Michael B. Dillencourt, Lubomir F. Bic, UCI Technical Report: TR-05-04, 2005

ABSTRACT OF THE DISSERTATION

Incremental Parallelization with Migration

By

Wenhui Zhang

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2014

Professor Lubomir F. Bic, Co-Chair
Professor Michael B. Dillencourt, Co-Chair

This dissertation presents a new methodology for developing parallel distributed programs in a series of incremental steps to achieve incremental parallelism and incremental performance improvement. The methodology takes advantage of threads that are able to migrate through the network and thus are able to follow distributed data. This allows the data to be partitioned and distributed first, which guarantees that elements that are used together in a computation are collocated on the same node. Next, the loops in the code are tiled to minimize migration among nodes. After deciding on the location at which each loop is to execute, the necessary migration and remote access statements are inserted to make the code executable. This process is repeated based on feedback obtained from the execution, which may improve the overall performance by suggesting a different data distribution or a different coarseness of tiling. We show each steps and the performance data for two well-known application. Also, we illustrate the trade-offs using a well-known application with two different data distributions.

Chapter 1

Introduction

1.1 Research Motivation and Target Problem

Generating a parallel program from a sequential one for a distributed-memory environment is an important and difficult problem. A common approach is to decompose the original program into smaller computations and then to construct a distributed schedule for the computations [5, 15, 20, 21, 28, 53, 60, 64, 69]. The distributed scheduling task consists of several subtasks: assigning each computation to a processor, chronologically ordering the computations on each processor, and scheduling the data movement so that each computation has the necessary data when it executes. This is difficult under the classical message-passing approach, in which processes are stationary and any remote data required by a process is communicated through send/receive primitives.

The Navigational Programming (NavP) [34] [39] approach to generating a distributed parallel program can be summarized as follows. (1) the data are distributed; (2) the program is divided into computations (tiles) based on the data distribution, and each computation is assigned to a processor (again based on the data distribution); (3) the computations are scheduled in those tiles and combined into parallel threads that migrate through the network based on their data dependencies.

NavP [40] is still considered SPMD (Single Program Multiple Data) in that different processes execute the same code with different data, and differs from the traditional SPMD view [40, 41] in that processes are able to access remote data by migrating to the target node. Hence the locus of any given computation is not stationary but follows the distribution of data as appropriate for best performance.

All three steps are extensions of existing techniques and tools. Data distribution (step 1) uses an affinity graph [43] produced by instrumenting the program and then partitioning the graph using Metis [26] . To break the program into subcomputations (step 2) we rely on the well-known techniques of tiling [23, 24, 52, 67, 68]. The main difference is that the execution provides feedback that guides the choice of the tile size. Scheduling (step 3) is performed only after each computation has been assigned a processor. It is straight forward because it follows the existing data dependencies; it is a form of dataflow scheduling [2, 56].

The NavP-based methodology provides two major advantages for distributed parallel programming:

1. **Incremental parallelism**. The sequential program evolves into the distributed program in a series of incremental steps: the data is distributed, the program follows the data, the program is divided into smaller computational units, the computational units are scheduled. At any point during this evolution there is a viable (executable) program that has the same semantics as the original sequential program. One consequence of this is that it is possible to return to any decision made during the generation of the parallel program, change the decision, and continue the generation process from that point forward.

2. **Incremental performance improvement**. The feedback mechanism provides per-

formance evaluations, including speedup and load balance, that can be used to adjust the output of the preceding steps. In this way, the methodology is a closed-loop system that incrementally improves the performance of the parallel and distributed program.

What enables the incremental parallelization and performance improvement is computation mobility. Our approach uses the principle of *pivot-computes*, which is different from the commonly used *owner computes* [16]. Pivot computes performs the computation on the node that contains the largest portion of the data, regardless of whether it owns it (writes it) or only reads it and then writes it on some other node. This is only possible if computations can migrate to the data. Consequently, the data layout can be decided first, the computations then follow the data distribution. This simplifies the programming task because it decouples the two main problems: data distribution/placement and code parallelization.

Another consequence of distributing data first is better performance, since migrating computation to data is frequently more efficient (only the state moves, not the code) than moving the data to computation.

This thesis presents a new methodology for developing parallel distributed programs in a series of incremental steps to achieve incremental parallelism and incremental performance improvement. The methodology takes advantage of threads that are able to migrate through the network and thus are able to follow distributed data. This allows the data to be partitioned and distributed first, which guarantees that elements that are used together in a computation are collocated on the same node. Next, the loops in the code are tiled to minimize migration among nodes. After deciding on the location at which each loop is to execute, the necessary migration and remote access statements are inserted to make the code executable. This process is repeated based on feedback obtained from the execution, which may improve the overall performance by suggesting a different data distribution or a different coarseness of tiling.

1.2 Dissertation Overview and Organization

Navigational Programming (NavP) offers a different approach to generating a distributed parallel program from a sequential one, in that the transformation occurs incrementally and produces an executable program at each stage. Under NavP, computations migrate using hop() statements inserted explicitly by the programmer. The cost of a hop() is essentially the cost of moving the data stored in its thread variables plus a small amount of state data. Although the state of the computation is moved on each hop, the code is not moved. The computations carry small amounts of data, such as intermediate results, as they migrate to large data structures that are stationary. The synchronization among different migrating computations is achieved by waiting on and posting of events.

In this dissertation, we describe the methodology of Incremental Parallelization with Migration. In Chapter 2, We introduce the navigational programming. Chapter 3 presents the methodology of Incremental Parallelization with Migration, and describes its steps using two examples [71] [44]. Chapter 4 shows the performance data for two examples, also compares the performance of NavP and MPI implementations [71] [44]. Chapter 5 gives a proof-of-concept extension of NavP on cloud with preliminary results and future improvement. Chapter 6 discusses related works. Finally, we conclude in Chapter 7 with a summary of the contributions of this dissertation and suggestions for further research.

Chapter 2

Navigational Programming

2.1 Overview

Navigational Programming (NavP) [34] [39] [30] is a methodology for distributed parallel programming based on the use of self-migrating computations. In NavP code, a programmer inserts navigational commands, i.e., hop() statements, to migrate the computation locus in order to access remotely distributed data and spread out computations. Small data is carried by the moving computation in agent variables, which are private to a computation thread and available to the thread wherever it migrates. Large data that stays on a computer is held in node variables that are resident on a particular PE (processing element) and are shared by all computation threads currently on that PE. The cost of a hop() is essentially the cost of moving the data stored in agent variables plus a small amount of state data. Although the state of the computation is moved on each hop, the code is not moved. The synchronization among different migrating computations is done through events (signalEvent() and waitEvent()). A programmer can inject, or spawn, a migrating thread at command line. The injection of a thread can also be done by another thread, called a spawner. All injections happen locally (i.e., a thread can spawn another thread only on the node on which it currently resides).There are two kinds of variables in NavP programming, one is called node variable, which is usually for large amount of data and stay in one node. It could be used for synchronizing between different threads/processes in the same node. The other is thread-carried variable which is useful for migrating with the threads.

NavP provides a different view of distributed computation from the classical SPMD (Single Program Multiple Data) view [40, 41]. The SPMD view describes distributed computations at stationary locations, while the NavP view describes a computation following the movement of its locus. The NavP view changes the way distributed parallel programs are composed and provides some new benefits.

The good example to show the differences of those two views is the train example [39]. A train goes across the cities and arrives and departures at certain stations at certain time. The train schedulers and the taxi drivers are interested in when a train arrives at what time at what station, whereas, the travelers are interested in ternary which shows the sequences of time and station that my train will arrive in order. Fig. 2.1 shows the traces of four travelers. So Fig. 2.2 [39] shows the view of schedulers and taxi drivers, which is SPMD MPI view, and Fig. 2.3 shows the view of travelers, which is NavP view. Those two views are all useful and for different purposes. NavP view is for programmers to navigate computation across distributed systems.

We use a code example [39] which only has three lines of sequence code to show the differences of the two different views (SPMD via NavP). The sequential algorithm is listed in Fig. 2.4. A and B are blocks with order n, and v1, v2 and v3 are vectors with size n. Line (1) gets the diagonal entries of matrix A and assigns them to the intermediate vector v1. Line (2) multiplies the matrix B by the intermediate vector v1, and assigns the intermediate result to the vector v2. At the end, ine (3) computes the matrix A by the intermediate vector v2 and assigns the result to the final vector v3.

We assume A and B are too large and cannot be hold in one physical node. Fig. 2.5 list the

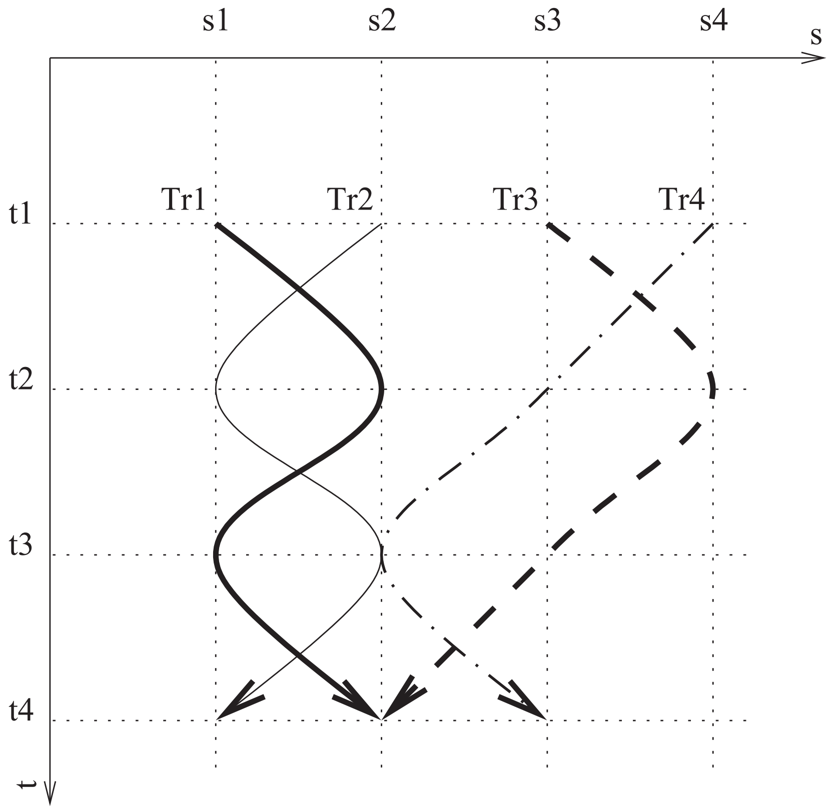


Figure 2.1: Four Traveler's Traces

	s1	s2	s3	s4
t1	Tr1	Tr2	Tr3	Tr4
t2	Tr2	Tr1	Tr4	Tr3
t3	Tr1	Tr2,Tr4	Tr3	
t4	Tr2	Tr1,Tr3	Tr4	

Figure 2.2: SPMD MPI View:Arrivals Departures at stations

	Tr1	Tr2	Tr3	Tr4
t1	s1	s2	s3	s4
t2	s2	s1	s4	s3
t3	s1	s2	s3	s2
t4	s2	s1	s2	s3

Figure 2.3: NavP View:Itineraries of trains

```
(1) v1 = diag (A)
(2) v2 = Bv1
(3) v3 = Av2
```

Figure 2.4: Example: Three lines of code computing on distributed data (a) Sequential

pseudocode for MPI, which needs to distinguish whether the node is PE 0 or PE1. If in PE0, it needs first getting A and sending to PE0, then, waiting for B from PE1, finally computing the final resultl; if in PE1, first it needs waiting from A from PE0, computing and sending intermediate result to PE0. Fig. 2.6 list the pseudocode for NavP, which is computational view, just needs to follow the execution flow going across the nodes.

Fig. 2.7 [39] shows the dependency graph in distributed system. Fig. 2.8 [39] shows the SPMD MPI view to distinguish different nodes and Fig. 2.9 [39] shows NavP view which just follow the execution flows going across distributed nodes.

A NavP application consists of a dynamically created, set of autonomous threads which can migrate through the network and communicated with each other in various ways.

```
(0.1) if (rank == PE0)
(1) v1 = diag (A)
(1.1) Send(v1, PE1)
(1.2) Recv(v2, PE1)
(3) v3 = Av2
(3.1) else if (rank == PE1)
(3.2) Recv(v1, PE0)
(2) v2 = Bv1
(2.1) Send(v2, PE0)
(2.2) end if
```

Figure 2.5: Example: Three lines of code computing on distributed data (b) MP

```
(1) mv1 = diag (A)
(1.1) hop(PE1)
(2) mv2 = B mv1
(2.1) hop(PE0)
(3) v3 = A mv2
```

Figure 2.6: Example: Three lines of code computing on distributed data (c) NavP

2.2 Characteristics

A preliminary version of the NavP model was presented in [29, 58], and NavP is keeping improved since then [34] and extends with Java version [57]. The following list the key characteristics of NavP:

- Self-migration: A process is able to pack up its state, recreate itself on another node within the logical network, and to continue executing at the remote site. If the original process then dies after recreating itself, this operation is called a hop operation. If the caller continues, this is called a clone operation.
- Priority-based migration: Certain applications require that processes arriving at a particular node from different locations are processed in a certain predefined order. To make this more efficient, we allow processes to specify a priority as part of their migration statement. The receiving node maintains all arriving processes in a priority

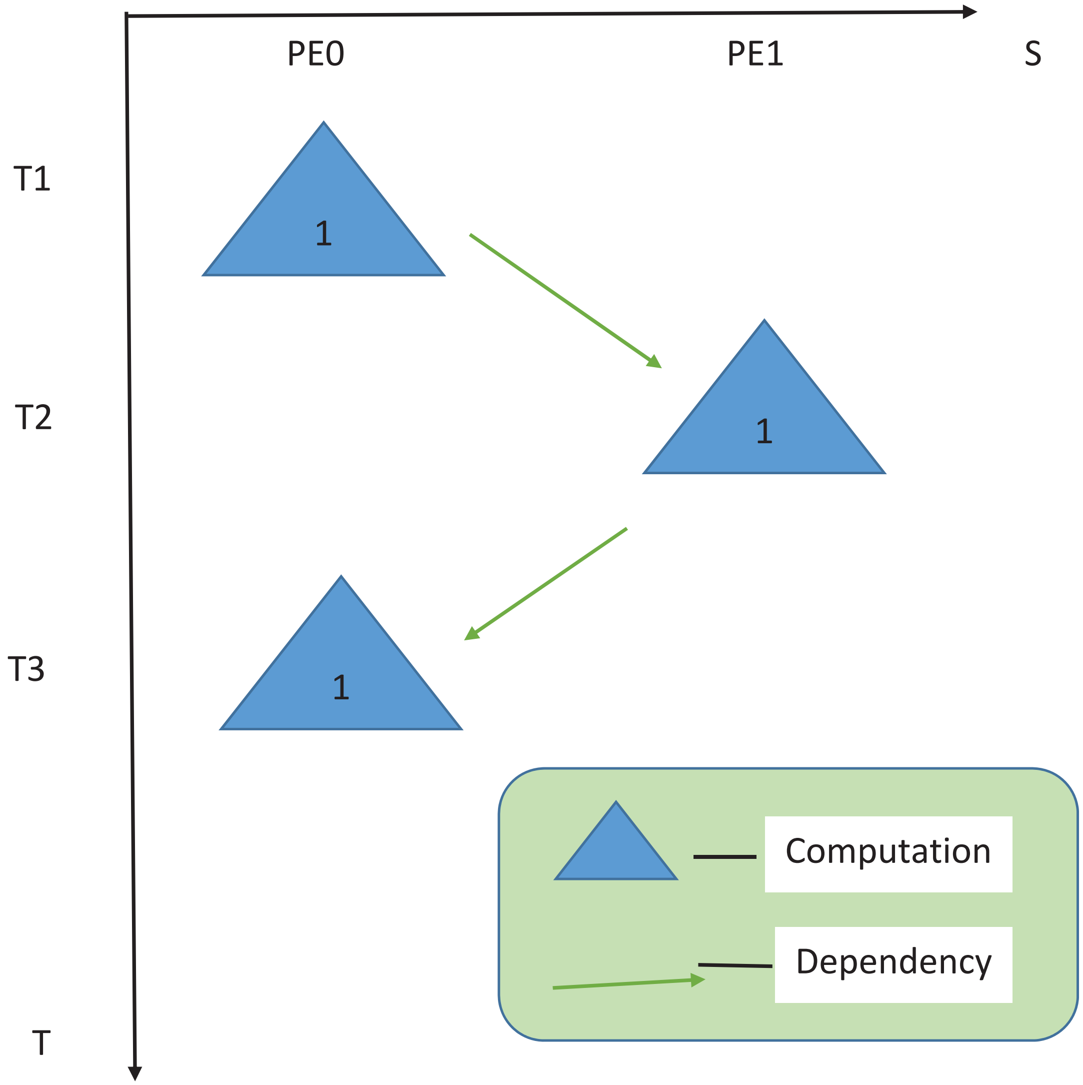


Figure 2.7: Dependency Graph

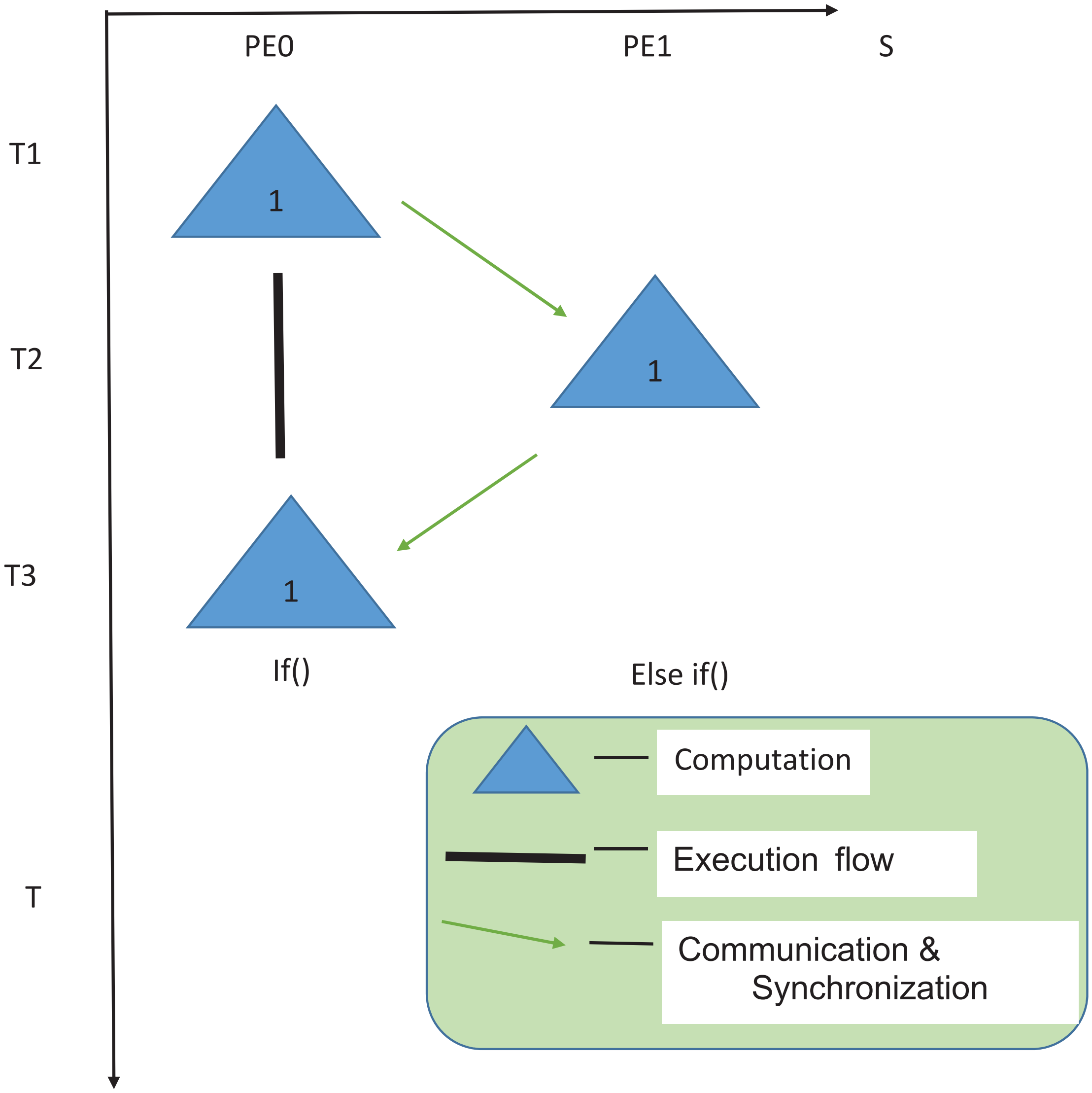


Figure 2.8: SPMD MPI View

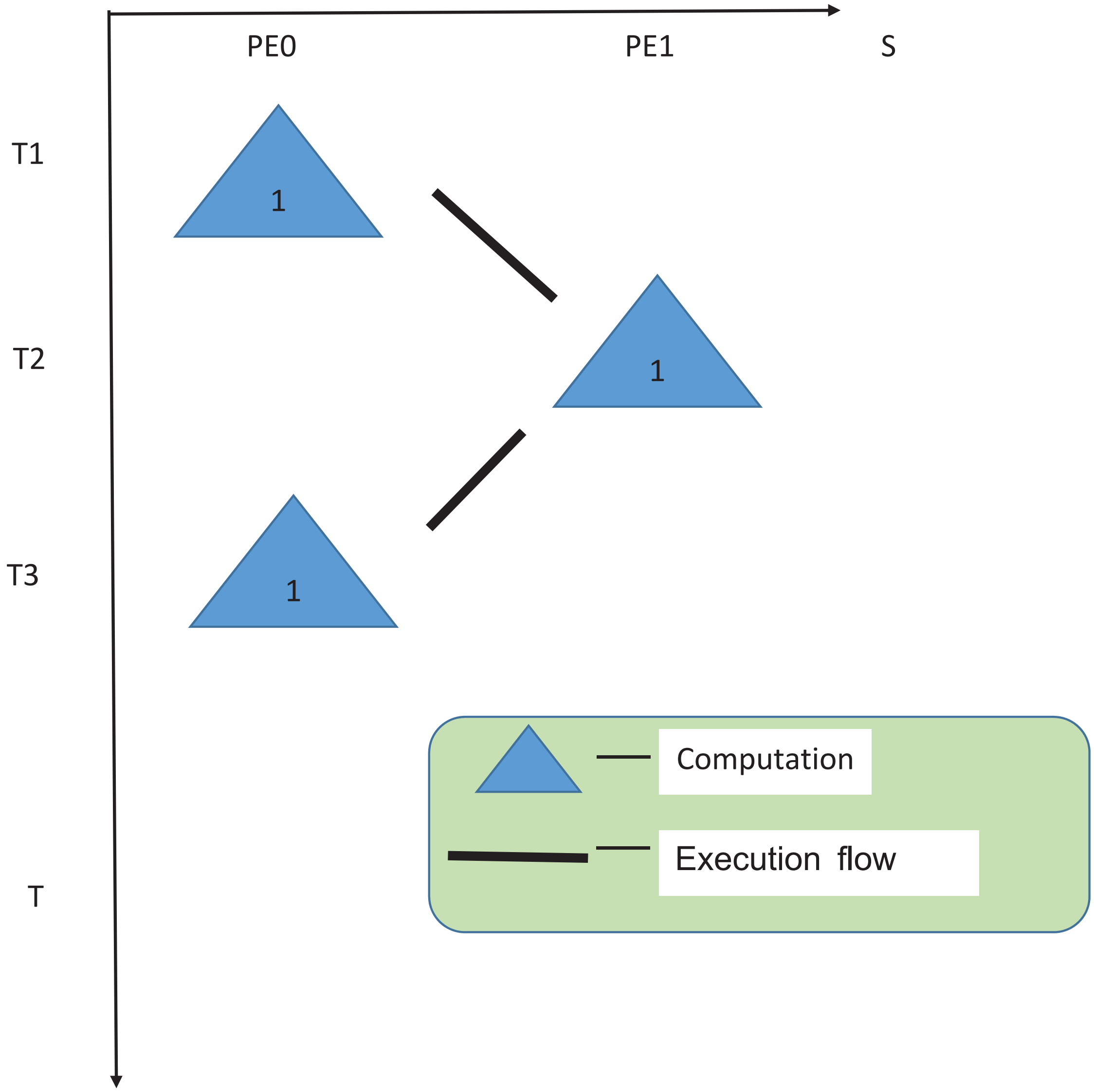


Figure 2.9: NavP View

queue ordered by process priority. When the currently-running process gives up the CPU, it is placed appropriately in the ready queue according to its priority, and the highest-priority process is selected to run next.

- Stationary and mobile data: There are two kinds of variables in NavP programs. Mobile data is private to each process and its carried with it as the process migrates among the machines. Stationary data is bound to a specific logical node and is accessible to processes currently residing on that logical node.

- Process interaction: Processes can interact with each other only when they reside on the same logical node. Data communicating is achieved through shared node variables. Process synchronization is achieved through events and wait/signal operations.

- Non-preemptive scheduling: NavP assumes a nonpreemptive scheduling discipline, which simplifies issues of synchronization. On each node, there is only one execution thread running at any time. An execution thread cannot be preempted. The only way it can be blocked is when it blocks itself by issuing specific commands such as wait(), yield(), or a blocking receive request. Nonpreemptive scheduling eliminates the need for explicit critical sections when accessing shared variables and reduces context-switching overhead.

- Synchronization: The extended NavP model provides events and the standard primitives of wait and signal for synchronizing threads on the same logical node. Applications can create any number of distinct events. If an event has not been signaled, a wait operation on that event causes the thread to become blocked. When the event is signaled, the process becomes unblocked and becomes eligible for scheduling. An event can only be signaled once. Any process that waits on an event that has already been signaled is not blocked. A process can wait on multiple events, specified as an array of events, all of which must be signaled before the process can continue. A process can also wait on a disjunction of events, in which case it will be awakened when at least one of the events has occurred and a return value will tell it one of the events that has occurred.

- Remote process interaction: In the basic NavP model, the only mechanism for interprocess communication is the shared node variable area. This means that for two processes to communicate, they must reside on the same logical node. This has the advantage of simplicity, but it may also introducing unnecessary migrations to access remote data. The extended NavP model solves this problem by introducing a form of remote communication between threads on different machines. This can take two different forms, both of which allow a process to deposit data into a named mailbox without migrating to the target. The difference is on the receiving side. In the first form, which corresponds to a conventional send/receive, there is a corresponding receive command in which the receiver specifies how data sent to that mailbox is to be handled. In the second form, which is a remote write operation, the receiver specifies once how data sent to this mailbox is to be handled. Each time data is sent to the mailbox, the specified handler is automatically invoked.

Chapter 3

Incremental Parallelization

3.1 Methodology Overview

Navigational Programming (NavP) [34] [39] offers a different approach to generating a distributed parallel program from a sequential one, in that the transformation occurs incrementally and produces an executable program at each stage. Under NavP, computations migrate u[34] [39] sing hop() statements inserted explicitly by the programmer. The cost of a hop() is essentially the cost of moving the data stored in its thread variables plus a small amount of state data. Although the state of the computation is moved on each hop, the code is not moved. The computations carry small amounts of data, such as intermediate results, as they migrate to large data structures that are stationary. The synchronization among different migrating computations is achieved by waiting on and posting of events.

The NavP approach to generating a distributed parallel program can be summarized as follows. (1) the data is distributed; (2) the program is divided into computations (tiles) based on the data distribution, and each computation is assigned to a processor (again based on the data distribution); (3) the computations are scheduled in those tiles and combined into parallel threads that migrate through the network based on their data dependencies.

The NavP methodology consists of the following steps.

The transformations under the NavP view are depicted in Fig. 3.1 and Fig. 3.2. The arrows indicate hop() operations. The basic idea behind the transformations is to spread out computations using self-migrating computation threads as soon as possible without violating any dependency conditions.

The basic idea of distributed sequential computing(DSC)[42] Transformation is that Large data is distributed among the PEs, and hop() statements are inserted into the sequential code in order for the computation to chase large data while carrying small data. The DSC Transformation is schematically depicted by Fig. 3.1(a) and (b). The resulting program performs distributed sequential computing. The immediate benefit of DSC is that, with a small amount of work, a sequential program can efficiently solve large problems that cannot fit in the main memory of one computer. By using a network of workstations, the DSC program removes paging overhead by trading it against a modest amount of network communication [29]. DSC also serves as the starting point of parallel program development in NavP.

Distributed and parallel computing (DPC) could have more than one forms. for example: Pipelining form: This transformation is depicted by (b) and (c). The basic idea is to overlap the execution of multiple DSC threads by staggering their starting times. Synchronization may be necessary to ensure that the data dependencies among the DSC threads are not violated. Or DPC could be Phase-shifting form: Sometimes the dependency among different computations allows different DSC threads to enter the pipeline from different PEs. In these situations, we can phase shift the DSC threads to achieve full parallelism, as depicted in (c) and (d).

In summary, the NavP methodology consists of the following steps, illustrated in Fig. 3.3 with an example.

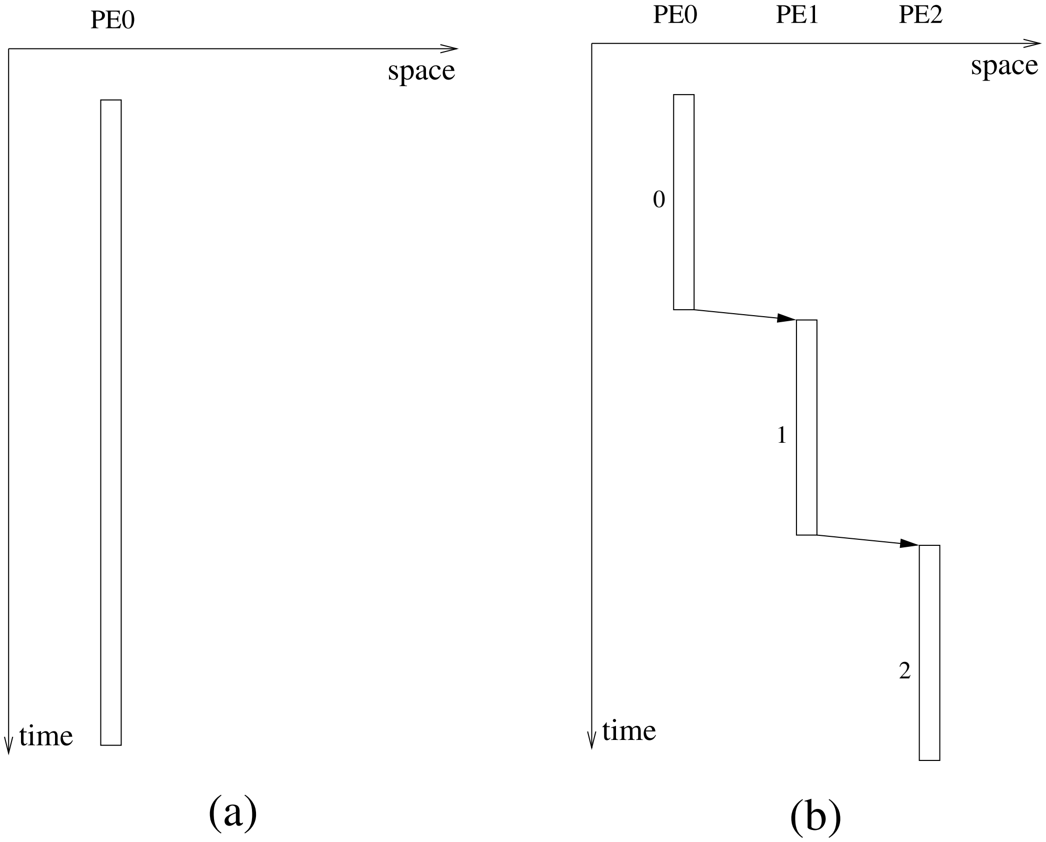


Figure 3.1: The code transformations in NavP. (a) Sequential. (b) DSC

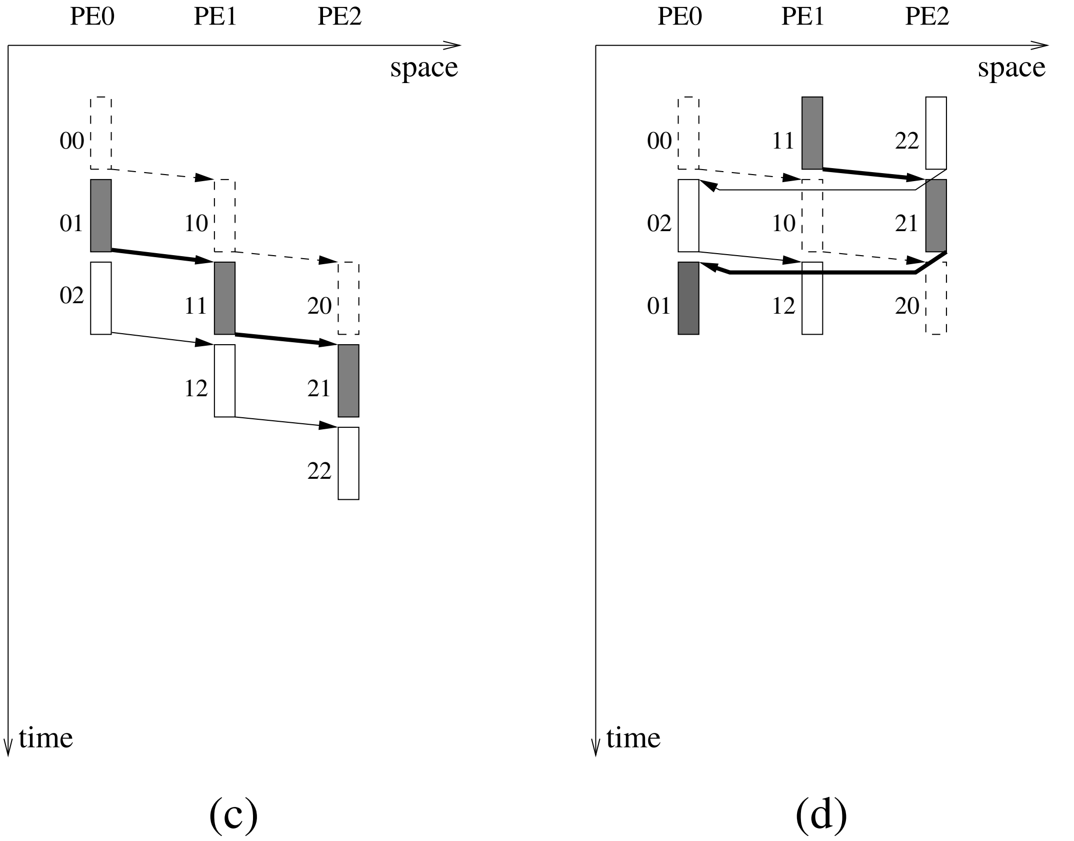


Figure 3.2: The code transformations in NavP. (c) Pipelining (d)Phase shifting

1. Data distribution: This step runs the sequential program using a small data sample and generates the initial data partitioning and distribution.

2. Code Transformation: This step distributes the computation. It partitions the original sequential program into smaller units, based on the given data distribution, it assigns each unit to execute on one of the nodes, and it inserts appropriate navigational statements into the code. The resulting code is still sequential but it runs in a distributed manner using migration. This is referred to as distributed sequential computing (DSC).

3. Parallelization: This converts the single-threaded DSC into multiple threads. Each thread combines a number of units created in the previous step and adds signals to synchronize between different threads. Each of the threads is scheduled to run as early as possible, subject to the constraint that all dependencies must be respected. The result of this step is to generate distributed and parallel computing (DPC).

4. Feedback: This evaluates the performance of the DPC program in terms of speedup and load balance, by taking into account parallelism, communication cost, computation cost, and other overhead such as context switch. The purpose of this step is to provide information for the previous steps to improve performance.

After Data distribution, the data is distributed into different nodes represented by different shapes. After the Code Transformation, distributed sequential computing (DSC) is generated, it is still a single and sequential DSC thread but runs in a distributed manner using migration.after Parallelization, distributed and parallel code (DPC) is generated, it splits the single DSC thread into three threads and schedules those three threads as early as possible, which means that the resulting code is multi-threaded code and each of the threads is scheduled to run as early as possible, subject to the constraint that all dependencies must be respected. So, in this example, these three threads starts at the same time at different nodes consistent with the constrains of data dependencies.

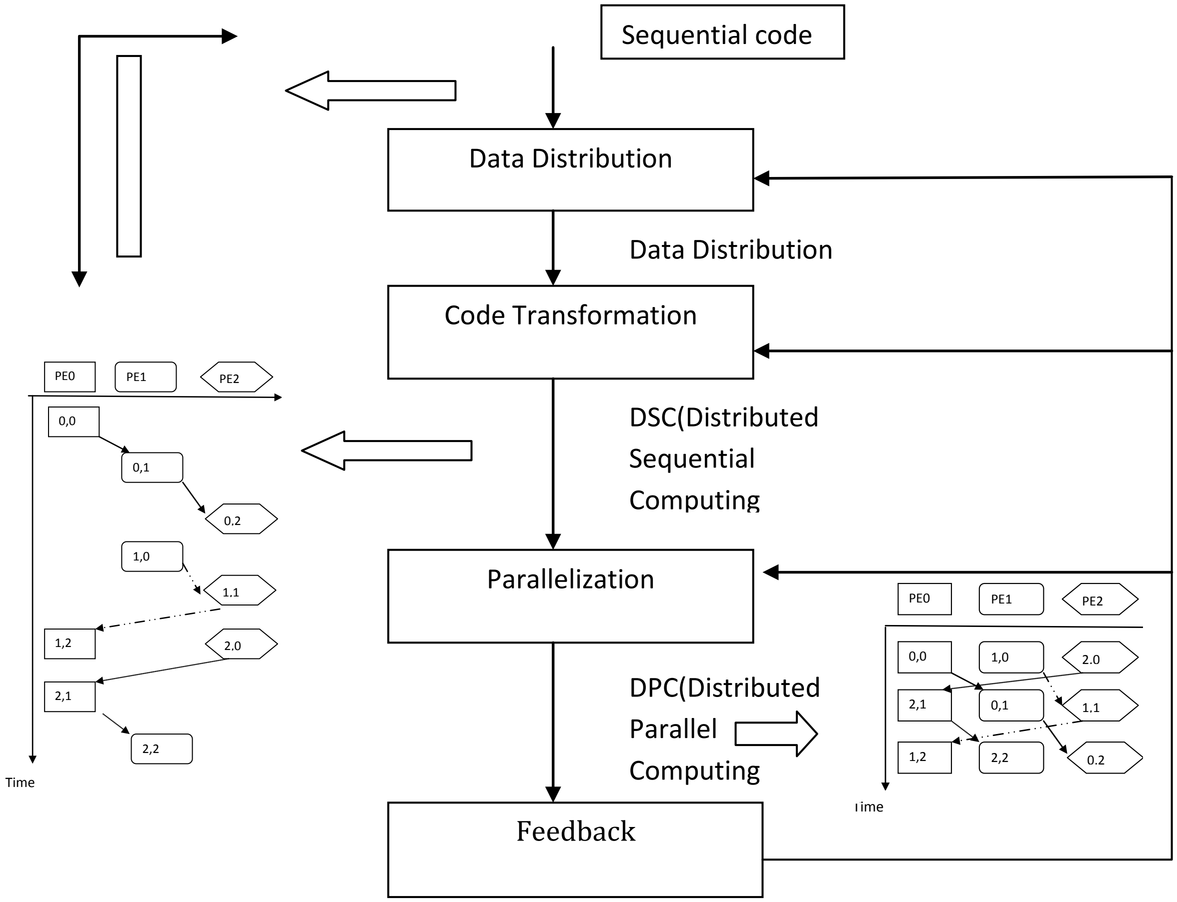


Figure 3.3: Methodology

```
do i=0,N-1
    do j=0,N-1
        t = 0.0
        do k=0,N-1
            t += A(i,k) * B(k,j)
        end do
        C(i,j) = t
    end do
end do
```

Figure 3.4: Matrix Multiplication Sequential Pseudocode

3.2 Two Examples

3.2.1 An Example of Matrix Multiplication

Matrix multiplication is a fundamental operation of many numerical algorithms. Pseudocode for sequential matrix multiplication is listed in Fig. 3.4. Throughout the paper, we assume N is the order of the square matrices.

Matrix A is divided by row blocks and matrix B is divided by column blocks (as shown in Fig. 3.5).

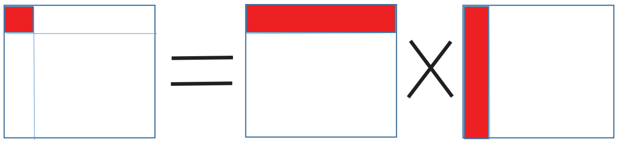


Figure 3.5: Matrix multiplication

3.2.2 An Example of ADI

We will describe our methodology using the code shown in Fig. 3.4 as an example [31], which is an abstraction of a method for solving a 2-dimensional heat equation originally presented in [31]. The example is very good application which shows extensive both data communication and data computation, also strong data dependencies.

It solves a 2D heat equation using the alternating direction implicit (ADI) method, which is a finite difference method for solving parabolic, hyperbolic and elliptic partial differential equations in numerical analysis [1]. It used the PeacemanRachford algorithm to formulate the numerical solution of the partial differential equation as a second-order approximation by solving two sets of tridiagonal systems of linear equations. The variables of the first set of tridiagonal systems correspond to elements from each column of an intermediate matrix, and the variables of the second set of traditional systems correspond to elements from each row of a target matrix [62]. Using the Thomas algorithm, It reduce a traditional system of linear equations to three sets of first-order recurrence equations .

The example is a variant of the alternating direction implicit (ADI) method, which makes multiple sweeps in different directions across an array representing the discrete domain. The example shows four 2D arrays involved in the computation (u, v, p, and q). It also shows the computation kernel which consists of an outer loop (line 1) that repeatedly performs a column sweep phase (lines (2)-(18)) followed by a row sweep phase (lines (19)-(35)). The row sweep phase and the column sweep phase consist of four loops each, for a total of 8 inner loops performed for each iteration of the outer loop. In Fig. 3.8, it shows the execution sequences of the major four loops along with the pseudocode. In loop2, the inner loop computes from left to right and outer loop sweep from top to bottom. In loop4, the inner loop computes from bottom to up and outer loop sweep from left to right. In loop6, the inner loop computes from left to right and outer loop sweep from top to bottom. In loop8, the inner loop computes from right to left and outer loop sweep from top to bottom.

The methodology discussed in this paper takes a specified data distribution as a starting point. We use the two data distributions illustrated in Fig. 3.7 to show the methodology and compare the respective performance. The data distributions are shown for 3 nodes, each represented by a different shade. The first was generated by our Data Distributor [43]; while the second is the well-known twisted data layout [59]. In each distribution, all four 2D arrays (u, v, p, and q) are distributed as shown in the corresponding figure.

```
Perform NT iterations
    for (t=1; t<= ITERATIONS;t++){
      //Column sweep
       for (i=1; i<=N ; i++){
               q[i][0]=v[0][i];
           }
       for (i=1; i<=N ;i++){
           for( j=1; j<=N ;j++){
              p[i][j]=F1(p[i][j-1]);
              q[i][j]=F2(u[j][i-1],u[j][i],
                      u[j][i+1],q[i][j-1],
                      p[i][j-1]);
           }
       }
       for (i=1; i<=N; i++){
           v[NX+1][i]= t+NX+1+i+DT+DX+DY;
           }
       for (j=1; j<=N ;j++){
           for (i=N ;i>=1;i--){
              v[i][j]=F3(p[j][i],v[i+1][j],
                      q[j][i]);
           }
       }
   // Row sweep
       for (i=1; i<=N; i++){
         q[i][0]=u[i][0];
       }
       for (i=1; i<=N; i++){
         for (j=1; j<=N ;j++){
          p[i][j]= F4(p[i][j-1]);
          q[i][j]=F5(v[i-1][j],v[i][j],
                  v[i+1][j],q[i][j-1],
                  p[i][j-1]);
         }
       }
       for (i=1; i<=N; i++){
         u[i][NY+1]= t*DT+i*DX+1.0;
       }
       for (i=1; i<=N; i++){
         for (j=N ;j>=1;j--){
          u[i][j]=F6(p[i][j], u[i][j+1],
                  q[i][j]);
         }
       }
    }
```

Figure 3.6: ADI pseudocode

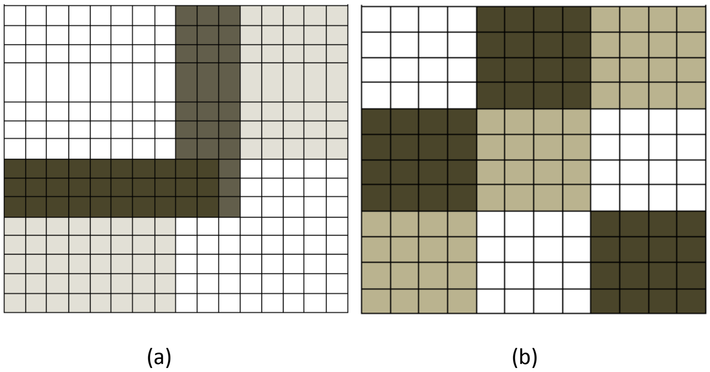
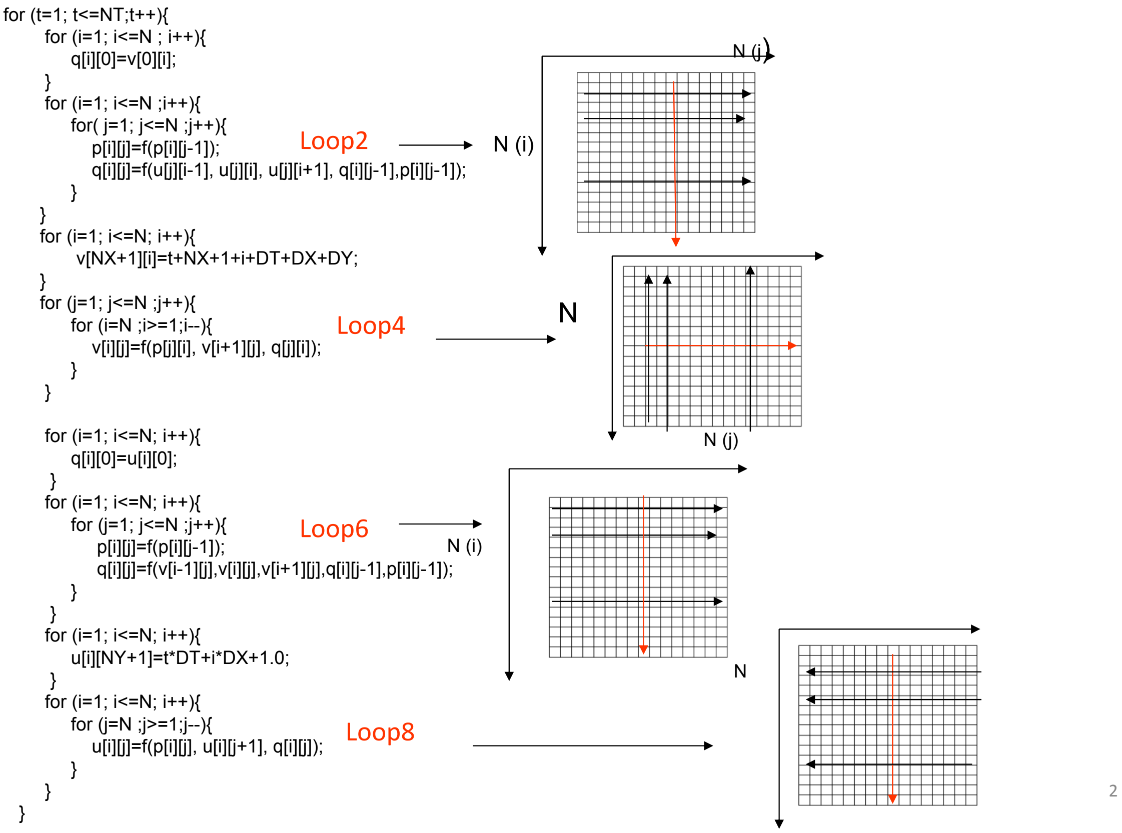


Figure 3.7: Two data distribution patterns. (a) unstructured data distribution (b) twisted data distribution



```
for (t=1; t<=NT;t++){
      for (i=1; i<=N ; i++){
          q[i][0]=v[0][i];
      }
      for (i=1; i<=N ;i++){
          for( j=1; j<=N ;j++){
             p[i][j]=f(p[i][j-1]);
             q[i][j]=f(u[j][i-1], u[j][i], u[j][i+1], q[i][j-1],p[i][j-1]);
          }
     }
     for (i=1; i<=N; i++){
           v[NX+1][i]=t+NX+1+i+DT+DX+DY;
     }
     for (j=1; j<=N ;j++){
          for (i=N ;i>=1;i--){
             v[i][j]=f(p[j][i], v[i+1][j], q[j][i]);
          }
      }

      for (i=1; i<=N; i++){
          q[i][0]=u[i][0];
       }
      for (i=1; i<=N; i++){
          for (j=1; j<=N ;j++){
              p[i][j]=f(p[i][j-1]);
              q[i][j]=f(v[i-1][j],v[i][j],v[i+1][j],q[i][j-1],p[i][j-1]);
          }
       }
      for (i=1; i<=N; i++){
          u[i][NY+1]=t*DT+i*DX+1.0;
       }
      for (i=1; i<=N; i++){
          for (j=N ;j>=1;j--){
             u[i][j]=f(p[i][j], u[i][j+1], q[i][j]);
          }
      }
  }
```

Figure 3.8: ADI Code and Execution Sequence

3.3 Code Transformation and Placement

Starting from the sequential code and an initial data distribution, we first build the DSC that navigates through the network and accesses the distributed data sequentially. The simplest way of constructing a working DSC program would be to insert conditional hop statements in the sequential program before each data access. The program resulting from this simple strategy would be correct, but it could be quite inefficient due to the large number of hops. To reduce the number of hops we first partition the iteration space into smaller tiles, which will be executed locally. As discussed below, we use tiling to reduce communication and increase the opportunity for parallelism. After the tiling operation is complete, we assign a location to each tile.

3.3.1 Partitioning phase 1: Adapt to the data distribution

Tiling proceeds in two phases. The goal of the first phase, which is performed once, is to minimize communication overhead. The second phase, discussed in the next subsection, may be repeated multiple times and is intended to increase opportunities for parallelism.

We define the **write set** and **read set** of a tile to be, respectively, the set of all data written by and read by the tile. The partitioning in this first phase is then based on the following principles:

1. **Homogenous write sets**: All the writes to memory from any given tile are to data on a single machine.

2. **Isothetic cuts**: The write sets of the tiles are all rectangles and the boundaries between write sets of tiles all fall on a common set of vertical and horizontal lines.

3. **Minimized communication**: The tiles should be as large as possible, consistent with

the first two conditions.

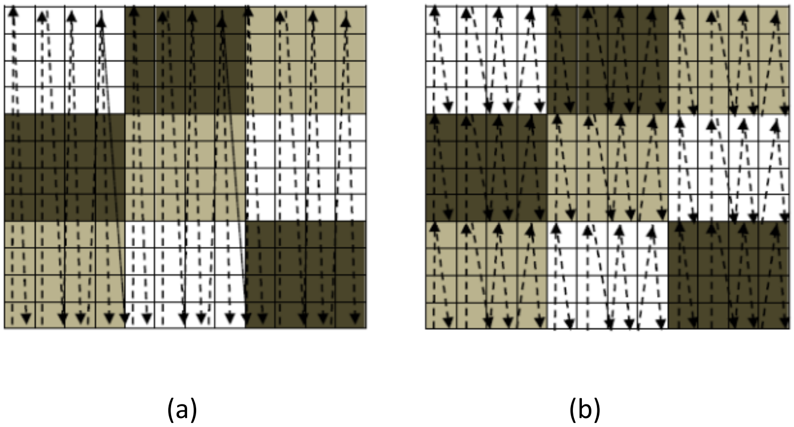
The read sets are not necessarily homogenous. Condition 2 implies that all write sets are rectangles, but the rectangles do not necessarily have the same size. This can be seen in Fig. 3.30(a), which shows the result of applying phase 1 partitioning to the data distribution of Fig. 3.7(a).

3.3.2 Partitioning phase 1: Example of ADI

The tile size can be changed without changing the codes. Fig. 3.10 Fig. 3.11 illustrates the first phase of tiling for a 3×3 twisted layout. Considering loop 4 (lines 14-18 of Fig. 3.6), if this loop is transformed to DSC simply by inserting hop statements, three hops would be required per column for a total of 3N hops (Fig. 3.10(a)). However, if the code is transformed as shown in Fig. 3.9, then only 9 hops are necessary for the entire loop as can be seen in Fig. 3.10(b).

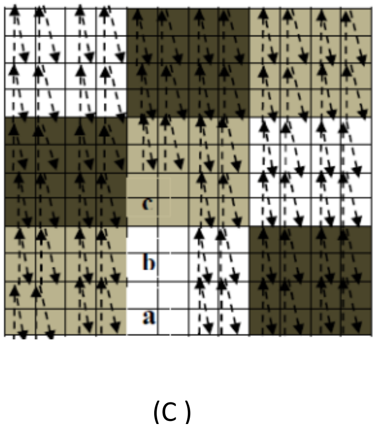
```
for (J=0; J < num_blk_J; J++)
    for (I = (num_blk_I-1); I >= 0; I--)
        for (j = j_start (I,J);
             j <= j_end(I,J); j++)
             for (i = i_end (I,J);
                  i >= i_start (I,J); i--)
                  v[i][j] = F3(p[j][i],
                            v[i+1][j],q[j][i]);
```

Figure 3.9: Computing sequences of different number of tiles


(a)

(b)

Figure 3.10: Computing sequences of different number of tiles. (a) Original computation sequence of loop4 (b) Computing sequence of loop4 after tiling



(C)

Figure 3.11: Computing sequences of different number of tiles. (c) Computing sequence of loop4 with sub-tiles

3.3.3 Partitioning phase 1: Example of Matrix Multiplication in one dimension

We first apply transformation to sequential matrix multiplication, as depicted in Fig. 3.12 where we assume N = 3. This DSC transformation essentially distributes the computation in the j dimension. The PE network is 1D in which each PE has a unique identifier HnodeID = 0, 1, ..., N - 1 from west to east. Again, the arrows represent hop() operations. Thick boxes contain node variables on different machines, and thin boxes carry agent variables. All PEs are assumed to be fully connected via a collision-free switch, rather than being connected as a ring. This assumption is true for most modern hardware environments, and it makes the initial staggering (i.e., moving the entries of the three matrices to the right places before any computation begins) faster, because each matrix entry can be shipped to any destination directly instead of having to go stepwise through a number of intermediate PEs.

Pseudocode for DSC matrix multiplication is listed in Fig. 3.12. In the pseudocode hereafter, A and B indicate node variables, whereas mA and mB represent agent variables. 1 Matrix A is loaded into agent variable mA and carried by the migrating thread.

In Fig. 3.13, matrix A is initially put on the PE with HnodeID = 0, and the columns of matrices B and C are distributed such that B(., j) and C(., j) are on the PE with HnodeID = j. In Fig. 3.13, node(j) maps to the PE that hosts column j of matrices B and C. Every time the computation thread hops back to node(0), it will pick up a different row of matrix A for the computation of the loop over j.

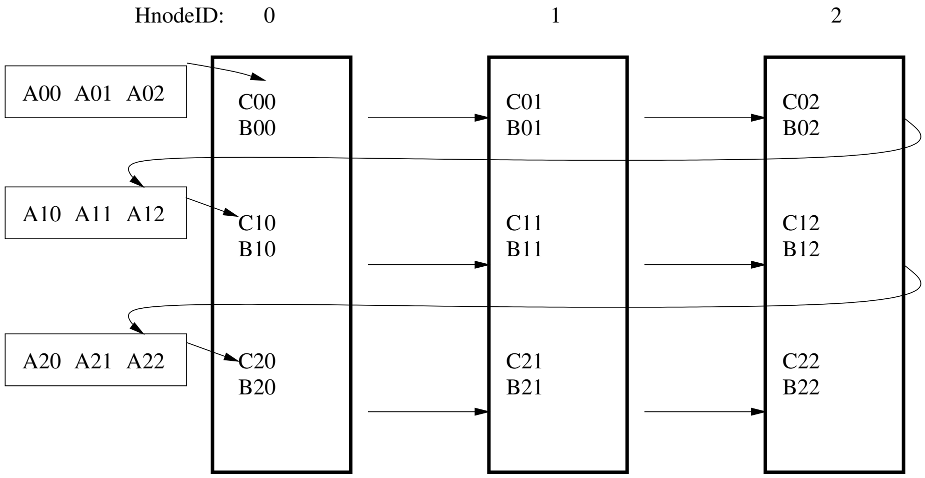


Figure 3.12: DSC

```
hop(node(0))
inject(RowCarrier)
RowCarrier
do mi=0,N-1
    do mj=0,N-1
    hop(node(mj))
    if(mj=0) mA(*) = A(mi,*)
    t = 0.0
        do k=0,N-1
        t += mA(k) * B(k)
        end do
        C(mi) = t
    end do
end do
end
```

Figure 3.13: Pseudocode for DSC

3.3.4 Partitioning phase 1: Example of Matrix Multiplication in second dimension

In second dimension, the first step is to introduce a 2D network in which each PE has a unique 2D identifier (HnodeID, VnodeID), where HnodeID = 0, 1, ..., N - 1 from west to east, and VnodeID = 0, 1, ..., N - 1 from north to south. Then the DSC Transformation is applied in the second dimension, as depicted in Fig. 3.15. Essentially, this DSC transformation further distributes the computations in the i dimension. Pseudocode for DSC in the second dimension is listed in Fig. 3.14. The rows of matrix A and columns of matrix B are carried in their corresponding agent variables mA and mB, respectively. The ColCarriers ship the B columns, and the RowCarriers use these B columns to compute with the A rows that they carry. The events are necessary because the consumers, i.e., the RowCarriers, need to hold on their computations until the producers, i.e., the ColCarriers, finish putting the columns of B in place. The matrices are initially distributed, as shown in Fig. 3.15 , such that A(N . 1 . l, .) and B(., l) are on node(N . 1 . l, l), and C(i, j) (initialized to 0) is on node(i, j), where node(i, j) maps to the PE that hosts entry (i, j) of matrix C.

```
do ml=0,N-1
    hop(node(N-1-ml,ml))
    inject(RowCarrier(N-1-ml))
    inject(ColCarrier(ml))
end do
RowCarrier(int mi)
mA(*) = A(*)
do mj=0,N-1
    hop(node(mi,(N-1-mi+mj)%N)
    waitEvent(EP(mi,(N-1-mi+mj)%N))
    do k=0,N-1
        C += mA(k) * B(k)
    end do
end do
end
ColCarrier(int mj)
mB(*) = B(*)
do mi=0,N-1
    hop(node((N-1-mj+mi)%N,mj))
    B(*) = mB(*)
    signalEvent(EP((N-1-mj+mi)%N,mj))
end do
end
```

Figure 3.14: Pseudocode for DSC in the 2nd dimension

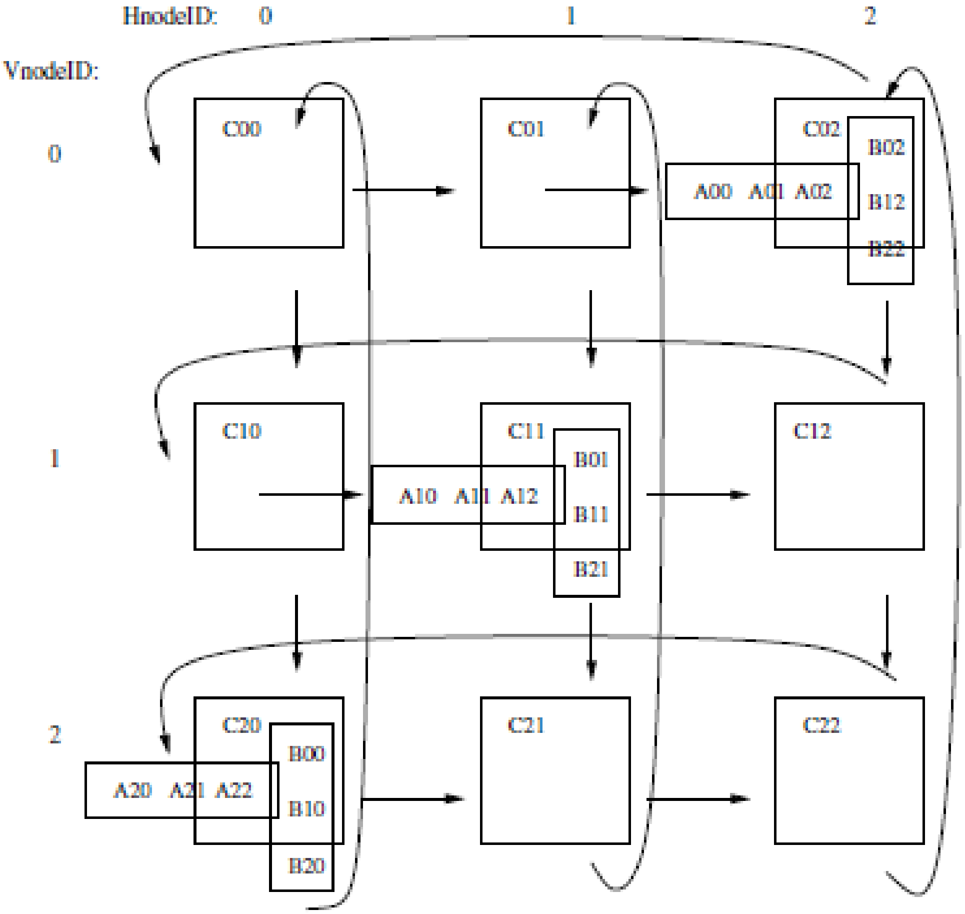


Figure 3.15: DSC in the second dimension

3.3.5 Partitioning phase 2: Increase opportunities for parallelism

The result of the first partitioning phase is a tiling with low communication cost. Splitting the tiles further will have two conflicting effects: it will increase the number of hops, but it will increase the potential for parallelism. The first effect will result in increased communication cost and hence decreased speedup, while the second effect will increase speedup.

3.3.6 Partitioning phase 2: Example of ADI

As an example, Fig. 3.10(c) shows the result of splitting each tile into 4 subtiles. This transformation increases the number of hops from 9 to 36. It increases the opportunity for parallelism because, for example, the tile with write set c can begin executing as soon as the tiles with write sets a and b have completed. The tradeoffs between the increased communication cost and the increased opportunity for parallelism, and the optimal level of tile splitting that should occur, are difficult to evaluate a priori, as they depend on the tile size and on the order in which tiles are evaluated. These considerations suggest the following strategy: start with the largest possible tile; split the tiles; estimate the resulting speedup; and repeat these steps for as long as the estimated speedup continues to increase. The resulting feedback loop is discussed in Section VII. The result of this further partitioning is code that is identical to the code in Fig. 3.9. The only difference is that the parameters describing the number and extent of the blocks have changed.

```
for (J=0; J < num_blk_J; J++){
 for (I = (num_blk_I-1); I >= 0; I--) {
   if (I < num_blk_I)
       x = load(v[i_end(I,J)+1]
            [j_start(I,J)..
             j_end (I,J)];
   hop(comp_loc(I,J));
   if (I < num_blk_I) unload(x);
   for (j = j_start(I,J);
     j <= j_end(I,J); j++)
     for (i = i_end(I,J);
        i >= i_start(I,J); i--)
        v[i][j] = F3(p[j][i],
             v[i+1][j],q[j][i]);

 }

}
```

Figure 3.16: Loop4 pseudocode after assigning locations for tiles

3.3.7 Partitioning phase 2: Example of Matrix Multiplication in second dimension

We apply the Phase 2 Transformation in both dimensions, as depicted in Fig. 3.17. Basically, a pair of A and B entries can move on along their pipelines respectively as soon as they finish computing and contributing the corresponding C entry. A producer BCarrier needs to make sure that the B entry produced by its predecessor in the pipeline is consumed before it puts the B entry it carries in place. This is the reason for a second event EC(., .). Pseudocode for DSC with pipelining in both dimensions is listed in Fig. ??. The entries of matrices A and B are carried in their corresponding agent variables mA and mB, respectively. The matrices are initially distributed, as shown in Fig. 3.17, such that A(N . 1 . l, .) and B(., l) are on node(N . 1 . l, l), and C(i, j) (initialized to 0) is on node(i, j). An event EC(i, j) is signaled on node(i, j) for all values of i, j initially.

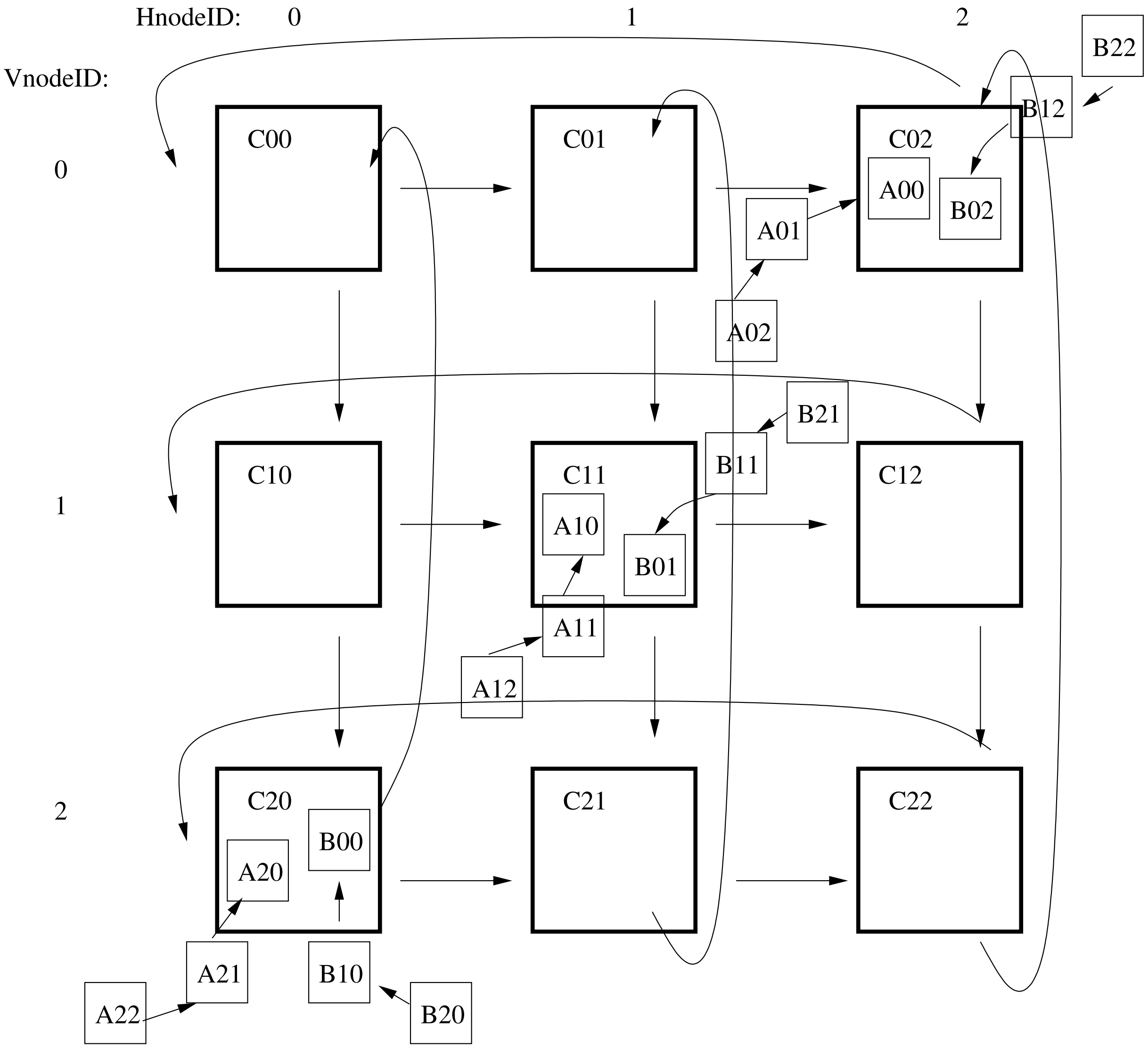


Figure 3.17: Sub DSC in the second dimension

3.3.8 Assigning a location to each tile

Once the size of the tiles has been chosen, each tile needs to be assigned to one of the nodes. In most cases this is a straightforward procedure: all writes from each tile are to variables on the same node, and often the best strategy is to assign the tile to that node. This is consistent with the well-known owner-computes strategy [16].

The strategy that we actually use is to assign each tile to the node that holds the most data accessed by the tile (either as part of the read set or the write set). We call this node the pivot node, and we call the resulting strategy pivot-computes [42]. The pivot-computes strategy can result in significantly less data movement in certain situations, such as a REDUCE-type operation where a large amount of data stored on one node is summarized in a few variables stored on a second node. In this case, performing the computation on the node that holds the read set is more efficient than the node that writes the final value. For a specific example see [41]. The two strategies frequently produce the same result, as they do with the examples of this paper.

Once a tile has been assigned to a node, additional code is inserted to ensure that the execution is performed on the chosen node and all necessary data is carried there. Fig. 5 shows the result of inserting this code in the code of Fig. 3.9. The additional partial row of v necessary for the tile computation is loaded into the local variable x. After the hop to the node where the computation will occur, the data carried in x is unloaded.

3.4 Parallelization Generation

The result of the previous steps is a set of code tiles, each of which has been assigned to a node. The next step is to turn these into a parallel program. The fact that each tile has already been assigned a node makes this step relatively straightforward: all that is needed is to group the tiles into threads and insert appropriate commands for synchronization and transfer of data among threads. This step proceeds in three stages. First we build a Weighted Dependence Graph which captures the essential dependency relations among the tiles. Next we combine the tiles into threads. At this point, we can pass the Weighted Dependence Graph and thread information to the Feedback Mechanism. Based on the feedback, we may decide to go back and change some decisions made during the earlier code transformation and placement phase or to change the data distribution. If we are satisfied with the results of the feedback, we proceed to the third state, which is the generation of the parallel code.

3.4.1 Building the Weighted Dependence Graph

The Weighted Dependence Graph is a precedence graph that captures the relationships among the tiles derived using the code transformation. Each node represents a tile. All edges are directed and indicate that the tile corresponding to the origin node must be computed before the tile corresponding to the destination node.

Each node is assigned a cost, which is the relative amount of computation required by the corresponding tile. Associated with each edge is the communication cost and context switching cost associated with that edge. The context-switching cost is taken to be constant. The communication cost is zero if the two endpoints of the edge are tiles assigned to the same node. Otherwise, the cost is a function of the amount of data that needs to be moved, using a piecewise-linear communication cost model that takes into consideration packet size, latency per packet of the given size, and bandwidth [65].

To estimate the communication cost, we use a piecewise linear communication cost model. Instead of applying fixed latency, we use different latencies in different packet size ranges. The latency for a packet is fixed if the packet size is within a certain range of size. The latency changes if the packet size increases to reach another range of size due to the presence of Maximum Transmission Unit (MTU). MTU is the packet size that a network can transmit and measured in bytes. The communication cost for a packet is per packet latency plus a per Byte bandwidth cost. A node may communicate with a number of other nodes at the same time, and many nodes may communicate with each other at the same time. So, given the limited network resources, the congestion and contention may happen and they have random costs. Our piecewise linear model assume congestion and contention free network which is the reality in most modern network for the purpose of high performance computing.

In mathematics, a transitive reduction of a binary relation R on a set X is a minimal relation R' on X such that the transitive closure of R' is the same as the transitive closure of R. The transitive reduction of a finite acyclic graph is unique. We apply transitive reduction to the Weighted Dependence graph. During the reduction, only the edges with the property of Zero cost can be reduced, because if an edge not only represents synchronization, but also carries data, the edge cannot be removed. If the edge only represents synchronization, the edge can be removed if other edges guarantee the data dependence relation of this edge.

Once the graph has been constructed, we simplify it using a restricted form of transitive reduction: If an edge in the Weighted Dependence Graph only represents synchronization (i.e., carries no data) and if other edges guarantee the precedence relation represented by the edge, the edge can be removed.

3.4.2 Building the Weighted Dependence Graph: Example of ADI

Fig. 3.18 shows original Weighted Dependence Graph corresponding to loops 2, 4, 6, and 8(Loops 1, 3, 5, and 7 are used only for initialization and are omitted to simplify the presentation). Fig. 3.19 shows the reduced or simplified Weighted Dependence Graph. The shapes of the graph nodes represent the processing node where the computation is executed. There are three different shapes because the data are distributed over three processing nodes. Each

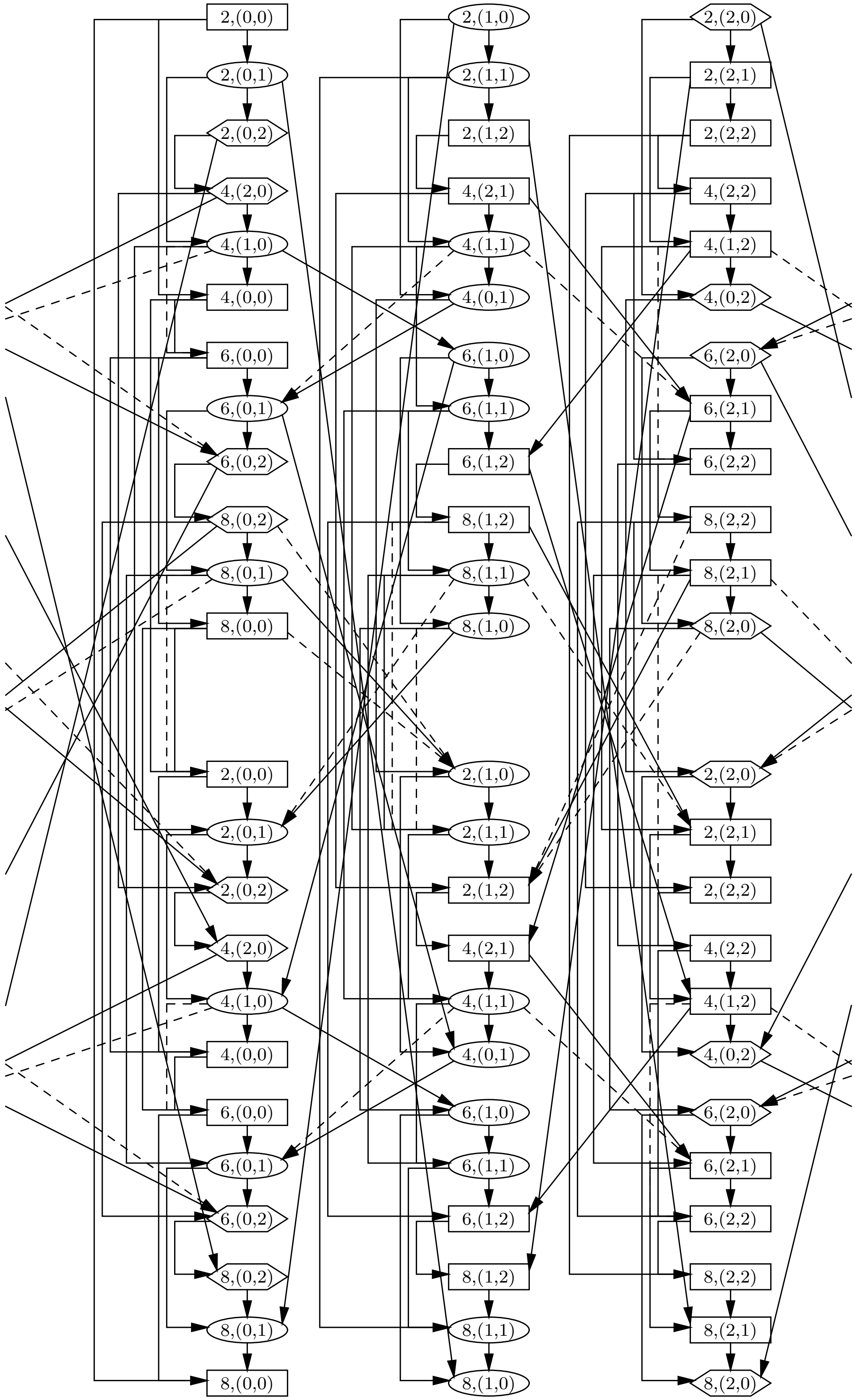


Figure 3.18: Weighted Dependence Graph

graph node is annotated L(I,J), where L is the loop number and the pair (I,J) is the write set of the tile. For example, 4(0,1) means loop4 and (0, 1) is its write set, where I=0 and J=1. A solid edge means that a whole data block has to be transferred, while a dashed edge means that the data dependence consists only of boundary data.

3.4.3 Building the Weighted Dependence Graph: Example of Matrix Multiplication

Since the matrix multiplication has no data dependencies, so, it could perform multiple element calculations concurrently as long as each core only compute one element at the same time. Therefore, there is no need to build dependence graph. However, since matrix A and B are not duplicated in each node and only have one copy, the carries of certain row of matrix A and certain column of matrix B needs to be at the same node at the right timing, so still needs synchronization to fully paralleled.

3.4.4 Combining tiles into threads

To create execution threads from the reduced graph, we use a bottom up approach. Initially each tile is considered a separate thread. These are then combined into longer threads.

3.4.5 Generating threads for ADI

There are many ways to combine the tiles into a thread. We apply a heuristic strategy, which combines threads inside a single loop within the same global iteration. Our heuristic threads together tiles that cannot be executed in parallel. In particular, it combines tiles connected by edges because the data dependencies require such tiles to be executed sequentially. For

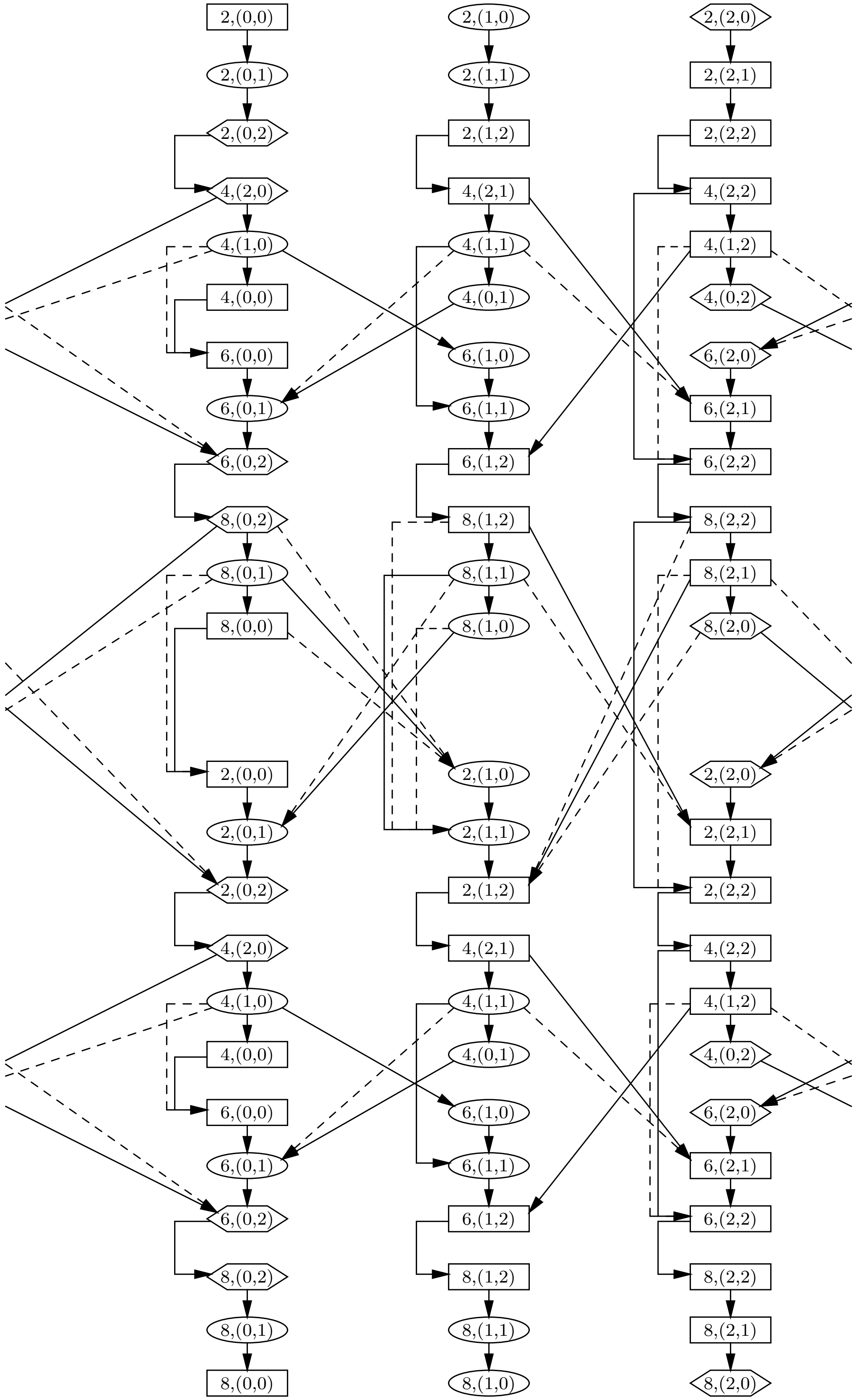


Figure 3.19: Simplified Weighted Dependence Graph

example, in Fig. 3.18, for loop2, tile (0,0), (0,1), (0,2) are grouped into a thread, tile (1,0), (1,1), (1,2) are grouped into a thread, and tile (2,0), (2,1), (2,2) are grouped into a thread.

3.4.6 Generating threads for Matrix Multiplication in one dimension

Since there is no data dependencies among the DSC threads, but still need synchronization to ensure the elements need to be computed are in the local node. According to the pipelining idea which is to overlap the execution of multiple DSC threads by staggering their starting times. The result of this pipelining is depicted in Fig. 3.20. Each row of matrix A is assigned to a different computation thread. Injected into the PE pipeline in order, these threads follow each other in the network to compute the corresponding C entries.

Sometimes the dependency among different computations allows different DSC threads to enter the pipeline from different PEs. In these situations, we can phase shift the DSC threads to achieve full parallelism, We apply our Phase-shifting Transformation to achieve a full DPC, as depicted in Fig. 3.21. This is possible because each row of A, though needed on all three PEs, can start its computation from any PE.

3.4.7 Generating threads for Matrix Multiplication in second dimension

Same as in one dimension, in two dimension, we apply our Pipelining and Phase-shifting in both dimensions, as depicted in Fig. 3.22 and Fig. 3.23. Basically, a pair of A and B entries can move on along their pipelines respectively as soon as they finish computing and contributing the corresponding C entry. A producer BCarrier needs to make sure that the B entry produced by its predecessor in the pipeline is consumed before it puts the B entry it carries in place. This is the reason for a second event EC(., .).

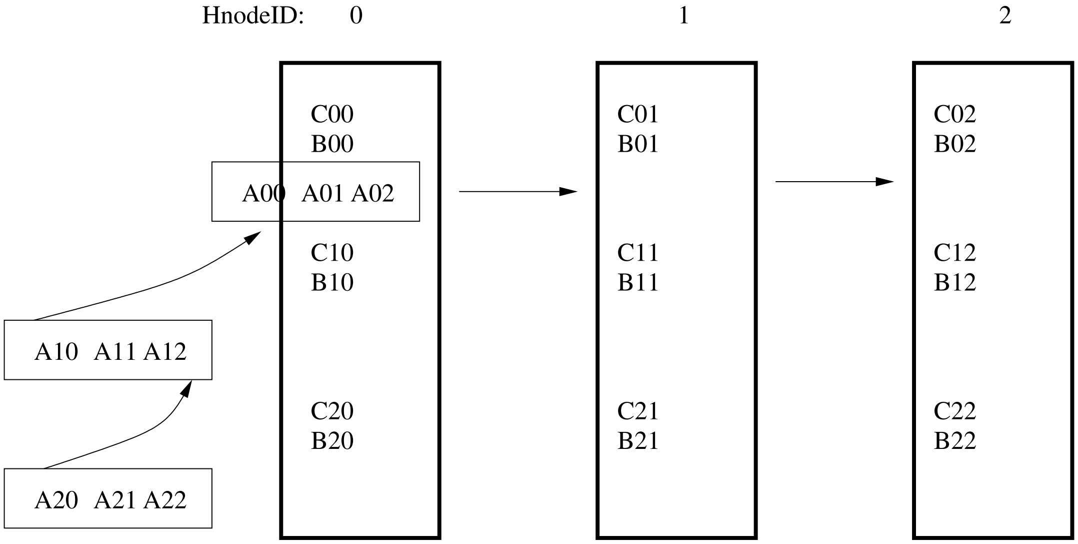


Figure 3.20: Pipelining in one dimensions

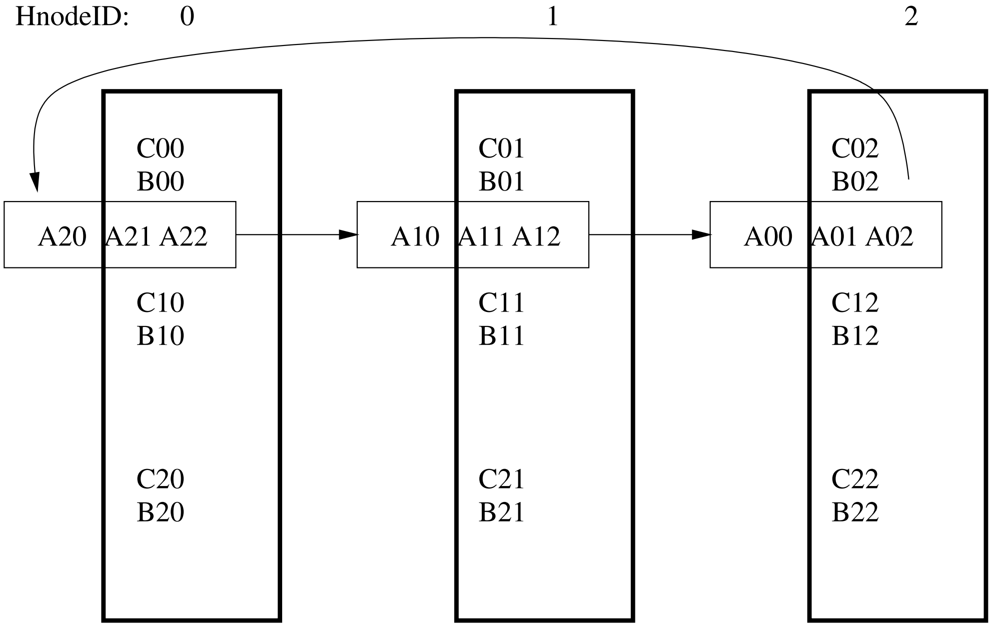


Figure 3.21: Phase shifting in one dimension

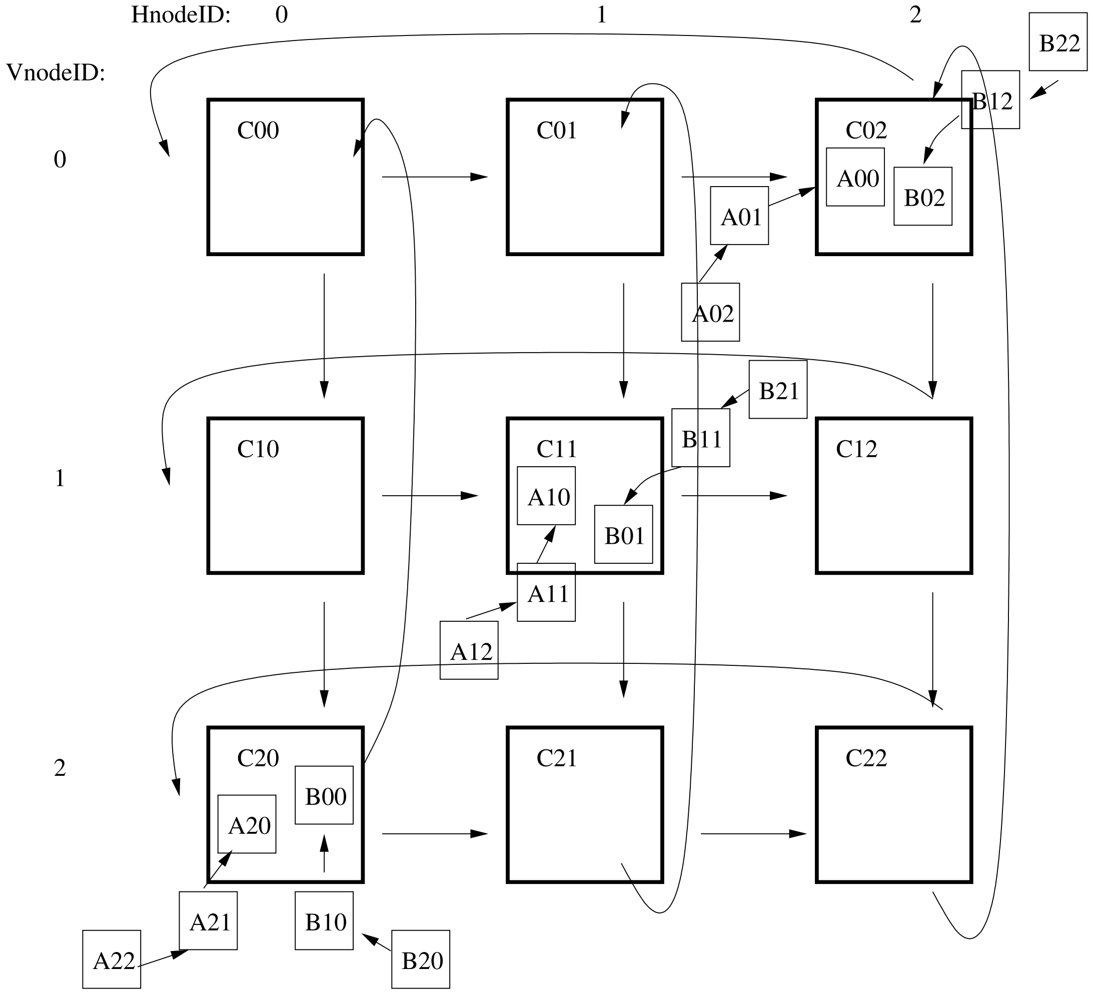


Figure 3.22: Pipelining in both dimensions

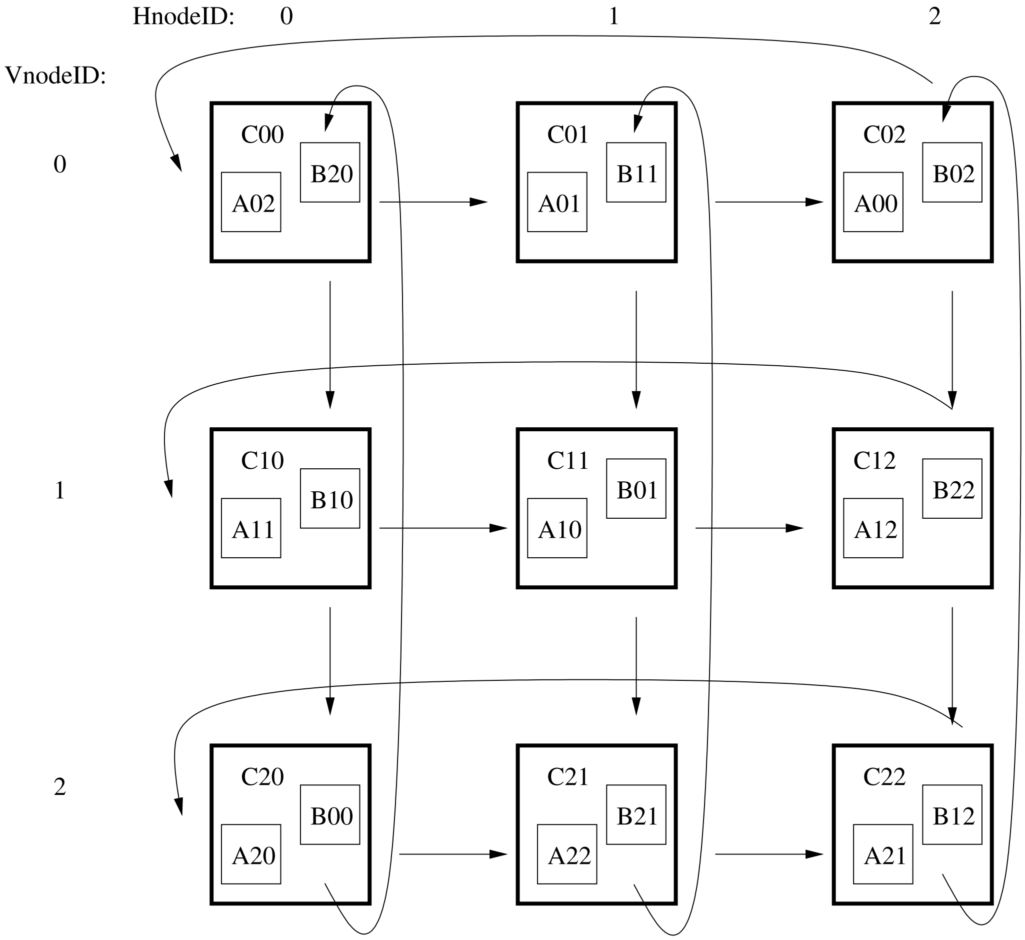


Figure 3.23: Phase shifting in both dimensions

3.4.8 Generating the Code

Generating the code requires inserting code so that the multiple threads synchronize with each other.

3.4.9 Generating the Code: Example of ADI

For instance, loop4 is responsible for computing v[i][j]. It must wait until p[j][i] and q[j][i] are ready, which are computed by other threads. Synchronization code must be added to wait for data computed by other threads and to signal that data is ready to be accessed. The basic principle is to insert signal/wait primitives at each end of a solid edge and add code to transfer data for any edge that carries data (edges with non-zero weight.) Fig. 3.24 shows the portion of the DPC code that represents the transformed Loop 4. Builder.msgr injects a spawner. loop4_spawner.msgr creates the multiple threads for loop4 (Loop4_sweeper) and injects them at the node where the computing starts for loop 4. Fig. 3.25 shows the portion of the DPC code that represents the transformed Loop 8.

Each of these threads processes a column of tiles. It executes as a doubly indexed loop (lines 9-10), where the outer loop runs through all iterations of the main loop in the original sequential program and the inner loop runs through all the tiles in the column. For each tile, it hops to the pivot node (line 11) synchronizes with other threads (line 12), computes the tile (line 13-17) , and synchronizes with other threads (lines 18-21). Since each hop is carrying data, we do not explicitly show load and unload data in this pseudo code.

In this example, no data transmission code is needed for solid edges because their weights are all zero. This is true in most cases because any two tiles that access the same data will, due to the pivot-computes principle, be assigned to the same node.

```
Builder.msgr(ITERATIONS)
 inject(Loop4_spawner,ITERATIONS);
loop4_spawner.msgr
for (J = 0; J < num_blks_J J++) {
     hop(comp_loc[0,J]);
     inject(Loop4_sweeper, J);
}
Loop4_sweeper(J):
for(iter=0; iter< ITERATIONS;iter--){
 for (I=(num_blk_I-1);I>=0;I--){
   hop(comp_loc[I,J]);
   if(I==(num_blk_I-1))
       wait(E(iter,loop4,J,I,p/q)) ;
   for(j=j_start(I,J); j<=j_end(I,J); j++){
    for(i=i_end(I,J); i>=i_start(I,J);i--){
      v[i][j]= F3(p[j][i],v[i+1][j],q[j][i];
     }
    }
    if(I==0)signal(E(iter,loop6,I,J,v) ;
    if(I!=(n_num_blks_I-1) && I!=0){
       Push with Signal E(iter,I,J,v_plus);
       Push with Signal E(iter,I,J,v_minus);}
   }
 }
}
```

Figure 3.24: DPC Code

```
loop8_sweeper.msgr
for (J= n_num_blks_per_row-1; J >=0; J--) {
    n_curr_node = n_node_map[I,J];
    if ( n_curr_node != myID )
       hop(link=n_links[n_curr_node]);
    if(J == n_num_blks_per_row-1){
      wait_element(m_iter,loop8,I,n_num_blks_per_row-1,u);}
    wait_element(m_iter,loop8,I,J,p/q);
    loop8_sweep(I,J);
    if(m_iter+1 <= ITERATIONS){
      set_element(m_iter+1,loop2,I,J,u);
      set_space(m_iter+1,loop2,I,J,p/q);
      if(n_reverse_fetch_minus[I,J]!=-1){
         inject(PUSH_U_MINUS,m_iter+1,I,J);
      }
      if(n_reverse_fetch_plus[I,J]!=-1){
        inject(PUSH_U_PLUS,m_iter+1,I,J);
       }
    }
}
```

Figure 3.25: DPC Code Loop8

3.4.10 Generating the Code: Example of Matrix Multiplication in one dimension

Pseudocode for pipelined DSC matrix multiplication is listed in Fig. 3.26. The matrix A is initially put on the PE with HnodeID = 0, and the columns of matrices B and C are distributed such that B(., j) and C(., j) are on the PE with HnodeID = j.

Pseudocode for Phase-shifting in one dimension is listed in Fig. 3.27. Rows of matrix A are carried by the corresponding agent variables mA. In this implementation, matrix A is initially distributed such that A(i, .) is on the PE with HnodeID = i, and the columns of matrices B and C are distributed such that B(., j) and C(., j) are on the PE with HnodeID = j.

```
hop(node(0))
do i=0,N-1
    inject(RowCarrier(i))
end do
RowCarrier(int mi)
mA(*) = A(mi,*)
do mj=0,N-1
    hop(node(mj))
    t = 0.0
    do k=0,N-1
        t += mA(k) * B(k)
    end do
    C(mi) = t
end do
end
```

Figure 3.26: Pseudocode for pipelined DSC

```
do mi=0,N-1
    hop(node(mi))
    inject(RowCarrier(mi))
end do
RowCarrier(int mi)
mA(*) = A(*)
do mj=0,N-1
    hop(node((N-1-mi+mj)%N))
    t = 0.0
    do k=0,N-1
         t += mA(k) * B(k)
    end do
    C(mi) = t
end do
end
```

Figure 3.27: Pseudocode for DPC in one dimension

3.4.11 Generating the Code: Example of Matrix Multiplication in second dimension

Pseudocode for DSC with pipelining in both dimensions is listed in Fig. 3.28. The entries of matrices A and B are carried in their corresponding agent variables mA and mB, respectively. The matrices are initially distributed, as shown in Fig. 3.28, such that A(N . 1 . l, .) and B(., l) are on node(N . 1 . l, l), and C(i, j) (initialized to 0) is on node(i, j). An event EC(i, j) is signaled on node(i, j) for all values of i, j initially.

```
do ml=0,N-1
    hop(node(N-1-ml,ml))
    inject(spawner(ml))
end do
spawner(int ml)
    do mk=0,N-1
        inject(ACarrier(N-1-ml,mk))
        inject(BCarrier(mk,ml))
    end do
end
ACarrier(int mi, int mk)
    mA=A(mk)
    do mj=0,N-1
        hop(node(mi,(N-1-mi+mj)%N))
        waitEvent(EP(mi,(N-1-mi+mj)%N))
        C += mA * B
        signalEvent(EC(mi,(N-1-mi+mj)%N))
    end do
end
BCarrier(int mk, int mj)
    mB=B(mk)
    do mi=0,N-1
        hop(node((N-1-mj+mi)%N,mj))
        waitEvent(EC((N-1-mj+mi)%N,mj))
        B = mB
        signalEvent(EP((N-1-mj+mi)%N,mj))
    end do
end
```

Figure 3.28: Pseudocode for pipelining in second dimensions.

Pseudocode for DPC in both dimensions is listed in Fig. 3.29. The entries of matrices A and B are carried in their corresponding agent variables mA and mB, respectively. The matrices are initially distributed such that A(i, j), B(i, j) and C(i, j) (initialized to 0) are on node(i, j). In the above figures such as Fig. 3.23, each sub-matrix block, e.g., A10 or C11, is called a distribution block in our implementation, as it is a basic unit of data distribution on a PE. To achieve better performance from a block algorithm, a further level of matrix decomposition is used [47] . A distribution block is decomposed into algorithmic blocks, and each algorithmic block of A or B is carried by a migrating thread (i.e., ACarrier or BCarrier). Our sequential and MPI (Message Passing Interface) implementations described below use algorithmic blocks as well.

```
do mj=0,N-1
    hop(node(0,mj))
    inject(spawner(mj))
end do
spawner(int mj)
do mi=0,N-1
    hop(node(mi,mj))
    signalEvent(EC(mi,mj))
    inject(ACarrier(mi,mj))
    inject(BCarrier(mi,mj))
end do
end
ACarrier(int mi, int mk)
mA = A
do mj=0,N-1
    hop(node(mi,(N-1-mi-mk+mj)%N)
    waitEvent(EP(mi,(N-1-mi-mk+mj)%N))
    C += mA * B
    signalEvent(EC(mi,(N-1-mi-mk+mj)%N))
end do
end
BCarrier(int mk, int mj)
mB = B
do mi=0,N-1
    hop(node((N-1-mj-mk+mi)%N,mj))
    waitEvent(EC((N-1-mj-mk+mi)%N,mj))
    B = mB
    signalEvent(EP((N-1-mj-mk+mi)%N,mj))
end do
end
```

Figure 3.29: Pseudocode for full DPC in second dimensions.

3.5 Feedback Mechanism

The long-term goal of the feedback loop is to be able to periodically improve the performance of the actual program by adjusting its current data distribution and level of parallelism at runtime based on information gathered automatically from the instrumented program. Currently, the feedback loop uses only a simulator, which executes a discrete-event simulation of the simplified Weighted Dependence Graph. The communication costs are estimated using the communication cost model discussed earlier. The computation time of each tile is estimated by the relative time to perform the computation. The simulator gives the following output to help us evaluate the performance of the parallel distributed program:

- **Total execution time**. This includes communication cost, computation cost, the overlap of communication cost and computation cost, overhead of context switch, and the time to wait for dependent data. The execution time is not real wall clock time; it is a relative time, but it can be scaled to real time. This feedback is used to evaluate and analyze the DPC code for further improvements.
- **Idle time**: This is the total of the idle times over all physical nodes.
- **Efficiency ratio**: This is the ratio of the total attained time (simulated CPU time) to the total execution time (attained time, communication time, and idle time)
- **Speedup**: This is the speedup of DPC codes over sequential codes.
- **Degree of load balance**: This measure captures how well the load is balanced over all physical nodes.

After evaluating the parallel distributed programs, the Feedback mechanism passes its evaluations to the previous steps. These evaluations include the speedup, the idle time, the efficiency ratio, and load imbalance, and serve as guide to improve the respective transformations as shown next.

3.5.1 Case 1: Feedback to Code Transformer

When the Code Transformer receives the speedup feedback, it tries to increase parallelism by reducing the tile size. As the number of tiles increases, the communication and context switch cost increase while the parallelism increases. Both changes directly affect speedup. The process of decreasing the tile size is repeated until the speedup reaches its peak. Beyond this point the speedup decreases despite increased parallelism because the gains are outweighed by the even faster increasing communication and context switching costs.

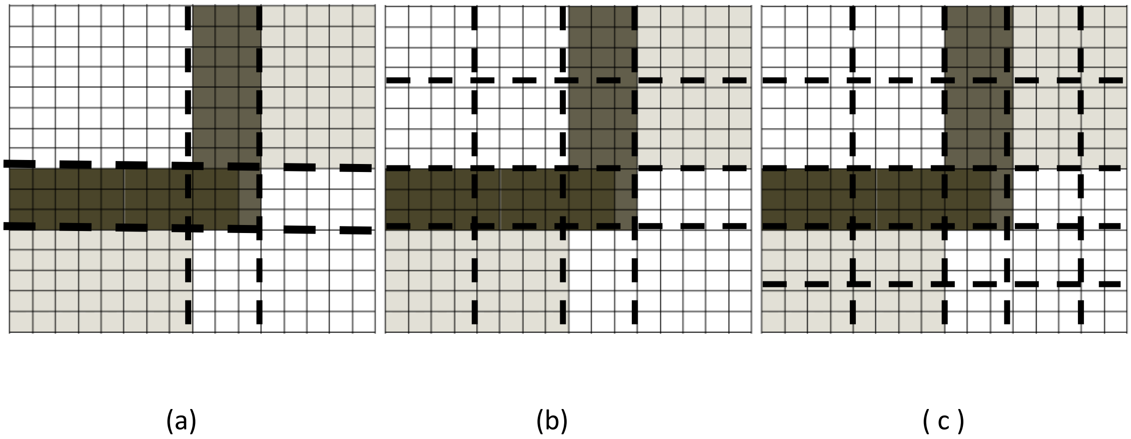


Figure 3.30: Different options of tiles for the same data distribution

Fig. 3.30 shows the same partitioning of loop4 but with different number of tiles. Fig. 3.30 (a) has 9 tiles, Fig. 3.30(b) has 16, and Fig. 3.30 (c) has 25. Clearly, the first case has the smallest communication cost but also the smallest opportunity for parallelism, while the third case shows the opposite. Fig. 3.31(a) shows the speedup for four different tile sizes.

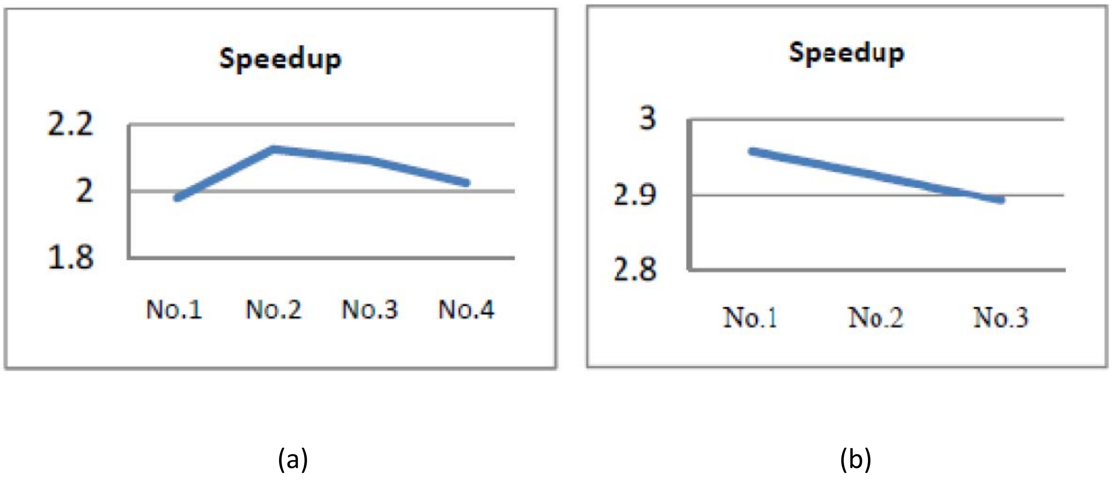


Figure 3.31: (a) Speedup for different tile options for the unstructured data distribution (b) Speedup for different tile options for the twisted data distribution

No.1 through No.3 correspond to the above three tile sizes, and No.4 corresponds to an even smaller tile size. The curve shows an improvement when tile size is increased from 9 to 16 but drops off with the next smaller tile size. Based on this, the second option, Fig. 3.30(b), is the best choice.

Fig. 3.31(b) shows the results for the same program but using the twisted data distribution. No.1 has 9 tiles, No.2 has 36 tiles, and No.3 has 81 tiles. The speedup decreases with the tile size. Hence the best choice for this distribution is the first option.

3.5.2 Case 2: Feedback to Data distributor

When the Feedback mechanism determines that the load is highly imbalanced, it will suggest to the Data Distributor to increase the number of partitions and try a different assignment of partitions to nodes.

Fig. 3.33, Fig. 3.34 and Fig. 3.35 shows three versions of data partitioning and their assignment to three different nodes. (a) has 6 partitions (resulting in 2 partitions assigned to each

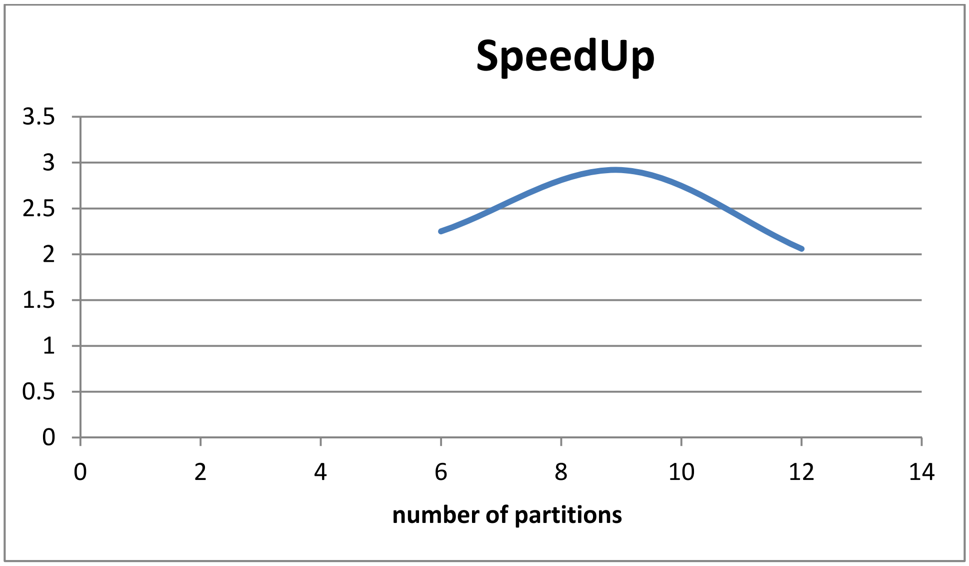


Figure 3.32: Speedup achieved for the three cases of Fig.10

node), (b) has 9 partitions (3 per node) and (c) has 12 partitions (4 per node). Fig. 3.32 shows the performance for the three cases. As the number of partitions increases, the distributor has more flexibility in assigning partitions to nodes, which increases the chances to keep the load balanced. However, more partitions result in more communication overhead. Thus when the number of partitions increases beyond its peak, the increased communication cost outweighs the benefits of the better load distribution. For the example, using 9 partitions was the best choice.

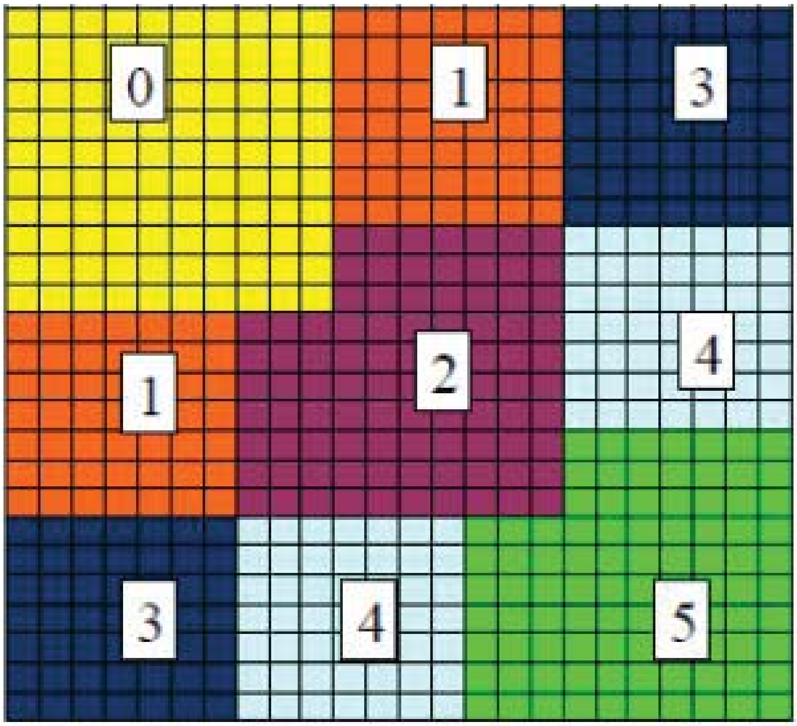


Figure 3.33: 6 partitions (2 partitions per node)

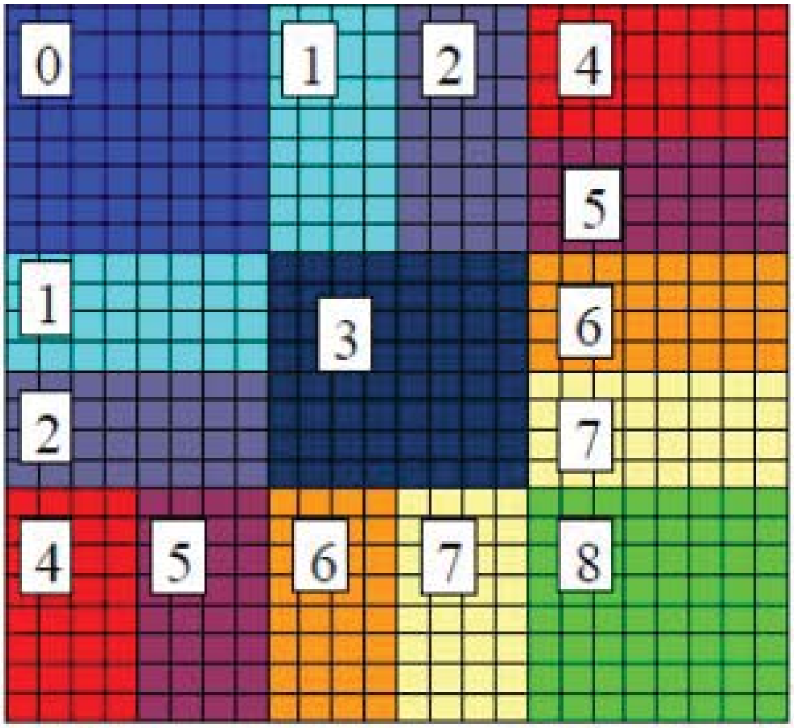


Figure 3.34: 9 partitions (3 per node)

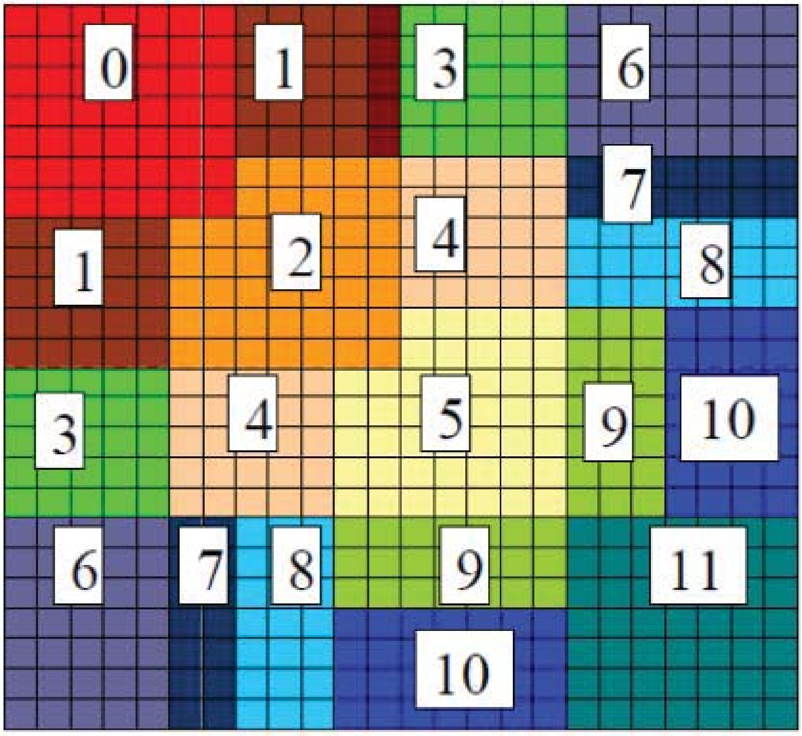


Figure 3.35: 12 partitions (4 per node)

Chapter 4

Experiments

4.1 Experiment of ADI

In this section, we present our experimental results for the target application. The data was obtained using a network of Linux cluster and 100Mbps of Ethernet connection with a collision-free switch and using the NFS file-sharing system.

Fig. 4.1 shows the performance when using the unstructured data distribution. Array size is the size of the arrays u, v, p and q. DPC is execution time for the parallel program, which is divided by the sequential time to get the speedup. Fig. 4.2 Fig. 4.2 Fig. 4.3 Fig. 4.4 Fig. **??** shows the performance results for the twisted data distribution, for array sizes of 5040×5040, 12600×12600, and 15120×15120. Both experiments show a significant speedup for all cases.

Number of Nodes	Array size	DPC Exec. Time	Sequential Exec. Time	Speedup
3	5040×5040	38.176	111.875	2.930
4	5056×5056	36.581	138.104	3.775
5	9600×9600	104.357	410.104	3.929
6	9600×9600	93.262	410.104	4.397

Table 4.1: Speedup given unstructured data distribution.

Number of Nodes	Array size	DPC Exec. Time	Sequential Exec. Time	Speedup
3	7560 × 7560	86.006	250.139	2.908
4	12600 × 12600	178.764	684.181	3.827
5	12600 × 12600	144.85	678.368	4.683
6	15120 × 15120	174.096	977.904	5.617

Table 4.2: Speedup given twisted data distribution A

Number of Nodes	Array size	DPC Exec. Time	Sequential Exec. Time	Speedup
3	5040 × 5040	38.556	110.266	2.859
4	5040 × 5040	34.586	109.469	3.165
5	5040 × 5040	23.597	108.539	4.599
6	5040 × 5040	22.407	108.656	4.849
7	5040 × 5040	20.437	108.213	5.294
8	5040 × 5040	28.486	109.813	3.854
9	5040 × 5040	15.350	106.811	6.958
10	5040 × 5040	14.330	108.110	7.544
11	5040 × 5040	13.959	107.985	7.735
12	5040 × 5040	13.593	108.064	7.949

Table 4.3: Speedup given twisted data distribution B

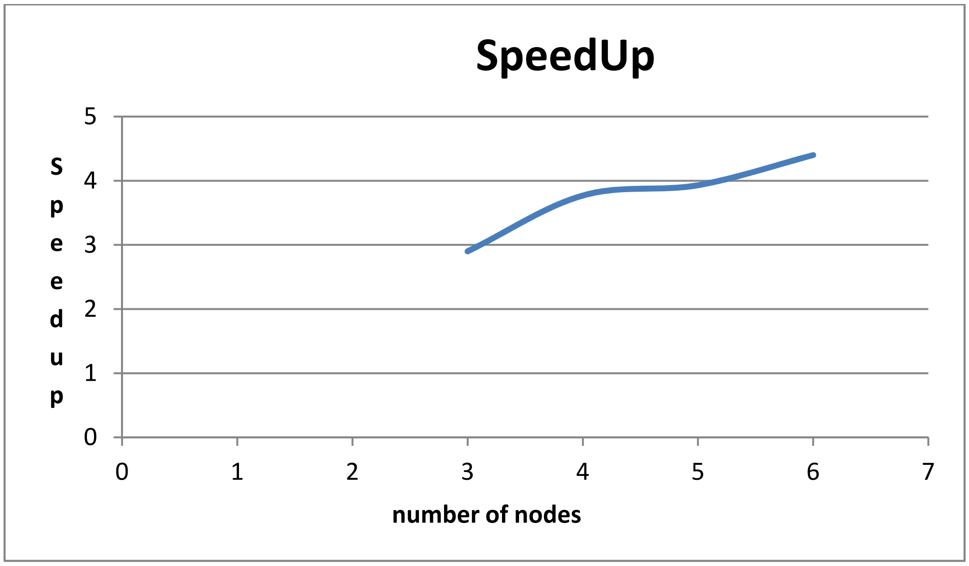


Figure 4.1: Speedup with unstructured data distribution

Number of Nodes	Array size	DPC Exec. Time	Sequential Exec. Time	Speedup
4	12600 × 12600	178.764	684.181	3.827
5	12600 × 12600	144.850	678.368	4.683
6	12600 × 12600	121.155	679.125	5.605
7	12600 × 12600	104.834	676.331	6.451
8	12600 × 12600	95.154	686.331	7.212
9	12600 × 12600	82.883	667.568	8.054
10	12600 × 12600	84.588	675.687	7.987
11	12600 × 12600	78.631	734.869	9.345
12	12600 × 12600	65.423	675.400	10.323

Table 4.4: Speedup given twisted data distribution C

Number of Nodes	Array size	DPC Exec. Time	Sequential Exec. Time	Speedup
6	15120 × 15120	174.096	977.904	5.617
7	15120 × 15120	149.346	973.917	6.521
8	15120 × 15120	134.631	988.317	7.340
9	15120 × 15120	118.279	961.299	8.127
10	15120 × 15120	106.704	972.990	9.118
11	15120 × 15120	101.219	1000.2391	9.881
12	15120 × 15120	106.514	972.576	9.130

Table 4.5: Speedup given twisted data distribution D

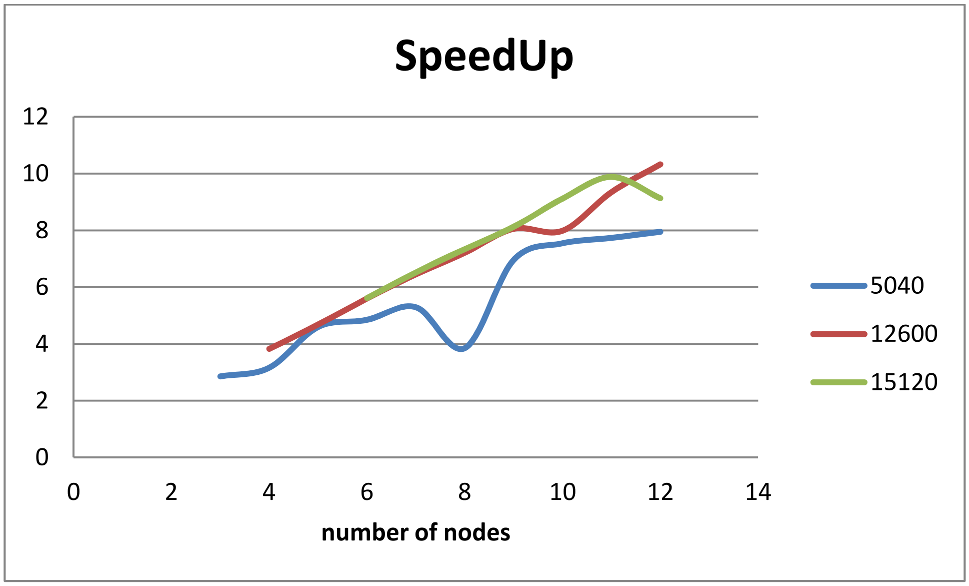


Figure 4.2: Speedup with twisted data distribution

4.2 Experiment of Matrix Multiplication

4.2.1 Gentlemans Algorithm of Matrix Multiplication

Gentlemans Algorithm [18] [49] is a classical SPMD algorithm for parallel matrix multiplication. The pseudocode is listed in Fig. 4.3, in which an arrow represents a combination of receive and send among remote PEs. During initial staggering, each entry of matrix A will stagger i times to the west, where i is the entrys row number, and each entry in matrix B will stagger j times to the north, where j is the entrys column number. An entry can be either a single value or a sub-matrix. Thus, a skewed transformation of matrices A and B results. Like the NavP pseudocode, our MPI implementation assumes a fully connected network, and matrix staggering is accomplished in a single step (not shown in Fig. 4.3) rather than in a series of steps. Throughout the entirety of Gentlemans Algorithm, matrix C remains stationary. Once the initial staggering completes, matrices A and B are multiplied and the results are placed in matrix C. For N - 1 iterations, matrix A shifts its columns one step to the west and matrix B shifts its rows one step to the north, and A and B are multiplied with the results added to the C matrix. In our implementation, non-blocking receives (i.e., MPI Irecv()) are used in conjunction with blocking sends to prevent deadlocking. MPI Wait(), which blocks until the incoming matrix has been received, assists in providing synchronization between PEs. As a result of using algorithmic blocks, many blocks are shifted from a PE to itself during the computation. Instead of sending an algorithmic block to a PE itself, or copying an algorithmic block from a local memory, we use pointer swapping to shift an algorithmic block locally.

```
do k=0,N-2
    doall node(i,j) where 0<=i,j<=N-1
        if i>k then
        A <- east(A)
        end if
        if j>k then
        B <- south(B)
        end if
    end do
end do
doall node(i,j) where 0<=i,j<=N-1
    C = A * B
end do
do k=0,N-2
    doall node(i,j) where 0<=i,j<=N-1
        A <- east(A)
        B <- south(B)
        C += A * B
    end do
end do
```

Figure 4.3: Pseudocode for Gentlemans Alg.

4.2.2 Performance data

We have implemented parallel matrix multiplication using both NavP and message passing. The NavP system used was MESSENGERS developed in Donald Bren School of Information Computer Sciences, University of California Irvine. The message passing system used was LAM from Indiana University [17]. The ScaLAPACK used was from University of Tennessee, Knoxville and Oak Ridge National Laboratory [9]. The performance data was obtained from SUN workstations and 100Mbps of Ethernet connection.

These workstations have a shared file system (NFS). When the working set of a sequential program exceeds the physical memory on a PE, thrashing happens and the performance degrades. In a distributed program, however, the data of a sub-problem may fit in the memory of a machine completely even if the entire problem is too large. In order to obtain fair speedup numbers, we calculate sequential timing for large problems using least squared curve fitting with a polynomial of order 3 using performance numbers collected with small problems.

When the working set of a sequential program exceeds the physical memory on a PE, thrashing happens and the performance degrades. In a distributed program, however, the data of a sub-problem may fit in the memory of a machine completely even if the entire problem is too large. In order to obtain fair speedup numbers, we calculate sequential timing for large problems using least squared curve fitting with a polynomial of order 3 using performance numbers collected with small problems.

Table 1 in Fig. 4.4 lists the performance data for NavP and ScaLAPACK on a 1D PE network of three machines. It can be seen that the performance improves as we go from NavP DSC to NavP pipelining and then to NavP phase shifting. For small problems, NavP 1D DSC is only marginally slower than the corresponding sequential execution; however, as the problem size grows, NavP 1D DSC becomes faster, as indicated by the data from actual runs (not curve-fitted data). Table 2 in Fig. 4.5 indicates that with several networked computers DSC performs almost as fast as the sequential program running with enough main memory, and it is significantly faster than the sequential program paging using virtual memory. With N = 9216, the total memory usage is about 1GB, but our machines each have only 256MB of main memory.

Tables 3 and 4 in Fig. 4.6 and Fig. 4.7 list the performance data for MPI, NavP, and ScaLAPACK on a 2D PE network of nine machines. Again, performance improves as we hierarchically apply the three NavP transformations in the second dimension. In both 1D and 2D cases, our DSC and pipelining programs achieve high performance. This can be attributed to the use of algorithmic blocks. The RowCarriers or ACarriers, each of which responsible for the computation of a row of algorithmic blocks or an algorithmic block, can spread out their computations to the entire network earlier than if a full distribution block on a PE has to be computed before these carriers can hop out. The MPI implementation used for the comparison was Gentlemans Algorithm modified to use block partitioning of matrices; moreover, pointer swapping was used in order to avoid unnecessary local data copying. ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique [16], so the block orders in the tables do not apply to the ScaLAPACK numbers.

The performance data indicates that the NavP implementation achieves a higher speedup than the MPI implementation.

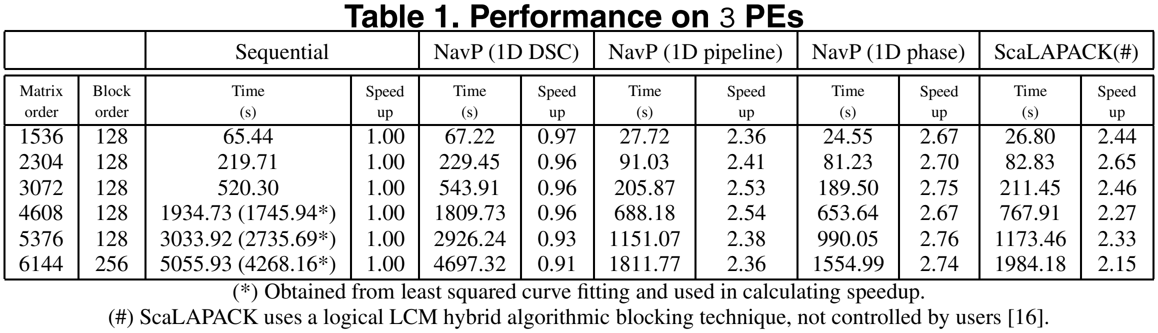


Table 1. Performance on 3 PEs

		Sequential		NavP (1D DSC)		NavP (1D pipeline)		NavP (1D phase)		ScaLAPACK(#)	
Matrix order	Block order	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
1536	128	65.44	1.00	67.22	0.97	27.72	2.36	24.55	2.67	26.80	2.44
2304	128	219.71	1.00	229.45	0.96	91.03	2.41	81.23	2.70	82.83	2.65
3072	128	520.30	1.00	543.91	0.96	205.87	2.53	189.50	2.75	211.45	2.46
4608	128	1934.73 (1745.94*)	1.00	1809.73	0.96	688.18	2.54	653.64	2.67	767.91	2.27
5376	128	3033.92 (2735.69*)	1.00	2926.24	0.93	1151.07	2.38	990.05	2.76	1173.46	2.33
6144	256	5055.93 (4268.16*)	1.00	4697.32	0.91	1811.77	2.36	1554.99	2.74	1984.18	2.15

(*) Obtained from least squared curve fitting and used in calculating speedup.
(#) ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique, not controlled by users [16].

Figure 4.4: Performance on 3 PEs

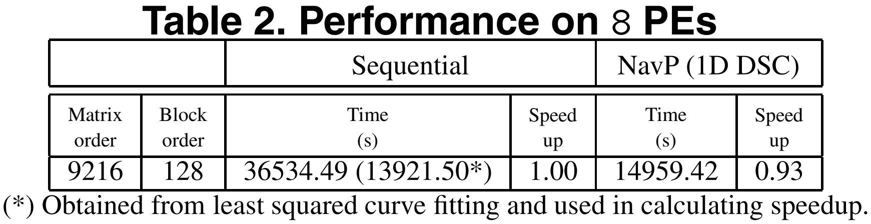


Table 2. Performance on 8 PEs

		Sequential		NavP (1D DSC)	
Matrix order	Block order	Time (s)	Speed up	Time (s)	Speed up
9216	128	36534.49 (13921.50*)	1.00	14959.42	0.93

(*) Obtained from least squared curve fitting and used in calculating speedup.

Figure 4.5: Performance on 8 PEs

The performance data indicates that the NavP implementation achieves a higher speedup than the MPI implementation. Some differences between these two implementations are discussed briefly below. More details can be found in our full-length technical report [45].

1. Communication. We use block algorithms for better cache and communication performance. The algo-rithmic blocks of C on a PE can be updated in different orders. In the

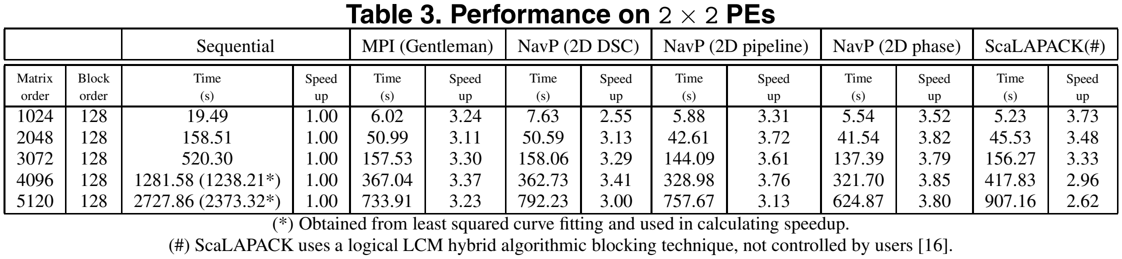


Table 3. Performance on 2×2 PEs

		Sequential		MPI (Gentleman)		NavP (2D DSC)		NavP (2D pipeline)		NavP (2D phase)		ScaLAPACK(#)	
Matrix order	Block order	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
1024	128	19.49	1.00	6.02	3.24	7.63	2.55	5.88	3.31	5.54	3.52	5.23	3.73
2048	128	158.51	1.00	50.99	3.11	50.59	3.13	42.61	3.72	41.54	3.82	45.53	3.48
3072	128	520.30	1.00	157.53	3.30	158.06	3.29	144.09	3.61	137.39	3.79	156.27	3.33
4096	128	1281.58 (1238.21*)	1.00	367.04	3.37	362.73	3.41	328.98	3.76	321.70	3.85	417.83	2.96
5120	128	2727.86 (2373.32*)	1.00	733.91	3.23	792.23	3.00	757.67	3.13	624.87	3.80	907.16	2.62

(*) Obtained from least squared curve fitting and used in calculating speedup.
(#) ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique, not controlled by users [16].

Figure 4.6: Performance on 2*2 PEs

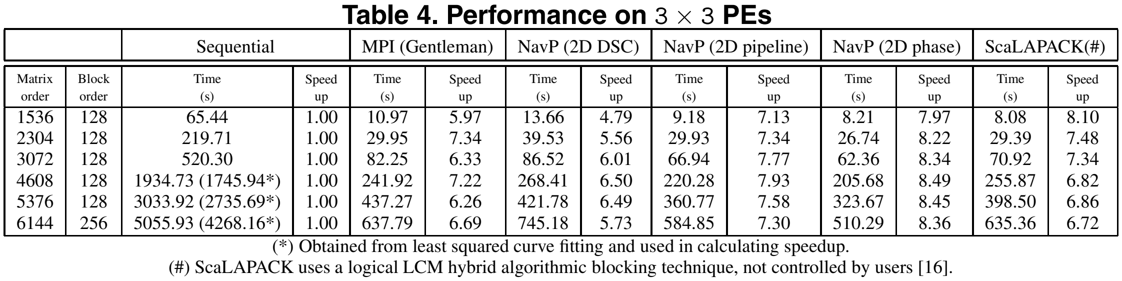


Table 4. Performance on 3×3 PEs

		Sequential		MPI (Gentleman)		NavP (2D DSC)		NavP (2D pipeline)		NavP (2D phase)		ScaLAPACK(#)	
Matrix order	Block order	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
1536	128	65.44	1.00	10.97	5.97	13.66	4.79	9.18	7.13	8.21	7.97	8.08	8.10
2304	128	219.71	1.00	29.95	7.34	39.53	5.56	29.93	7.34	26.74	8.22	29.39	7.48
3072	128	520.30	1.00	82.25	6.33	86.52	6.01	66.94	7.77	62.36	8.34	70.92	7.34
4608	128	1934.73 (1745.94*)	1.00	241.92	7.22	268.41	6.50	220.28	7.93	205.68	8.49	255.87	6.82
5376	128	3033.92 (2735.69*)	1.00	437.27	6.26	421.78	6.49	360.77	7.58	323.67	8.45	398.50	6.86
6144	256	5055.93 (4268.16*)	1.00	637.79	6.69	745.18	5.73	584.85	7.30	510.29	8.36	635.36	6.72

(*) Obtained from least squared curve fitting and used in calculating speedup.
(#) ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique, not controlled by users [16].

Figure 4.7: Performance on 3*3 PEs

case of NavP, the order is not predefined and the CPU cycles are thus efficiently utilized in computations as the data they need arrives. An efficient run-time task scheduling, handled by the queuing mechanisms built into the MESSENGERS daemon, is provided to the NavP programmers. As a result, NavP programmers only need to concern themselves with the two event handling commands as the interface to the queuing mechanisms that are otherwise hidden at the system level. It is the NavP view that allows us to focus on describing the application level computations following their movement and to factor out the functionality associated with scheduling code that describes behaviors at fixed locations. In MPI, the situation is quite different. The straightforward way to program the block implementation is to have a loop over all the algorithmic blocks of C on a PE. The loop introduces an artificial sequential order to the communications and computations even though they are actually independent of each other and hence may result in slower performance. Possible ways to remove the artificial sequencing are proposed [45], but they all require significantly more programming work.

2. Cache performance. The NavP and the sequential programs have a similar cache performance because in both cases during the execution there is an algorithmic block (of C for the sequential program and of A for the NavP program, respectively) that would stay in the cache for the duration of computation using other two algorithmic blocks. In contrast, in the block-oriented MPI program, triplets of A B C blocks are frequently fresh in the cache, which leads to less efficient cache use. A simple analysis shows that this cache performance of NavP can account for as much as a 4over MPI [45].

3. Initial staggering. The NavP program uses reverse staggering for matrices A and B. That is, the chain of a row or a column is both shifted and reverseordered. In contrast, both Gentlemans Algorithm and Cannons Algorithm [10] [48] use forward staggering, which only shifts the positions of the entries without reversing the order. It is shown [45] that reverse staggering never requires more than two communication phases, while forward staggering often requires three communication phases. It would be possible to improve the performance of the MPI code by subtle fine-tuning at a cost of considerably more programming effort. Nevertheless, the data makes it clear that the NavP program is faster than a straightforward implementation of Gentlemans Algorithm and competitive with a highly tuned version.

In incremental parallelization, a programmer uses sequential code as the starting point and introduces parallelism in a step by step fashion, until satisfactory performance is achieved or a time/resource constraint is reached. Oftentimes, programmers begin with the performance critical hot spots in a program and gradually parallelize other parts of the program.

Message passing programming is less amenable to incremental parallelization. Transforming a sequential program into a message-passing one is an abrupt break, since data must be distributed and code structure is often dramatically changed. This is seen in the matrix multiplication example one either gets no parallelism at all with the sequential code, or one gets all parallelism with Gentlemans Algorithm. Going directly from the sequential code to a parallel algorithm such as Gentlemans Algorithm requires considerable ingenuity. Nevertheless, message passing programming usually leads to good performance. This phenomenon can be attributed to the message passing programmers explicit control of data distribution and careful avoidance of communication contention and extra data movement. In NavP, the DSC Transformation involves data distribution and insertion of migration statements (i.e., hop()). The other two code transformations exploit parallelism by decomposing the long DSC threads and properly managing the synchronization among the shorter threads. The programmability of NavP is similar to that of HPF in that they both require explicit control of data distribution and explicit synchronization (through the use of barriers, events, critical regions, etc. in HPF, and events in NavP). Similar to HPF, synchronization errors are more likely to happen in NavP than in message passing. Unlike HPF, NavP requires its programmers to handle details in communication by using agent variables to carry data around. As a result, the NavP programmers know exactly how much is communicated to where at what time. NavP composes parallel code from shorter DSC threads, and the parallel code is structurally the same as the original sequential code. This property of NavP is referred to as Algorithmic Integrity [29].

Our NavP matrix multiplication implementation is faster than our MPI code. This is mainly because the NavP code successfully hides some of the communication overhead using an efficient but transparent run-time scheduling. This task scheduling functionality is factored out from the application code under the NavP view and put into the MESSENGERS daemon. Although it is entirely possible to achieve better task scheduling in the MPI code, with the MPI environment available today, the code that implements this will have to be developed for each and every application and will be interleaved with the application code. In this sense, message passing is harder to use than NavP.

Chapter 5

Preliminary Extension

5.1 Navigational Programming on Cloud

Fig. 5.1 shows the architecture of Navigational Programming [57]. The physical network could be computing machines connected with high speed Ethernet or cluster connected with fiber networks. constitutes the underlying computational processing elements.Superimposed on the physical layer is the daemon network, which is the NavP runtime system. Each daemon is responsible for receiving, processing, executing, dispatching, scheduling and synchronizing. In Java version of NavP, JaMes [57] runs on top of JVM. JaMes is a superset of Java, in that it provides a collection of methods that implement the navigational programming model. In addition to the basic capabilities of migration and non-preemptive scheduling, JaMes extends the basic model by allowing processes to collaborate both locally, through standard synchronization primitives, and remotely, by two different forms of send and receive operations.

The logical network is fully connected network on top of the daemon network, which is an application-specific computation network. NavP applications assume a fully connected logical network whose size is specified by the application programmer. JaMes maps logical nodes to daemon node and more than one logical nodes can be mapped to one daemon node.This makes NavP programs independent of available physical resources.

To start up the JaMes daemon network, the user first selects the physical nodes on which the system should run and designates one of these nodes as the master node, the rest being slave nodes. An application can be injected on any node that is running a daemon. The default is to choose a daemon on the physical node where the inject command is issued, but this can be modified by first calling the static method JaMes.setDestination(int dest), which causes the application to be injected on the node with the given rank. To shut down the JaMes daemon network, the user issues the shutdown command on the physical machine where the master node is running. This causes all daemons in the daemon network to stop running.

Migration is implemented by creating a remote object and then invoking a method on the newly created object This causes a remote object to be created on the destination node. The destination node specified by giving its rank, and object Class specifies the class of the object be created. The method is then invoked asynchronously on the destination node, and execution continues on the current node (similar to a fork). If the current node terminates (e.g., by reaching the end of the current function) the effect is a hop operation. If the current node continues executing, the effect is a clone operation.

Events within JaMes are represented by objects of class JMEvent. JaMes provides static functions to create an event with (JaMes.createJMEvent())

wait on one or more events with (JaMes.waitEvent()),

wait until one of a group of events has been signaled with (JaMes.selectWait())

and signal an event with (JaMes.signalEvent())

When an execution thread is waits for an event that has not yet been signaled it preempts itself, thus allowing another execution thread to continue running instead. JaMes also provides a function (JaMes.yield()) that causes a process to preempt itself.

Data transfer between executing threads running on different nodes takes two forms. One form corresponds to a conventional send-receive operation. The sender issues the command: JaMes.send(int dest, String mailboxName, Object data)

This causes the data to be placed in a mailbox on the remote machine. The receiver on the target machine issues the command

JMEvent JaMes.receive(String mailboxName, JMDataHandler dataHandler);

The handler specifies the name of a class that implements the JMDataHandler interface. This interface consists of a single method, handleShipment(), which contains user-provided code for unpacking the data. The data parameter on the send command consists of one or more objects. The receive command is non-blocking, but it returns an event on which the receiving thread can wait if it chooses to do so. This event will be signaled when the unpacking of the data is complete. The second form does not require an explicit matching receive. Instead, the receiving thread makes a single call, using the following syntax:

JMEvent JaMes.persistentReceive(String mailboxName, JMDataHandler dataHandler);

With this form, the mailbox is persistent in that multiple sends can be issued to it. The persistent receive command specifies the data handler that will be used to unpack data each time it arrives at the mailbox. It is the responsibility of the application programmer to synchronize the execution thread that is using the data with the data handler thread that is unpacking the data. To facilitate this synchronization, an event is returned by the call to the persistent receive function.

Therefore, the Java version of Navigational Programming, JaMes, has the characteristics of supporting heterogeneous infrastructure (software heterogeneous and hardware heterogeneous), and NavP applications are totally independent of available underlaying computation resources. On the Cloud, the physical nodes are no longer physical, but they are virtual nodes.

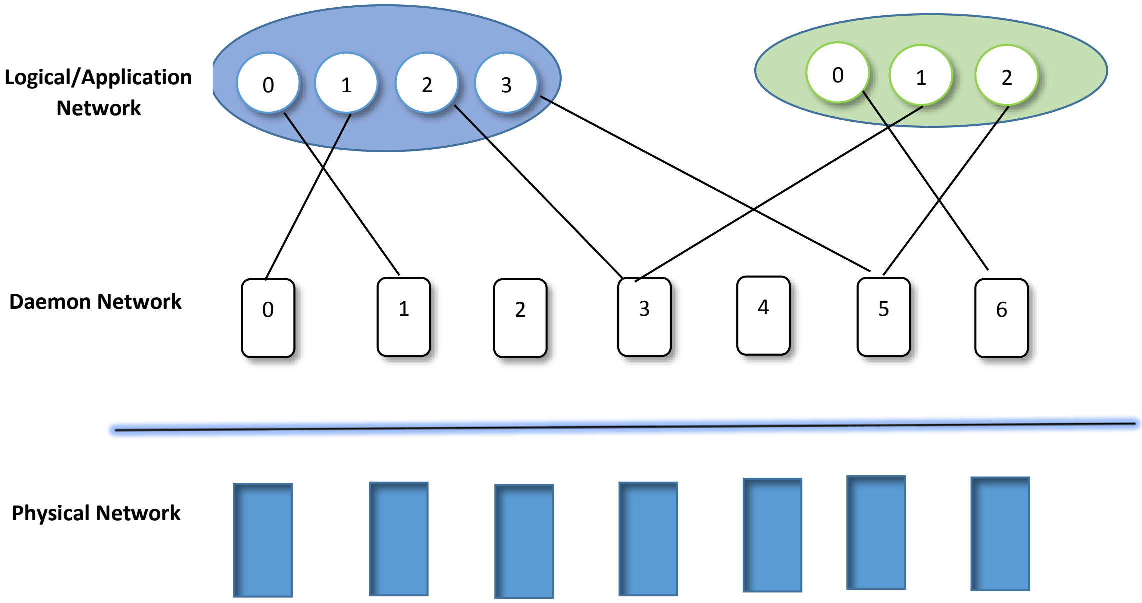


Figure 5.1: Navigational Programming Architecture

5.2 Investigation and Future Improvement

There are a number of commercial cloud providers such as: AmazonEC2, Microsoft Azure [7], Google Cloud [19], Rackspace [50] and Salesforce Service Cloud [55]. We investigates running NavP on Cloud with EC2 (Amazon Elastic Compute Cloud)[4]. Amazon EC2 provides resource management tool to configure computing instances, to allow users to setup customized instances, such as identifying the security group, and allow customers to create multiple instances with the same characteristics just once.

All the instances are created from US East (N. Virginia) data center of Amazon. Fig. 5.1 shows the preliminary performance results.

Number of Nodes	Array size	Speedup with UCI Linux	Speedup with Amazon EC2 Cloud
3	5040 × 5040	1.367048	1.888784
4	5040 × 5040	1.44899	2.012114
5	5040 × 5040	1.593074	1.870056
6	5040 × 5040	1.296688	2.191897
7	5040 × 5040	1.458795	1.670094
8	5040 × 5040	1.337289	1.421071

Table 5.1: ADI Performance on Cloud

This investigation is for proof-of-concept that NavP runs on cloud and provides an promising methodology for HPC applications. In order to make NavP an programming methodology for scientific applications on Cloud efficiently, we have to do a few improvements in near future.

First, we will improve the deployment of JaMes and Applications, to make it fully automatical deployment. There are two tasks we will address, one is the automated tools to setup Cloud environment and take cares of all of the steps required by Cloud provides, the other one is detecting the right set of instances, with have both the good quality of computing performance and relatively even and fast interconnect communication.Especially, for communication intensive scientific applications, the allocating or selecting the right set of virtual nodes have a great influential effect on overall speedup performance.

Second, since the diversity of latency and bandwidth, the network cost model needs to be improved to adapt the cloud environment. As [17] [70] observe, The network environment on Cloud is different from the network structure with cluster. On Cloud, the network infrastructure is hierarchic, a tree topology, machines are first grouped into racks, and then racks are connected with high speed switchers. The interesting of tree structure is that the bandwidth is not uniform, depends on the switching network between machines. Also, as [25] finds out that there is a strong bound between communication time and overall performance on EC2 resources, and application with intensive global communication are affected the most.

Third, due to performance variation of virtual nodes, Feedback mechanism will play more roles on detecting the slow nodes, monitoring the application programs real run on Cloud both with computational spending and communication cost. Since some Cloud provider, such as Amazon EC2, offers a feature of launching a snapshot (Images) and can be deployed to other instances no matter the same type or different types. JaMes could do reallocation/redeployment certain daemon node on the virtual node without applications' awareness. More, JaMes will optimize its synchronization mechanism to reduce the overhead.

Chapter 6

Related Work/Literature Review

6.1 Distributed Parallel Programming

Message Passing Interface (MPI) [35] [8] is the widely used standard for high-performance parallel programming. In concept, MPI is very simple because it could just includes a small number of primitives used directly by programmers. MPI-2 was released in 2000, adding few additional features such as one-sided communication based on remote memory access, parallel I/O. MPI-2.1 (2008) and MPI-2.2 (2009) were released with some corrections to the standard and small features. MPI-3 (2012) added several new features to MPI, such as new one sided functions and semantics, nonblocking collective communication, neighborhood collectives, MPI tool interface, improvements in Language Bindings, fault tolerance/resiliency. MPI just offers a standard communication interface and several high performance implementations such as MPICH [36] and OpenMPI [37] are implemented following the standards.

Important considerations while using MPI : ”All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs” [8]. The programmer must restructure any given sequential algorithm to make it work in a distributed memory environment by using MPI primitives.

NavP provides a different view of distributed computation from the classical SPMD (Single Program Multiple Data) view. The SPMD MPI view describes distributed computations at stationary locations, while the NavP view describes a computation following the movement of its locus. The NavP view changes the way distributed parallel programs are composed and provides some new benefits.

Although MPI grows with continues improvement and standards, its fundamental principles is not changed. basic use of MPI could be simplified down to six primitives, with MPI Send() and MPI Recv() being mostly used.

In NavP code, a programmer inserts navigational commands, i.e., hop() statements, to migrate the computation locus in order to access remotely distributed data and spread out computations. Small data is carried by the moving computation in agent variables, which are private to a computation thread and available to the thread wherever it migrates. Large data that stays on a computer is held in node variables that are resident on a particular PE (processing element) and are shared by all computation threads currently on that PE. The cost of a hop() is essentially the cost of moving the data stored in agent variables plus a small amount of state data. Although the state of the computation is moved on each hop, the code is not moved. The synchronization among different migrating computations is done through events (signalEvent() and waitEvent()). A programmer can inject, or spawn, a migrating thread at command line. The injection of a thread can also be done by another thread, called a spawner. All injections happen locally (i.e., a thread can spawn another thread only on the node on which it currently resides).

Navigational Programming (NavP) offers a different approach to generating a distributed parallel program from a sequential one, in that the transformation occurs incrementally and produces an executable program at each stage. It has two major advantages for distributed parallel programming:Incremental parallelism which it has a executable programs in the process of parallelizing from sequence code and increasing speedup performance, and Incremental performance improvement, which is a closed-loop system to feedback and control the speedup in next run. The feedback mechanism provides performance evaluations, including speedup and load balance, that can be used to adjust the output of the preceding steps.

6.2 Parallelizing Compilers

Parallelizing compilers target to parallel the programs fully and automatically. There are many existing high performance parallelizing compilers, such as SUIF [66], Parascope [27]. Many of of them are targeted at shared memory machines, and some compilers targets distributed memory computers. The techniques these compilers applied ranging from loop parallelization to complex inter-procedural analysis.

Although these compilers work comparatively well for regular applications, when meeting irregular application, these compiler can only handle limited patterns. Manuel Arenaz [6] addresses the automatic generation of parallel code in the scope of complex loop nests where today's parallelizing compilers fail. These examples focus on complex loops that contain computational kernels frequently found in real codes, namely, irregular assignment and consecutively written array.

Large number of parallelizing compilers focus on fine-grain parabolization such as loop transformation by exploring data parallelism transferring sequential loops to parallel loops.

Jingling Xue works proposed loop transformations and computing distribution first and Xue [68] , discussed the problem of the choice of the tiling parameters to solve the communication-minimal tiling optimally.

Du [11] proposed coarse-grained pipelined parallelism, the processing associated with an application is carried out in several stages. These stages are executed on a pipeline of computing units.

In contrast, NavP is a manual programming approach allows its programmers to exploit coarse level parallelism and develop irregular applications. A NavP program can also utilize parallelizing compilers and enjoy a speedup on every computer node. We believe that exploiting the relationship between the NavP view (including the NavP code transformations) and building parallelizing compilers is likely to be an important research direction in the near future.

To break the program into subcomputations we rely on the well-known techniques of tiling [23, 24, 52, 67, 68].

6.3 Cloud Computing

As Cloud computing provides a virtual and elastic infrastructure computing resources on demand, it attracts researchers to investigate the feasibility of running scientific high performance computing on Cloud [63] [38] [25] [22] [51] [3].

Luo [33] observes some Cloud system platforms have heterogeneous compute environments likelike AWS EC2 that provides both Intel Xeon and AMD Opteron while others have varying generations of CPUs.

Jackson [25] compares the cloud systems with real cluster by running HPC scientific applications. It indicates that the speed of interconnected networks is the major factor to affect overall performance.

Strazdins [61] evaluates the performance results for for a set of benchmark kernels (OSU MPI micro-benchmark, the NASA NAS macro-benchmarks and two large scientific application in climate and biology science). It finds out that communication bound applications, especially those which used short messages, needs to handle the disadvantage of interconnection on Cloud.

Roloff [54] compares different Cloud services of Rackspace, Azure, Amazon and a cluster in terms of deployment, performance and cost-efficiency, and provides comprehensive analysis of the above three important aspects, by running a set of well-known HPC benchmarks. The analysis shows that Cloud computing provides a viable infrastructure for running HPC applications, although it has some disadvantages in the deployment. For a number of benchmarks, the cloud infrastructure offers a higher performance and cost efficiency than the cluster. up to 27% and 41%, respectively. In terms of efficiency, it calculates by taking account of two factors: performance and cost.

Fan [14] investigates the communication intensive applications with Message Passing Interface (MPI), which is greatly affected by the network connections between the selected nodes, proposes clustering-based method to select cloud nodes for deploying MPI programs with communicate intensive applications, validates by deploying several well-known MPI programs on a real-world cloud. Moreover, Fan [13] proposes an automatic topology detection method,and a deployment method based on the topology information, that can improve the performance of a scientific application and validates with large scale real-world experiments.

Cloud computing provides a virtual infrastructure which hides the network topology and underlying computation resources. Gong [70] exams the network performance of Amazon EC2 and points out significant network performance unevenness (i.e., the performance varies significantly for different virtual machine pairs). Also, it observes that network performance of two virtual machines in Amazon is not symmetric, the communication cost of A to B might be varied from B to A on Cloud. Then, it proposes a network performance hierarchy to capture the network performance based on latency and bandwidth matrices.

Li [32] proposes an adaptive resource allocation algorithm for the cloud system by updating the actual task executions.

Hormozi [12] investigates using the machine learning methods to allocate and manage resources automatically, rely on machine learning techniques to efficiently decide the amount of resources necessary for the service.

Pawar [46] proposes an dynamic resource allocation algorithm to take account of preemptive task execution and multiple SLA parameters such as memory, network bandwidth, and required CPU time.

Chapter 7

Conclusion

7.1 Contribution

There are five major contributions of this dissertation.

1. This thesis presents a new methodology for incremental parallelism and incremental performance improvement. The methodology takes advantage of threads that are able to migrate through the network and thus are able to follow distributed data. This allows the data to be partitioned and distributed first, which guarantees that elements that are used together in a computation are collocated on the same node. Next, the loops in the code are tiled to minimize migration among nodes. After deciding on the location at which each loop is to execute, the necessary migration and remote access statements are inserted to make the code executable. This process is repeated based on feedback obtained from the execution, which may improve the overall performance by suggesting a different data distribution or a different coarseness of tiling.

2. We illustrate the truly incremental procedure in the context of matrix multiplication to show that each step represents a functioning program and every intermediate program is an improvement over its predecessor.As a result, no abrupt change in code will

happen between any consecutive steps;Every intermediate program is an improvement from its predecessor. If program development is limited by time or resources, any one of the intermediate programs can be taken as production code;Our final stage is similar to the classical Gentlemans Algorithm, The well-known message passing solution to the same problem. However, one either gets no parallelism at all with the sequential code, or one gets all parallelism with Gentlemans Algorithm. We also show the brief comparison of the two implementations and performance comparison to demonstrate the advantages of our methodology.

3. We use a case study of ADI, which is an abstraction of a method for solving a 2-dimensional heat equation and has characteristics of extensive both data communication and data computation, also strong data dependencies, to demonstrate incremental performance improvement. It starts with two data distribution patterns, builds the Weighted Dependence Graph which captures the essential dependency. We apply transitive reduction to the Weighted Dependence graph and apply heuristic bottom up approach to create create execution threads, and based on the feedback, we go back and change some decisions made during the earlier code transformation and placement phase or to change the data distribution. The implementation are built both on cluster and cloud environment.

4. A simulation tool is developed to execute a discrete-event simulation of the simplified Weighted Dependence Graph. These evaluations include the Total execution time, speedup, the idle time, the efficiency ratio, and load imbalance, and serve as guide to improve the respective transformations.

5. Examined/Investigated of Navigational Programming on cloud with ADI application and points out the future work to improve: a) auto deployment b) topology analysis/ Network cost analysis c) Feedback to dynamic reallocation/deployment (migrate node d) Improve efficiency of synchronization

7.2 Future Works

The long-term goal of the feedback loop is to be able to periodically improve the performance of the actual program by adjusting its current data distribution and level of parallelism at runtime based on information gathered automatically from the instrumented program.

One promising extension of our method is to eliminate the simulator and instead use the actual program. The feedback mechanism would then monitor the performance of the code, suggest a new data distribution and/or a new tiling strategy, and at some point cut over to the new code. This would represent incremental parallelism in the fullest sense of the phrase: a working program continually improving itself as it performs its task. The major challenges here are to efficiently gather runtime performance data and to manage the transition from the old program to the new program.

NavP is an promising methodology that will be applied on Cloud computing for scientific high performance computing. In order to make NavP an programming methodology run on Cloud efficiently, we have to do a few improvements in near future. 1) automatical deployment based on both the good quality of computing performance and interconnect communication cost; 2) improvement of communication cost model to reflect of topology analysis on Cloud; 3) Due to Feedback Mechanism's control return, reallocation/redeployment of certain daemon node on the virtual node without applications' awareness; 4) JaMes will optimize its synchronization mechanism to reduce the overhead, such as wait/signal and send/recevive mechanism.

Bibliography

[1] Alternating direction implicit method. `http://en.wikipedia.org/wiki/Alternatingdirectionimplicitmethod`.

[2] I. Ahmad, Y. Kwok, and M. Wu. Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In *Proceedings of the Second International Symposium on Parallel Architectures, Algorithms, and Networks*, 1996.

[3] Amazon inc. high performance computing (hpc) on aws. `http://aws.amazon.com/hpc-applications/`.

[4] Amazon ec2. `http://aws.amazon.com/ec2/`.

[5] T. Andronikos, F. M. Ciorba, P. Theodoropoulos, D. Kamenopoulos, and G. Papakonstantinou. Cronus: A platform for parallel code generation based on computational geometry methods. *J. Syst. Softw*, 81, August 2008.

[6] M. Arenaz, J. Tourino, and R. Doallo. Compiler support for parallel code generation through kernel recognition. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th IEEE International*, 2004.

[7] Microsoft's cloud platform —. `https://azure.microsoft.com`.

[8] P. Balaji. Advanced parallel programming with mpi. `http://www.mcs.anl.gov/~balaji/2013-06-16-isc-mpi.pptx`.

[9] L. S. Blackford, J. Choi, and et al. Scalapack users guide. *Society for Industrial and Applied Mathematics*, 1997.

[10] L. E. Cannon. *A cellular computer to implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, 1969.

[11] W. Du. Compiler support for exploiting coarse-grained pipelined parallelism. *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[12] M. K. A. Elham Hormozi, Hadi Hormozi and M. S. Javan. Using of machine learning into cloud environment managing and scheduling of resources in cloud systems. In *2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 2012.

[13] P. Fan, Z. Chen, J. Wang, Z. Zheng, and M. R. Lyu. Topology-aware deployment of scientific applications in cloud computing. In *2012 IEEE Fifth International Conference on Cloud Computing*, 2012.

[14] P. Fan, J. Wang, Z. Zheng, and M. R. Lyu. Toward optimal deployment of communication-intensive cloud applications. In *2011 IEEE 4th International Conference on Cloud Computing*, 2011.

[15] Y. Fann, C. Yang, S. Tseng, and C. Tsai. An intelligent parallel loop scheduling for parallelizing compilers. *Journal of Information Science and Engineering*, 16, 2000.

[16] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[17] R. D. e. a. G. Burns. Lam: An open cluster environment for mpi. In *In Proceedings of Supercomputing Symposium*, 1994.

[18] W. M. Gentleman. Some complexity results for matrix computations on parallel computers. *Journal of the ACM*, 1978.

[19] Google cloud computing. `https://cloud.google.com/`.

[20] G. Goumas, A. Sotiropoulos, and N. Koziris. Minimizing completion time for loop tiling with computation and communication overlapping. In *International Parallel and Distributed Processing Symposium (IPDPS 2001)*, San Francisco CA, April 2001.

[21] M. Hakem and F. Butelle. Critical path scheduling parallel programs on an unbounded number of processors. *International Journal of Foundations of Computer Science*, 17(2), April 2006.

[22] Q. He, S. Zhou, B. Kobler, D. Duy, and T. McGlynn. Case study for running hpc applications in public clouds. In *In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.

[23] E. Hodzic and W. Shang. On time optimal supernode shape. *IEEE Transactions of Parallel and Distributed Systems*, 13(12), December 2002.

[24] K. Högstedt, L. Carter, and J. Ferrante. On the parallel execution time of tiled loops. *IEEE Transactions on Parallel and Distributed Systems*, 14(3), March 2003.

[25] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, and et al. Performance analysis of high performance computing applications on the amazon web services cloud. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, 2010.

[26] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1998.

[27] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice and Experience*, 1993.

[28] K. Kyriakopoulos, A. T. Chronopoulos, and L. Ni. An optimal scheduling scheme for tiling in distributed systems. In *IEEE International Conference on Cluster Computing*, September 2007.

[29] L. F. B. L. Pan and M. B. Dillencourt. Distributed sequential computing using mobile code: Moving computation to data. In *Proceedings of the 2001 International Conference on Parallel Processing (ICPP 2001)*, September 2001.

[30] M. K. Lai. *State-Migration Shared-Variable Programming.* PhD thesis, UNIVERSITY of CALIFORNIA,IRVINE, 2009.

[31] P. Lee and Z. M. Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(1), January 2002.

[32] J. Li, M. Qiu, J.-W. Niu, Y. Chen, and Z. Ming. Adaptive resource allocation for preempt able jobs in cloud systems. In *10th International Conference on Intelligent System Design and Application*, January 2011.

[33] W. Luo, Golpavar, C. N., and A. C., Chronopoulos. Benchmarking joyent smartdatacenter for hadoop mapreduce and mpi operations. In *Proceedings of 2013 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2013.

[34] Messengers project. `http://www.ics.uci.edu/~bic/messengers`.

[35] Message passing interface. `http://www.mpi-forum.org`.

[36] Message passing interface ch. `http://www.mcs.anl.gov/mpi/mpich2`.

[37] Open source high performance computing. `http://www.opem-mpi.org`.

[38] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D.Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *In Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, 2010.

[39] L. Pan. *Navigational Programming: Toward Structured Distributed Parallel Programming.* PhD thesis, UNIVERSITY of CALIFORNIA,IRVINE, 2005.

[40] L. Pan, L. F. Bic, M. B. Dillencourt, and M. K. Lai. NavP versus SPMD: Two views of distributed computation. In *IASTED International Conference on Parallel and Distributed Computing and Systems (PCDCS 2003)*, Marina Del Rey CA, November 2003.

[41] L. Pan, M. K. Lai, M. B. Dillencourt, and L. F. Bic. Mobile pipelines: Parallelizing left-looking algorithms using Navigational Programming. In *12th International Conference on High Perfomance Computing (HiPC 2005)*, Goa, India, December 2005.

[42] L. Pan, M. K. Lai, K. Noguchi, J. J. Huseynov, L. F. Bic, , and M. B. Dillencourt. Distributed parallel computing using navigational programming. *International Journal of Parallel Programming*, 32(1), February 2004.

[43] L. Pan, J. Xue, M. B. Dillencourt, and L. F. Bic. Toward automatic data distribution for migrating computations. In *International Conference on Parallel Processing (ICPP 07)*, Xian, China, September 2007.

[44] L. Pan, W. Zhang, A. Asuncion, M. K. Lai, M. B. Dillencourt, , and L. F. Bic. Incremental parallelization using navigational programming: A case study. In *International Conference on Parallel Processing (ICPP 2005)*, Oslo, Norway, June 2005.

[45] L. Pan, W. Zhang, A. Asuncion, M. K. Lai, M. B. Dillencourt, and L. F. Bic. Incremental parallelization using navigational programming: A case study. School of Information and Computer Sciences Technical Report TR 05-04, University of California, Irvine, Irvine, CA, March 2005.

[46] C. S. Pawar and R. B. Wagh. Priority based dynamic resource allocation in cloud computing with modified waiting queue. In *2013 IEEE International Conference on Intelligent Systems and Signal Processing (ISSP)*, March 2013.

[47] A. P. Petitet and J. J. Dongarra. Algorithmic redistribution methods for block-cyclic decompositions. *IEEE Transactions on Parallel and Distributed Systems*, 1999.

[48] N. Petkov. *Systolic Parallel Processing*. Elsevier Science Publishers, Amsterdam, North-Holland, 1993.

[49] M. J. Quinn. *Parallel computing theory and practice*. Wadsworth Brooks Cole Advanced Books Software, McGraw-Hill, 1994.

[50] rackspace. `https://www.rackspace.com`.

[51] L. Ramakrishnan, R. S. Canon, K. Muriki, I. Sakrejda, and N. J.Wright. Evaluating interconnect and virtualization performance for high performance computing. In *SIGMETRICS Performance Evaluation Review 40(2)*, 2012.

[52] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for non-shared memory machines. In *ACM/IEEE Conference on Supercomputing (SC 1991)*, Albuquerque NM, November 1991.

[53] F. Rastello, A. Rao, and S. Pande. Optimal task scheduling to minimize inter-tile latencies. *Parallel Computing*, 29(2), 2003.

[54] E. Roloff, M. Diener, A. Carissimi, and P. O. A. Navaux. High performance computing in the cloud:deployment, performance and cost efficiency. In *2012 IEEE 4th International Conference on Cloud Computing Technology and Science*, 2012.

[55] Salesforce service cloud. `https://www.salesforce.com/`.

[56] S.Ha and E.A.Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7), 1997.

[57] Q. Shang, M. Fukuda, W. Zhang, L. Bic, and M. B. Dillencourt. James: A java-based system for navigational programming. In *Computational Problem-Solving (ICCP), 2011 International Conference on*, 2011.

[58] D. Shiao. Mobile agents: A new model of intelligent distributed computing. *IBM Developer Works*, 2004.

[59] T. Shindo, H. Iwashita, S. Kaneshiro, T. Doi, and J. Hagiwara. Twisted data layout. In *8th ACM International Conference on Supercomputing (ICS 1994)*, Manchester UK, July 1994.

[60] O. Sinnen. *Task scheduling for parallel systems.* Wiley-Interscience, May 2007.

[61] P. E. Strazdins, J. Cai, M. Atif, and J. Antony. Scientific application performance on hpc, private and public cloud resources: A case study using climate, cardiac model codes and the npb benchmark suite. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum*, 2012.

[62] J. C. STRIKWERDA. *The Alternating Direction Implicit (ADI) Method.* Wadsworth Brooks/Cole Advanced Books Software, Pacific Grove, Calif, 1989.

[63] H. J. e. a. Subhash Saini, Steve Heistand. An application-based performance evaluation of nasas nebula cloud computing platform. In *2012 IEEE 14th International Conference on High Performance Computing and Communications*, 2012.

[64] T. Thanalapati and S. Dandamudi. An efficient adaptive scheduling scheme for distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 12(7), 2001.

[65] J. L. Traff and A. Ripke. Optimal broadcast for fully connected networks. In *High Performance Computing and Communications (HPCC 2005)*, Sorrento, Italy, September 2005.

[66] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, and et al. The suif compiler system: a parallelizing and optimizing research compiler. *Technical Report CSL-TR-94-620, Stanford University, Computer Systems Laboratory, Stanford*, 1994.

[67] M. Wolfe. More iteration space tiling. In *ACM International Conference on Supercomputing (ICS 1989)*, Reno NV, November 1989.

[68] J. Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Norwell, MA, 2000.

[69] T. Yang and A. Gerasoulis. DSC: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions of Parallel and Distributed Systems*, 5(9), September 1994.

[70] J. Z. Yifan Gong, Bingsheng He. Network performance aware mpi collective communication operations in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, PP(99), March 2013.

[71] W. Zhang, L. Pan, Q. Shang, L. F. Bic, and M. B. Dillencourt. Incremental parallelization with migration. In *IIn Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages pp. 223–230, 2012.