

UNIVERSITY OF CALIFORNIA,  
IRVINE

State-Migration Shared-Variable Programming  
and its Runtime Support for  
Fine-Grained Cooperative Tasks

DISSERTATION

submitted in partial satisfaction of the requirements  
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Ming Kin Lai

Dissertation Committee:  
Professor Lubomir F. Bic, Co-Chair  
Professor Michael B. Dillencourt, Co-Chair  
Professor Alexandru Nicolau

2009



The dissertation of Ming Kin Lai  
is approved and is acceptable in quality and form for  
publication on microfilm and in digital formats:



---



---

Committee Co-Chair



---

Committee Co-Chair

# DEDICATION

To  
my family, particularly my brother

# TABLE OF CONTENTS

	Page
<b>LIST OF FIGURES</b>	<b>vii</b>
<b>LIST OF TABLES</b>	<b>ix</b>
<b>ACKNOWLEDGMENTS</b>	<b>x</b>
<b>CURRICULUM VITAE</b>	<b>xi</b>
<b>ABSTRACT OF THE DISSERTATION</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.1.1 Programming model . . . . .	1
1.1.2 Run-time support . . . . .	2
1.2 Contribution . . . . .	5
1.3 Overview and organization . . . . .	6
<b>2 MESSENGERS Programming Model</b>	<b>7</b>
2.1 Distributed programming using program migration and location-scoped variables . . . . .	8
2.1.1 Location and migration of program . . . . .	8
2.1.2 State-migration programming . . . . .	9
2.1.3 State-migration programming vs message passing . . . . .	10
2.1.4 Location-scoped variables . . . . .	17
2.1.5 State-migration location-scoped-variable programming . . . . .	18
2.1.6 State-migration location-scoped-variable programming vs message passing . . . . .	19
2.1.7 Migration during execution of subprogram . . . . .	21
2.2 Concurrent programming using multiple tasks sharing variables . . . . .	24
2.2.1 Task . . . . .	24
2.2.2 Shared-variable programming . . . . .	25
2.3 MESSENGERS programming model . . . . .	25
2.3.1 Messenger migration . . . . .	26
2.3.2 Physical node and logical node . . . . .	26
2.3.3 Messenger migration on logical nodes and physical nodes . . . . .	28

2.3.4	MESSENGERS language . . . . .	30
2.3.5	Dynamic binding of node variables . . . . .	32
2.3.6	Scheduling of Messengers . . . . .	35
2.3.7	Computation and data distribution using MESSENGERS . . . . .	36
2.3.8	Summary . . . . .	38
<b>3</b>	<b>Original MESSENGERS System</b>	<b>40</b>
3.1	MESSENGERS compiler . . . . .	41
3.2	Single-CPU MESSENGERS runtime . . . . .	45
3.2.1	Implementation of logical nodes and Messengers . . . . .	46
3.2.2	Overview of execution of daemon . . . . .	49
3.2.3	Formation of daemon network . . . . .	50
3.2.4	Injection of Messenger . . . . .	51
3.2.5	Ready queue and scheduling of Messengers . . . . .	52
3.2.6	Execution of Messenger . . . . .	53
3.2.7	References to node variables . . . . .	55
3.2.8	Multithreading of daemon and sending/receiving queues . . . . .	57
3.2.9	Support for fine-grained tasks and migration . . . . .	57
<b>4</b>	<b>Case Study of MESSENGERS Programming</b>	<b>59</b>
4.1	Mapping of physical and logical nodes on network . . . . .	63
4.2	Mapping of physical and logical nodes on uni-processor computer . . . . .	64
<b>5</b>	<b>Vertical Scaling and Multithreading</b>	<b>71</b>
5.1	Vertical and horizontal scaling . . . . .	71
5.2	Processor support for multiple CPUs on computer . . . . .	73
5.2.1	Symmetric multiprocessing . . . . .	73
5.2.2	Chip multiprocessing . . . . .	74
5.2.3	Simultaneous multithreading . . . . .	75
5.3	Operating system support for multiple CPUs and multithreading . . . . .	76
5.3.1	General . . . . .	76
5.3.2	Support by Linux . . . . .	78
5.3.3	Impact of asymmetric multi-cores . . . . .	80
5.3.4	Impact of NUMA topology . . . . .	81
5.3.5	Binding process/thread to CPU . . . . .	81
<b>6</b>	<b>Multiple-CPU MESSENGERS Runtime System</b>	<b>83</b>
6.1	Inadequacy of single-CPU MESSENGERS runtime . . . . .	83
6.2	Flow of control: process vs thread . . . . .	84
6.2.1	Scaling using processes . . . . .	85
6.2.2	Parallelization using threads . . . . .	85
6.3	Implementation of multiple-CPU MESSENGERS runtime using multiple processes . . . . .	86
6.4	Implementation of multiple-CPU MESSENGERS runtime using multiple threads . . . . .	91

6.4.1	Domain decomposition of MESSENGERS runtime program . . . . .	91
6.4.2	Possible choices for domain . . . . .	92
6.4.3	Implementation . . . . .	93
6.5	Performance evaluation . . . . .	97
6.6	Discussion . . . . .	103
<b>7</b>	<b>Related Work</b>	<b>104</b>
7.1	Navigational programming . . . . .	104
7.2	Thread migration systems . . . . .	106
7.3	Forms of mobility . . . . .	109
7.4	Mobile agents and strong/weak mobility . . . . .	110
7.5	High Performance Fortran . . . . .	113
7.5.1	Migration of abstract processors . . . . .	115
7.5.2	Hierarchy of abstract processors on one computer . . . . .	115
<b>8</b>	<b>Future Work</b>	<b>117</b>
8.1	Hierarchy of physical and logical nodes . . . . .	117
8.2	Static binding of node variables . . . . .	117
8.3	Additional experiment on computers with larger number of CPUs and/or asymmetric CPUs . . . . .	118
8.4	Data redistribution by migrating physical and/or logical nodes . . . . .	118
<b>9</b>	<b>Conclusion</b>	<b>120</b>
	<b>Bibliography</b>	<b>121</b>

# LIST OF FIGURES

	Page
2.1 Program migration . . . . .	9
2.2 Program transformation - sequence, using state-migration programming	11
2.3 Program transformation - selection, using state-migration programming	11
2.4 Program transformation - repetition, using state-migration programming	12
2.5 Message-passing programming vs programming using goto's . . . . .	14
2.6 Program modifiability - state-migration programming vs message-passing	16
2.7 Location-scoped variables . . . . .	17
2.8 Saving the value of a location-scoped variable into a migratable variable	18
2.9 Program transformation - sequence, using state-migration location-scoped-variable programming. . . . .	20
2.10 Program transformation - selection, using state-migration location-scoped-variable programming. . . . .	21
2.11 Program transformation - repetition, using state-migration location-scoped-variable programming . . . . .	22
2.12 Migratable and location-scoped variables of a subprogram in a state-migration location-scoped-variable program . . . . .	23
2.13 Locations, physical nodes and logical nodes . . . . .	27
2.14 Messenger migration with respect to logical nodes and physical nodes	29
2.15 A simple Messenger definition and a simple node file . . . . .	30
2.16 Location-based scope for location-scoped variables . . . . .	33
2.17 Using the MESSENGERS programming model for computation and data distribution . . . . .	38
2.18 Incremental partitioning of computation and data using the MESSENGERS programming model . . . . .	39
3.1 Translation of declarations of Messenger variables to the definition of a structure by the MESSENGERS preprocessor . . . . .	42
3.2 Translation of a Messenger definition into a Messenger C program consisting of <code>_msggr_functions</code> . . . . .	43
3.3 Initialization functions in a Messenger C program . . . . .	44
3.4 Three levels of networks in the MESSENGERS execution environment	45
3.5 <code>struct NCB</code> and <code>struct NVA</code> . . . . .	47
3.6 <code>struct MCB</code> . . . . .	48
3.7 Simplified definition of <code>run_daemon()</code> in the runtime program . . . . .	49

3.8	Definition of <code>create_mcb()</code> in the runtime program . . . . .	52
3.9	Definition of <code>exec_msgs()</code> in the runtime program . . . . .	53
3.10	Definition of <code>exec_mcb()</code> in the runtime program . . . . .	54
3.11	Definitions of functions sending a MCB to an remote daemon . . . . .	56
4.1	Sequential program for Crout factorization . . . . .	60
4.2	Working set of an iteration of the outermost loop in the Crout factorization algorithm . . . . .	60
4.3	Definition of Messenger for Crout factorization . . . . .	61
4.4	Mapping of physical and logical nodes on a network of uni-processor computers . . . . .	63
4.5	Mapping of physical and logical nodes on a uni-processor computer . . . . .	68
6.1	Address spaces of the processes running the daemons on a dual-CPU computer with the multiple-CPU MESSENGERS runtime . . . . .	90
6.2	Three levels of networks in the MESSENGERS execution environment when two daemons, each embeds a logical node, are created on a dual-CPU computer. . . . .	91
6.3	<code>struct NCB</code> and <code>struct node_ready_queue</code> . . . . .	95
6.4	Definition of <code>run_node()</code> in the multiple-CPU MESSENGERS runtime program with multiple computation threads . . . . .	96
6.5	Migration of an MCB from (the ready queue of) one logical node to (the ready queue of) another. . . . .	96
6.6	Definition of <code>exec_msgs()</code> in the new runtime program. . . . .	97
6.7	Address space of a multithreaded process running the daemon on a dual-CPU computer with the multiple-CPU MESSENGERS runtime . . . . .	98
6.8	Three levels of networks in the MESSENGERS execution environment when one daemon, consisting of two computation threads each embedding a logical node, is created on a dual-CPU computer. . . . .	98
6.9	Mapping of physical and logical nodes on a multiprocessor computer . . . . .	102

# LIST OF TABLES

	Page
4.1 Block distribution of a $3000 \times 3000$ upper triangular matrix, each entry of which is 4 bytes large, to 15 logical nodes each allocated 200 columns	66
4.2 Block-cyclic distribution of a $3000 \times 3000$ upper triangular matrix, each entry of which is 4 bytes large, to 2 logical nodes using a block size of 200 columns . . . . .	67
4.3 Performance of running the MESSENGERS program for Crout factorization on a uniprocessor Sun Ultra-5 workstation with 2 MB cache .	69
6.1 Performance of running the MESSENGERS program for Crout factorization on a Sun Fire V210 workstation with two UltraSPARC III-i processors . . . . .	100
6.2 Performance of running the MESSENGERS program for Crout factorization on a TYAN computer with two AMD Opteron 248 processors	100
6.3 Performance of running the MESSENGERS program for Crout factorization on a HP computer with two dual-core AMD Opteron 270 processors . . . . .	100
6.4 Performance of running the MESSENGERS program for Crout factorization on a Sun Fire V210 workstation with two UltraSPARC III-i processors . . . . .	101
6.5 Performance improvement in running the MESSENGERS program for Crout factorization by using two logical nodes over one logical node per physical node with the multi-process version of the MESSENGERS runtime on a dual-processor Sun Fire V210 workstation . . . . .	101
6.6 Performance improvement in running the MESSENGERS program for Crout factorization by using two logical nodes per physical node with the multi-process version over (using two logical nodes with) the multi-computation-thread version of the multiple-CPU MESSENGERS runtime on a dual-processor Sun Fire V210 workstation . . . . .	102

# ACKNOWLEDGMENTS

I would like to thank my advisors Professors Lubomir Bic and Michael Dillencourt for their advices, insight, and encouragement throughout this work. The MESSENGERS group led by them is an easy-going and pleasant environment that fosters free and independent research. They are also nice people with great sense of humor.

I feel privileged to have Professor Alex Nicolau on my dissertation committee, and have Professors Lichun Bao of the Department of Computer Science and David A. Smith of the Department of Sociology on my PhD Candidacy Advancement Committee. I thank each of them for their time and helpful comments.

My thanks also go to the past and current members of the MESSENGERS group, including Lei Pan, Koji Noguchi, Munehiro Fukuda, Christian Wicke, Wenhui (Wendy) Zhang, Matthew L. Badin, and Hairong Kuang. I am grateful to Munehiro and Christian for being instrumental in building the MESSENGERS system from which all of us benefit for a long period of time.

I definitely would thank my family. Kai Huen Lai, my father, Mo Ching Leung, my mother, Ming Bik Lai, my sister, and especially Ming Lap Lai, my beloved brother, are always in my mind. So are my grandma, Wai Ying Leung, and my great aunt, Wai Ling Leung. I owe many thanks to my friends, in particular Raymond Ka Sing Leung, Yi Tong Tse, George William Huntley III, and Hoi Wong. Indeed, I am indebted to all those who have helped and supported me.

Lastly, I thank the U. S. Government for supporting me through the Graduate Assistantship in Areas of National Needs (GAANN) during most of my study.

# CURRICULUM VITAE

**Ming Kin Lai**

## **Education**

1981–1985	The Chinese University of Hong Kong Hong Kong
December 1985	Bachelor of Business Administration (Honours) The Chinese University of Hong Kong
1987–1988	The University of North Carolina at Chapel Hill Chapel Hill, North Carolina, U.S.A.
December 1988	Master of Accounting The University of North Carolina at Chapel Hill
1993–1995	California State University, Los Angeles Los Angeles, California, U.S.A.
1999–2000	The University of Georgia Athens, Georgia, U.S.A.
2000–2009	University of California, Irvine Irvine, California, U.S.A.
December 2003	Master of Science University of California, Irvine
June 2009	Doctor of Philosophy University of California, Irvine

## **Selected awards**

1994	Lewis & Urner Scholarship, California State University, Los Angeles
1995	Charles Clark Scholarship, California State University, Los Angeles
2000–2007	Graduate Assistantship in Areas of National Needs, U.S. Department of Education

# ABSTRACT OF THE DISSERTATION

State-Migration Shared-Variable Programming  
and its Runtime Support for  
Fine-Grained Cooperative Tasks

By

Ming Kin Lai

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2009

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

This dissertation dissects a programming model that makes concurrent programming on a network of computers easier than using message passing, expands the understanding of this model by clarifying its concepts, shows how it is used to improve data reuse in the cache on a uni-processor, and demonstrates, by using two implementations of a runtime system, that it also supports programming on a multiple-CPU computer.

Message passing is the predominant programming model for distributed concurrent programming. An alternative programming model proposed in previous works, called MESSENGERS in this dissertation, was enhanced and defined. The model was improved to differentiate and expand the roles played by two abstractions in the model, namely, physical node and logical node. An in-depth comparison with the message passing model was analyzed to offer insight on the ease-of-programming advantage of this MESSENGERS model.

A language, a compiler and a runtime system were developed in prior works to support programming fine-grained cooperative tasks under the MESSENGERS model.

Whereas prior efforts focused upon using MESSENGERS on a network of uni-processor computers leveraging its increased processing and memory power, this dissertation shows how to take advantage of MESSENGERS's cooperative scheduling to improve data reuse in the cache of a single computer.

The runtime system was modified to demonstrate that this model can be used to benefit from multiple processors or cores often found in contemporary computers. Two different versions of the runtime system were implemented. One creates as many processes (by creating multiple instances of the runtime) as there are CPUs on a computer, with a single computation thread in each process. The other creates as many threads as there are CPUs, all within the runtime process on the multiple-CPU computer. These versions were analyzed on different platforms and they achieved satisfactory improvement over the original runtime. A program running on the version using multiple processes was able to benefit from the technique that improves cache performance and thus achieve better speedup.

# Chapter 1

## Introduction

### 1.1 Motivation

#### 1.1.1 Programming model

Message passing is the predominant programming model for concurrent programming on distributed computer systems. Among many alternative models proposed by researchers, there exists one that is based on the migration of concurrent tasks and the dynamic binding of variables based on locations. This model, called MESSENGERS in this dissertation, allows the programmer to transform a sequential program to a program under this model more easily than to a message-passing program, and such a program can be more easily transformed in order to adapt to different computation distribution schemes than a message-passing program can. This model can be uniformly used for shared-memory and distributed-memory programming. However, concepts in the MESSENGERS model had not been clearly defined. No attempts had been made to apply it on a single computer in a way that can improve data reuse

in the cache. Nor were there attempts to explain how this model can be applied to a computer with multiple CPUs so that these CPUs can be adequately utilized.

### 1.1.2 Run-time support

There were systems such as Ariadne that uses Standard C Library functions such as `setjmp()` and `longjmp()` (see Engelschall's paper [18], for example, for details), and Arachne that uses an alternative approach, to capture and restore the state of a thread. These systems send the captured state of a migrating thread at the source machine and receive and restore it at the destination machine of the same network. A model based on task migration is therefore supported. A thread, which can be viewed as a particular realization of a task, can then migrate to a remote computer to exploit data locality or to balance workload.

Multiprocessing has become the mainstream processor architecture for both server and desktop systems. Multithreading and technologies based thereon have been the conventional method to utilize the multiple CPUs on these machines. On a single-CPU machine, creating multiple threads allows the overlap of computation and communication, as well as the expression of logical concurrency. On a multiple-CPU machine, that logical concurrency becomes physical parallelism and the CPUs can be utilized.

As more and more processors and/or cores are put into a computer, it is difficult for current technology to allow each processor or core to access all the memory locations in the computer with equal amounts of time. In other words, the trend for the multiprocessors and multi-cores is toward non-uniform memory access (NUMA). In these emerging NUMA architectures, thread migration would be useful, to exploit data locality, for instance, as it is on a network of computers. Programmer-directed

thread migration within a computer can be accomplished by system calls recently added to various operating systems. For example, the Linux 2.6 kernel provides the `sched_setaffinity()` system call that allows the application programmer to designate the CPU on which the calling thread is to be executed. By calling this system call with a CPU different from the one the thread is currently executed on as parameter, thread migration is achieved.

Consequently, the migration of tasks both within a computer and across the network of computers can be made possible by a system that combines the techniques mentioned above. However, a large number of threads on a computer hurt performance because of the overhead associated with managing them. For program development purposes, the programmer may need to express concurrency by a large number of threads, but for performance purposes, only as few threads as the number of CPUs may be desired.

In regard to threads at least, Gillespie [27] asserted the following:

The speedup available by threading a specific region of code depends in part upon how many threads that region generates. Typically, it is desirable to create no more threads than the available processor resources can accommodate simultaneously. Thus, for a two-processor quad-core system that supports Hyper-Threading Technology, the maximum number of desirable threads would be 16 (2 processors  $\times$  4 cores per processor  $\times$  2 virtual processors per core). This is an important consideration, because creating more threads than can practically be used by the execution hardware generates overhead in creating threads that will not provide additional performance benefit.

Many concurrent programming models and languages support dynamic creation of tasks. These tasks express the logical concurrency in the program. The level of

logical concurrency, namely, the number of tasks, often exceeds the level of physical parallelism, namely, the number of machine instructions actually executed in parallel. Despite the inefficiency of running threads in a number much larger than the number of CPUs available, it is more logical from the perspective of software development to divide a program into a large number of tasks based on their functionality than just a handful based on the number of CPUs.

Norman and Thanisch [50] defined a fine-grained parallel computation as one that “comprises a large number of modules in relation to the number of processors, each of which takes a relatively small amount of time to execute.” In consistency with this definition, the tasks that outnumber the processors are referred to as fine-grained tasks. When these tasks are implemented as threads, they are often called fine-grained threads.

There may be considerable costs associated with the time to initialize a thread and switch between thread contexts, the space occupied by its stack, etc. Therefore, the large number of fine-grained threads prompted research [29, 28, 64] into ways to reduce these costs.

Wicke and others developed a compiler and a runtime system that support task migration on a network. These tasks are not implemented as threads. Rather, the number of threads are statically determined and the tasks which are dynamically created are mapped to these threads. This MESSENGERS system has the potential to allow efficient execution of fine-grained tasks, at the price of tight coupling between the compiler and the runtime, and of using an approach that is more involving than one that takes advantage of certain system calls provided by the operating system, and Standard C Library functions, as described above, for the purposes of migrating tasks on homogeneous computers. This runtime system, however, was not able to adequately utilize all the multiple CPUs on a contemporary computer.

## 1.2 Contribution

As discussed above, there is a need for a programming model and a system to support the migration of fine-grained threads, or fine-grained tasks in general, both within a multiple-CPU computer and on a network, so that logical concurrency can be fully expressed without hurting performance. Past works on MESSENGERS were limited to supporting such programming on a network of uniprocessors.

By explaining it using concepts in the study of programming languages, I expanded the understanding of the MESSENGERS model for such programming, and compared it in detail with the message-passing model. This dissertation offers a description of this MESSENGERS programming model, with improvements thereof, in more rigorous terms than previous works did.

Using new interpretation of two abstractions in the MESSENGERS model, namely, physical node and logical node, I demonstrated that one can improve data reuse in the cache by manipulating the number of logical nodes on a single computer, with certain data distribution schemes.

By modifying the runtime system, guided by the improved MESSENGERS model, I demonstrated that it can support programming on a computer with multiple processors or cores. I implemented and assessed two approaches to developing a MESSENGERS runtime that utilizes multiple CPUs on a computer, one using multiple processes in a way consistent with the interpretation of physical node and logical node, and an alternative one using multiple threads. The multi-process implementation performed better when used with the technique that relies on the enhanced MESSENGERS model to improve cache utilization.

## 1.3 Overview and organization

The MESSENGERS programming model is derived from the distributed model of state-migration location-scoped-variable programming and the concurrent model of shared-variable programming, as explained in Chapter 2. The original MESSENGERS system is described in Chapter 3. A scientific-computing application kernel, Crout factorization, is presented in Chapter 4 to illustrate how the MESSENGERS model can be used to develop a distributed concurrent program for a network of computers and to efficiently utilize the cache in a uni-processor. A brief review of the multiprocessing technologies follows, in Chapter 5. Different approaches to developing a runtime system to support MESSENGERS programming so that multiple CPUs on a computer can be utilized are discussed in Chapter 6. A couple of these designs are implemented and these implementations on three different platforms are assessed. I review related work and offer possible directions for further research in Chapter 7 and Chapter 8 respectively. Finally the contribution of this dissertation is summarized in Chapter 9.

## Chapter 2

# MESSENGERS Programming Model

It is useful to bear in mind what a programming model means. In Skillicorn and Talia [63]’s words,

A model of parallel computation is an interface separating high-level properties from low-level ones. More concretely, a model is an abstract machine providing certain operations to the programming level above and requiring implementations for each of these operations on all of the architectures below. ... models exist at many levels of abstraction. For example, every programming language is a model in our sense, since each provides some simplified view of the underlying hardware.

In the following, a distributed concurrent programming model, based on the migration of tasks synchronized and communicating via location-scoped variables, will be presented.

## 2.1 Distributed programming using program migration and location-scoped variables

### 2.1.1 Location and migration of program

A program can contain loops and subprograms, that can be called multiple times, and therefore each statement in the program may have multiple execution instances, each is called a statement instance.

In non-distributed programming, there is only one dimension, namely, time, for a program. That is, if there is an attribute, called location, associated with a statement instance, then the values of this attribute for all statement instances in the program are the same.

One can add a spatial dimension to a program so that a program can span more than one location, meaning that the location attributes for some statement instances in the (distributed) program have one value, whereas those for others have other values, for example.

A statement instance is said to be associated with a location if the location attribute for that statement instance has that location as its value. And the statement instance is said to be computed at the location associated with it. If one statement instance in a program is associated with one location and the statement instance following it is associated with another location, then it can be said that there is a change of location in the program. Alternatively speaking, the program migrates from one location to another. The destination location of the migration is the new location after the change of location.

A distributed program's current location is the location associated with the statement

instance being executed. It can be said that a program is (located) on its current location.

### 2.1.2 State-migration programming

If the migration, i.e., change of location, of a program is accomplished by a statement, namely, a migration statement, then obviously this statement would make reference to the migration destination. The result of executing this statement is that the location attribute of all the statement instances thereafter has a new value — the destination location, up to the point where another migration statement changes the location again.

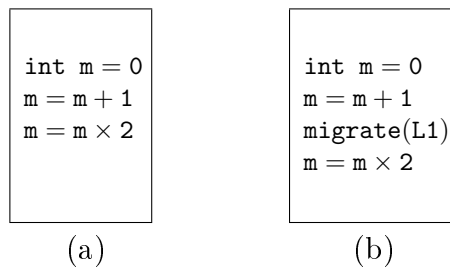


Figure 2.1: State-migration programming. (a) Non-distributed program. (b) Distributed state-migration program.

Figure 2.1 illustrates the idea of placing a `migrate()` statement in a program to change the location of the program. In Fig. 2.1(b), assumed that the program starts at Location L0, the addition statement is computed at L0. After the `migrate(L1)` statement, the multiplication statement is computed at Location L1. It is assumed that the same `m` is visible to (i.e., can be referenced by) the program irrespective of its current location.

*State-migration programming* is the distributed programming technique with which a program migrates, using a migration statement, among locations at any of which all

program variables are visible, subject to scoping rules if subprograms exist.

### 2.1.3 State-migration programming vs message passing

A popular programming model for distributed (and concurrent) computing is message passing. In the message-passing model, a program is explicitly divided into multiple parts each is stationary on a location, that is, each part is always associated with one particular location, and there is no change of locations in each part. Each part communicates and synchronizes with another by sending (using a `send` statement) and receiving (using a `receive` statement) messages. On the other hand, each part of the program has its own distinct non-overlapping namespace for variables. That is, no variable is visible to more than one part of a message-passing program. Message Passing Interface (MPI) [46, 47], a widely used interface for distributed concurrent programming, is a typical implementation of the message-passing model. As will be shortly elaborated in this section, a rank in MPI in effect refers to a location.

An important difference between state-migration programming and message-passing programming is that the `migrate()` statement takes only location-related parameter(s) — the destination location and possibly the route(s) used to reach that location, for instance, whereas the `send()` and `receive()` statements specify the data to be sent and received respectively, in addition to the source and destination locations. Another difference is that all variables in a state-migration program are visible to the program, subject to scoping rules if subprograms exist, whereas in message-passing, variables at a location are visible only to the part of the program on that location.

Figures 2.2 through 2.4 illustrate how non-distributed programs with sequence, selection, and repetition control structures can be transformed into programs distributed over two locations using state-migration programming, and then further into message-

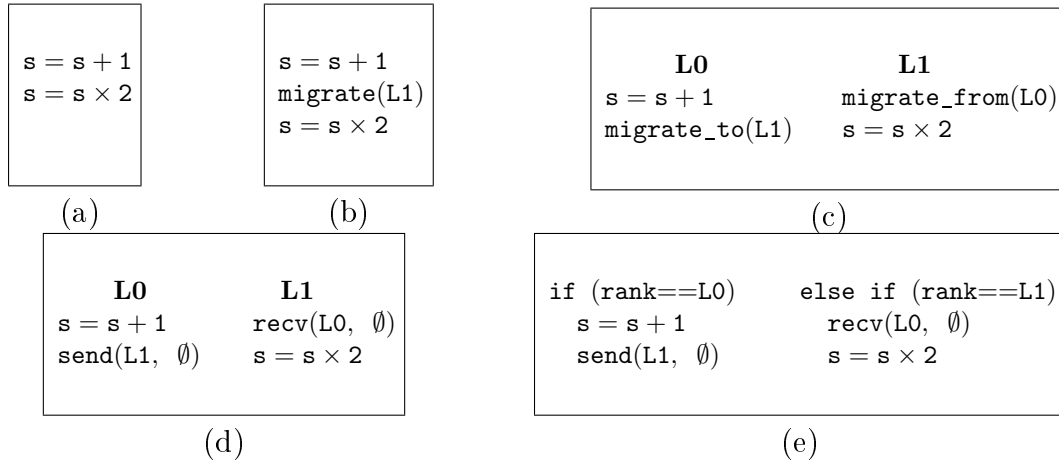


Figure 2.2: Straightline code. (a) Non-distributed program. (b) State-migration program. (c) State-migration programs, MPMD style. (d) Message-passing programs, MPMD style, assuming variables visible in all locations. (e) Message-passing program, SPMD style, assuming variables visible in all locations.

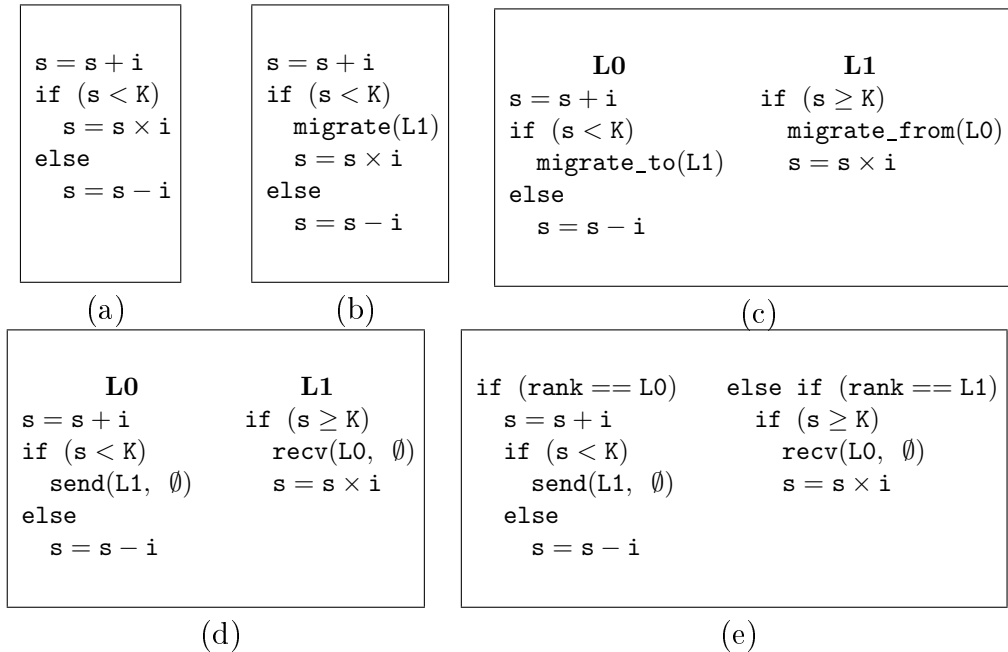


Figure 2.3: Selection code. (a) Non-distributed program. (b) State-migration program. (c) State-migration programs, MPMD style. (d) Message-passing programs, MPMD style, assuming variables visible in all locations. (e) Message-passing program, SPMD style, assuming variables visible in all locations.

```

for (i = 1; i < J; i++)
  s = s + i

```

(a)

```

for (i = 1; i < J; i++)
  s = s + i
  if (i > K) migrate(L1)

```

(b)

```

L0
for (i = 1; i ≤ K; i++)
  s = s + i
migrate_to(L1)

```

```

L1
migrate_from(L0)
for (i = K + 1; i < J; i++)
  s = s + i

```

(c)

```

L0
for (i = 1; i ≤ K; i++)
  s = s + i
send(L1, ∅)

```

```

L1
recv(L0, ∅)
for (i = K + 1; i < J; i++)
  s = s + i

```

(d)

```

if (rank == L0)
  for (i = 1; i ≤ K; i++)
    s = s + i
  send(L1, ∅)

```

```

else if (rank == L1)
  recv(L0, ∅)
  for (i = K + 1; i < J; i++)
    s = s + i

```

(e)

Figure 2.4: Repetition code. (a) Non-distributed program. (b) State-migration program. (c) State-migration programs, MPMD style. (d) Message-passing programs, MPMD style, assuming variables visible in all locations. (e) Message-passing program, SPMD style, assuming variables visible in all locations.

passing programs. As said, the message-passing model assumes at each location a distinct namespace for variables. However, for a fair comparison with its state-migration counterpart, the message-passing programs in Figs. 2.2 through 2.4 assume that the same  $s$  and  $i$  are visible in both Locations L0 and L1. Therefore, there is no need

to communicate any data from L0 to L1. The send and receive statements serve for synchronization purpose only, and the  $\emptyset$  therein indicates that there are no data involved. It is assumed that a distributed program starts at L0. At the end of all the programs, the variables named `s` have the same value.

The MPMD (Multiple Program Multiple Data) style means all processes on the same location uses the same program text, but processes on another location use another program text. The SPMD (Single Program Multiple Data) style means that processes on all locations use the same program text. A process is a program in execution on an execution location; in other words, a process is a runtime instance of a program on an execution location.

As a state-migration program can be viewed as an intermediate step in the process of transforming a non-distributed program into a message-passing program, it can be claimed that it is easier to develop a distributed program using state-migration programming than using message passing.

Note that an important step in the entire transforming process from a non-distributed or a state-migration program into a message-passing program is the transformation into the MPMD style where the statements are distributed among locations and the lexical structure of the program may be changed as a result.

The control structures in a message-passing program is more complex than those in the corresponding state-migration program. Though not shown, one can anticipate that the message-passing program in the case of nested selection and repetition, or combination thereof, will be much more complex. To gain a better understanding of why a message-passing program is hard to develop and understand, one can look at a contrived example shown in Fig. 2.5.

The programs in Figs. 2.5(a), (b) and (c) are legitimate C programs. Their control

```

int main()
{
    int s;
    s = f(s);
    if (s<K)
        s = g(s);
    else
        s = h(s);
    s = i(s);
    return 0;
}

```

(a)

```

int main()
{
    int s;
    s = f(s);
    if (s<K) {
        s = m(s);
        s = i(s);
    } else {
        s = h(s);
        s = n(s);
    }
    return 0;
}

int m(int s)
{
    return g(s);
}

int n(int s)
{
    return i(s);
}

```

(b)

```

int main()
{
    int s;

    {
        s = f(s);
        if (s<K) {
            goto A:
        } else {
            s = h(s);
            goto C:
        }
    }

    B:  s = i(s);
    goto D:

    D:  return 0;
}

{
    A:  s = g(s);
    goto B;
    C:  s = i(s);
    goto D;
}
}

```

(c)

```

s = f(s)
if (s<K)
    s = g(s)
else
    s = h(s)
s = i(s)

```

(d)

```

s = f(s)
if (s<K)
    migrate(L1)
    s = g(s)
    migrate(L0)
else
    s = h(s)
    migrate(L1)
s = i(s)

```

(e)

```

if (rank==L0)
    s = f(s)
    if (s<K)
        send(L1, s)
        recv(L1, s)
        s = i(s)
    else
        s = h(s)
        send(L1, s)
else if (rank==L1)
    if (s<K)
        recv(L0, s)
        s = g(s)
        send(L0, s)
    else
        recv(L0, s)
        s = i(s)

```

(f)

Figure 2.5: Comparing message-passing programming with programming using goto's. (a) A simple C program. (b) A C program derived from and equivalent to (a). (c) A C program derived from and equivalent to (b). (d) A non-distributed program. (e) A state-migration program obtained by partitioning the computation in (d). (f) A message-passing program equivalent to (e), assumed that the same  $s$  is visible in all locations.

flows are the same and they generate the same results, assumed that functions  $f()$ ,  $g()$ ,  $h()$ ,  $i()$ , and constant  $K$  are defined. The program in (b) distributes the  $s$

= `i(s)` statement into the conditional, and extracts out some statements inside the conditional in (a) to form a couple of functions (i.e., `m()` and `n()`). One can transform (b) to (c) by replacing the function calls by `goto`'s (and labels which can be considered as `comefrom`'s) and combining the two functions `m()` and `n()` into a block. (d) is simply a pseudocode equivalent of (a). Based on a computation distribution scheme, which associates statement instances in a program to different locations, one can place migration statements in the program as shown in (e); and (f) is the corresponding message-passing program. For comparability purposes, it is assumed that variables in the message-passing program are visible irrespective of the program's current location. Therefore, the `send` and `receive` statements in (f) are for synchronization only, the value of `s` needs not be communicated. One can easily see the similarity of (c) and (f), with a `goto` corresponding to a `send`, a label (i.e., `comefrom`) to a `recv`. This suggests that the complexity of a message-passing program is at least due to the facts that: 1. there are no single points of entry and exit in each part (one for each rank) of the SPMD message-passing program, and 2. there may be duplicate code. A state-migration program does not suffer from these two drawbacks.

From the perspective of software development, state-migration programming is superior to message-passing programming because a state-migration program does not duplicate the statements in the original non-distributed program and preserves their relative order in the program text.

It is noteworthy that state-migration programming is amenable to changes in computation distribution. (Computation distribution will be further discussed in Sec. 2.3.7.) The way computation is partitioned may be different on different architectures. A state-migration program requires minimal changes, by changing the position of the `migrate()` statement, to accommodate the change in computation distribution. On the other hand, a message-passing program needs more extensive changes. Fig-

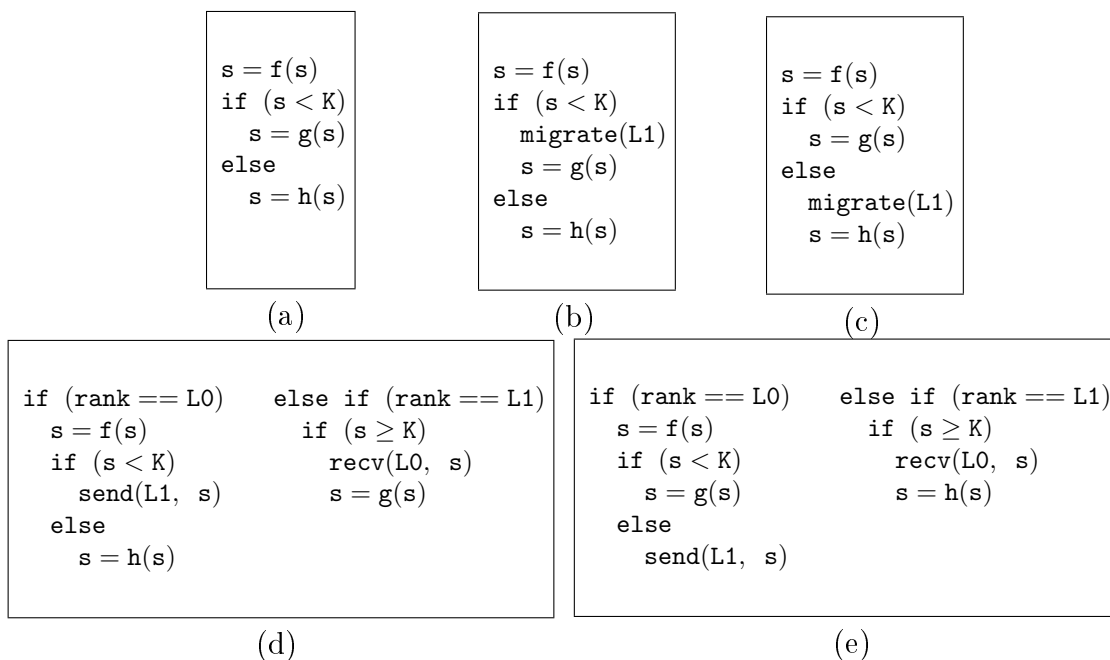


Figure 2.6: Program modifiability - state-migration programming vs message-passing. (a) Non-distributed program. (b) State-migration program, based on a certain computation distribution scheme. (c) State-migration program, based on another computation distribution scheme. (d) Message-passing program corresponding to (b). (e) Message-passing program corresponding to (c).

Figure 2.6(a) shows a non-distributed program. It is assumed that the program originally (and initially) is executed on Location L0. If it is decided that  $s = g(s)$  should be computed on L1 and  $s = h(s)$  on L0, the `migrate()` statement should be placed as in Fig. 2.6(b). However, if it is decided that  $s = g(s)$  should be computed on L0 and  $s = h(s)$  on L1, then the `migrate()` statement should be placed as in Fig. 2.6(c). Figures 2.6(d) and 2.6(e) are message-passing programs corresponding to Figs. 2.6(b) and 2.6(c) respectively. It is apparent that it is more involving to rewrite one message-passing program into another in order to adapt to the changes in computation distribution. Again, for comparability with the state-migration programs, it is assumed that the  $s$ 's in the message-passing programs are visible in all locations. The  $s$ 's in

the receive and send statements are not needed and are there just to conform with the conventional syntax of the send/receive statements.

### 2.1.4 Location-scoped variables

In a distributed program, where there is a spatial dimension, just as a computation step, or a statement, can be associated with a location, data can be associated with a location. That is, a variable can be associated with a location in such a way that the variable is visible to a program only when the program is currently located on that location. Such a variable is said to be a location-scoped variable; it is considered declared in that location.

A location-scoped variable associated with one location and another location-scoped variable associated with another location can share the same name because these two variables will not be referenced by the same program at the same time as a program can be located on only one location at any given time. In fact, these two variables are independent of each other.

```
int n = 0 on L0, L1  
  
n = n + 1  
migrate(L1)  
n = n + 1
```

Figure 2.7: Location-scoped variables.

In Fig. 2.7, there are an `n` declared in `L0`, and another `n` declared in `L1`; they are not the same variable. The `n` in `L0` is visible to the program only when the program is located on `L0`, and the `n` in `L1` is visible to the program only when the program is located on `L1`. Suppose that the program starts on `L0`, the `n` before the `migrate(L1)`

statement refers to the `n` declared in `L0`, and the `n` after the `migrate(L1)` statement refers to the `n` declared in `L1`. In other words, after its migration to `L1`, the program can no longer reference the `n` declared in `L0`. Both `n`'s are initialized to 0 and have a value of 1 when the program ends.

### 2.1.5 State-migration location-scoped-variable programming

*State-migration location-scoped-variable programming* is defined as a distributed programming technique with which a program migrates among multiple locations and computes using location-scoped variables. In other words, state-migration location-scoped-variable programming is state-migration programming with location-scoped variables.

Since a variable declared in one location is no longer visible to a program after the program migrates to another location, the program needs to save, before the migration, the value of the location-scoped variable into a variable that is visible in all locations, a so-called migratable variable, if the program needs to use that value after the migration. A migratable variable is not associated with any location.

```
int n0 = 0 on L0
int n1 = 0 on L1

int m
m = n0
migrate(L1)
n1 = m
```

Figure 2.8: Saving the value of a location-scoped variable into a migratable variable.

In Fig. 2.8, `m` is a migratable variable, `n0` is declared in Location `L0`, `n1` is declared in Location `L1`. It is again assumed that the program starts at `L0`. If the program

needs to use the pre-migration value of `n0` after it migrates, it needs to save it in `m` before the migration. And in this particular example, the value is restored to `n1` at L1. Note that since `n0` and `n1` are independent, they can be renamed to `n` and the outcome would be the same.

## 2.1.6 State-migration location-scoped-variable programming vs message passing

Section 2.1.3 compares state-migration programming and message passing with variables not associated with locations. Here, Figs. 2.9 through 2.11 illustrate how non-distributed programs with sequence, selection, and repetition control structures can be transformed into programs distributed over two locations using state-migration location-scoped-variable programming, and then further into message-passing programs. The traditional message-passing model with all variables being location-scoped is used here. In the state-migration location-scoped-variable programs, the name `s` refers to either a variable declared in L0 or a variable declared in L1, depending on the current location of the program. And `m` and `i` are migratable variables, not associated with any location. It is again assumed that a distributed program starts at L0. At the end of the state-migration location-scoped-variable and message-passing programs, the variables declared in L1 have the same value.

In Figs. 2.9(b) and (c), `s` at L0 is supposed to be not needed to store the result of its increment. However, in Fig. 2.11, the accumulated sum `s` is supposed to be stored in a location-scoped variable at that location. If the accumulated sum at L0 in Fig. 2.4 is supposed to be stored, an additional variable is needed; and it is visible to the program regardless where the program is located, due to the lack of location-scoped variables in the state-migration model.

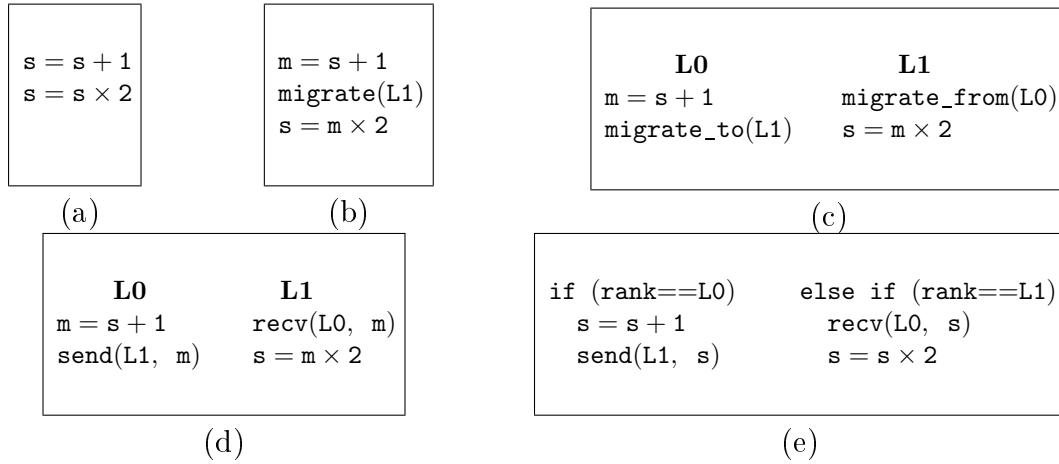


Figure 2.9: Straightline code. (a) Non-distributed program. (b) State-migration location-scoped-variable program. (c) State-migration location-scoped-variable programs, MPMD style. (d) Message-passing programs, MPMD style. (e) Message-passing program, SPMD style.

Like a state-migration program, a state-migration location-scoped-variable program can be viewed as an intermediate step in the process of transforming a non-distributed program into a message-passing program, it can be claimed that it is easier to develop a distributed program using state-migration location-scoped-variable programming than using message passing. From the perspective of software development, state-migration location-scoped-variable programming is superior to message-passing programming because a state-migration location-scoped-variable program does not duplicate the statements in the original non-distributed program and preserves their relative order in the program text, although there may be some additional statements saving values from location-scoped variables to migratable variables and restoring the values from migratable variables to location-scoped variables.

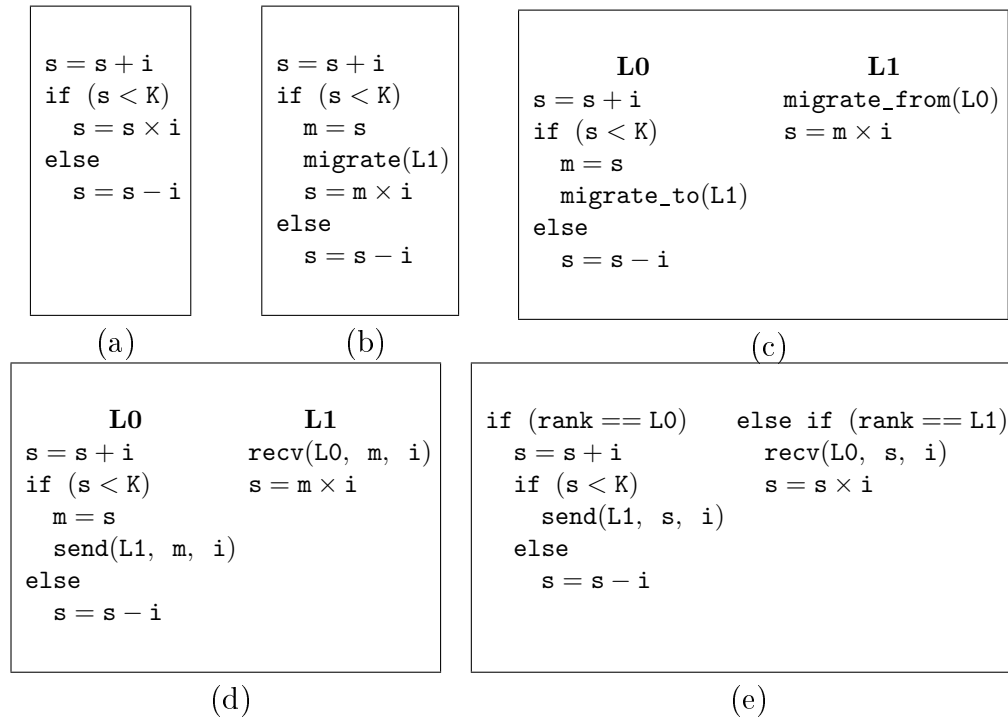


Figure 2.10: Selection code. (a) Non-distributed program. (b) State-migration location-scoped-variable program. (c) State-migration location-scoped-variable programs, MPMD style. (d) Message-passing programs, MPMD style. (e) Message-passing program, SPMD style.

### 2.1.7 Migration during execution of subprogram

Subprograms are the fundamental building blocks of programs often used to achieve procedural (or functional) abstraction. A subprogram usually has local variables as well as parameters. In the previous discussion, simple programs, without any subprograms, are used to illustrate the concept of state-migration location-scoped-variable programming. A program can migrate when a subprogram of which is being executed. For the sake of simplicity, the following discussion assumes that no nested subprograms are allowed.

In Fig. 2.12,  $m$  and  $p$  are a local variable and a parameter, respectively, declared in a subprogram `func()`, whereas there is an  $n$  declared in Location  $L0$ , and an  $n$  declared

```

for (i = 1; i < J; i++)
  s = s + i

```

(a)

```

for (i = 1; i < J; i++)
  s = s + i
  if (i > K)
    m = s
    migrate(L1)
    s = m

```

(b)

<b>L0</b>	<b>L1</b>
<pre> for (i = 1; i ≤ K; i++)   s = s + i m = s migrate_to(L1) </pre>	<pre> migrate_from(L0) s = m for (i = K + 1; i &lt; J; i++)   s = s + i </pre>

(c)

<b>L0</b>	<b>L1</b>
<pre> for (i = 1; i ≤ K; i++)   s = s + i m = s send(L1, m) </pre>	<pre> recv(L0, m) s = m for (i = K + 1; i &lt; J; i++)   s = s + i </pre>

(d)

```

if (rank == L0)
  for (i = 1; i ≤ K; i++)
    s = s + i
    send(L1, s)
else if (rank == L1)
  recv(L0, s)
  for (i = K + 1; i < J; i++)
    s = s + i

```

(e)

Figure 2.11: Repetition code. (a) Non-distributed program. (b) State-migration location-scoped-variable program. (c) State-migration location-scoped-variable programs, MPMD style. (d) Message-passing programs, MPMD style. (e) Message-passing program, SPMD style.

in Location L1. There is also a global variable `g`, which is not associated with any location. `m` and `p` are migratable variables, and so is `g`; all of them are visible to `func()` at any location.

```
int n = 2 on L0, L1

int g

func(int p)
{
  int m = 0
  if (p > 0)
    m = n + p
  migrate(L1)
  n = n × m × p
  g = n
}
```

Figure 2.12: Migratable and location-scoped variables of a subprogram in a state-migration location-scoped-variable program.

The principle of state-migration location-scoped-variable programming is unchanged in the case of a migration inside a subprogram. Note that, however, in general one may not readily determine with which location `n` is associated by merely looking at the program text. The `n` in the addition statement is associated with the location where `func()` is called, that is, the current location of `func()`'s caller when the call is made.

## 2.2 Concurrent programming using multiple tasks sharing variables

### 2.2.1 Task

It is useful to distinguish between (physical) parallelism and (logical) concurrency. In parallelism, multiple processors are available, and a program is divided into a number of program units, executed at the same time each on a different processor. Concurrency allows the programmer to assume that there are multiple processors available, when in fact the actual execution of these program units is taking place in interleaved fashion on a single processor.

Many researchers such as Sebasta [59] use the term *task* to refer to a runtime instance of a unit of program that is similar to a subprogram but which can be in concurrent execution with other instances of units of the same program. As Sebasta [60, 59] explained, both a subprogram and a task have a single point of entry; that is, the caller cannot pass control to an arbitrary statement inside the body of the subprogram/task. However, the program unit that calls a subprogram is suspended during the execution of the subprogram, whereas the program unit that invokes a task can continue its execution concurrently with the task. Also, when the subprogram's execution terminates, control always returns to the caller, but when the execution of a task is completed, control may or may not return to the program unit that invokes the task.

A task is the execution instance of a task definition. The relationship between a task and a task definition is similar to that between an object and a class in an object-oriented programming language. As recursive calls to a subprogram generate multiple subprogram instances, calling a task definition multiple times creates multiple tasks

which can be in concurrent execution.

### **2.2.2 Shared-variable programming**

Shared-variable programming is a model for concurrent programming which involves two or more tasks, and some variables that are visible to more than one task. In the shared-variable model, a program is explicitly decomposed into multiple tasks and each communicates and synchronizes with another via variables that are visible to all of them. Concurrency is achieved by having at the same time two or more tasks active, meaning that they have begun but not completed execution. There is no notion of location; but there may be a distinct identifying value associated with each task. Some kind of statements to create a task and to synchronize, both cooperatively and competitively, tasks are needed. The pthreads (see Sec. 5.3.1) model is a representative of this model.

Message passing can also be used in concurrent programming. A message-passing program is decomposed into multiple parts, each being a task. Under the message-passing model, there exist no variables that can be referenced by more than one tasks, even at different times. The tasks communicate and synchronize with each other by passing messages — sending data (i.e., values) to and receiving data from each other.

## **2.3 MESSENGERS programming model**

The MESSENGERS programming model is basically a combination of the state-migration location-scoped-variable model and the shared-variable model.

### 2.3.1 Messenger migration

In the MESSENGERS programming model, a distributed concurrent program is composed of a number of migrating tasks called Messengers.

A Messenger optionally takes value parameters but does not return any value; it may declare local variables, called Messenger variables. Messenger variables and parameters are the migratable variables discussed in Sec. 2.1.5. At the end of the execution of a Messenger, control is not passed back to its caller.

One Messenger can fork, or inject, in MESSENGERS terminology, another Messenger. Using the shared-variable model, one can create multiple concurrent Messengers which synchronize and communicate with each other via variables declared in these Messengers' current locations. There are no global variables such as the  $g$  in Fig. 2.12. Therefore, the only shared variables are the location-scoped variables.

A Messenger and its children, namely, those injected by it, are independent of each other in the sense that the parent does not have the authority to terminate its children and they all have independent lifetimes, meaning that the children would not die even if their parent dies.

### 2.3.2 Physical node and logical node

A location as discussed in Sec. 2.1.1 is implemented in the MESSENGERS model by two abstractions: physical node and logical node. A physical node embodies the processing capability of a location, whereas a logical node embodies the storage capability of a location. There is a 1:N relationship between a physical node and a logical node (related to the same location); that is, there can be multiple logical nodes associated with one physical node, and any one logical node can be matched

to only one physical node. Figure 2.13 illustrates the relationship between locations, physical nodes and logical nodes. A location is represented by a physical node and a number of logical nodes associated with that physical node. Note that the term *physical node* here does not refer to a physical computer, though it can be used to embody the processing capability of a physical computer.

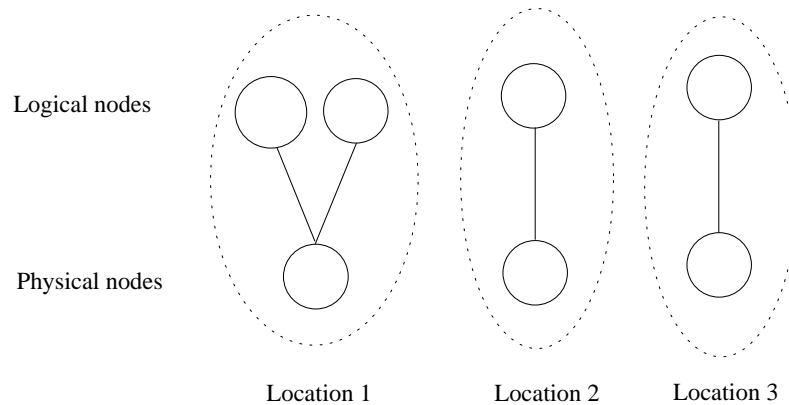


Figure 2.13: Locations, physical nodes and logical nodes.

Just like a distributed program, subprogram, or task is associated with, or located on, one location at any given time, a Messenger is located on one logical node and the physical node associated therewith at any given time.

The view that a physical node is an abstraction of the processing capability of a location implies that Messengers on different physical nodes can use different sets of processing resources, or different instances of the same set of processing resources, and they can run concurrently.

As said in Sec. 2.2.1, a task is a unit of concurrency: two tasks run concurrently. In MESSENGERS, however, that concurrency is subject to a restriction: two Messengers are concurrent only if they are on different physical nodes. In other words, two Messengers each on one logical node do not run concurrently unless these two logical nodes are on different physical nodes. Two Messengers on the same logical node are

not concurrent.

The location-scoped variables discussed in Sec. 2.1.4 are called node variables in the MESSENGERS model; and they are associated with logical nodes.

Because two Messengers on the same logical node are not concurrent, they never have concurrent references to shared variables, and consequently there is no need for mutual exclusion locks on node variables to protect them from concurrent accesses by multiple Messengers on the same logical node. If two Messengers are on different logical nodes, one cannot reference the node variables, which are location-scoped, the other may be accessing, so there is no need for mutual exclusion lock on any variables they access.

### **2.3.3 Messenger migration on logical nodes and physical nodes**

In light of the separation between a physical node and a logical node, the semantics of the migration of a Messenger needs further clarification: its migration is from one logical node to another and whether there is also a migration from one physical node to another depends on whether the source and destination logical nodes are associated with the same physical node. Because no more than one physical node is associated with a particular logical node, the migration of a Messenger from one logical node to another can also be used for the purposes of migration from one physical node to another. Figure 2.14(a) illustrates this point: In the case that both the source and destination logical nodes are associated with the same physical node, the migration can be seen as a no-op, or do-nothing operation, from the standpoint of the physical node; that is, a Messenger is to access a different set of storage resources but not a different set of processing resources. If the source and destination logical nodes are on different physical nodes, then there is also migration from one physical node to

another; the migrating Messenger is accessing different sets of storage resources and processing resources. On the other hand, if more than one physical nodes could be associated with one logical node, as shown in Fig. 2.14(b), then the above arguments would not hold. And there probably would need two different types of migration statements, one for migrating on logical nodes, another for migrating on physical nodes.

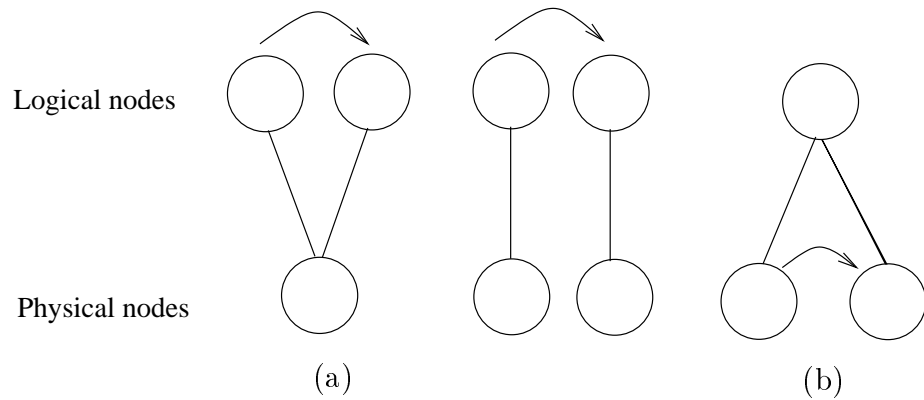


Figure 2.14: Messenger migration with respect to logical nodes and physical nodes. (a) Migration from one logical node to another, which may or may not result in migration from one physical node to another. (b) Migration from one physical node to another, in the case that two physical nodes are associated with one logical node, which is not supported by the MESSENGERS model.

As discussed in Sec. 2.1.3, the state-migration programming model, a generic component of the MESSENGERS model, facilitates computation distribution, by placing the migration statement at the point where the computation is partitioned.

As a Messenger migrates on logical nodes only, if one wishes to partition computation between two physical nodes, one needs to create a logical node for each of these two physical nodes and migrate a Messenger from one logical node to another.

### 2.3.4 MESSENGERS language

A MESSENGERS program (see Fig. 2.15 for an example) is a collection of Messenger definitions, each of which may call one or more other Messenger definitions and any of which may include one or more node files, which contain declarations of node variables. The execution of a MESSENGERS program starts with the execution of one Messenger definition. The name of the Messenger definition is not included in the program text; rather, it is implied from the name of the file the Messenger definition is written in. A node file is `#included` at the top of a Messenger definition so that the node variables declared in the node file can be referenced in that Messenger definition. A node variable is considered declared in, or associated with, a logical node if the node file declaring it is `#included` in a Messenger definition which makes references to that logical node.

<pre>#include "simple.node" int \$arg1; // Messenger parameter int x; // Messenger variable create(node = "nodeA";     physical = "tarheel.ics.uci.edu"); x = \$arg1 + y; printf("%d\n", x);</pre>	<pre>int y; // Node variable</pre>
(a)	(b)

Figure 2.15: A MESSENGERS program consisting of a Messenger definition and a node file. (a) A simple Messenger definition. (b) A simple node file.

There is no `main()` function like the one in C in a MESSENGERS program. In addition, there are no global variables like those in C.

The MESSENGERS language is based on the C language. Specifically, it has many, if not most of, C language constructs. On the other hand, because it needs to implement

the abstraction of Messengers, physical nodes and logical nodes and their operations, it provides four major statements, which can be included in a Messenger definition:

1. `inject()`: creating another Messenger on the same logical node the calling Messenger is located. The `inject()` function takes the name of a library object file (see Chapter 3) associated with the injected Messenger as its only parameter.
2. `hop()`: migrating the calling Messenger to a logical node specified as the parameter of the call. The main forms of the `hop()` statement make reference to either the name or the internal address of the destination logical node.
3. `signalEvent()` and `waitEvent()`: synchronizing Messengers. The main form of either the `signalEvent()` function or the `waitEvent()` function takes a node variable of `event` type as its parameter.
4. `create()`: creating a logical node (and migrating the calling Messenger to the created logical node) and a link between the source logical node and the destination logical node created. The main form of the `create()` statement has two attributes, `node` and `physical`, taking as their values the name of the logical node to be created, and the name of the physical node with which the logical node is associated, respectively.

Note that the `create()` statement is included in a Messenger definition. The node file contains only declaration statements.

Also, either `inject()` or `hop()` associates a Messenger with a logical node, and `create()` associate a logical node with a physical node.

Both the `hop()` and the `create()` statements can be considered migration statements because the calling Messenger migrates to another logical node.

A Messenger calls the `stall()` statement to yield the CPU. Another Messenger, if any, shall be executed next.

There is no nesting of Messenger definitions. Therefore, a Messenger cannot reference the Messenger variables of the Messenger that injects it.

Each Messenger definition can call functions in external libraries such as the Standard C Library and user-created libraries, but such a function cannot inject a Messenger or include a migration statement. Nor can it reference node variables or Messenger variables. It is allowed to use only value parameters and local variables.

The MESSENGERS User's Manual [22] provides the detailed syntax of the MESSENGERS language.

### **2.3.5 Dynamic binding of node variables**

Section 2.1.4 describes how different location-scoped variables with the same name are referenced by a migrating program. In the MESSENGERS language, the mechanism is realized by dynamic binding of node variables.

#### **Dynamic binding in general**

It is useful at this point to distinguish between a variable and a variable identifier. A variable is an abstraction of a memory location that stores a value. A variable identifier is a name for a variable.

In general, binding is the association of an attribute with an entity, or of an operation with a symbol, for example. With respect to variables in a program, binding, or name binding to be more specific, refers to the association of a variable identifier with a

variable. In a language with hierarchical scopes, binding of a variable identifier in a subprogram to a variable can be done in compile time (in the case of static binding) based on the lexical structure of the program or run time (in the case of dynamic binding).

### Location-based scope

A scope of a variable is a range of program statement instances in which the variable is visible. Usually a scope is associated with a program unit such as a subprogram. In a distributed program with a spatial dimension, it is possible to define, for a variable, a location-based scope that is associated with a location as the range of statement instances in the subprograms that are associated with that location. (Recall that a statement instance is associated with a location if the location attribute for that statement instance has that location as its value, as defined in Sec. 2.1.1.

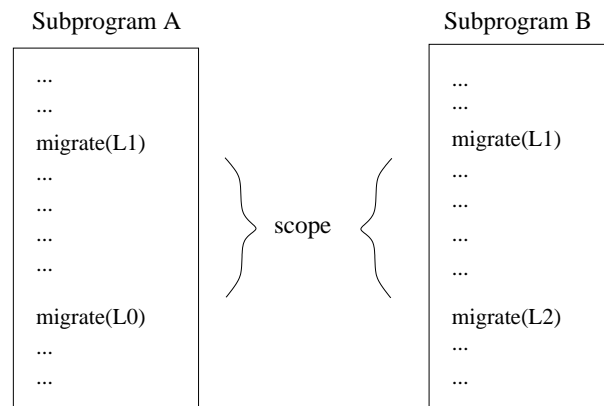


Figure 2.16: Location-based scope for location-scoped variables associated with L1.

Figure 2.16 shows two subprograms which are located on a location at different times. The location-based scope for location-scoped variables associated with L1 consists of the range of statement instances between `migrate(L1)` and `migrate(L0)` in Subprogram A and the range of statement instances between `migrate(L1)` and `migrate(L2)`

in Subprogram B.

The scoping rule for a location-scoped variable of the MESSENGERS language is that one needs to look for the declaration of that variable in the logical node on which the Messenger referencing that variable is currently located.

### **Migration and dynamic binding of node variables**

The name of a node variable referenced by a Messenger currently on a logical node is bound to a variable declared in that logical node. The binding is determined dynamically, i.e., at run time because the current location of the referencing Messenger can be determined only at run time.

With respect to dynamic binding of node variables, the semantics of a migration statement, e.g., a `hop()`, is such that it rebinds names of node variables to variables declared in the location-based scope associated with the destination logical node of the migration.

Sebasta [61] defined the state of a program as a set of ordered pairs:  $\{\langle i_1, v_1 \rangle, \langle i_2, v_2 \rangle, \dots, \langle i_n, v_n \rangle\}$  where each  $i$  is the name of a variable, and the associated  $v$ 's are the current values of those variables. The next state of a program is generated by the execution of a statement instance on the current state.

Assumed that there are only two variable names, or variable identifiers, in a program:  $m$  and  $n$ , where  $m$  is a migratable variable and  $n$  is a location-scoped variable. There are an  $n$  declared in  $L_0$ , and an  $n$  declared in  $L_1$ . The state of the program is therefore  $\{\langle m, v_m \rangle, \langle n, v_n \rangle\}$ . If the current value of  $n$  at  $L_0$  is 0 and the current value of  $n$  at  $L_1$  is 1, the execution of a `migrate(L1)` statement at  $L_0$  when the state of the program is  $\{\langle m, v_m \rangle, \langle n, 0 \rangle\}$  will produce a new state  $\{\langle m, v_m \rangle, \langle n, 1 \rangle\}$ .

## Other binding possibilities

There are other binding possibilities for location-scoped variables in general. In addition to being bound to a variable which is declared in the location on which the Messenger referencing it is currently located, as dictated by the MESSENGERS model, the name of a location-scoped variable can either be dynamically bound to a variable declared in the location where the referencing Messenger was injected, or statically bound to a variable declared in a location specified in the program by the programmer. In the former case, it is a one-time binding. The binding does not change even when the Messenger migrates away from the location where it was injected. In the latter case, the name of the location-scoped variable can be prepended by the location to which it is statically bound, e.g., `L1::n`. Again, in MESSENGERS, dynamic binding to the referencing Messenger's current location is the only permitted scoping rule for node variables.

### 2.3.6 Scheduling of Messengers

#### Messenger as cooperative task

One key difference between traditional tasks and Messengers is that Messengers are scheduled cooperatively. That is, a Messenger runs until it terminates, voluntarily gives up the CPU control, or blocks by cooperative synchronization. One can therefore say that the MESSENGERS language has a run-to-completion-or-cooperation semantics.

## Cooperative threads

A thread, or more generally a task, that is scheduled preemptively can be interrupted externally out of its own control. This makes reasoning on programs difficult. As a result, some researchers developed cooperative threads such as FairThreads [11]. In these systems, the control of the CPU cannot be forcefully taken away from a thread; instead, a thread needs to give it up by calling a function such as `yield()`. Cooperative threads are a minority in the thread packages developed.

## Deterministic context switching

The caller of the `yield()` statement usually gives another task only the possibility of getting the CPU; it is possible that no context switching actually happens. In MESSENGERS, a `stall()` call to give up the CPU actually turns the CPU over to another Messenger, if there is one.

The migration of a Messenger also causes it to give up the CPU and thus an actual context-switching. Another Messenger, if any, shall get the CPU from the migrating Messenger.

### 2.3.7 Computation and data distribution using MESSENGERS

In the arena of concurrent programming on distributed-memory computers, data and computation distribution is important for performance. It aims at optimizing locality (alternatively, communication) and parallelism, thus minimizing the overall execution time of a program.

In (explicitly) distributed (explicitly) concurrent programming, the programmer needs

to control the distribution of computation and data in order to minimize the run time. That is, it is the programmer's job to maximize concurrency and minimize communication. One has to find an optimal point to strike a balance between the level of concurrency and the volume of communication. There is much research on how to find computation and data distribution schemes (for example, see Anderson and Lam [2], Anderson et al [1], and Lee and Kedem [41]). Once such distribution is found, one can develop a program based on whatever programming model one uses.

Section 2.1.3 shows how computation can easily be distributed on locations using the state-migration programming model. In the case of MESSENGERS, computation can be similarly distributed on physical nodes, as physical nodes are abstraction of processing capability of a location, as discussed in Sec. 2.3.2. Two migration statements, not intervened by any others, in MESSENGERS in effect place the computation between them on a physical node associated with the logical node referred to in the first migration statement, and the computation after the second migration statement to a physical node associated with the logical node referred to in the second migration statement. Data distribution can be achieved by declaring variables for those data in node files and associating these node files with different logical nodes. A physical node is mapped to a CPU, and a logical node is mapped to the memory associated with that CPU.

Figure 2.17 shows how computation and data are distributed using the MESSENGERS model. It is assumed that only one logical node is associated with one physical node. Level of concurrency is determined by the number of physical nodes. Volume of communication is determined by the sizes of the Messengers and the number of their migrations.

Through its state-migration location-scoped-variable component, the MESSENGERS model supports incremental partitioning of computation and data. One can add a

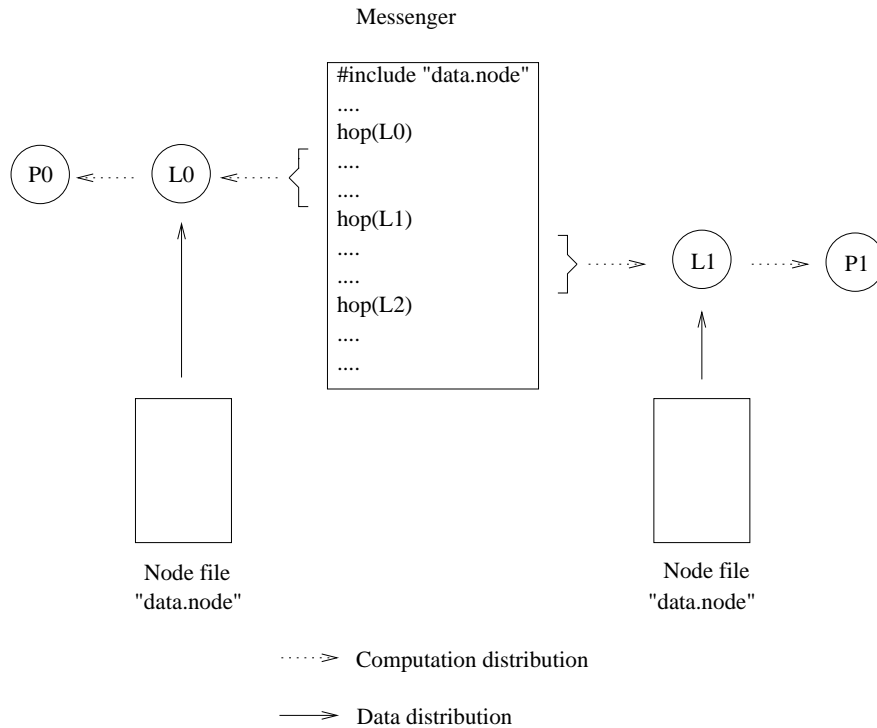


Figure 2.17: Using the MESSENGERS programming model for computation and data distribution.

migration statement and a pair of logical and physical nodes to increment the number of partitions. As an example, consider Fig. 2.17. Suppose one wishes to further partition the data assigned to L0 between L0 and a new logical node L3, and further partition the computation assigned to P0 between P0 and a new physical node P3, he can create a new pair of logical node and physical node, L3 and P3 respectively, and place a `hop()` statement somewhere between `hop(L0)` and `hop(L1)`, as shown in Fig. 2.18. Note that the level of concurrency is increased when a new physical node, namely, P3, is added.

### 2.3.8 Summary

To sum up, MESSENGERS programming combines the distributed programming technique of migration and using location-scoped variables and the concurrent pro-

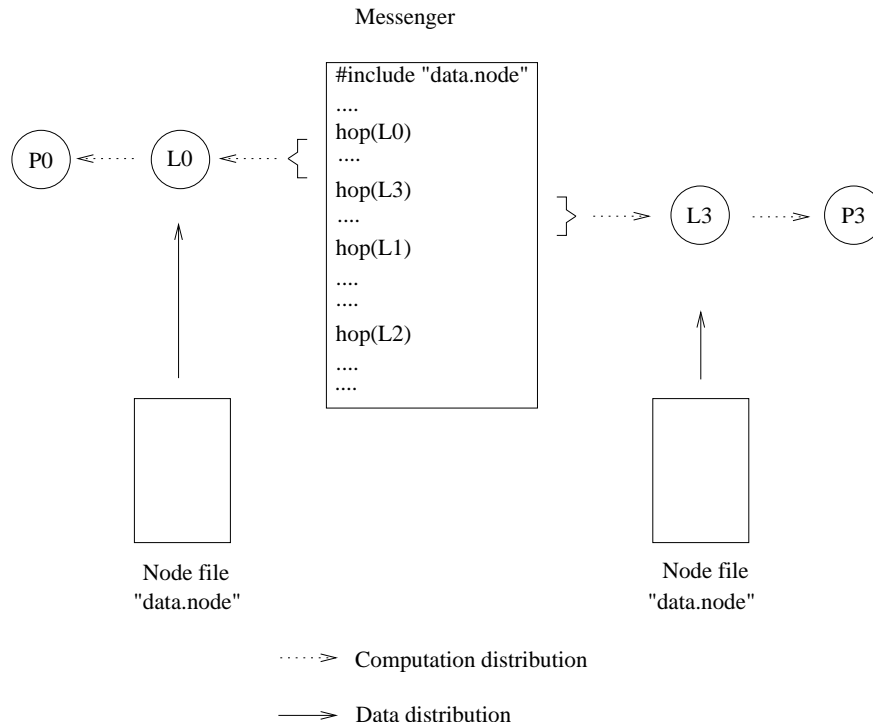


Figure 2.18: Incremental partitioning of computation and data using the MESSENGERS programming model.

programming technique of concurrent tasks communicating and synchronizing via shared variables. Since the two component techniques address separate concerns and thus are orthogonal, they supplement and extend each other without changing their existing individual properties. The MESSENGERS programming model inherits the benefits of both state-migration location-scoped-variable and shared-variable models. For example, the shared-variable model offers the ease of incremental parallelization (of computation) and the state-migration location-scoped-variable model offers the ease of incremental partitioning (of computation and data); MESSENGERS inherits both.

## Chapter 3

# Original MESSENGERS System

The MESSENGERS system is a programming system to support the MESSENGERS model for fine-grained cooperative tasks on distributed-memory parallel computers (i.e., networks of computers) and is composed of a translator and a runtime system.

Fukuda [20] developed the first MESSENGERS system, known as MESSENGERS-I, for programming on distributed systems using the MESSENGERS programming model and language, where the code of a Messenger is dynamically distributed and executed by an interpreter. It can be considered one of the early mobile agents systems.

Subsequently, Wicke [73] modified MESSENGERS-I so that the code of a Messenger is statically distributed over different computers. That is, the Messenger code is compiled and accessible from the local file system of the participating computers. This version is referred to as MESSENGERS-C. MESSENGERS-C supports the MESSENGERS programming language but does not involve code mobility. The MESSENGERS system discussed in this dissertation is based on MESSENGERS-C; and hereafter the MESSENGERS-C system is simply referred to as the MESSENGERS

system.

### 3.1 MESSENGERS compiler

The MESSENGERS translator, or compiler, was initially developed by Christian Wicke and later enhanced and updated by Hairong Kuang, me, and particularly Koji Noguchi, who optimized the translation of the node variables, among other things.

The MESSENGERS compiler is composed of a preprocessor and a C compiler. The source-to-source preprocessor translates a program written in the MESSENGERS language into a C source program, using the technique discussed below, and the C compiler translates the C source program into a shared library file. At present, the MESSENGERS compiler calls the GNU C compiler for the second stage.

The MESSENGERS compiler's preprocessor translates a Messenger definition into a C program, hereafter referred to as a Messenger C program. The preprocessor outputs in the Messenger C program a structure named `msgr` whose members are the Messenger variables declared in the definition of a Messenger. It also outputs a structure named `MB` that is identical to `struct MCB` as described in Sec. 3.2.1 except that its last member is of type `struct msgr`, whereas the last member of `struct MCB` is a flexible array. In the runtime program, a pointer to `struct MB` and a pointer to `struct MCB` are cast to each other as needed. Figure 3.1 shows the declarations of two Messenger variables in a Messenger definition and the corresponding `struct msgr` in the Messenger C program generated by the MESSENGERS preprocessor.

The MESSENGERS preprocessor splits a Messenger definition into a number of functions at each statement (in the Messenger definition) that places a Messenger onto a different logical node. That is, the Messenger C program consists of definitions

```
int    x;
double y;
```

(a)

```
struct msgr {
    int    x;
    double y;
};
```

(b)

Figure 3.1: Translation of declarations of Messenger variables to the definition of a structure by the MESSENGERS preprocessor. (a) Messenger variables declared in a Messenger definition. (b) Corresponding `struct msgr` output by the MESSENGERS preprocessor.

of functions, called `_msgr_functions`, that are generated from the Messenger definition. Each `_msgr_function` takes a pointer to a `struct MB` as its only parameter and returns a number for the scheduler to use in determining what to do next after the execution of this `_msgr_function` is complete. Each `_msgr_function` also sets a field, `next_function_nr`, in the MB to a number representing the `_msgr_function` that is to be executed next, if and when the scheduler continues to execute this MB. In addition, the compiler inserts in these `_msgr_functions` functions defined in the MESSENGERS runtime system program. Figure 3.2 shows a Messenger definition and the `_msgr_functions` in the corresponding Messenger C program. The `_msgr_functions` in Fig. 3.2(b) are derived from the statements in the Messenger definition and they call functions defined in the runtime such as `msgr_dequeue()` and `route_name_search()`. The compiler also writes certain initialization functions, such as those shown in Fig. 3.3, in the Messenger C program. Those initialization functions are called by a `create_mcb()` function which will be discussed in Sec. 3.2.4.

Further details of the MESSENGERS compiler can be found in Wicke's thesis [73] and Wicke et al's work [76, 75, 74].

```

int x;
create(node = "nodeA"; physical = "tarheel.ics.uci.edu");
x = 1;
printf("%d\n", x);

```

(a)

```

int _msgr_function_0(struct MB *mcb)
{
    msgr_dequeue(mcb);
    destroy_mcb(mcb);
}

int _msgr_function_1(struct MB *mcb)
{
    struct LINK *new_link = get_new_link();
    ...
    strcpy(new_link->dst_ncb_name, "nodeA");
    mcb->next_function_nr = 2;
    new_link->route = route_name_search("tarheel.ics.uci.edu");
    ...
    move_msgr_to_route(mcb, new_link->route);
    return 0;
}

int _msgr_function_2(struct MB *mcb)
{
    struct LINK *new_link = get_new_link();
    ...
    new_link->ncb = create_node(cs->dst_node_name);
    ...
    assign_ncb(mcb, new_link->ncb);
    ...
    mcb->next_function_nr = 3;
    return 0;
}

int _msgr_function_3(struct MB *mcb)
{
    mcb->msgr_variables_area.x = 1;
    (c_func(0))("%d\n", mcb->msgr_variables.x);
    mcb->next_function_nr = 0;
    return 0;
}

```

(b)

Figure 3.2: Translation of a Messenger definition into a Messenger C program consisting of `_msgr_functions`. (a) A simple Messenger definition. (b) `_msgr_functions` in the corresponding Messenger C program.

```

typedef int t_c_function();
typedef t_c_function *p_c_function;
#define c_func(n) ((t_c_function*)mcb->functions[n+4])
typedef int t_msgr_function();
typedef t_msgr_function *p_msgr_function;
static p_msgr_function *functions = 0;

static void init_msgr_functions(void *handle)
{
    int i;
    char name[256];
    strcpy(name, "_msgr_function_"); // length of the string is 15
    for (i = 0; i < 4; i++) {
        sprintf(name + 15, "%d", i);
        functions[i] = (p_msgr_function)dlsym(handle, name);
    }
}

static void init_external_functions(p_c_function *c_funcs_ptr)
{
    *(c_funcs_ptr++) = (p_c_function*)dlsym(libraries, "printf");
}

void init_library(void *handle)
{
    ...
    functions = (p_msgr_function*)malloc(funcs_size);
    init_msgr_functions(handle);
    init_external_functions((p_c_function*)&functions[4]);
    ...
}

struct MB *create_msgr(char **argv)
{
    struct MB *mcb = malloc(MSGR_SIZE);
    mcb->next_function_nr = 1;
    mcb->functions = functions;
    ...
    return mcb;
}

```

Figure 3.3: Initialization functions in a Messenger C program.

## 3.2 Single-CPU MESSENGERS runtime

The MESSENGERS runtime system as implemented by Wicke [73] and subsequently modified and improved by others including Kuang, Noguchi and me provides the execution environment for MESSENGERS applications on (networks of) single-CPU computers. I re-engineered a significant part of the code for parallelization, portability, robustness and other purposes. For example, the part that involves the UNIX signaling mechanism was rewritten to use reliable signals to allow the system to meet current system software requirements.

The runtime program is written in C for performance and portability reasons.

Three levels of networks, as illustrated in Fig. 3.4, can be distinguished in the MESSENGERS execution environment.

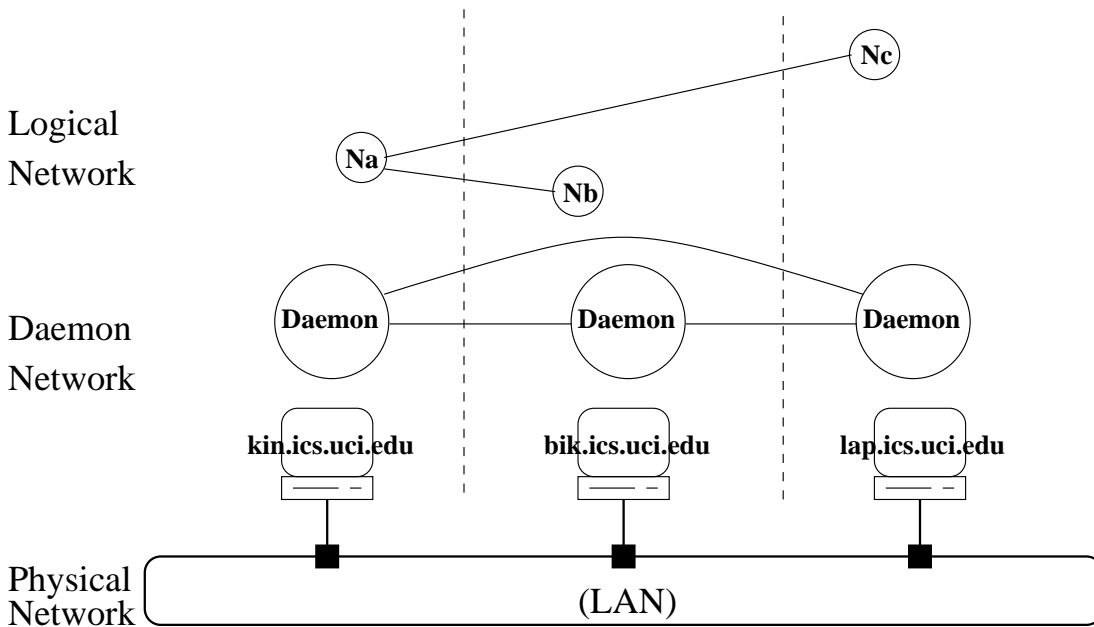


Figure 3.4: Three levels of networks in the MESSENGERS execution environment. (Adapted from Fukuda's dissertation [20].)

1. Physical network: This is the network of computers (i.e., hosts), most likely a local area network.

2. Daemon network: This is a fully connected network, on top of the physical network, on which the MESSENGERS runtime system runs. In systems programming, a *daemon* usually refers to a process that almost always run with superuser privilege, is started at boot time of a host and runs until system shutdown, and has the init process as its parent. However, in MESSENGERS terminology, a daemon merely is a process that runs the MESSENGERS runtime program on a host. In essence, the daemon network is an inter-process communication network. The user running MESSENGERS application creates this daemon network by specifying in a configuration file called MESSENGERS profile the hosts on which the daemons runs and executing the runtime program on each host. The daemon network is fixed at the beginning of execution. That is, the number of daemons and their locations (i.e., the hosts they run on) usually cannot be changed in the middle of execution although it is possible to migrate a daemon from one host to another host at run time (see Sec. 8.4). A daemon actually implements a physical node as described in Sec. 2.3.2. That is, there is a daemon for each physical node.

3. Logical network: This is a network of the logical nodes as described in Sec. 2.3.2. The logical network is constructed by the MESSENGERS programmer by using the `create()` statement, for example, as described in Sec. 2.3.4, in a Messenger definition. (The physical node specified in a `create()` statement must be one of those hosts specified in the MESSENGERS profile.) It is this logical network on which Messengers migrate.

### 3.2.1 Implementation of logical nodes and Messengers

Whereas a physical node is represented by a daemon, and thus no separate data structure for a physical node is implemented in the MESSENGERS runtime pro-

gram, logical nodes and Messengers are implemented by data structures defined in the MESSENGERS runtime program.

### Implementation of logical node

A logical node is implemented by a data structure called Node Control Block (NCB), as described in Fig. 3.5, defined in the MESSENGERS runtime program. One member of `struct NCB` is a list of Node Variables Areas (NVAs). An NVA stores the node variables declared in a node file. The list of NVAs therefore includes the node variables in all the node files loaded to a logical node. A node file is loaded to a logical node when a Messenger thereon references variables declared in the node file. The `link_list` member is a list of `struct LINKs`, through them an NCB is directly or indirectly connected to other NCBs forming a logical network.

```
struct NCB {
    char                name[64];
    uint32_t            address[2];
    struct splay_tree   *nva_list;
    struct doubly_linked_list *link_list;
    ... //other members
};
```

(a)

```
struct NVA {
    char    name[64];
    struct NCB *ncb;
    int     data_size;
    char    *data;
    int     heap_size;
    char    *heap;
    ... //other members
};
```

(b)

Figure 3.5: `struct NCB` and `struct NVA`. (a) NCB. (b) NVA.

## Implementation of Messenger

A Messenger is implemented by a data structure called Messenger Control Block (MCB), as described in Fig. 3.6, defined in the MESSENGERS runtime program.

```
struct MCB {
    char    name[64];
    struct NCB *NCB;
    int     next_function_nr;
    int     (** functions)(struct MCB *);
    ...
    int     msgr_variables_area[];
};
```

Figure 3.6: struct MCB.

The `name` member in `struct MCB` stores the name of the shared library file associated with the Messenger represented by the MCB. The `next_function_nr` member is an identification number for a `_msgr_function` in a Messenger C program generated, in a process described in Sec. 3.1, from the Messenger definition corresponding to the Messenger. This `_msgr_function` is the one to be executed next when the current `_msgr_function` finishes its execution. The `functions` member is a linked list of `_msgr_functions` and functions in external libraries called in a Messenger definition. The `NCB` member points to the logical node where the Messenger represented by the MCB is currently located. The Messenger references node variables by following this pointer. The `msgr_variables_area[]` is a variable-length array to store the Messenger variables of the Messenger. Not counting `msgr_variables_area[]`, the size of `struct MCB` is about 220 bytes.

### 3.2.2 Overview of execution of daemon

The MESSENGERS user starts one daemon on each of the participating hosts, as specified in the MESSENGERS profile, in the physical network, by executing the runtime program, named `messengers`, on each host. These daemons form the daemon network, as described in Sec. 3.2.3. Then each daemon enters a loop (see Fig. 3.7) which is exited only upon the receipt of a SIGINT signal generated by the user pressing Control-C on the keyboard, causing the `is_to_shut_down` flag to be set. Inside the loop are calls to functions in the runtime program that process the Messengers, if any, injected or hopped from other daemons, followed by a call to `wait_for_msgr_or_packet()`, which in turn calls the Unix system call `sigwait()`, which suspends the daemon until awoken by a UNIX signal SIGUSR1.

```
void run_daemon(bool is_this_the_only_daemon)
{
    bool is_to_shut_down = false;
    int ret = 0;
    while (!is_to_shut_down) {
        retrieve_msgrs(ready_queue, interface_with_minject);
        exec_msgrs(ready_queue);
        if (!is_this_the_only_daemon) {
            receive_new_pckts(ready_queue, packets_queue);
            ret = exec_daemon_pckts(packets_queue);
            if (ret == TERMINATION) {
                is_to_shut_down = true;
                break;
            }
        }
        if (is_mcb_list_empty(ready_queue)) {
            ret = wait_for_msgr_or_packet();
            if ((ret == SIGINT) || (ret == ERROR))
                is_to_shut_down = true;
        }
    }
}
```

Figure 3.7: Simplified definition of `run_daemon()` in the runtime program

To run a MESSENGERS application, the user executes, in one of the hosts, a pro-

gram, named `minject`, to inject a Messenger. (This very first Messenger may call `inject()` to inject other Messengers.) The `minject` program takes as its parameters the name of the shared library file associated with the Messenger, and its parameters, if any. This `minject` program sends `SIGUSR1` and, through the Unix inter-process communication mechanism of message queue, data representing the injected Messenger to the daemon running on the host. The daemon, awoken by `SIGUSR1`, starts the execution of this Messenger, as described in Sec. 3.2.5, and other Messengers injected by it. When the execution is complete and there are no more Messengers ready to be executed, the daemon calls `wait_for_msgr_or_packet()` to suspend itself.

The runtime program is multithreaded using POSIX pthreads. Each daemon has

1. one thread, referred to as the computation thread, to execute the Messengers on the logical nodes created on the physical node that the daemon represents,
2. one thread to run the socket sending outgoing data (representing Messengers), and
3. a number of threads, one for each of the other daemons, to run the sockets receiving incoming data (representing Messengers).

The sockets are discussed in Sec. 3.2.3.

### **3.2.3 Formation of daemon network**

Upon execution, the daemons learn about each other by reading the `MESSENGERS` profile and set up Internet domain (IP v.4) Stream sockets among themselves to create the daemon network. The socket application programming interface (API) is supported by POSIX-compliant systems. A daemon has an outgoing socket respon-

sible for sending data to all other daemons, and a number of incoming sockets each responsible for sending data to one other daemon in the daemon network. The host names, e.g., tarheel.ics.uci.edu, are used to set up the sockets.

### 3.2.4 Injection of Messenger

Upon the receipt of a SIGUSR1 signal and a Messenger injected by the `minject` program, as described in Sec. 3.2.2, a daemon calls the `retrieve_msgrs()` function which in turn calls the `create_mcb()` function, a simplified version of whose code is shown in Fig. 3.8, to create an MCB based on the arguments of the `minject` program. There exists a dynamic linking library function `dlopen()` that allows a program to load a library at run time and is often used to load a plug-in program. With `dlopen()`, the `create_mcb()` function loads the shared library file associated with the injected Messenger, if it is not already loaded, using the dynamic linking loader. Then it obtains, by `dlsym()`, another dynamic linking library function, the addresses of a number of pre-specified initialization functions in the Messenger C program (see Fig. 3.3) associated with the injected Messenger, and executes a couple of them, setting the `functions` member of the `struct MCB` being created to the `functions` linked list in the Messenger C program, which stores individual `_msgr_functions`. The `next_function_nr` member of the `struct MCB` being created is set to 1, meaning that the next function to be executed (see Sec. 3.2.6) is `functions[1]`, i.e., `_msgr_function_1`. The MCB is enqueued to a ready queue (see Sec. 3.2.5) and initially associated with a pre-defined initial logical node created by the runtime on the host where the Messenger is injected. The MCB is created in the heap area of the address space of the daemon.

When a Messenger is injected by an `inject()` statement in a Messenger definition,

a similar process takes place to create an MCB, except that the MCB is associated with the same logical node where the injecting Messenger is located.

```
struct MCB *create_mcb(const char *msgr_name,
    const char *argv[])
{
    struct MCB *new_mcb = NULL;
    struct msgr_lib *msgr_lib;
    void (*init_lib)(void *);

    msgr_lib->handle = dlopen(msgr_name, 1);
    strcpy(msgr_lib->name, msgr_name);
    msgr_lib->create = dlsym(msgr_lib->handle, create_msgr);
    msgr_lib->init = dlsym(lib->handle, init_msgr);
    init_lib = dlsym(lib->handle, init_library);
    (* init_lib)(lib->handle);
    new_mcb = (*(msgr_lib->create))(argv);
    strcpy(new_mcb->name, msgr_name);
    ...
    return new_mcb;
}
```

Figure 3.8: Definition of `create_mcb()` in the runtime program.

### 3.2.5 Ready queue and scheduling of Messengers

A ready queue is a data structure collecting the units of scheduling, Messengers in the case of the MESSENGERS system, arranged in the order of execution based on some scheduling policy. Given that a physical node is an abstraction of some processing capability, one ready queue is associated with one physical node, or alternatively speaking, one daemon. Each daemon creates a ready queue, implemented by an “internal-storage”-based doubly linked list. The ready queue has all the MCBs ready for execution on the logical nodes associated with the physical node (i.e., created in the daemon). The MCBs are executed on a first-come-first-served (FCFS) basis. The runtime program calls a function `exec_msgrs()`, a simplified version of whose code is

shown in Fig. 3.9, to loop through all the MCBs on the ready queue. Each iteration of the loop calls `exec_mcb()` (see Sec. 3.2.6) that returns an integer. Based on the value of this integer, the runtime program decides whether to continue executing the current MCB or execute another MCB. Specifically, the next `_msgr_function` of the current MCB is to be executed unless it is derived from a `stall()`, a `create()` or a `hop()` statement, in which cases the MCB is moved to the end of the ready queue (and thus (a `_msgr_function` of) the next MCB on the ready queue is to be executed).

```

void exec_msgrs(struct mcb_list *ready_queue)
{
    struct MCB *current_mcb, *next_mcb, *prev_mcb;
    int next_stmt_kind;

    prev_mcb = NULL;
    current_mcb = get_head_of_mcb_list(ready_queue);
    while (!is_mcb_list_empty(ready_queue)) {
        next_mcb = get_next_of_mcb_list(ready_queue, current_mcb);
        next_stmt_kind = exec_mcb(current_mcb);
        if (prev_mcb == NULL)
            current_mcb = get_head_of_mcb_list(ready_queue);
        else
            current_mcb = get_next_of_mcb_list(ready_queue, prev_mcb);
        if (next_stmt_kind == RELEASE_CPU) {
            prev_mcb = current_mcb;
            current_mcb = next_mcb;
        }
    }
}

```

Figure 3.9: Definition of `exec_msgrs()` in the runtime program.

### 3.2.6 Execution of Messenger

The execution of a Messenger is implemented as the execution of a sequence of functions stored in the `functions` linked list in the `struct MCB` representing the Messenger. As pointed out before, the functions in this `functions` linked list are those `_msgr_functions` in the corresponding Messenger C program. Each such execution

of a (single) `_msgr_function` is performed by the `exec_mcb()` function, whose code is shown in Fig. 3.10, defined in the runtime program. During the execution of a `_msgr_function`, the `next_function_nr` of the MCB is set to indicate the next `_msgr_function` (of the same MCB) to be executed, as shown in Fig. 3.2. The `exec_mcb()` function returns an integer indicating the kind of statement in the Messenger definition that corresponds to the next `_msgr_function` to be executed. That is, the integer indicates whether the next `_msgr_function` is derived from a `create()` statement, a `hop()` statement, or a `stall()` statement, for example, in the Messenger definition.

```
int exec_mcb(struct MCB *mcb)
{
    return (* mcb->functions[mcb->next_function_nr])(mcb);
}
```

Figure 3.10: Definition of `exec_mcb()` in the runtime program.

### **Creation of a logical node**

When a logical node is created by the `create()` statement in a Messenger definition, an NCB is created in the heap area of the address space of the daemon that runs the calling Messenger, or more precisely, that loads the shared library associated with the corresponding Messenger definition.

### **Injection of another Messenger**

The mechanism involved in the injection of another Messenger by a Messenger is described in Sec. 3.2.4.

## Migration of Messenger

When a `_msgr_function` derived from a `create()` or a `hop()` statement in a Messenger definition is executed to migrate a Messenger to a logical node on another daemon (i.e., another physical node), the MCB representing this Messenger is dequeued from the daemon's ready queue and enqueued to a sending queue — see Fig. 3.11(a). A sending thread retrieves this Messenger from the sending queue and writes it, as a block of bytes, to an outgoing socket associated with the daemon for the destination logical node — see Fig. 3.11(b). The receiving thread of the destination daemon receives this block of bytes and creates a `struct MCB` from it. If the Messenger migrates to a logical node on the same daemon, the MCB representing this Messenger is moved from the front to the end of the ready queue.

## Synchronization

In the case of `waitEvent()`, the corresponding `_msgr_function` in the Messenger C program returns a value indicating the current MCB should continue to be executed if the event is already signaled, or a value indicating that the next MCB should be executed if the event is not yet signaled. In the latter case, the current MCB is moved to a wait queue, and will be put back to the ready queue when the event is signaled.

### 3.2.7 References to node variables

Taking advantage of the dynamic binding of node variables, the MESSENGERS runtime does not need to implement mechanism for some type of address resolution and address translation for and maintaining consistency of node variables had they been accessible from a remote computer. The dynamic binding of node variables ensures

```

void move_msgr_to_route(struct MCB *mcb, struct ROUTE *route,
    struct mcb_list *ready_queue)
{
    ...
    if (route != daemon_body.incoming_route) { /* MCB goes to
                                                another daemon */
        remove_from_mcb_list(ready_queue, mcb);
        enqueue_packet_for_sending(route->channel->packet_queue,
            (struct packet *)mcb);
    }
}

```

(a)

```

int send_packet(const struct channel *chnl, struct packet *p)
{
    int sd;
    sd = chnl->socketDescriptor;
    if (p->size > 0)
        ret = write_int32_via_socket(sd, p->size);
    if (ret == sizeof(p->size))
        ret = write_chars_via_socket(sd, (char *)p+sizeof(p->size),
            p->size-sizeof(p->size));
    ...
    return ret;
}

```

(b)

Figure 3.11: Definitions of functions sending a MCB to an remote daemon. (a) `move_msgr_to_route()` executed by the computation thread to put an MCB to the sending queue. (b) `send_packet()` executed by the sending thread to write the data in the MCB to an outgoing socket.

that they are visible only to a Messenger on the local computer, assumed that only one computer can be mapped to one physical node. In fact, the main advantage of the dynamic binding is the ease of implementing the support for the programming model. If a Messenger is on one logical node but could reference a node variable declared in another logical node, and these two nodes are on different computers, then the runtime needed to find the logical node where the node variable is located, which may not be easy. Further, there would be potential consistency problem for the node variable if it is replicated on different computers. Some kind of consistency model

and mechanism would be needed.

### **3.2.8 Multithreading of daemon and sending/receiving queues**

The runtime program is multithreaded using the pthreads interface. As described in Sec. 3.2.2, the execution of Messengers is performed by a computation thread, the receiving of Messengers from other daemons is performed by a receiving thread, and the sending of Messengers to other daemons is performed by a number of sending thread each for a destination daemon. The receiving thread enqueues an incoming Messenger to the receiving queue, sends a SIGUSR1 signal to the computation thread, which then dequeues the Messenger from the receiving queue and enqueues it to the ready queue. The communication (sending and receiving) threads have a higher priority than the computation thread.

### **3.2.9 Support for fine-grained tasks and migration**

The MESSENGERS runtime system described above efficiently supports fine-grained tasks, on a single-CPU host, by running all Messengers on that host sequentially. The creation of a Messenger does not entail the creation of a thread ever. There is no overhead of managing a large number of threads. The single computation thread and the multiple communication threads achieve overlapping of computation and communication. To take advantage of the run-to-completion-or-cooperation semantics of a Messenger, the runtime system employs a first-come-first-served scheduler.

As (the computation of) all Messengers on a host are executed in one thread, the state of a Messenger, the migrating entity, cannot be captured and restored using conventional approaches associated with threads, such as the use of the Standard C

Library functions `setjmp()` and `longjmp()` employed by systems like Ariadne, as will be discussed in Chapter 7. Instead, the values of the migratable variables of a Messenger are captured and restored as part of the MCB data structure, rather than as part of some machine-dependent stack, etc. (That is, the so-called logical state, which depends on the language only, is saved and restored.) The values of the node variables at the source logical node of the migration are not needed at the destination logical node, nor are links to these node variable needed.

In addition to being able to satisfy the need to efficiently run fine-grained tasks, the capture and restoration of the logical state of a Messenger rather than the machine-level context of a thread allows for the possibility of migrating a Messenger on heterogeneous systems. In fact, most, if not all, of migration systems running on heterogeneous systems uses an approach to capture and restore the logical state of the migrating entity.

# Chapter 4

## Case Study of MESSENGERS

### Programming

There is a spectrum for the design space of a programming model with respect to its abstraction. On one end, a model may need to be as abstract as possible to hide the details of the hardware. One major reason for that is that different hardware architectures have different characteristics and if a programming model is not abstract enough, there may be different models for different architectures. On the other end, a model may need to be as closely mapped to a particular architecture as possible for performance reason.

The MESSENGERS programming model is abstract enough and can hide differing characteristics of a large number of architectures so that it can accommodate them, but concrete enough so that it can allow the programmer to take advantage of the characteristics of each architecture. This chapter shows how to use the MESSENGERS model to take advantage of the different characteristics of different architectures when programming a numeric algorithm on:

1. a network so that the communication cost can be minimized and the level of parallelism maximized, and
2. a (uniprocessor) computer so that the cache can be efficiently utilized.

A numeric algorithm called Crout factorization by Hughes et al [34] that factorizes an  $N \times N$  matrix is presented as follows. Two matrices L and D are obtained as a result by the algorithm where  $K = L^TDL$ , K is a symmetric and positive-definite matrix to be factorized, L is a non-singular lower triangular matrix,  $L^T$  is an upper triangular matrix with unit diagonal entries, and D is a diagonal matrix.

```

for (j = 1 .. N)
  for (i = 1 .. j - 1)
     $K_{ij} = K_{ij} - \sum_{h=1}^{i-1} K_{hi} \times K_{hj}$ 
  for (i = 1 .. j - 1)
    t =  $K_{ij}$ 
     $K_{ij} = t / K_{ii}$ 
     $K_{jj} = K_{jj} - t \times K_{ij}$ 

```

Figure 4.1: Sequential program for Crout factorization.

Figure 4.1 shows the sequential program for Crout factorization. The algorithm works in place and K is overwritten by L and D.

Due to K's symmetry, only its upper triangular part is stored.

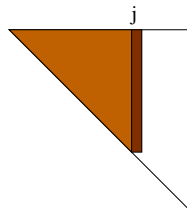


Figure 4.2: Working set of an iteration of the outermost loop in the Crout factorization algorithm.

The algorithm has good reference locality. The values of the entries in the  $j^{\text{th}}$  column

are updated using the values of the entries in the first through  $(j - 1)^{\text{th}}$  columns. That is, the first  $j - 1$  columns are the working set when the  $j^{\text{th}}$  column is updated. Figure 4.2 shows a triangular matrix, with the shaded area being the working set for the  $j^{\text{th}}$  column.

The algorithm is amenable to pipeline concurrency. The updating of the  $j^{\text{th}}$  column can be performed concurrently with the updating of the  $(j - 1)^{\text{th}}$  column as long as when the updating of the  $j^{\text{th}}$  column is about to read the values of the entries in the  $(j - 1)^{\text{th}}$  column, the updating of the latter is complete.

```

#include "data.node" // node file declares K, evt, and node[]
int $arg1 // Messenger parameter
int h,i,j // Messenger variables
float t // Messenger variable
float col[],diag[] // Messenger variables
j = $arg1
hop(node[j])
col[] = K*_j
for (i = 1 .. j - 1)
  hop(node[i])
  if (j > 1 && i == 1) waitEvent(evt,j - 1)
  diag[i] = Kii
  col[i] = col[i] -  $\sum_{h=1}^{i-1} K_{hi} \times col[h]$ 
  if (i == 1) signalEvent(evt,j)
hop(node[j])
K*_j = col[]
for (i = 1 .. j - 1)
  t = Kij
  Kij = t / diag[i]
  Kjj = Kjj - t  $\times$  Kij

```

Figure 4.3: Definition of Messenger for Crout factorization.

One way to express the concurrency in this algorithm is to decompose the work into  $N$  parts, each corresponding to an iteration of the outermost loop, and execute each concurrently. Using MESSENGERS, each one of these  $N$  units of work is performed by a Messenger called `factorizer`. Figure 4.3 shows the pseudocode for `factorizer`. `node[]` is an array that maps a column number to a logical node with which a node

variable for that column is associated. There are another Messenger that builds a fully connected logical network, maps logical nodes to physical nodes, and injects these  $N$  factorizers in a starting node, and some other Messengers that load the matrix data. The definitions of these Messengers are not shown here.

A number of logical nodes are created. A column of the matrix is the unit of data distribution, and the matrix is distributed among these logical nodes so that there is certain number of columns in the node variable in each logical node. One factorizer is assigned to update each of the  $N$  columns of the matrix. The  $j^{\text{th}}$  factorizer first hops to the logical node where the  $j^{\text{th}}$  column is stored, saves the column from the node variable to its Messenger variable, and hops to the logical nodes where the previous  $(j - 1)^{\text{th}}$  columns are stored and performs the computation involving each of these columns and the  $j^{\text{th}}$  column in its Messenger variable. Finally, the Messenger hops back to the logical node the  $j^{\text{th}}$  column is stored and updates the values of the entries in the column in the node variable.

It is very likely that  $N$  is much larger than the number of CPUs available in the physical network; as a result, each Messenger is a fine-grained task as described in Sec. 1.1.2.

In the MESSENGERS implementation, the upper triangular matrix is stored in column-major order in a linear array.

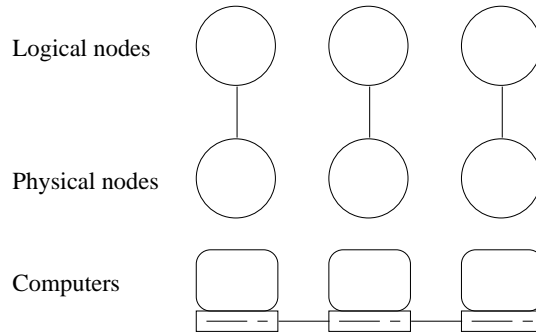


Figure 4.4: Mapping of physical and logical nodes on a network of uni-processor computers.

## 4.1 Mapping of physical and logical nodes on network

In prior work [52], one physical node and one logical node were created for and mapped to each uni-processor computer in the network, as shown in Fig. 4.4: the physical node represents the processing resources of the computer and is the object for computation distribution, whereas the logical node represents the storage resources of the computer and is the object for data distribution.

The need for a Messenger on one logical node (and hence one physical node) to migrate to another logical node (and hence another physical node) to access data, and therefore incurring some cost of migration, makes clearer that the cost of accessing that (remote) data is higher than the cost of accessing data on the current logical node. The cost of accessing remote data is not the cost of moving the remote data to the local node; rather it is the cost of moving the Messenger to the remote node. The size of the Messenger variables of a Messenger measures the communication cost, i.e., the time needed to send the data in the Messenger variables from the computer mapped to the physical node associated with the source logical node to the computer mapped to the physical node associated with the destination logical node. The `hop()`

statement signifies the incurrence of the communication cost.

One can use a block-cyclic data distribution scheme to distribute the matrix  $K$  onto different logical nodes, and the `hop` statement to distribute computation on different physical nodes, as described in Sec. 2.3.7. A Messenger migrates directly to a logical node to access data thereon and indirectly to the physical node associated with that logical node to compute.

Experiments were performed on a network of 12 uni-processors and satisfactory speed-up was achieved.

Additional examples of running MESSENGERS on a network of uni-processors with various numeric algorithms can be found in other papers [53, 54].

## 4.2 Mapping of physical and logical nodes on uni-processor computer

On a uni-processor computer, one can decrease the execution time of a program by increasing the number of times a piece of data is used while it is resident in the cache. How to do that using MESSENGERS is explained below.

A single physical node is used to represent the processing capability of the uni-processor. As such, there is only one ready queue. According to the scheduling policy of the MESSENGERS model, as described in Sec. 3.2.5, when a Messenger is to migrate from one logical node to another logical node, both associated with the same physical node, it is moved to the back of the ready queue, and another Messenger (now at the front of the queue) is to be executed. As a result, all Messengers ready to be executed on one logical node would finish the part of their execution,

before the migration, on that node before any one of those which has just hopped to another logical node is executed. In other words, the part of computation, of any Messenger, on a given logical node before its migration to another logical node, is completed before any other part of computation (of any Messenger) on another logical node starts.

To improve cache performance, multiple logical nodes can be created in such a way that the size of (the node variables on) a logical node, or a part thereof, that all the Messengers are located on before any of them migrates to another logical node is limited to the size of the cache on the computer. To illustrate the idea, first consider the case using block distribution. Assumed that the  $N$  columns of the matrix  $K$  are distributed to  $n$  logical nodes, and that the cache is fully associative, that is, any cache line can store the contents of any memory location. The first  $N/n$  columns are distributed to the first logical node, the second  $N/n$  columns to the second logical node, and so on. If the size of a logical node is larger than the cache size, then at least part of the cache needs to be replaced before the execution on that logical node of a particular Messenger is complete, as a Messenger runs to completion unless it migrates (or yields the CPU voluntarily, or blocks due to synchronization). The next Messenger would again access the data in this logical node and the access would cause cache replacement because the cache is not large enough to hold all the data in that logical node. On the other hand, if the size of a logical node is less than the cache size, then there is no cache replacement needed (until all the Messengers on that logical node finish their part of execution on that node). When all the Messengers on the ready queue is associated with another logical node (namely, the destination node, as a result of their migration), the data related to the previous logical node can be flushed out from the cache and will never be used again. That is, during a particular phrase of the program execution, all the data it needs is in the cache, and when that phrase is complete, those data are no longer needed.

As an example, assumed that the cache is 2 MB large and the matrix has 3000 columns, each entry of the matrix is a 4-byte-long `float`, and a block distribution scheme with a block size of 200 columns is used to distribute the matrix to 15 logical nodes. Table 4.1 shows the size of each logical node based on this distribution scheme. For the sake of simplicity, it is assumed that the matrix is the only piece of data stored as node variable, and that the sizes of the Messenger variables are insignificant. Only the last two logical nodes have sizes slightly larger than 2 MB. Therefore, the utilization of the cache can be improved by this data distribution, combined with MESSENGERS’s scheduling policy.

Table 4.1: Block distribution of a 3000×3000 upper triangular matrix, each entry of which is 4 bytes large, to 15 logical nodes each allocated 200 columns. If  $s$  is the accumulated number of columns in the current and all previous logical nodes, accumulated size is  $s \times (s + 1)/2 \times 4$ , and size of a logical node is its accumulated size less the accumulated size of its previous node.

Logical node	Accumulated no. of cols.	Accumulated size (bytes)	Size of logical node (bytes)
1	200	80,400	80,400
2	400	320,800	240,400
3	600	721,200	400,400
4	800	1,281,600	560,400
5	1000	2,002,000	720,400
6	1200	2,882,400	880,400
7	1400	3,922,800	1,040,400
8	1600	5,123,200	1,200,400
9	1800	6,483,600	1,360,400
10	2000	8,004,000	1,520,400
11	2200	9,684,400	1,680,400
12	2400	11,524,800	1,840,400
13	2600	13,525,200	2,000,400
14	2800	15,685,600	2,160,400
15	3000	18,006,000	2,320,400

Note that because a column is a unit of data distribution and the matrix to be factorized,  $K$ , is symmetric and only its upper triangular half is stored, the  $j^{\text{th}}$  column is ‘longer’ than the  $(j - 1)^{\text{th}}$  column in the sense that the former has one more entry than the latter. Therefore, when block data distribution is used, the total size of data distributed to one logical node is different from the total size of data distributed

to another logical node. When block-cyclic data distribution is used, it is also likely to be so. For example, if the matrix has 6 columns and is to be distributed to two logical nodes in a block-cyclic manner with a block size of one column, then columns 1, 3, and 5 are distributed to the first logical node, whereas columns 2, 4, and 6 to another. As column 2 is ‘longer’ than column 1, column 4 is ‘longer’ than column 3, and column 6 is ‘longer’ than column 5, the total size of data distributed to the second logical node is larger than that to the first node.

It is worth noting that if the size of each column were the same, then one could have created less than 15 logical nodes with the block distribution scheme.

Table 4.2: Block-cyclic distribution of a 3000×3000 upper triangular matrix, each entry of which is 4 bytes large, to 2 logical nodes using a block size of 200 columns. If  $s$  is the accumulated number of columns in the current and all previous blocks, accumulated size is  $s \times (s + 1)/2 \times 4$ , and size of a block is its accumulated size less the accumulated size of its previous block.

Block	Accumulated no. of cols.	Accumulated size (bytes)	Size of block (bytes)	Logical node 1	Logical node 2
1	200	80,400	80,400	80,400	
2	400	320,800	240,400		240,400
3	600	721,200	400,400	400,400	
4	800	1,281,600	560,400		560,400
5	1000	2,002,000	720,400	720,400	
6	1200	2,882,400	880,400		880,400
7	1400	3,922,800	1,040,400	1,040,400	
8	1600	5,123,200	1,200,400		1,200,400
9	1800	6,483,600	1,360,400	1,360,400	
10	2000	8,004,000	1,520,400		1,520,400
11	2200	9,684,400	1,680,400	1,680,400	
12	2400	11,524,800	1,840,400		1,840,400
13	2600	13,525,200	2,000,400	2,000,400	
14	2800	15,685,600	2,160,400		2,160,400
15	3000	18,006,000	2,320,400	2,320,400	

Instead of creating 15 logical nodes with block distribution of data as above, one can create two logical nodes, as shown in Fig. 4.5, with block-cyclic distribution. Table 4.2 shows how the 15 blocks, each has 200 columns, are cyclically allocated to two logical nodes. The Messengers would access data in the first block (on the first logical node) and then after, and only after, the execution using those data is

complete, they would access data in the second block (by migrating to the second logical node). The total size of the first 5 blocks are approximately 2 MB. That is, there is no cache replacement until the sixth block is accessed. Once the data in the cache is replaced, they do not need to be reloaded. The total size of the sixth and the seventh blocks are approximately 2 MB. When the Messengers finish with the data in the seventh block, which is allocated to the first logical node, the part of their execution using data in the eighth block starts. A change of the logical node associated with these Messengers (from the first logical node to the second, or from the second to the first) indicates that the data in the cache just accessed can be replaced without being reloaded. Except for the last two blocks with slightly larger sizes, the sizes of all the blocks are smaller than the cache size. That means no cache replacement takes place if the otherwise replaced data would be again needed.

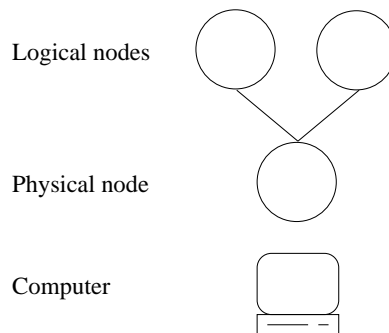


Figure 4.5: Mapping of physical and logical nodes on a uni-processor computer.

Using the above approach to improve cache reuse, the size of the node variables referenced by a Messenger on a logical node before migration represents the cache replacement cost because when a Messenger migrates to another logical node, a new set of node variables is loaded to the cache and the cache lines storing the old set may be flushed. The `hop()` statement executed by the last Messenger on a logical node before a Messenger on another logical node is executed signifies the starting point of potentially incurring the cache replacement cost.

Table 4.3: Performance of running the MESSENGERS program for Crout factorization on a uniprocessor Sun Ultra-5 workstation with 2 MB cache.

Order of matrix	# logical nodes	Block size (# cols)	Size of largest block (MB)	Time (s)	Speed up
2000	1	-	8	51.31	1.00
2000	2	100	0.8	45.08	1.14
2000	2	200	1.5	43.81	1.17
2000	2	400	2.9	44.32	1.16
2000	2	800	3.8	49.72	1.03
3000	1	-	18	180.31	1.00
3000	2	100	1.2	162.46	1.11
3000	2	200	2.3	167.21	1.08
3000	2	400	4.2	170.83	1.06
4000	1	-	32	464.10	1.00
4000	2	100	1.6	422.13	1.10
4000	2	200	3.1	437.52	1.06
4000	2	500	7.5	480.23	0.97

Experiments, using the same MESSENGERS program for Crout factorization as in Sec. 4.1 in conjunction with block-cyclic data distribution on two logical nodes, were performed on a Sun Ultra-5 workstation with one 400 MHz UltraSPARC-IIi processor, 128 MB memory and 2 MB external 2-way, set associative level-2 cache, that runs Solaris 9. The MESSENGERS compiler called GCC 3.4.4 and the MESSENGERS runtime was also compiled using GCC 3.4.4. As shown in Table 4.3, there was noticeable performance improvement when two logical nodes, instead of one, were used with the block-cyclic data distribution, as described above. Best speed-up was achieved when the sizes of all the blocks were below the cache size.

The way this approach, taking advantage of MESSENGERS's scheduling policy of context-switching upon migration and the use of logical nodes to partition data, works is similar to the use of loop tiling to improve cache performance. Loop tiling [77] is a widely-used optimization technique in compilers with which the loop iteration space is divided into smaller blocks (tiles) in such a way that the working set for the tile fits into the cache, resulting in the maximization of the reuse of array data within each tile.

In conclusion, the appropriate use of multiple logical nodes, under the MESSENGERS cooperative scheduling policy, orders the computation in such a way that cache misses, resulting from eviction from the cache of reusable data due to cache size limitation, are reduced and cache thrashing is prevented.

# Chapter 5

## Vertical Scaling and Multithreading

### 5.1 Vertical and horizontal scaling

Two approaches to adding more processing capability to a computer system can be discerned: vertical scaling (also known as scaling up) and horizontal scaling (also known as scaling out). Vertical scaling is basically adding more processors (or cores) within a computer; horizontal scaling is adding more computers to the existing one(s).

A single computer, vertically scaled, in general is easier to manage than a network of computers. After a single computer is vertically scaled, an application can still run without modification though the performance may not improve. And all the processors (cores) share within that computer the same resources, which may constrain the scalability of applications. One drawback of this approach is that the computer is a single point of failure. And because the processors share the same memory, the time cost of communication among processors is low relative to that among computers. If different cores happen to share the same cache, the cost is even lower. Consequently, vertical scaling is most effective in cases in which the processes or threads have to

share significant amounts of data.

In the case of horizontal scaling, the resources scale with the number of computers added. However, the time cost of communication between two computers is higher, suggesting that this approach is better when there is relatively little data sharing among computers. Further, a non-distributed program on one single computer may need some modification, or some additional runtime support, to run after more computers are added.

Of course vertical scaling and horizontal scaling can be combined (and the approach is called diagonal scaling by some). Many of today's top performance computer systems are in the form of clusters of multiprocessors or multi-core computers. According to the TOP500 Project [65], 81.2% of the world's most powerful high-performance computers at November 2007 were clusters, and multi-core processors were the dominant chip architecture. In the November 2008 ranking [66], eight out of the top ten high-performance systems were clusters of mostly quad-core nodes.

As will be described in the following, significant progress has been made in recent years in the technology in vertically scaling a computer. A large amount of research has also been performed in designing software to keep in line with this trend of hardware scaling.

The MESSENGERS runtime system as designed by Wicke (see Sec. 3.2) focused on horizontally scaled systems. The work described in the latter parts of this dissertation aims at improving the performance of the MESSENGERS runtime system on vertically scaled systems.

## 5.2 Processor support for multiple CPUs on computer

In this dissertation, a CPU is defined as an execution engine capable of executing a single stream of instructions, namely, a thread. There can be more than one CPU on a stand-alone computer. These CPUs share memory, I/O facilities, and an operating system in the computer. The multiplicity of CPUs on a computer node can be achieved by three major processor technologies:

1. symmetric multiprocessing
2. chip multiprocessing
3. simultaneous multithreading

These designs can be combined together and there can be, for example, a computer with dual processors, each has two cores, and simultaneous multithreading enabled on each core to create two virtual processors per core, making a total of eight CPUs on the computer. Alternatively, simultaneous multithreading can be enabled on a dual uni-core processor or a single dual-core processor, for instance.

### 5.2.1 Symmetric multiprocessing

A processor is a physical chip on a circuit board (i.e., a package) which is then plugged into a socket inside a computer. The traditional approach of scaling up a computer is to plug in multiple boards, each may have multiple processors. A multiprocessor computer can be UMA (or SMP) or NUMA.

Although every processor in all (shared-memory) multiprocessors can access all of the memory, the processors in the uniform memory access (UMA) machines can read every memory word as fast as every other memory word, but the processors in the non-uniform memory access (NUMA) machines cannot.

In a symmetric multiprocessor (SMP) architecture, every processor is identical. On the other hand, one or more processor in the asymmetric multiprocessor architecture may be specialized processor(s), or otherwise different from the rest of the processors. Using the asymmetric configuration, each processor can optimize a specific portion of the work.

Whereas it is theoretically possible that processors in a UMA machine are not identical or processors in an SMP machine do not access memory in identical time, in practice UMA and SMP generally are equivalent.

SMP couples processors and their memory together more closely than NUMA does. NUMA partitions all of the main memory into local and shared memory; accessing local memory is faster than accessing shared (non-local) memory. NUMA costs less to manufacture than SMP for computers with more than four processors, and as a result server makers adopt the NUMA design [70].

### **5.2.2 Chip multiprocessing**

Multi-core refers to multiple execution engines (cores) on an integrated circuit (IC) chip in the same IC package. This chip multiprocessing (CMP) technology allows one socket to provide access to multiple cores. Each of the cores has its own resources such as architecture state, registers, execution units. Each core has its private level 1 (L1) cache and either has a private level 2 (L2) cache or share a common L2 cache

with other cores. A shared level 3 cache is also possible. A multi-core arrangement with low-clock-speed cores is designed to minimize power consumption and deliver lower heat output than configurations relying on a single high-clock-speed core [19].

Multi-core architectures are becoming popular in recent processor designs. Many expect to see tens or even hundreds of cores per processor (so-called many-cores) within the next few years [10]. As the number of cores on a chip increases, a shared bus connecting the cores to the memory is overwhelmed. As a result, some cross-bar interconnect, such as AMD's HyperTransport and Intel's QuickPath Interconnect, is employed. This makes the configuration a NUMA one. Both AMD's Operton multi-cores and Intel's Nehalem multi-cores, the latter released in November 2008, as well as Intel's upcoming Tukwila multi-cores, have the NUMA characteristics. It was reported by Kanter [39] that with Nehalem, access latency of remote memory is 1.5 times that of local memory. On the other hand, recent research [4, 6, 26] proposed performance-asymmetric (or heterogeneous) multi-core configuration, in which the cores have the same instruction set but different performance characteristics such as clock speed and issue width.

### 5.2.3 Simultaneous multithreading

Simultaneous multithreading (SMT), or known as hyper-threading as implemented by Intel, adds multiple circuitry and functionality into a traditional processor to enable one physical processor to appear as multiple (two in Intel's case) separate virtual processors, so that one thread can execute on each virtual processor simultaneously with others. Scheduling two (or more) threads on the same processor core allows better use of the processor resources such as cache, registers, and execution units. SMT permits several threads to issue instructions to these virtual processors in a

single cycle. Whereas it may increase processor utilization, it can add substantial complexity to the design [68]. Operating system support is needed to recognize the virtual processors and BIOS support is needed to enable/disable SMT. SMT aims at using single-core resources more efficiently whereas CMP tries to increase computation throughput by providing multiple complete sets of execution resources.

## 5.3 Operating system support for multiple CPUs and multithreading

### 5.3.1 General

Instead of having a private copy of the operating system for each processor in a multiprocessor computer, a multiprocessor operating system allows all the processors to share the operating system code and make private copies of only the data. The scheduling in a multiprocessor operating system is two-dimensional: the scheduler has to decide which process/thread to run next, and which processor it is to run on.

A process is a construct used by the operating system to encapsulate resources such as address space and open files. A process is independent of each other though multiple processes can share memory or I/O. A thread has a weight (i.e., size of the context) lighter than a process. The POSIX specification [35] defines a process as

An address space with one or more threads executing within that address space, and the required system resources for those threads

and a thread as

A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, errno value, thread-specific key/value bindings, and the required system resources to support a flow of control. Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via `malloc()`, directly addressable storage obtained through implementation-defined functions, and automatic variables, are accessible to all threads in the same process.

There used to be only a single thread within a process. More recent operating systems support multiple (kernel) threads within a process; they share many of the process's resources but each has its own stack. A problem encountered by a thread may kill the entire process. In systems such as Solaris 10, a (kernel) thread is a unit of scheduling for a CPU. That is, the operating system schedules on the basis of a (kernel) thread, not a process. Instructions in each thread are executed independently of those of other processes and threads.

The Linux kernel does not exactly distinguish between a process and a thread. Two so-called tasks, one forked by another, are related to each other as either two processes or two threads depending on how individual flags in the data structure representing the child task are set when it is created. For example, if the flags are set so that the child task does not share the same address space, etc. with its parent, they relate to each other as independent processes. If the flags are set so that the child task shares the same address space, etc. with its parent, then they can be considered as threads within a process. A task is the unit of scheduling on Linux. For the purposes of this dissertation, a Linux task is considered as a kernel thread.

In addition to the kernel threads that are known and scheduled by the operating system, or if kernel threads are not supported by the operating system, user-level

thread libraries are implemented for programmers to develop multithreaded programs. These user-level threads are not known by the operating system. If kernel threads are not supported by the operating system, or in other words, threads are implemented at the user level only, then all (user-level) threads in a process are mapped to one execution context, namely, that process. On the other hand, if kernel threads are supported by the operating system, there are various ways to map user-level threads to kernel threads, and the threading model can be 1:1 (one-to-one), N:1 (many-to-one), and M:N (many-to-many), depending on how many user-level threads are mapped to how many kernel threads. On Solaris 10, the situation is a bit more complex: multiple user-level threads can be mapped to a so-called light-weight process (LWP), and each LWP is mapped to a kernel thread. User-level threads are scheduled by the scheduler in the thread library.

The POSIX specification [36] standardizes the programming interface of multithreaded programming but it does not take a position on which threading model should be used. The term *POSIX threads* (pthreads) refers to threads programmed with the POSIX interface. C is the only language for which POSIX defines bindings with pthreads. As said by Nakano [48], pthreads programming is mainly used by professional systems programmers to support advanced parallel computing technologies such as OpenMP [5].

Most modern operating systems support multiple CPUs and multithreading. The Linux 2.6 kernel is discussed further below.

### 5.3.2 Support by Linux

The Linux 2.6 kernel has an SMP version and a NUMA version. To support the memory hierarchy in NUMA computers, the kernel internals understand one processor

or one memory pool's relation to I/O devices and each other, and the scheduler attempts to optimize tasks for best use of local resources [57]. Linux 2.6 supports at least 32 processors [70]. The Linux 2.6 kernel is also aware of virtual processors created by SMT, and schedules tasks between virtual processors such that no one underlying single physical processor is overburdened. The pre-2.6 runqueue (a list of all ready-to-run tasks) is global to the entire system, only one CPU can manipulate it at any given time. A lock is maintained to protect the runqueue from simultaneous access from multiple CPUs. Once one CPU obtains this lock, any other CPU that becomes freed up has to wait until the lock is released. Consequently, the scheduler code can be executed only by one CPU at any one time. This lock contention goes up as the number of CPUs increases. The pre-2.6 scheduler selects the best candidate task to run from this global runqueue regardless of which CPU the task was on. This may ping-pong a task between CPUs. The scheduler of the 2.6 kernel has one runqueue for each CPU. This addresses the lock contention problem and the task bouncing problem. An interrupted task will resume on the same CPU it was running before the interruption because a CPU picks the task from its local runqueue. "CPU affinity" is thus achieved, and cache and, therefore, workload performance is improved. On the other hand, in order to balance the load among CPUs, the scheduler, when appropriate, migrates tasks between CPUs. Linux 2.6 includes in-kernel support for the new, POSIX-conforming Native Posix Threading Library (NPTL), which is based on a 1:1 threading model. And there is no limit on the number of threads created.

The Linux 2.6 scheduler prior to version 2.6.23 performs its tasks in complexity  $O(1)$ ; the scheduler can schedule tasks within a constant amount of time regardless of the number of tasks that are running. Motivated by the failure of the pre-2.6.23 scheduler to predict whether a task is interactive (I/O-bound) or CPU-bound, kernel 2.6.23 employs a so-called completely fair scheduler (CFS) [40]. The complexity of the scheduling algorithm, however, is increased to  $O(\log n)$  where  $n$  is the number of

tasks in the runqueue [3]. Cache hierarchy is considered in scheduling with the help of the Linux load-balancer. The concept of scheduling domains was introduced in the 2.6.7 load-balancer and the the scheduling domain for multi-cores was added with version 2.6.17. For example, on a computer with two dual-core processors (packages), the scheduler, with the help of this load-balancer, will schedule two tasks on different packages, if both are idle, so as to maximize the use of the combined larger cache [78].

When asked about the CFS’s impact on NUMA, SMT, and multi-core machines, Ingo Molnar, CFS’s designer, responded: “it will inevitably have some sort of effect — and if it’s negative, I’ll try to fix it.” [3] That seems to suggest that particular processor architectures were not given special consideration when the new scheduling algorithm was designed.

As of this writing, the latest stable Linux kernel version is 2.6.28 released in December 2008 and the CFS and scheduling domains are still used.

A review of the latest Linux scheduler is provided in papers by Zangerl [78] and Siddha [62], for example.

### **5.3.3 Impact of asymmetric multi-cores**

In light of the development of asymmetric multi-cores, as described above, there have been studies on operating-system support, particularly in scheduling, for those architectures. Li et al [42], for example, pointed out that operating-system “schedulers traditionally assume homogeneous hardware and do not directly work well on asymmetric architectures” and therefore implemented another scheduler, in the Linux 2.6.16 kernel, for performance-asymmetric architectures.

### 5.3.4 Impact of NUMA topology

Schüpbach et al [58] observed that “interconnect technologies like HyperTransport and QuickPath mean that the inside of a general-purpose computer will increasingly resemble a network, and cost of messaging between cores may depend significantly on hop length and routing.” Whereas the contemporary operating systems commonly hide the NUMA nature of the underlying hardware, Schüpbach et al noted that “[t]here is, however, a need that more information about the underlying hardware architecture, and more control over the allocation of resources, should be available to applications for better scalability.” The NUMA topology and the core heterogeneity prompted Schüpbach et al to study how the operating system and language runtime system should be architected and interact for the emerging manycore systems.

### 5.3.5 Binding process/thread to CPU

As discussed before, a multiprocessor operating system can be optimized by leveraging CPU affinity, that is, scheduling a process or thread to a CPU where the process or thread was run on. This type of CPU affinity is called soft affinity because it is just an attempt by the operating system to keep processes/threads on the same CPU as long as possible. Recent versions of some operating systems offer the application programmer the ability to bind a process or thread to a particular CPU. It is enforced as a requirement so this type of affinity is called hard affinity. For example, Linux, starting from version 2.6, offers the `sched_setaffinity()` system call that allows a programmer to force the execution of a task to take place only on a set of CPUs. Solaris 9 and 10 provide the `processor_bind()` call that binds a thread to one particular CPU. The desirability of such binding is dependent on the application, and the programmer needs knowledge about the application in order for the binding

to achieve better performance. If one thread is bound to a CPU while other threads to other CPUs, that thread pretty much receives the full attention of the CPU it is bound to.

# Chapter 6

## Multiple-CPU MESSENGERS

### Runtime System

#### 6.1 Inadequacy of single-CPU MESSENGERS runtime

The version of the runtime system discussed in Chapter 3 does not fully utilize all available CPUs on a single host. Even though there are multiple threads within the daemon running on a multiple-CPU host, execution of the Messengers can be performed on only one CPU at any given time because there is only one computation thread. Though each daemon process is multithreaded, if there is no communication going on, then only one CPU can be utilized — by the single computation thread. If there are multiple CPUs on the host, CPUs other than the one running the computation thread are idle unless they have active communication threads to run. Even when there is heavy communication, the number of communication threads, plus the computation thread, may be less than the number of CPUs, thus resulting in

under-utilization of the computer's processing resources.

In addition, only one daemon is allowed by the runtime system to run on each host. It is not possible to execute multiple daemons on one host in order to have multiple computation threads to utilize the multiple CPUs. Under the runtime system implementation that a physical node, as described in Sec. 2.3.2, is represented by a daemon process, the restriction that there can only be one daemon running on each host means that from the perspective of the MESSENGERS programming model, there can only be one physical node, set up on each host. And the MESSENGERS programming model is such that two Messengers are concurrent only if they are on two physical nodes. Therefore, Messengers on a host cannot run concurrently even if there are multiple CPUs.

## 6.2 Flow of control: process vs thread

A flow of control is sometimes called a thread of control; but the term *flows of control* is preferred here because *threads of control* may be confused with threads, which as one will see are a particular way to realize flows of control. (In the field of concurrent computing theory, the term *process* is commonly used to describe flow of control. For a more rigorous definition of a process (not to be confused with a process in an operating system) as used in formalism, see Broy and Streicher's work [14], for example.) There are many mechanisms to implement multiple flows of controls, Zheng et al [79] reviewed some of these, such as processes, threads, event-driven objects, etc.

There are two widely used ways to allow a single program to utilize multiple CPUs on a computer: by creating multiple processes out of the program and creating multiple threads within the program. A process can be created in almost every operating

system, and multithreading is supported by the majority of contemporary operating systems. In particular, there is a pthreads binding with C, the language used to write the single-CPU MESSENGERS runtime, and pthreads is already used to functionally decompose the runtime program, as described in Sec. 3.2.8.

Two variants of the MESSENGERS runtime were implemented that can utilize multiple CPUs on a host, one using multiple processes, and another using multiple threads. The pros and cons, in general, of using these two approaches are summed up below, based on Perez's paper [55].

### 6.2.1 Scaling using processes

The principal advantage of using multiple processes is that it is relatively easy to implement.

The disadvantages are:

1. If a process is preempted, the cost of context-switching a process is higher than the cost of context-switching a thread because the size of a process's context is larger than the size of a thread's context.
2. Inter-process communication is more costly than inter-thread communication.
3. A process needs more space than a thread. Two processes do not share address space.

### 6.2.2 Parallelization using threads

The advantages of using multiple threads include:

1. The cost of context-switching a thread is lower.
2. Inter-thread communication is less costly.
3. A thread needs only a private stack. Threads needs lesser space.

The main disadvantage is that it is error-prone to synchronize threads in a multi-threaded program. Consequently, the development cost is higher.

### **6.3 Implementation of multiple-CPU MESSENGERS runtime using multiple processes**

A key insight from the MESSENGERS programming model is that a physical node is an abstraction of the processing capability of a location. If one wishes to increase the processing capability of a location, one needs to add more physical nodes. Being able to better utilize multiple CPUs on a host can be viewed as increasing the processing capability of a location available to a program. If a physical node continues to be implemented by a daemon (i.e., a process), then one needs to be able to create more daemons on a host in order to have more physical nodes. On the other hand, a physical node can be implemented by a thread instead; Sec. 6.4.2 considers the alternative of adding more physical nodes by creating more threads in that case.

The need for the MESSENGERS programmer/user to add more physical nodes on a multiple-CPU host underscores the philosophy that the underlying hardware configuration should/need not be transparent to the programmer/user. As discussed in Chapter 5, contemporary computers exhibit NUMA characteristics, and some researchers suggested that hiding these characteristics may impede application performance. The design of the multiple-CPU MESSENGERS runtime is based on the idea

that exposing the underlying hardware, to a certain extent, to the programmer/user would allow him/her to use that knowledge to improve application performance.

Under the premise that a physical node continues to be implemented by a daemon, the single-CPU runtime program was modified so that the code setting up the daemon network can accommodate and differentiate multiple daemons on the same host. This is the most straightforward way to allow MESSENGERS to utilize multiple CPUs on a host. Since a daemon represents a physical node as described in Sec. 2.3.2, setting up multiple daemons on a host is equivalent to setting up multiple physical nodes on a host. As mentioned in Sec. 3.2.3, the daemon network is formed by assuming that the daemon on a host uses the host name to set up its sockets. If there are multiple daemons on a host, obviously the host name (alone) cannot serve as a unique identifier for a daemon. The MESSENGERS programmer and user need to give different names to the daemons running on the same host, and the runtime program was modified accordingly. Specifically, two daemons running on the tarheel.ics.uci.edu host would be named tarheel.ics.uci.edu:0 and tarheel.ics.uci.edu:1, for example. These daemon names, or physical node names, replace the host names in the MESSENGERS profile and act as the names of physical nodes in the `create()` statements in a Messenger definition. In addition, if there are multiple daemons on a host, to start running each daemon, the user needs to use an option flag `-d` followed by the name given to the daemon. An example would be `messengers -d tarheel.ics.uci.edu:0`. A daemon can be executed on any of the CPUs on the host it is created; the mapping between daemons and CPUs is done by the operating system. It is probable that a daemon runs entirely on one particular CPU due to the operating system's CPU affinity scheduling and that daemons are assigned to CPUs in such a way that there is load balancing among CPUs.

Just as a runqueue can be associated with either one CPU (in the case of a dedicated

runqueue) or multiple CPUs (in the case of a shared runqueue) in the design of a multiprocessor operating system, it is possible to associate one ready queue with either one physical node or all physical nodes on a host. However, if a physical node is implemented by a daemon, then any data structure shared by multiple daemons has to be implemented by some data structures shared by multiple processes, and accessed via some inter-process communication mechanism. Such implementation is likely to be complicated and therefore a one-to-one mapping between a physical node and a ready queue is chosen. That is, there is one ready queue per daemon.

There is no need to change the code dealing with the injection, execution, migration and scheduling of Messengers in the single-CPU runtime, as described in Sec. 3.2. In particular, like in the single-CPU runtime, when a migrating Messenger arrives at the destination logical node, it is enqueued at the end of the ready queue in the destination daemon. The MESSENGERS programmer/user can still create multiple logical nodes in each daemon (or in other words, on each physical node). However, like in the single-CPU runtime, Messengers on two logical nodes do not run concurrently if they are on the same daemon. Two Messengers are concurrent only if on different physical nodes. If two Messengers are to run concurrently, they need to be placed on two different logical nodes, one on a separate daemon, whether the daemons are on the same host or not.

### **Binding MESSENGERS daemon to CPU**

The MESSENGERS model is silent on how a physical node, embodying the processing capability of a location, is to be associated with an actual processor, allowing flexibility in the actual application of the model in this aspect. One flexibility is in terms of the number of physical nodes created — for example, one can create one or multiple physical nodes for one particular CPU. That issue is beyond the scope of this

dissertation. Another flexibility is on whether a physical node is always associated with one particular CPU during the execution of the entire program. It is possible that a physical node is mapped to one CPU during part of the execution, and to another CPU during the rest of the execution, for instance. As a physical node is implemented by a daemon, that means that the possibilities to bind or not bind a physical node to a CPU would be realized by either binding or not binding a daemon to a particular CPU.

When multiple daemons are run using the `-d` flag as described above, they are not bound to CPUs. In fact, under that scheme, the number of daemons on a host can be greater than the number of CPUs on that host.

The multiple-CPU MESSENGERS runtime system gives the user and programmer the option to bind a daemon, which is a process, to a CPU on the host it runs on. To use this option, the user uses a `-n` flag followed by an extended host name when invoking the runtime program on a host. The extended host name is formed by appending a CPU ID to the host name. For example, `tarheel.ics.uci.edu:0` would represent CPU 0 of `tarheel.ics.uci.edu`. The daemon thus created is bound to a particular CPU on the host. These extended host names replace the host names in the MESSENGERS profile and each acts as the physical node name in the `create()` statement. Internally, the runtime program uses the operating system calls described in Sec. 5.3.5 to bind a daemon to a CPU. Note that this CPU ID must be one recognized by the operating system, whereas the daemon ID used in the daemon name in the `-d` option is simply an arbitrary non-negative integer.

From the view of the operating system, more than one process can be bound to a particular CPU. However, in the above scheme each daemon is in effect named by the extended host name, this limits the number of daemons bound to a particular CPU to 1. That is, if a daemon is bound to `tarheel.ics.uci.edu:0`, effectively the daemon is

named tarheel.ics.uci.edu:0, and there cannot be another daemon that is also bound to tarheel.ics.uci.edu:0 and thus named tarheel.ics.uci.edu:0 too.

It should also be noted that when a process is bound to a particular CPU, all the threads within that process are bound to that CPU. That is, if a daemon is bound to a particular CPU, the computation thread and the communication threads are all bound to that CPU.

Figure 6.1 illustrates a situation where two daemons are executed on a dual-CPU host, and each daemon has a single computation thread and one logical node set up. It is assumed that the operating system employs one runqueue for each CPU.

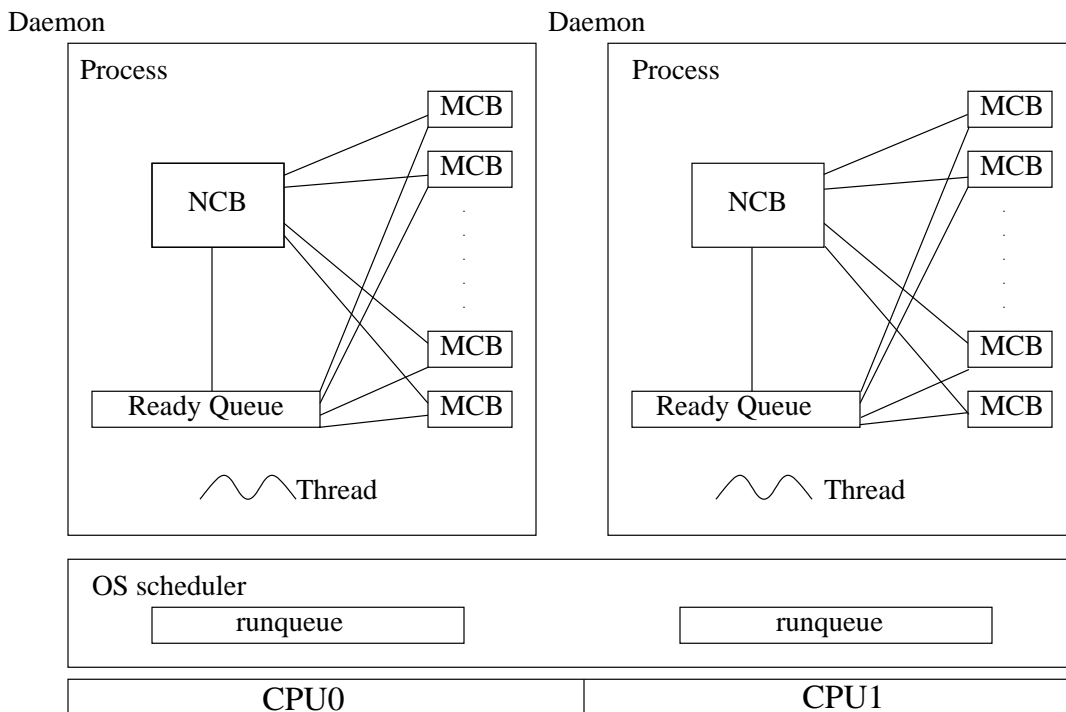


Figure 6.1: Address spaces of the processes running the daemons on a dual-CPU computer with the multiple-CPU MESSENGERS runtime. (Only the computation threads are shown.)

Figure 6.2 shows the three levels of networks in the MESSENGERS execution environment with a physical network of dual-CPU hosts, a daemon network of daemons,

two on each computer, and a logical network of nodes, two on each computer. The daemon network is still fully connected but how logical nodes are connected is entirely up to the MESSENGERS programmer.

Figures 6.1 and 6.2 do not indicate whether the daemons are bound to CPUs.

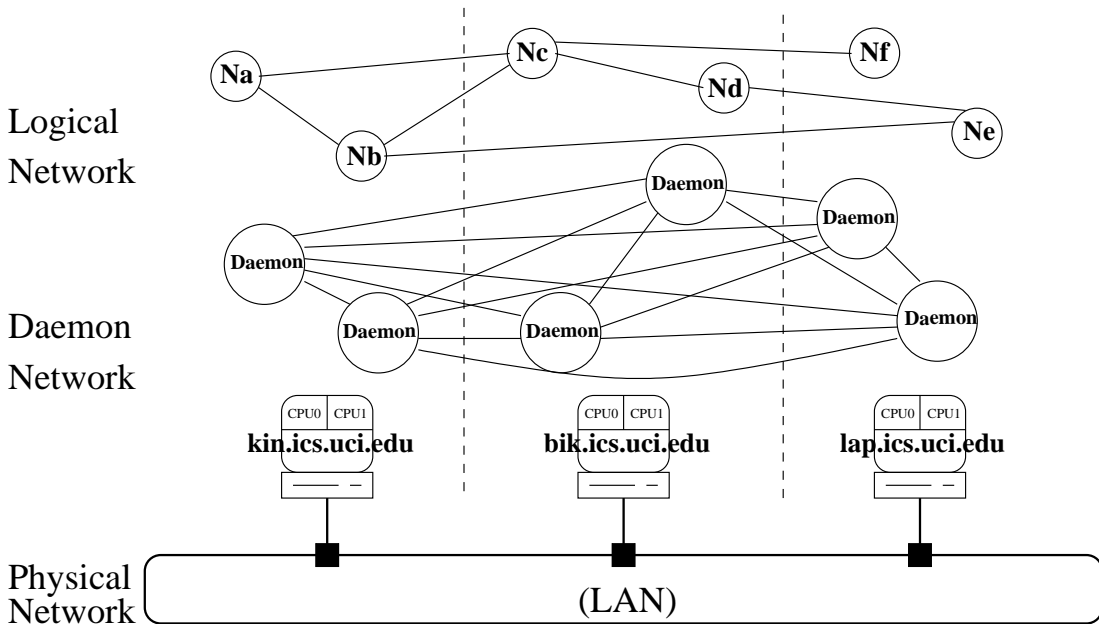


Figure 6.2: Three levels of networks in the MESSENGERS execution environment when two daemons, each embeds a logical node, are created on a dual-CPU computer.

## 6.4 Implementation of multiple-CPU MESSENGERS runtime using multiple threads

### 6.4.1 Domain decomposition of MESSENGERS runtime program

There are two main approaches to parallelizing a program: domain decomposition and functional decomposition. Such approaches can be used to parallelize the MESSEN-

GERS runtime program. In fact, the single-CPU MESSENGERS runtime program is already parallelized using functional decomposition: the part of the program dealing with the execution of Messengers is performed by the computation thread, the part receiving Messengers from other daemons by the receiving thread, and the part sending Messengers to other daemons by the sending threads.

With domain decomposition, data and work is distributed into a number of domains and the work in the domains is performed concurrently. Each domain is mapped to a flow of control. The part of the runtime program that executes Messengers can be parallelized by domain decomposition, with the computation in each domain performed by a distinct flow of control, e.g., a thread.

### **6.4.2 Possible choices for domain**

There are three abstractions in the MESSENGERS programming model that can be choices for a domain:

1. Messenger
2. Physical node
3. Logical node

#### **Messenger**

As discussed in Sec. 3.2.9, the MESSENGERS runtime is designed to support the running of Messengers as fine-grained tasks, and a Messenger is a unit of logical concurrency for program development purposes. Multiple Messengers (on a host)

are mapped to one thread so that the execution of these fine-grained tasks is more efficient. Consequently, a Messenger should not be considered as a domain.

### **Physical node**

As a reminder, a physical node as used here is not necessarily a physical CPU; it is an abstraction of the processing resources of a location. The single-CPU MESSENGERS runtime couples a daemon with a physical node. Therefore, there is no separate data structure implementing a physical node. If a physical node is to be selected as a domain the computation therein is carried out by a thread, then there needs to be a data structure, to be instantiated multiple times, in the MESSENGERS runtime program to represent physical node. Significant modification to the MESSENGERS runtime program is required. Therefore, this approach is not considered further.

### **Logical node**

The only choice left is logical node. However, choosing a logical node as a domain is not consistent with the interpretation of a logical node as an abstraction of storage capability in the MESSENGERS model, because in effect a logical node is then represented by a thread, which is an abstraction of a flow of execution. Nevertheless, such a configuration was implemented in order to examine its implication and performance.

### **6.4.3 Implementation**

A MESSENGERS runtime program with multiple computation threads was developed so that a user-level thread is created whenever a logical node is created. The single

computation thread in the original runtime is now mainly responsible for initialization and housekeeping. The execution of Messengers is now performed by the computation threads associated with the logical nodes the Messengers are on.

Similar to the choices between a dedicated runqueue and a shared runqueue in a multiprocessor operating system, there can be one ready queue either for one logical node or for all logical nodes in a physical node (i.e., daemon). A shared ready queue would be a multiple-producer-multiple-consumer queue in the sense that each thread may enqueue Messengers to the queue when a thread migrates a Messenger from its associated logical node to another logical node, and each thread may dequeue Messengers from the queue in order to execute them. A dedicated ready queue would be a multiple-producer-single-consumer queue because each thread may enqueue Messengers to the queue when a thread migrates a Messenger from its associated logical node to another logical node but there is only one thread that may dequeue Messengers from the queue in order to execute them.

The implementation of a multiple-producer-single-consumer queue is simpler and therefore such was implemented, where the queue is protected by mutual exclusion locks from concurrent accesses by multiple threads. Each enqueueing and dequeueing operation involves locking and unlocking.

A new member, `ready_q`, is added to `struct NCB`, as shown in Fig. 6.3, so that there is a ready queue associated with each logical node; and all the Messengers ready to run on that logical node are queued in that ready queue. A thread for a logical node loops through and executes all the MCBs in the ready queue for that node and calls `wait_for_migrating_msgrs()`, which calls `sigwait()` when the queue is empty, awaiting for a SIGUSR2 signal (see Fig. 6.4). When a Messenger hops from one logical node to another in the same daemon, the thread for the source node removes the corresponding MCB from the ready queue for that node, enqueues it to

the ready queue for the destination node and sends a SIGUSR2 signal to the thread for the destination node. Figure 6.5 shows that the thread for the source node sends SIGUSR2 while executing `exec_msgrs()` and the thread for the destination node is executing `run_node()` while the signal is received. The `exec_msgrs()` function, a simplified version of whose definition is shown in Fig. 6.6, was modified from the one in the single-CPU runtime by adding code to test whether there is a change in the NCB associated with the MCB being executed. (The change would have occurred during the execution of `exec_mcb()`.) If there is, that means the Messenger has hopped to another logical node, and `exec_msgrs()` makes the ready queue switch and sends the signal.

When a migrating Messenger arrives at the destination logical node, it is enqueued at the end of the ready queue associated with that logical node.

<pre> struct NCB {     char                name[64];     uint32_t            address[2];     struct splay_tree   *nva_list;     struct doubly_linked_list *link_list;     ... //other members     struct node_ready_queue *ready_q }; </pre>	<pre> struct node_ready_queue {     struct shared_mcb_list *mcb_list;     pthread_t              *tid;     ... //other members }; </pre>
(a)	(b)

Figure 6.3: `struct NCB` and `struct node_ready_queue`. (a) `NCB`. (b) `node_ready_queue`.

It should be noted that there is nothing to prevent the MESSENGERS programmer/user from creating more logical nodes on each host than there are CPUs on the host. The programmer/user needs to exercise caution in creating logical nodes. If there are too many logical nodes, thus too many threads, compared to the number of CPUs, there will be unnecessary overhead in thread management.

```

void run_node(struct node_ready_queue *ready_queue)
{
    for (;;) {
        exec_msgrs(ready_queue->mcb_list);
        if (is_shared_mcb_list_empty(ready_queue->mcb_list))
            wait_for_migrating_msgr();
    }
}

```

Figure 6.4: Definition of `run_node()` in the multiple-CPU MESSENGERS runtime program with multiple computation threads.

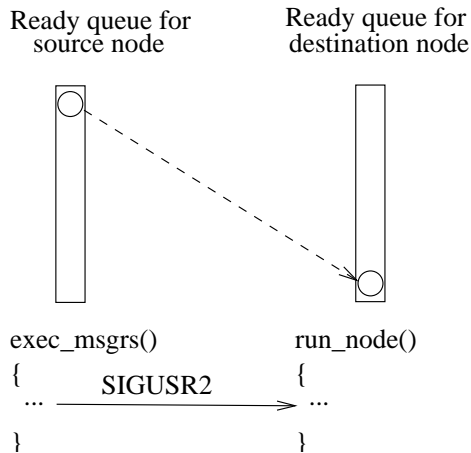


Figure 6.5: Migration of an MCB from (the ready queue of) one logical node to (the ready queue of) another.

Figure 6.7 illustrates a situation where one daemon is executed on a dual-CPU host, the daemon has two computation threads, and each thread runs one logical node. It is assumed that the operating system employs one runqueue for each CPU.

Figure 6.8 shows the the three levels of networks in the MESSENGERS execution environment with a physical network of dual-CPU computer nodes, a daemon network of daemons, one on each computer, and a logical network of nodes, two on each computer. The daemon network is still fully connected but how logical nodes are connected is entirely up to the MESSENGERS programmer.

```

void exec_msgrs(struct shared_mcb_list *q)
{
    struct MCB *current_mcb, *next_mcb, *prev_mcb;
    int next_stmt_kind;

    prev_mcb = NULL;
    current_mcb = get_head_of_shared_mcb_list(q);
    while (!is_shared_mcb_list_empty(q)) {
        next_mcb = get_next_of_shared_mcb_list(q, current_mcb);
        next_stmt_kind = exec_mcb(current_mcb);
        if (prev_mcb == NULL)
            current_mcb = get_head_of_shared_mcb_list(q);
        else
            current_mcb = get_next_of_shared_mcb_list(q, prev_mcb);

        if (is_valid_in_shared_mcb_list(q, current_mcb)) {
            ncb = get_ncb_located(current_mcb);
            if (ncb != get_associated_ncb(q)) {
                remove_from_shared_mcb_list(q, mcb);
                add_to_shared_mcb_list(ncb->ready_q->mcb_list, mcb);
                ret = notify_of_msgr_migration(ncb->ready_q->tid);
                if (prev_mcb == NULL)
                    current_mcb = get_head_of_shared_mcb_list(q);
                else
                    current_mcb = get_next_of_shared_mcb_list(q, prev_mcb);
            }
        }

        if (next_stmt_kind == RELEASE_CPU) {
            prev_mcb = current_mcb;
            current_mcb = next_mcb;
        }
    }
}

```

Figure 6.6: Definition of `exec_msgrs()` in the new runtime program.

## 6.5 Performance evaluation

The two approaches to utilizing the multiple CPUs on a host were implemented. One, referred to here as the multi-process version, involves executing multiple daemons (alternatively speaking, physical nodes) on a host and has two options — either binding or not binding a daemon to a particular CPU on the host. It is possible to execute more daemons than there are CPUs on the host, but its implication is outside

Daemon

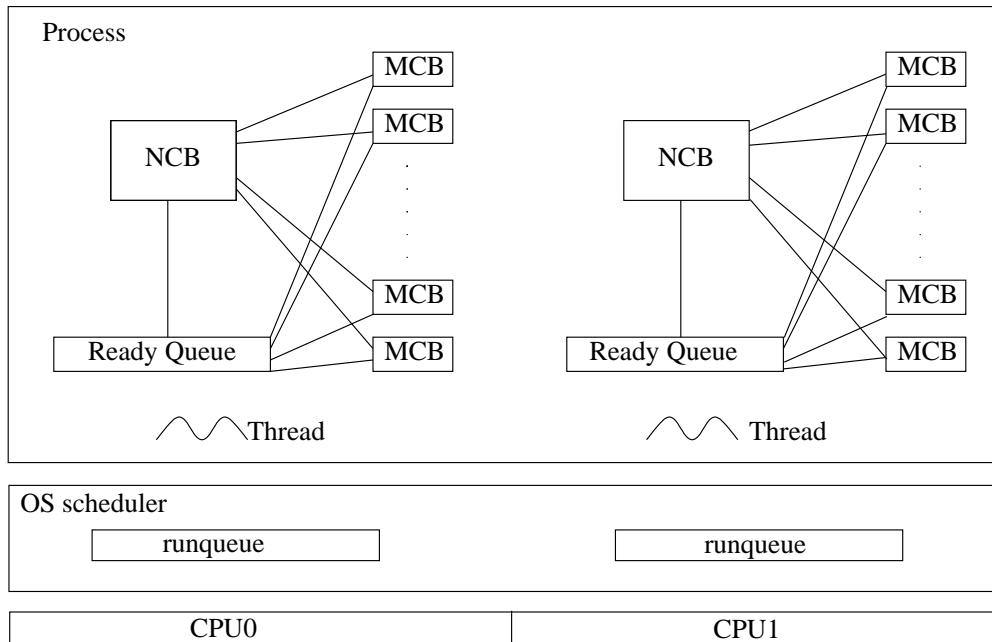


Figure 6.7: Address space of a multithreaded process running the daemon on a dual-CPU computer with the multiple-CPU MESSENGERS runtime. (Only the computation threads are shown.)

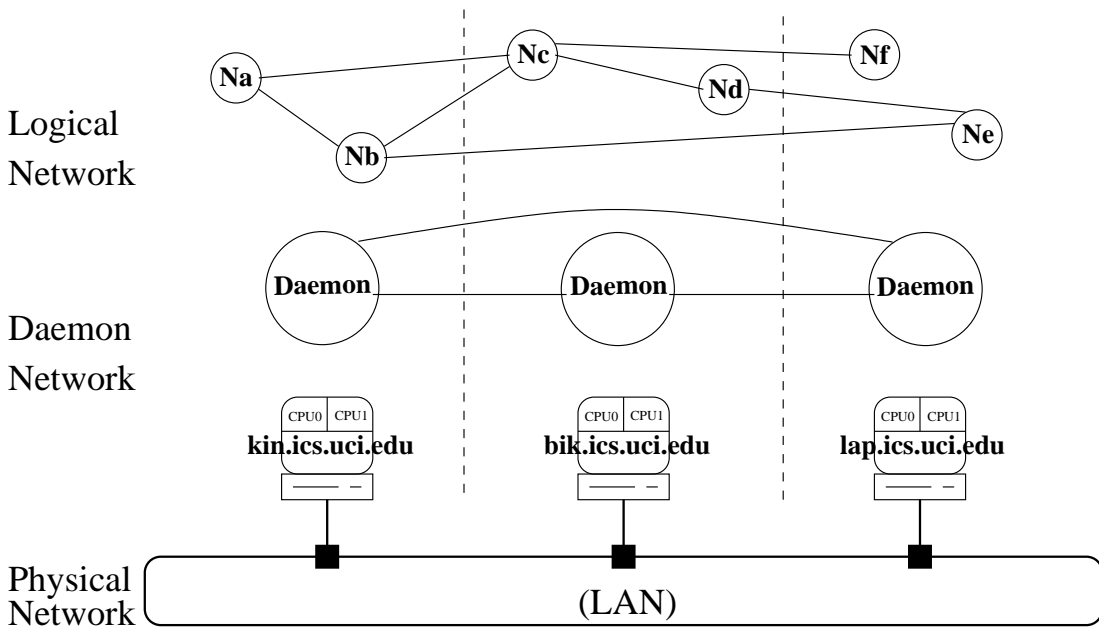


Figure 6.8: Three levels of networks in the MESSENGERS execution environment when one daemon, consisting of two computation threads each embedding a logical node, is created on a dual-CPU computer. (The threads are not shown.)

the scope of the discussion here. It is assumed that there is one daemon for each CPU. Another approach, referred to here as the multi-computation-thread version, creates a thread for each logical node and there is only one daemon on a host. It is assumed that the MESSENGERS programmer creates one logical node for one CPU on the host.

The MESSENGERS program for Crout factorization, as described in Chapter 4, was used to evaluate the performance of these approaches. A number of experiments, with different matrix sizes and a block-cyclic distribution scheme with a block size of 100 columns, were run on the following three platforms, which employ the SMP and CMP technologies discussed in Chapter 5:

1. A Sun Fire-V210 workstation having two 1 GHz UltraSPARC-IIIi processors, and 4 GB memory, running Solaris 10 — the MESSENGERS compiler called GCC 3.4.4 and the MESSENGERS runtime was compiled using GCC 3.4.4
2. A HP ProLiant BL35p G1 computer with two dual-core 2 GHz AMD Opteron 270 processors and 2 GB memory, running Solaris 10 — the MESSENGERS compiler called GCC 4.2.1 and the MESSENGERS runtime was compiled using GCC 4.2.1
3. A TYAN Thunder K8S Pro computer with two 2.2 GHz AMD Opteron 248 processors and 4 GB memory, running CentOS 5 (GNU/Linux 2.6.18) — the MESSENGERS compiler called GCC 4.1.2 and the MESSENGERS runtime was compiled using GCC 4.1.2

The multi-process version of the multiple-CPU MESSENGERS runtime, with the `-d` and the `-n` options, was evaluated against the single-CPU MESSENGERS runtime on all three platforms. As shown in Tables 6.1 through 6.3, the overall speedup was

Table 6.1: Performance of running the MESSENGERS program for Crout factorization on a Sun Fire V210 workstation with two 1.34 GHz UltraSPARC III-i processors and 4 GB memory. The data under the Single-CPU, Multiproc -nonbinding, and Multiproc -binding columns were collected using the Single-CPU MESSENGERS runtime, the multi-process version with -d option, and the multi-process version with -n option of the multiple-CPU MESSENGERS runtime, respectively.

Using 2 physical nodes		Single-CPU		Multiproc -nonbinding		Multiproc -binding	
Order of matrix	Block size	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
3000	100	127	1.00	55	2.31	55	2.31
4000	100	297	1.00	136	2.18	136	2.18
5000	100	597	1.00	279	2.14	278	2.14
6000	100	1032	1.00	498	2.07	499	2.07
7000	100	1652	1.00	824	2.00	818	2.02
8000	100	2421	1.00	1248	1.94	1249	1.94

Table 6.2: Performance of running the MESSENGERS program for Crout factorization on a TYAN Thunder K8S Pro computer with two 2.2 GHz AMD Opteron 248 processors and 4 GB memory. The data under the Single-CPU, Multiproc -nonbinding, and Multiproc -binding columns were collected using the Single-CPU MESSENGERS runtime, the multi-process version with -d option, and the multi-process version with -n option of the multiple-CPU MESSENGERS runtime, respectively.

Using 2 physical nodes		Single-CPU		Multiproc -nonbinding		Multiproc -binding	
Order of matrix	Block size	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
4000	100	64	1.00	22	2.91	24	2.67
5000	100	124	1.00	49	2.53	49	2.53
6000	100	214	1.00	88	2.43	88	2.43

Table 6.3: Performance of running the MESSENGERS program for Crout factorization on a HP ProLiant BL35p G1 computer with two dual-core 2 GHz AMD Opteron 270 processors and 2 GB memory. The data in the single-CPU row were collected using the single-CPU MESSENGERS runtime. The data in the Multiproc -nonbinding, and the Multiproc -binding rows were collected using the multi-process version with -d option and the multi-process version with -n option, respectively, of the multiple-CPU MESSENGERS runtime.

Order of matrix/Block size		4000/100		5000/100		6000/100	
	# physical nodes	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
Single-CPU	1	67	1.00	133	1.00	227	1.00
Multiproc -nonbinding	2	34	1.97	65	2.05	114	1.99
Multiproc -binding	2	34	1.97	66	2.02	114	1.99
Multiproc -nonbinding	4	21	3.19	38	3.50	67	3.39
Multiproc -binding	4	19	3.52	38	3.50	67	3.39

satisfactory, and there was insignificant performance difference between the `-d` and the `-n` options.

Table 6.4: Performance of running the MESSENGERS program for Crout factorization on a Sun Fire V210 workstation with two 1.34 GHz UltraSPARC III-i processors and 4 GB memory. The data in the single-CPU row were collected using the single-CPU MESSENGERS runtime. The data in the Multiproc `-nonbinding`, the Multiproc `-binding`, and the Multithreaded rows were collected using the multi-process version with `-d` option, the multi-process version with `-n` option, and the multi-computation-thread version, respectively, of the multiple-CPU MESSENGERS runtime. LNs stand for logical nodes.

Order of matrix/Block size	3000/100		4000/100		5000/100		6000/100		7000/100		8000/100			
	# daemons	# LNs per daemon	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
Single-CPU	1	1	127	1.00	297	1.00	597	1.00	1032	1.00	1652	1.00	2421	1.00
Multiproc <code>-nonbinding</code>	2	1	55	2.31	136	2.18	279	2.14	498	2.07	824	2.00	1248	1.94
Multiproc <code>-binding</code>	2	1	55	2.31	136	2.18	278	2.14	499	2.07	818	2.02	1249	1.94
Multiproc <code>-nonbinding</code>	2	2	54	2.35	130	2.28	265	2.25	478	2.16	782	2.11	1200	2.02
Multiproc <code>-binding</code>	2	2	52	2.44	129	2.30	265	2.25	480	2.15	782	2.11	1205	2.01
Multi-threaded	1	2	53	2.40	131	2.26	277	2.16	525	1.97	833	1.98	1231	1.97

Table 6.5: Performance improvement in running the MESSENGERS program for Crout factorization by using two logical nodes over one logical node per physical node with the multi-process version of the multiple-CPU MESSENGERS runtime on a dual-processor Sun Fire V210 workstation. Non-binding and binding refer to the `-d` and `-n` options respectively of the multi-process version. Block-cyclic data distribution was used with a block size of 100 columns.

Order of matrix	3000	4000	5000	6000	7000	8000
Non-binding	1.5%	4.4%	5.0%	3.8%	5.1%	3.8%
Binding	5.5%	5.2%	4.7%	3.8%	4.4%	3.5%

The multi-process version was further evaluated against the multi-computation-thread version of the multiple-CPU MESSENGERS runtime on the dual-processor Sun Fire-V210 workstation, which has a 1 MB on-chip 4-way, set associative level-2 cache on each processor. The results are shown in Tables 6.4 through 6.6. When only one logical node was set up in one daemon in the multi-process version, as shown in Fig. 6.9(a), in general there was insignificant performance difference between the multi-process version and the multi-computation-thread version. However, when two

Table 6.6: Performance improvement in running the MESSENGERS program for Crout factorization by using two logical nodes per physical node with the multi-process version over (using two logical nodes with) the multi-computation-thread version of the multiple-CPU MESSENGERS runtime on a dual-processor Sun Fire V210 workstation. Non-binding and binding refer to the `-d` and `-n` options respectively of the multi-process version. Block-cyclic data distribution was used with a block size of 100 columns.

Order of matrix	3000	4000	5000	6000	7000	8000
Non-binding	-1.9%	0.8%	4.3%	9.0%	6.1%	2.5%
Binding	1.9%	1.5%	4.3%	8.6%	6.1%	2.1%

logical nodes were set up in each daemon in the multi-process version, as shown in Fig. 6.9(b), the cache reuse improvement as described in Sec. 4.2 allows the multi-process version to have better performance. It is not possible to create multiple logical nodes in the multi-computation-thread version to benefit from the cache reuse improvement because the creation of a logical node means the creation of a separate thread and a separate ready queue in that version, where a Messenger would be on a separate queue to be executed by a separate thread when it migrates to a new logical node.

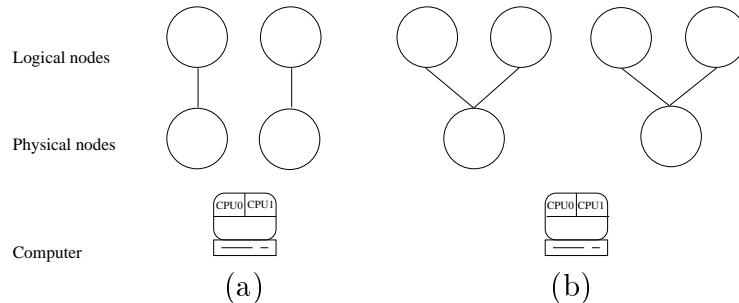


Figure 6.9: Mapping of physical and logical nodes on a multiprocessor computer. (a) One logical node for a physical node. (b) Two logical nodes for a physical node.

A blocked algorithm partitions a matrix into blocks to reduce data movement in the algorithm. It is conceivable that there may exist some concurrent blocked algorithm that on one hand parallelize the factorization and on the other hand exploit cache locality. However, the MESSENGERS programming model with the MESSENGERS

runtime offers a uniform programming approach for both shared-memory computers (and exploit both cache locality and parallelism) and distributed-memory computers (and exploit parallelism). The MESSENGERS programmer does not need to write a separate blocked algorithm.

## 6.6 Discussion

There was no significant difference in the performance between using the `-n` (binding) and the `-d` (non-binding) options of the multi-process version of the multiple-CPU runtime. In other words, whether binding the daemon to a CPU did not have material effect on performance. That means either the operating systems are doing a good job in placing the processes/threads on CPUs or the number of CPUs on a computer in the experiments were too small to cause a performance difference between the two options. If there are a large number of CPUs on a computer and as a result, the design of the architecture has a NUMA flavor, as described in Chapter 5, then the process/thread placement by the operating system is more important than for a SMP/UMA computer. If a process/thread is placed on the “wrong” CPU, or if there is excessive bouncing of processes/threads among CPUs, then performance would be adversely impacted. (Recall that currently Linux does not take the NUMA, SMT and multi-core characteristics into consideration when its scheduler was designed — see Sec. 5.3.2.) In that case, allowing the programmer/user the flexibility to take control of process/thread placement, by using the binding option, may have a positive impact on performance. Another possible use of the binding option would be on a computer with asymmetric CPUs. A daemon process can be bound to a particular CPU which is most suitable to perform the computation assigned to that process.

# Chapter 7

## Related Work

### 7.1 Navigational programming

At different times, Bic, Dillencourt, Fukuda, Wicke, Pan, and etc. used the term *navigational programming* to describe an idea similar to the MESSENGERS programming model discussed in Chapter 2. However, that term is used in a rather confusing way. The MESSENGERS-I system used by Fukuda et al [21] employs code migration techniques and they defined navigational programming as a programming style “based on the concept of autonomous ‘self-migrating’ messages,” namely, “messages that have their own identity and behavior, which permit them to actively navigate through the underlying computational network, perform various tasks at the nodes they visit, and coordinate their activities with other such messages constituting the distributed computation.” The autonomous nature of the messages, namely, Messengers, stresses the code mobility aspect of the programming style (“[a]utonomous messages, also referred to as mobile agents or self-migrating threads”); in contrasting the navigational style to message passing, Fukuda et al pointed out that “[a]ll functionality of the application

[written in the navigational style] is embedded in the individual Messengers, i.e., the programs carried by Messengers as they navigate through space” whereas messages in message passing “are only passive carriers of data.”

On the other hand, Wicke [73]’s MESSENGERS system does not involve code migration but Wicke et al [76] claimed that it facilitates self-migrating computations, which, “commonly referred to as mobile agents, are self-contained entities that can navigate autonomously through the network ...” Pan [51]’s definition of navigational programming is very concise: “Navigational programming is defined as the programming of self-migrating threads.” Pan did not explain what he meant by the term *self-migrating* and his work was based on Wicke’s system. He claimed that MESSENGERS is a thread migration system and did not distinguish between MESSENGERS and Ariadne (see Sec. 7.2), for example, the former maps multiple tasks to a thread and it is the task, not the thread, that migrates (on the language level), and the latter migrates threads on the operating system level.

Whereas the state of a program is defined in Chapter 2, Pan defined the state of a Messenger as “program counter and Messenger variables,” not including node variables as part of the state of a Messenger. Consequently the concept of dynamically binding the node variables was not explored. Furthermore, no significance was assigned to a physical node, and no concurrency model for Messengers running on a single computer was presented.

Therefore, to avoid confusion, the term *MESSENGERS programming model* is used in this dissertation to describe the idea of combining state-migration location-scoped-variable programming and shared-variable programming, and the term *MESSENGERS program* to describe a program written in the MESSENGERS programming language.

## 7.2 Thread migration systems

The idea of using migration to exploit resource (data) locality is not new — Ariadne [43, 45], for example, is one of the so-called thread migration systems that exploit data locality by migrating threads.

The Ariadne programming language is based on C more closely than MESSENGERS is. In particular, it keeps the `main()` function and global variables. The object (target) for migration is a process on a remote computer and the migrating entity is a thread, out of possibly many threads, of a process on the local computer. In Mascarenhas and Rego [44]’s words, “[t]he state of the thread is preserved during migration (i.e., all local variables are left intact and available when the thread resumes on the destination machine). All references to global variables after the migration will involve instances of the global variables on the destination machine.” Though the programming models used by Ariadne and MESSENGERS are very similar, Ariadne’s implementation is different from MESSENGERS’s; it migrates user-level threads (and aims at medium- to coarse-grained tasks). As such, it does not need a compiler that decomposes functions as MESSENGERS does. It merely needs to implement some library calls, e.g., the `a_migrate()` function invoked when a migration is called for, to save and restore the state of the migrating user-level thread. And calls to the Standard C Library functions `setjmp()` and `longjmp()` are embedded in `a_migrate()`. On the other hand, MESSENGERS maps multiple Messengers, the migrating entities, to a user-level thread. Therefore, the MESSENGERS approach requires much less user-level threads than the Ariadne approach does. A high number of threads involves overhead in thread creation and synchronization. Accordingly, MESSENGERS is potentially more efficient than Ariadne on a single computer because in most applications there would be considerably less threads created by the MESSENGERS runtime than those created by the Ariadne programmer.

Whereas Ariadne used `setjmp()` and `longjmp()` to capture and restore machine-dependent (i.e., physical) state of a thread and thus can support migration on only homogeneous computers, Arachne [16, 17]’s approach to capture and restore the logical state of a user-level thread allows it to support migration on heterogeneous computers. Like MESSENGERS, Arachne used a preprocessor to insert code into a (C++) program that contained the migration function, etc. and output a C++ source program file suitable for compilation with a conventional compiler. The way Arachne captured the state of a migrating thread is also very similar to that used by MESSENGERS. For each migrating thread, Arachne created a so-called simulated activation record which was a structure and each member thereof was a local variable of the thread. When the thread was migrated, the values of these members were marshalled, the structure was sent to the destination, and the values were then demarshalled. The destination of the migration in the migration function was expressed as the ID of the destination process. The threads were preemptively scheduled based on priority. If there were multiple processors on a computer, Arachne was not able to execute more than one thread — only one active thread was allowed within each process. (However, two or more Arachne processes could be run on a computer.) Presumably this limitation suggests that a single program was not able to be parallelized, with the conventional shared-variable model, using multiple threads on a multiprocessor.

Whereas an Ariadne thread, like a Messenger, no longer can access data in the original location before its migration, threads in a number of packages later developed focused on allowing these accesses usually using software distributed shared memory (DSM). Itzkovitz et al [37] developed a kernel-thread migration system called MILLIPEDE on top of a DSM system. The migration was on homogeneous computers running Windows NT and aimed for coarse-grained computation. The global data is stored in the DSM so that a thread can still access them after migrating to another computer. They claimed that Ariadne’s approach did not work if pointers were involved and

that in general compiler support was needed. They did not discuss Arachne. Their approach was based on making sure the stacks allocated by the operating system occupy the same addresses on all computers. There was not much detail on how the capture and migration of the state of a thread was implemented except the remark that standard Win32 API was used. The migration was to achieve load balancing and it was the runtime system (based on system workload), not the programmer, which decided when to migrate a thread. Therefore, there was no API for the programmer to direct migration.

Active Threads [71, 72] is a user-level thread package that used some implementation of a shared address space to allow for access to global data and the “iso-address” approach for keeping pointers valid after a thread migration. The package provided API for programmer-directed migration. There were no details provided on how to capture the state of a thread, however.

Zheng et al [79] reviewed different issues associated with migrating flows of control such as threads. Holtkamp [33] discussed the problems involving pointers specifically.

A number of studies in the thread migration research investigated how to capture and thus migrate the state of a thread from one computer to another. In MESSENGERS the migrating entity is a task, as defined in Sec. 2.2.1, not a thread. The state of a task is defined by the programming language, whereas the state of a thread is defined by the operating system. There is greater flexibility in migrating a task in the sense that the state, in particular the migratable state, namely, the part of the state at the source location that is still accessible at the destination location after migration, of a task depends on how the programming language defines it. MESSENGERS defines the migratable state in such a way that implementation is easier. Some thread migration systems tackled the problem of maintaining the link to the global variables defined in the process where a thread therein migrates. MESSENGERS defines away

the problem by limiting the variables a Messenger can access to its private Messenger variables and node variables associated with the logical node it is currently located on. On the other hand, using a thread as the migrating entity offers greater portability. Pointers are not supported in MESSENGERS; therefore, no consideration is given to keep them valid after a Messenger migrates.

### 7.3 Forms of mobility

Cardelli [15] defined control mobility<sup>1</sup> and data mobility, among other forms of mobility, using remote procedure call (RPC) and remote method invocation (RMI) as examples: control mobility takes place with a RPC or RMI call, where a flow of control is considered moving from one computer to another and back; data mobility takes place when data is marshalled, communicated, and demarshalled. In this sense, control mobility is similar to the idea of program migration discussed in Chapter 2. Data mobility, along with code mobility, are issues arisen when control mobility is implemented. That is, control mobility can be realized by data mobility, and possibly code mobility. When code mobility is involved, the migrating entity is often called mobile object or mobile agent.

State mobility can be considered as a form of data mobility, as state is essentially data about the mobile entity. In Sec. 2.3.5, *state* is some abstract mathematical construct — a set of ordered pairs; and migration is simply the change of the values in the location attributes for program statements. Here, the term *state* refers to some concrete data; and state mobility is about physically communicating these data from one computer to another. Migration, or mobility, of this concrete state involves physical

---

<sup>1</sup> In literature on the migration of process and threads, the word *migration* is usually used. In the mobile code and mobile agent field, the word *mobility* is used instead. Here, they are used interchangeably.

communication. The migratable state of a Messenger in the case of MESSENGERS and a thread in the case of Ariadne and Arachne is the set of local variables declared in the Messenger/thread. It is captured either logically, i.e., independent of machine representation, as in the case of MESSENGERS and Arachne, or physically, i.e., dependent of machine representation, as in the case of Ariadne. The migratable states of a Messenger and an Ariadne thread are partial states because they do not include the node variables and global variables, respectively.

## 7.4 Mobile agents and strong/weak mobility

There is no consensus as to what a mobile agent is. Braun and Rossak [12] argued that mobile agents can be defined from the artificial intelligence viewpoint and the distributed systems viewpoint. In the latter, which is closer to the focus of this dissertation, mobile agents are “self-contained and identifiable computer programs, bundled with their code, data, and execution state, that can move within a heterogeneous network of computer systems. They can suspend their execution on an arbitrary point and transport themselves to another computer system. During this migration the agent is transmitted completely, that is, as a set of code, data, and execution state. At the destination computer system, an agent’s execution is resumed at exactly the point where it was suspended before.”

The MESSENGERS runtime system does not support mobile agents in the sense that it does not support code mobility. In other words, Messengers running on the MESSENGERS runtime are not mobile agents.

Many mobile agent systems are based on Java. Truyen et al [67] defined three types of state of an object in Java:

1. program state: bytecode of the object's class
2. data state: contents of the object's instance variables
3. execution state: the program counter register, and the private Java stack of each Java Virtual Machine (JVM) thread that is involved in executing an object (each Java object executes in one or more JVM threads)

For the purposes of the following discussion, the mobility involving the program state is considered code mobility.

Braun and Rossak [13] offered a similar distinction with respect to the components of a mobile agent, not necessarily Java-based:

1. code: some kind of executable representation of computer programs, e.g., source code, bytecode or executable machine language
2. data (also called object state): data that are owned and directly controlled by an agent and are movable; an agent's instance variables if an agent is an instance of a class in an object-oriented language
3. execution state: information usually controlled by the processor and the operating system; might be quite complete information from within the underlying (virtual) machine about call stack, register values, and instruction pointers

Java supports code mobility with its use of machine-independent bytecode and class loading mechanism. It supports data state (or called object state) mobility with its serialization mechanism. However, JVM threads are not implemented as serializable, and the Java language does not define any abstractions for capturing and restoring the execution state inside the JVM.

Strong and weak mobility is often mentioned in the mobile agents literature (for instance, see works by Bettini and De Nicola [9], Walsh et al [69], Ghezzi and Vigna [25]); at issue is code and execution state mobility. Common definitions of strong and weak mobility assume code mobility. According to Fünfroeken [23], in non-transparent or weak mobility, the programmer has to provide explicit code to read and reestablish agents' state, and manually encode agents' "logical" execution state into data state, whereas in transparent or strong mobility, the programmer does not. There are a large volume of research into how to capture the execution state of an object (for example, see Hohlfeld and Yee's paper [32]).

With respect to a thread, at least a pthread, the distinction between data state and execution state is not that clear. Many researchers, such as Jiang and Chaudhary [38], define the state of a thread as that "consists of a program counter, a set of registers, and a stack of procedure records containing variables local to each procedure." If the execution state of a thread, at least a pthread, refers to the local variables of functions called by the thread, then the data state of a thread possibly refers to the thread-specific data (or alternatively called thread-local storage). Whereas there are Standard C Library functions such as `setjmp()` and `longjmp()` that can be used to capture the state (presumably at least the execution state) of a thread, and thus facilitate thread migration on homogeneous computers, there has been little research specifically addressing the capture and migration of thread-specific data.

In the case of MESSENGERS, a Messenger is not a thread, at least not a pthread. A Messenger can call C functions but the migration cannot take place when a C function is being executed. If the local variables in the C functions are deemed execution state, and the Messenger variables data state, then there is never need to save the execution state into the data state when a Messenger migrates. If the Messenger variables are deemed execution state, then it is not clear what data state is and into what the

programmer would have needed to save the execution state, if strong mobility is not supported, assumed the same concept of strong/weak mobility in mobile agents applies.

## 7.5 High Performance Fortran

There are three major approaches to distributed concurrent programming: message passing, shared-variable programming, and data-parallel programming. The first two have been discussed in Chapter 2. Shared-variable programming, not accompanied by migration, on a distributed environment is usually achieved by doing it, using a language such as OpenMP, on top of a software DSM layer.

The High Performance Fortran (HPF) language, a well-known realization of the data-parallel model, provides the notion of abstract processor, which is an abstraction of a real (i.e., physical) processor (and its associated memory). Certain data is distributed to these abstract processors, which are said to own those data. No such data is distributed to, or owned by, more than one abstract processor at any given point of time. On the language level, at least for versions 1.0 and 1.1, the role of an HPF abstract processor as an entity for data distribution is greater than its role as an entity to execute statements, i.e., a place for computation distribution. Computation distribution is usually determined by the compiler using the owner-computes rule [31], for example. The owner-computes rule followed by most HPF compilers assigns computations to abstract processors based on which abstract processor is assigned, by the data distribution directives, to own the data on the left-hand side of an assignment statement. Each abstract processor is mapped to a physical processor by the compiler.

HPF version 2.0 provides the ON directive as an Approved Extension. According

to the HPF 2.0 Language Specification [30], “[t]he ON directive facilitates explicit computation partitioning. The site of recommended execution of a computation can be specified either as an explicitly identified subset of a processor arrangement, or indirectly as the set of processors onto which a data object or template is mapped.” In other words, “[t]he ON directive partitions computations among the processors of a parallel machine (much as the DISTRIBUTE directive partitions the data among the processors).” It should be noted that “[l]ike the mapping directives ALIGN and DISTRIBUTE, this is advice rather than an absolute commandment; the compiler may override an ON directive.” The HPF Language Specification provides an example on how to use the ON clause:

```
!HPF$ INDEPENDENT
DO I = 2, N - 1
  !HPF$ ON HOME(A(I))
  A(I) = (B(I) + B(I - 1) + B(I + 1))/3
  !HPF$ ON HOME C(I + 1)
  C(I + 1) = A(I) * D(I + 1)
ENDDO
```

The computation of the first assignment statement is assigned to the owner of  $A(I)$  whereas that of the second assignment statement is assigned to the owner of  $C(I+1)$ . This code essentially results in a computation distribution that the owner-computes rule would.

The way HPF distributes data among abstract processors is very similar to the way MESSENGERS distributes node variables among logical nodes. And the change in the places of computation by using an ON clause in HPF is similar to that by using a hop in MESSENGERS. One difference between MESSENGERS and HPF is that a node variable in MESSENGERS is visible only to Messengers on a logical node in which the node variable is declared, whereas distributed data in HPF are visible to the

entire program, which is not decomposed into concurrent tasks. And as there are no (explicit) multiple tasks in an HPF program, there is no explicit synchronization. On the other hand, different Messengers may need to synchronize cooperatively via node variables. MESSENGERS is a control-parallel model whereas HPF is a data-parallel one.

### **7.5.1 Migration of abstract processors**

Perez [55, 56] investigated approaches to migrating an abstract processor on a computer in the context of HPF. Basically the two approaches he outlined are similar to the two implemented in this dissertation: one abstract processor per process, and abstract processors as threads. He completed a preliminary implementation of the latter approach only. His work aimed at migrating abstract processors to achieve load balancing, and used the PM<sup>2</sup> system [49] for thread migration. Like most migration systems that aimed at load balancing, Perez’s migration was directed by a runtime system, not the programmer. Perez did not consider the impact of multiple-CPU computer in his work; that is, he implicitly assumed that each computer had only one CPU, and multithreading was used only to overlap computation and communication.

### **7.5.2 Hierarchy of abstract processors on one computer**

Benkner et al [7, 8] proposed a scheme to extend HPF for SMP clusters which enabled the compiler to adopt a parallelization strategy combining distributed- and shared-memory parallelism. They proposed a hierarchical mapping scheme by adding a new node grid in addition to the processor grid, so that hosts are mapped to nodes and CPUs on the hosts are mapped to processors. In their words, “Each node process generates a set of threads which emulate the abstract processors mapped to a node

and which execute concurrently in the shared address space of a node [7].” Therefore, their approach is similar to the one-thread-for-one-logical-node approach discussed in Sec. 6.4 in terms of the use of threads. The distributed-memory parallelization among processes are implemented using MPI, as is usually the case with HPF compilation, and the shared-memory parallelization among threads using OpenMP.

# Chapter 8

## Future Work

### 8.1 Hierarchy of physical and logical nodes

It is possible to extend the structure of physical and logical nodes to represent the hierarchical structure of real CPUs and storage. There is a hierarchy of CPUs in the sense that a processor may consist of a number of cores. Hence, a physical node on a higher level may be used to represent the processor, and a physical node on a lower level may be used to represent a core. Similarly, there are different levels of cache in addition to the main memory. Different levels of logical nodes can be used to represent them.

### 8.2 Static binding of node variables

The possibility of binding node variables at compile time is discussed in Sec. 2.3.5. It is conceivable that a computation allocated to a computer may need to read data allocated to another computer. A Messenger is created to compute on a physical/-

logical node on the former computer but it may need to read data on (a logical node associated with a physical node on) the latter. Migrating that Messenger to another node just to bring back the data in its Messenger variables does not seem to be elegant. It is more convenient for the Messenger to be able to read it remotely. Note that a read operation does not involve memory consistency. The possibility of binding a node variable in such a static manner is worth studying.

### **8.3 Additional experiment on computers with larger number of CPUs and/or asymmetric CPUs**

As discussed in Sec. 6.6, the ability of the multi-process version of the MESSENGERS runtime to bind a daemon to a particular CPU was not fully explored as the experimental platforms did not have larger number of CPUs and/or asymmetric CPUs. It would be worthwhile to assess the performance of the MESSENGERS runtime on these architectures.

### **8.4 Data redistribution by migrating physical and/or logical nodes**

There are applications that have distinct computation phases, each having a different optimal data distribution pattern. One way to deal with this is to perform data re-distribution between two phases. There is a large volume of literature discussing the mechanism of data redistribution. However, the vast majority assumes a message passing programming model is used. HPF does provide a REDISTRIBUTE directive for data redistribution.

It is possible to achieve data redistribution on MESSENGERS by migrating logical nodes. Gentleman [24] implemented some mechanism to achieve migration of a logical node, together with Messengers located thereon but did not offer any application for that functionality. An alternative would be to migrate a physical node, with any logical nodes associated with it. That may be accomplished by process/thread migration — recall that a physical node is implemented by a daemon process.

The idea of migrating a physical node and/or a logical node and its usefulness is worth exploring.

# Chapter 9

## Conclusion

The MESSENGERS programming model for distributed concurrent programming has been presented. Prior works suggested that it satisfactorily supports parallel programming on distributed-memory computer systems, namely, networks of computers. This dissertation has presented ways to beneficially use this model on uni-processor computers and multiple-CPU computers. In the process, a runtime system that supports the MESSENGERS model on multiple-CPU computers was built, that runs on Linux 2.6 on the IA-32 and AMD64 architectures and Solaris 9 and 10 on the UltraSPARC and IA-32 architectures.

Major contributions of this dissertation are summarized as below:

1. Concepts of state-migration location-scoped-variable programming used in the MESSENGERS programming model were explained using established notions such as dynamic binding of variables.
2. State-migration location-scoped-variable programming was compared with message passing in-depth to illustrate the advantage of the former in terms of pro-

gram development.

3. Roles of logical nodes and physical nodes in the MESSENGERS programming model were clarified. As a result, it is possible to exploit the use of logical nodes to improve cache performance. The role of physical nodes helped the design of a runtime system that supports MESSENGERS programming on a computer with multiple processors or cores.
4. A numeric algorithm was used to demonstrate the use of the MESSENGERS model and the runtime system to improve data reuse in the cache on a uni-processor computer.
5. Guided by the MESSENGERS programming model, a runtime system that allows better utilization of computers with multiple processors or cores was designed and implemented. Two different versions of such runtime were evaluated on different platforms and satisfactory performance results were obtained.

In conclusion, this research has shown that the MESSENGERS programming model, with the runtime support, is a promising programming tool for both sequential and concurrent programming on shared-memory or distributed-memory systems.

# Bibliography

- [1] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In Richard L. Wexelblat, editor, *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178, July 1995.
- [2] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 112–125, June 1993.
- [3] Jerry Andrews. Linux: The Completely Fair Scheduler, April 2007. <http://kerneltrap.org/node/8059>.
- [4] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl’s Law through EPI throttling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society.
- [5] OpenMP Architecture Review Board. OpenMP specifications, 1997–2008. <http://www.openmp.org>.
- [6] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.
- [7] Siegfried Benkner and Thomas Brandes. High-level data mapping for clusters of SMPs. In Frank Mueller, editor, *Proceedings, 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2001)*, volume 2026 of *Lecture Notes in Computer Science*, pages 1–15, Berlin, Germany, April 2001. Springer-Verlag.
- [8] Siegfried Benkner and Viera Sipkova. Exploiting distributed-memory and shared-memory parallelism on clusters of SMPs with data parallel programs. *International Journal of Parallel Programming*, 31(1):3–19, February 2003.
- [9] Lorenzo Bettini and Rocco De Nicola. Translating strong mobility into weak mobility. In Gian Pietro Picco, editor, *Proceedings, 5th International Conference*

- on Mobile Agents, MA 2001*, volume 2240 of *Lecture Notes in Computer Science*, pages 182–197, Berlin, Germany, December 2001. Springer-Verlag.
- [10] Shekhar Y. Borkar, Pradeep Dubey, Kevin C. Kahn, David J. Kuck, Hans Mulder, Stephen S. Pawlowski, and Justin R. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. White paper, Intel Corporation, 2005. [http://tbp.berkeley.edu/~jdonald/research/cmp/borkar\\_2015.pdf](http://tbp.berkeley.edu/~jdonald/research/cmp/borkar_2015.pdf).
  - [11] Frédéric Boussinot. FairThreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice & Experience*, 18(5):445–469, 2006.
  - [12] Peter Braun and Wilhelm Rossak. *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkits*, chapter 2, page 11. Morgan Kaufmann, Amsterdam, Netherlands, 2005.
  - [13] Peter Braun and Wilhelm Rossak. *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkits*, chapter 3, pages 37–38. Morgan Kaufmann, Amsterdam, Netherlands, 2005.
  - [14] Manfred Broy and Thomas Streicher. View of distributed systems. In Marisa Venturini Zilli, editor, *Proceedings, Advanced School on Mathematical Models for the Semantics of Parallelism (1986)*, volume 280 of *Lecture Notes in Computer Science*, pages 114–143, Berlin, Germany, 1987. Springer-Verlag.
  - [15] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian D. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94, Berlin, Germany, 1999. Springer-Verlag.
  - [16] Bozhidar Dimitrov. Arachne: A compiler-based portable threads architecture supporting migration on heterogeneous networked systems. Master’s thesis, Purdue University, May 1996.
  - [17] Bozhidar Dimitrov and Vernon Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):459–469, May 1998.
  - [18] Ralf S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 239–250, Berkeley, California, USA, June 2000. The USENIX Association.
  - [19] John Fruehe. Planning considerations for multicore processor technology. *Dell Power Solutions*, pages 67–72, May 2005.
  - [20] Munehiro Fukuda. *MESSENGERS: A Distributed Computing System Based on Autonomous Objects*. PhD thesis, Dept. of Information and Computer Science, University of California, Irvine, California, USA, 1997.

- [21] Munehiro Fukuda, Lubomir F. Bic, and Michael B. Dillencourt. Messages versus Messengers in distributed programming. *Journal of Parallel and Distributed Computing*, 57:188–211, 1999.
- [22] Munehiro Fukuda, Christian Wicke, Hairong Kuang, Eugene Gendelman, Koji Noguchi, and Ming Kin Lai. *MESSENGERS User's Manual (version 3.2)*. Department of Computer Science, University of California, Irvine, California, USA, 2009.
- [23] Stefan Fünfroeken. Transparent migration of Java-based mobile agents. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings of the Second International Workshop on Mobile Agents MA '98*, volume 1477 of *Lecture Notes in Computer Science*, pages 39–49, Berlin, Germany, September 1998. Springer Verlag.
- [24] Yevgeniy Gendelman. *Virtual Infrastructure for Mobile Agent Computing*. PhD thesis, Dept. of Information and Computer Science, University of California, Irvine, California, USA, 2002.
- [25] Carlo Ghezzi and Giovanni Vigna. Mobile code paradigms and technologies: A case study. In Kurt Rothermel and Radu Popescu-Zeletin, editors, *Proceedings, First International Workshop on Mobile Agents, MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 39–49, Berlin, Germany, April 1997. Springer Verlag.
- [26] Soraya Ghiasi, Tom Keller, and Freeman Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 199–210, New York, NY, USA, 2005. ACM.
- [27] Matt Gillespie. Transitioning software to future generations of multi-core. *Intel Software Network*, 2007. <http://software.intel.com/en-us/articles/transitioning-software-to-future-generations-of-multi-core>.
- [28] Seth Copen Goldstein. Lazy threads: Compiler and runtime structures for fine-grained parallel programming. Technical Report CSD-1997-975, Department of Computer Science, University of California, Berkeley, California, 1997.
- [29] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37:5–20, 1996.
- [30] High Performance Fortran Forum. High Performance Fortran language specification version 2.0, January 1997. <http://hpff.rice.edu/versions/hpf2/hpf-v20/index.html>.
- [31] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

- [32] Matthew Hohlfeld and Bennet Yee. How to migrate agents. Technical Report CS98-588, Department of Computer Science and Engineering, University of California, San Diego, California, USA, 1998.
- [33] Michael Holtkamp. Thread migration with Active Threads. Technical Report TR-97-038, International Computer Science Institute, Berkeley, California, USA, September 1997.
- [34] Thomas J. R. Hughes, Robert M. Ferencz, and Arthur M. Raefsky. DLEARN – a linear static and dynamic finite element analysis program. In Thomas J. R. Hughes, editor, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, chapter 11. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1987.
- [35] IEEE and The Open Group. 9945-1:2003 (ISO/IEC) [IEEE/ANSI Std 1003.1 2004 edition] Standard for information technology – Portable Operating System Interface (POSIX) – part 1: Base definitions, 2004.
- [36] IEEE and The Open Group. 9945-1:2003 (ISO/IEC) [IEEE/ANSI Std 1003.1 2004 edition] Standard for information technology – Portable Operating System Interface (POSIX) – part 2: System interfaces, 2004.
- [37] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread migration and its applications in distributed shared memory systems. *Journal of Systems and Software*, 42(1):71–87, July 1998.
- [38] Hai Jiang and Vipin Chaudary. MigThread: Thread migration in DSM systems. In *Proceedings, International Conference on Parallel Processing Workshop on Compile/Runtime Techniques for Parallel Computing*, pages 581–588, Alamos, Calif., August 2002. IEEE Computer Society.
- [39] David Kanter. Inside Nehalem: Intel’s future processor and system, April 2008. <http://www.realworldtech.com/page.cfm?ArticleID=RWT040208182719>.
- [40] KernelNewbies. Linux 2.6.23, 2008. [http://kernelnewbies.org/Linux\\_2\\_6\\_23](http://kernelnewbies.org/Linux_2_6_23).
- [41] Peizong Lee and Zvi Meir Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Transactions on Programming Languages and Systems*, 24(1):1–50, January 2002.
- [42] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.
- [43] Edward Mascarenhas and Vernon Rego. Ariadne: Architecture of a portable threads system supporting thread migration. *Software – Practice and Experience*, 26(3):327–356, March 1996.

- [44] Edward Mascarenhas and Vernon Rego. Ariadne user manual version 3.1. Technical Report 94-081, Department of Computer Science, Purdue University, W. Lafayette, Indiana, USA, 1998.
- [45] Edward Mascarenhas and Vernon Rego. Migrant threads on process farms: parallel programming with Ariadne. *Concurrency: Practice and Experience*, 10(9):673–698, 1998.
- [46] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, 1995. <http://www.mpi-forum.org>.
- [47] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 1997. <http://www.mpi-forum.org>.
- [48] Junji Nakano. Parallel computing techniques. In James E. Gentle, Wolfgang Härdle, and Yuchi Mori, editors, *Handbook of Computational Statistics: Concepts and Methods*, chapter II.8, pages 237–266. Springer, Berlin, Germany, 2004.
- [49] Raymond Namyst and Jean-François Méhaut. PM<sup>2</sup> parallel multithreaded machine: A multithreaded environment on top of PVM. In Jack Dongarra, M. Gengler, B. Tourancheau, and X. Vigouroux, editors, *Proceedings, Second Euro PVM Users Group Meeting EuroPVM'95*, pages 179–184, September 1995.
- [50] Michael G. Norman and Peter Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, 1993.
- [51] Lei Pan. *Navigational Programming: Toward Structured Distributed Parallel Programming*. PhD thesis, Dept. of Computer Science, University of California, Irvine, California, USA, 2005.
- [52] Lei Pan, Ming Kin Lai, Michael B. Dillencourt, and Lubomir F. Bic. Mobile pipelines: Parallelizing left-looking algorithms using navigational programming. In David A. Bader, Manish Parashar, Varadarajan Sridhar, and Viktor K. Prasanna, editors, *Proceedings, 12th IEEE International Conference on High Performance Computing – HiPC 2005*, volume 3769 of *Lecture Notes in Computer Science*, pages 201–212, Berlin, Germany, December 2005. Springer-Verlag.
- [53] Lei Pan, Ming Kin Lai, Koji Noguchi, Javid J. Huseynov, Lubomir F. Bic, and Michael B. Dillencourt. Distributed parallel computing using navigational programming. *International Journal of Parallel Programming*, 32(1):1–37, February 2004.
- [54] Lei Pan, Wenhui Zhang, Arthur Asuncion, Ming Kin Lai, Michael B. Dillencourt, Lubomir F. Bic, and Laurence T. Yang. Toward incremental parallelization using navigational programming. *IEICE Transactions on Information and Systems*, E-89D(2):390–398, February 2006.

- [55] Christian Perez. Load balancing HPF programs by migrating virtual processors. Research Report 3037, INRIA, Montbonnot Saint-Martin, France, November 1996.
- [56] Christian Perez. Load balancing HPF programs by migrating virtual processors. In *Proceedings of the 1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS '97)*, pages 85–92, Los Alamitos, California, USA, April 1997. IEEE Computer Society.
- [57] Joseph Pranevich. The wonderful world of Linux 2.6. *Linux Gazette*, 98, January 2004. Reprinted from <http://kniggit.net/wwol26.html>.
- [58] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, pages 199–210, New York, NY, USA, June 2008. ACM.
- [59] Robert W. Sebasta. *Concepts of Programming Languages*, chapter 13, page 539. Pearson Addison-Wesley, Boston, Massachusetts, USA, seventh edition, 2006.
- [60] Robert W. Sebasta. *Concepts of Programming Languages*, chapter 9, pages 378–379. Pearson Addison-Wesley, Boston, Massachusetts, USA, seventh edition, 2006.
- [61] Robert W. Sebasta. *Concepts of Programming Languages*, chapter 3, page 166. Pearson Addison-Wesley, Boston, Massachusetts, USA, seventh edition, 2006.
- [62] Suresh Siddha. Multi-core and Linux kernel. Whitepaper, Intel Software Network. <http://software.intel.com/sites/oss/pdf/mclinux.pdf> Also titled as Linux kernel scheduler optimizations for multi-core.
- [63] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.
- [64] Kenjiro Tauro and Akinori Yonezawa. Fine-grain multithreading with minimal compiler support – a cost effective approach to implementing efficient multithreading languages. *ACM SIGPLAN Notices*, 32(5):320–333, May 1997.
- [65] Top500.Org. 30th edition of TOP500 list of world’s fastest supercomputers released, big turnover among the top 10 systems, November 2007. [http://www.top500.org/blog/2007/11/09/30th\\_edition\\_top500\\_list\\_world\\_s\\_fastest\\_supercomputers\\_released\\_big\\_turnover\\_among\\_top\\_10\\_systems](http://www.top500.org/blog/2007/11/09/30th_edition_top500_list_world_s_fastest_supercomputers_released_big_turnover_among_top_10_systems).
- [66] Top500.Org. Jaguar chases roadrunner, but can’t grab top spot on latest list of world’s TOP500 supercomputers, November 2008. <http://www.top500.org/lists/2008/11/press-release>.

- [67] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in Java. In David Kotz and Friedemann Mattern, editors, *Agent Systems, Mobile Agents, and Applications; Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*, pages 29–43. Springer-Verlag, September 2000.
- [68] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multi-threading: Maximizing on-chip parallelism. In *Proceedings, 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.
- [69] Tim Walsh, Paddy Nixon, and Simon Dobson. As strong as possible mobility: An architecture for stateful object migration on the Internet. Technical Report TCD-CS-2000-11, Department of Computer Science, Trinity College, Dublin, Ireland, 2000.
- [70] Justin Ward. Linux 2.6: Let’s take a look under the hood. *The Linux Beacon*, 1:4, February 2004. <http://midrangeserver.com/tlb/tlb021704-story01.html>.
- [71] Boris Weissman. Active Threads: an extensible and portable light-weight thread system. Technical Report TR-97-036, International Computer Science Institute, Berkeley, California, USA, September 1997.
- [72] Boris Weissman and Benedict Gomes. High-performance thread migration on clusters of SMPs. In Rajkumar Buyya and Clemens Szyperski, editors, *Cluster computing*, pages 85–96. Nova Science Publishers, Commack, New York, USA, 2001.
- [73] Christian Wicke. Implementation of an autonomous agents system. Diploma thesis, Universität Karlsruhe, Karlsruhe, Germany, 1998.
- [74] Christian Wicke, Lubomir F. Bic, and Michael B. Dillencourt. Compiling for fast state capture of mobile agents. In E. H. D’Hollander, G. R. Joubert, F. J. Peters, and H. J. Sips, editors, *Parallel Computing: Fundamentals & Applications; Proceedings of the International Conference ParCo99*, pages 714–721. Imperial College Press, August 1999.
- [75] Christian Wicke, Lubomir F. Bic, Michael B. Dillencourt, and Munehiro Fukuda. Automatic state capture of self-migrating computations. In *Proceedings, International Workshop on Computing and Communication in the Presence of Mobility*, April 1998.
- [76] Christian Wicke, Lubomir F. Bic, Michael B. Dillencourt, and Munehiro Fukuda. Automatic state capture of self-migrating computations in MESSENGERS. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings, Second International Workshop on Mobile Agents, MA ’98*, volume 1477 of *Lecture Notes in Computer Science*, pages 68–79, Berlin, Germany, September 1998. Springer-Verlag.

- [77] Michael Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 655–664, New York, NY, USA, November 1989. ACM.
- [78] Thomas Zangerl. Optimisation: Operating system scheduling on multi-core architectures, 2008. <http://homepage.uibk.ac.at/~csae6894/MulticoreScheduling.pdf>.
- [79] Gengbin Zheng, Laxmikant V. Kalé, and Orion Sky Lawlor. Multiple flows of control in migratable parallel programs. In *Proceedings, 2006 International Conference on Parallel Processing Workshops (ICPPW'06) (The 8th Workshop on High Performance Scientific and Engineering Computing)*, pages 435–444. IEEE Computer Society, August 2006.