High Performance Linda Using a Class Library

Cover design: Crasborn Graphic Designers bno, Valkenburg a.d. Geul.

This book is number 243 of the Tinbergen Institute Research Series, established through cooperation between Thela Thesis and the Tinbergen Institute. A list of books which already appeared in the series can be found in the back.

High Performance Linda Using a Class Library

(EFFICIENT LINDA MET EEN KLASSENBIBLIOTHEEK)

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR
AAN DE ERASMUS UNIVERSITEIT ROTTERDAM
OP GEZAG VAN DE RECTOR MAGNIFICUS
PROF.DR.IR. J.H. VAN BEMMEL
EN VOLGENS BESLUIT VAN HET COLLEGE VOOR PROMOTIES

DE OPENBARE VERDEDIGING ZAL PLAATSVINDEN OP
VRIJDAG 19 JANUARI 2001 OM 13:30 UUR

DOOR

ROEL ANTOIN VAN DER GOOT
GEBOREN TE GELEEN

Promotiecommissie

Promotoren:	Prof.dr. Arie de Bruin
	Prof.dr. Jonathan Schaeffer
Overige leden:	Prof.dr.ir. Mehmet Aksit
	Prof.dr.ir. Henri E. Bal
	Dr.ir. Jan van den Berg
	Prof.dr.ir. Rommert Dekker
	Prof.dr. Ron M. Lee

ISBN 90-517-0817-3

To my mum and dad

At the end of my formal education
I can tell you
that it was your informal education
that taught me most.

Contents

1 Introduction **1**

1.1 Imperative Programming 1
1.2 Parallel Programming 3
1.2.1 Parallel Architectures 4
1.3 Contributions 7
1.4 Features 8
1.4.1 *Blossom* 8
1.4.2 *Roots* 9
1.5 Contents 11

2 Linda **13**

2.1 The Linda Model 13
2.1.1 Tuples 14
2.1.2 Value Tuples 14
2.1.3 Query Tuples 14
2.1.4 Matching 15
2.1.5 The Operation `out` 15
2.1.6 The Operation `in` 16
2.1.7 The Operation `rd` 17
2.1.8 The Operation `inp` 17
2.1.9 The Operation `rdp` 18
2.1.10 The Operation `eval` 18
2.1.11 General Comments 19
2.2 Examples 19
2.2.1 Shared Queue 19
2.2.2 Shared Stack 21
2.2.3 Dining Philosophers 22
2.2.4 Matrix Multiplication 23
2.2.5 Primes 24

3 Enhancements **27**
3.1 Model Enhancements . . . 27
3.1.1 Multiple Tuple Spaces . . . 27
3.1.2 Array Fields . . . 28
3.1.3 Wild Card Fields . . . 29
3.1.4 Synchronous `out` . . . 30
3.2 Efficiency Enhancements . . . 31
3.2.1 Homogenization of the Tuple Space . . . 31
3.2.2 Distribution of the Tuple Space . . . 32
3.2.3 The Operation `in`-update-`out` . . . 33
3.2.4 Prefetching . . . 34
3.2.5 Tuple Access Patterns . . . 35
3.2.6 The Operation `collect` . . . 36
3.2.7 The Operation `copy-collect` . . . 36
3.3 Paradigm Enhancements . . . 38
3.3.1 Persistence . . . 38
3.3.2 Fault Tolerance . . . 38
3.3.3 Openness . . . 39
3.4 Software Engineering Enhancements . . . 39

4 *Blossom* **41**
4.1 Introduction . . . 41
4.1.1 The Library Approach . . . 41
4.1.2 The Precompiler Approach . . . 43
4.1.3 The Type Parameterized Library Approach . . . 44
4.2 The Basic *Blossom* Language . . . 46
4.2.1 Multiple tuple spaces . . . 46
4.2.2 Tuple space operations . . . 46
4.2.3 Process Creation . . . 47
4.2.4 Tuple Arguments . . . 47
4.3 Software Engineering Enhancements . . . 48
4.3.1 Strongly Typed Tuple Spaces . . . 50
4.3.2 Field Access Patterns . . . 51
4.3.3 Tuple Space Access Patterns . . . 52
4.3.4 Tuple Space Assertions . . . 54
4.4 Example . . . 55
4.5 Current Status of *Blossom* . . . 56
4.5.1 Model Enhancements . . . 56
4.5.2 Efficiency Enhancements . . . 58
4.5.3 Paradigm Enhancements . . . 60
4.5.4 Software Engineering Enhancements . . . 60
4.6 Conclusions . . . 63

5 *Blossom* Architecture **65**
5.1 Example . . . 65
5.2 Front End . . . 67
5.3 Communication . . . 68
5.3.1 Member Function `put` . . . 69
5.3.2 Member Function `put_s` . . . 69
5.3.3 Member Function `get` . . . 69
5.3.4 Member Function `copy` . . . 70
5.3.5 Member Function `get_nb` . . . 71
5.3.6 Member Function `copy_nb` . . . 71
5.4 Back End . . . 71

6 *Roots* **75**
6.1 Introduction . . . 75
6.2 Communication primitives . . . 76
6.2.1 Active Messages . . . 77
6.2.2 Sending Active Messages . . . 78
6.2.3 Receiving Active Messages . . . 78
6.2.4 Variable Sized Active Messages . . . 79
6.3 Threads . . . 80
6.4 Implementation . . . 81
6.5 Example: Traveling Salesman Problem . . . 83
6.6 Conclusions . . . 84

7 *Blossom* Implementation **87**
7.1 Type Parameterization . . . 87
7.2 Type Parameterization in C++ . . . 88
7.2.1 Traditional Classes . . . 88
7.2.2 Templated Classes . . . 90
7.2.3 Templated Functions . . . 91
7.2.4 Template Specialization . . . 92
7.2.5 Partial Template Specialization . . . 92
7.3 Tuples . . . 93
7.3.1 Default Template Arguments . . . 94
7.3.2 Template Specialization . . . 95
7.4 Tuple Space Back End . . . 97
7.4.1 Class `TupleSpaceStorage` . . . 97
7.4.2 Query Tuples . . . 98
7.4.3 Value Tuples . . . 99
7.5 Communication . . . 100
7.5.1 Member Function `put` . . . 100
7.5.2 Member Function `get` . . . 101
7.5.3 Member Function `get_nb` . . . 102
7.6 Tuple Space Front End . . . 103

7.6.1 Class `TupleSpace` . . . 105
7.6.2 Class `TupleSpaceTuples` . . . 106
7.6.3 Class `TupleSpaceFields` . . . 109
7.6.4 Class `TupleSpaceAccessPattern` . . . 111
7.7 Type Conversions . . . 114
7.7.1 `TupleSpaceFields` Selection . . . 114
7.7.2 Query Tuple Selection . . . 118
7.7.3 Synchronous or Asynchronous . . . 120

8 Experiments 123
8.1 Experiments Design . . . 123
8.2 Experiment: `Replicated` Tuple Space . . . 124
8.3 Experiment: `Local` Tuple Space . . . 126
8.4 Experiment: `Local` and `Replicated` Tuple Spaces . . . 129
8.5 Experiment: Futures . . . 131
8.6 Conclusions . . . 135

9 Conclusions and Future Work 137

A *Blossom* Syntax 139
A.1 Tuple Spaces . . . 139
A.2 Arguments . . . 140

B Results 143
B.1 Traveling Salesman Problem . . . 144
B.1.1 `Distributed` jobs and `Local` minimum . . . 144
B.1.2 `Distributed` jobs and `Replicated` minimum . . . 146
B.1.3 `Local` jobs and `Local` minimum . . . 148
B.1.4 `Local` jobs and `Replicated` minimum . . . 150
B.1.5 Sequential . . . 152
B.2 Reaction-Diffusion . . . 153
B.2.1 Without Futures . . . 153
B.2.2 With Futures . . . 154
B.2.3 Sequential . . . 155

Preface

I desperately wanted to do a Ph.D. after finishing my Master's degree, not knowing what was ahead of me. The road to a Ph.D. is full of unexpected obstacles, arduous hours of work, complicated circumstances, numerous frustrations, but yet rewards.

Unfortunately and sadly, my promoter and supervisor Dr. Jan van den Bos was diagnosed with an incurable cancer ten months into the program and passed away three months later. A new professor, Dr. Arie de Bruin, was assigned to me. At that time, the Faculty of Economics at the Erasmus University Rotterdam had tremendous financial problems and one third of the personnel were laid off work. Arie was heading the reorganization within the Computing Science Department which left him little time to supervise me. My wall climbing activities and skills soared during this period! Fortunately, when the difficulties of the reorganization had settled, Arie arranged for me to attend the Pooma Conference in Santa Fe, New Mexico, and meet Dr. Gregory V. Wilson. It was unheard of that somebody was allowed to go to a conference without presenting anything.

After meeting Greg at the conference, he was willing to help me out and invited me to Toronto. However the conference also taught me that others were much further ahead in my research topic (a parallel/distributed language based on objects with an access protocol on the methods [53]). Back in Rotterdam during the dinner after Aske Plaat's Ph.D. defense, his second promoter, Dr. Jonathan Schaeffer, challenged that if I would solve a puzzle that he had just given to Aske as a present, he would invite me to Edmonton. The puzzle was solved in ten minutes.

In September 1996, I went to Canada, where I first visited IBM Toronto Labs. I set myself the goal to find a new research topic. Greg Wilson encouraged me to read many excellent papers in parallel and distributed computing and wanted to push me in the direction of data parallelism using templates in C++. One evening, I commented that using templates would be ideal for Linda, unaware that Greg did his Ph.D. on Linda. Greg was immediately enthusiastic and in the following month, I implemented the first version of what I would later name *Blossom*. During the following visit to Edmonton, a new and improved version of *Blossom* was implemented and the first article on my new research topic was written. Back in the Netherlands where I did not have a decent C++ compiler nor a modern Unix computer (one with Posix threading), work mainly stagnated because the link to Edmonton was usually slow. I wrote another article.[1]

[1] The article was accepted for EuroPar'97. The organization of EuroPar later changed the paper to a poster with

The decision to visit the University of Alberta in the Summer of 1997 was an easy one. In September I attended Coordination'97 and EuroPar'97 in Berlin and Bonn, Germany. When I returned I started writing my thesis. However after half a year only three chapters had been finished. Disillusioned, I went back to Edmonton where Jonathan acquired a one month test license for SCA Linda, which would run on a cluster of IBMs. I started porting *Blossom* to AIX (IBM's Unix operating system) but could not get it to work due to an undocumented return value of a system call. However, I was successful in implementing the Cowichan problems in SCA Linda and wrote two more chapters. Back in the Netherlands there was more editing and rewriting of chapters and I got more frustrated. In December that year, I went to the Vrije Universiteit in Amsterdam to finish the comparison between SCA Linda and *Blossom*. I converted the Cowichan problems from Linda to *Blossom* and ran them on a cluster of Suns. When the decision was made that the results could not be used (partially because the SCA Linda license did not allow a comparison), I decided to stop pursuing a Ph.D. and take a breather from editing and rewriting chapters. In May 1998 I moved to Edmonton, accepted a Summer job and later a full time position at MACI (Multimedia and Advanced Computing Institute) both at the University of Alberta. Jonathan persevered in encouraging me to finish my thesis, but I did not give in until the Summer of 2000.

Acknowledgements

I am extremely grateful and indebted to Jonathan Schaeffer who did an awful lot for me. He invited me to Edmonton several times, gave me financial support during these visits, arranged the Summer job and position at MACI, and assisted me with a good reference and in finding my current job at YottaYotta. He is mainly to blame that I finished my thesis. Jonathan, thank you very, very much. I owe you.

My parents are equally to blame for my graduation. I am pretty sure that they are prouder of what I achieve(d) then I am myself. Even though they knew how bad I felt about the thesis at several points in time, they kept believing in me. Mum and dad, this thesis is for you, because you are such excellent parents.

Many other people have to be thanked, and I will do so by city. I will not mention everybody by name because it would be too easy to upset somebody when I forget to mention his or her name.

At the Erasmus University Rotterdam, I would like to thank all my colleagues at the Department of Computer Science and the system administrators. Special thanks go to my former room mates Patrick van der Laag, Aske Plaat and Mark Polman for keeping the work environment enjoyable and to Cia Scholte for all the conversations and errands she did for me while I was in Edmonton. I am also indebted to my promoter Arie de Bruin who was assigned to supervise me although he did not know much about distributed computing. He made it possible for me to visit Santa Fe, Toronto and Edmonton.

publication of an abstract. After the conference they decided to not publish abstracts for posters.

Henri Bal of the Vrije Universiteit in Amsterdam was so nice to help me out by allowing me to use a network of Suns to finish the comparison and offering his constructive comments.

In Delft I have fond memories of my room mates Kim van Schagen, Pieter Hafkamp, Marc Bisschops and Erik van der Starre. It was great relaxing with you after work. I spent a lot of my student time with Gorgel, a group of students whom I hope to keep seeing. Friends for life are the members of the *Lustrum Commissie* with whom I spend a year organizing the 95th anniversary of the *Delftsche Studenten Bond*.

I owe gratitude to Greg Wilson who kindly invited me to Toronto and housed me for a month. Greg, thank you for sharing all your knowledge with me in the form of articles and discussions. The second month of my stay in Toronto, Diego Moscoso offered to house me. I fondly remember the many good hours we spent together. Thank you. At IBM in Toronto I had many technical discussions with Bill O'Farrell and Josée Lajoie, who taught me a lot about C++ and were always willing to answer any C++ questions I had.

In Edmonton, many fond memories go back to the Parallel Lab, home of the Parallel group as well as the Games group. The environment was so invigorating that at times it was hard to get any work done! Thanks guys. Special memories go back to the Games nights and the Tuesday movie nights. At CNS (Computing and Network Services) where I worked for MACI, I would like to thank all the people from the Research group. I would also like to thank the people with whom I jogged, played soccer or played squash during lunch breaks. Nowadays, I am thrilled to go to work to meet my bright colleagues at YottaYotta. It is nice to be outgeeked!

Special thanks go to Stephanie Schaeffer for the many nice meals she cooked for me. While Stephanie's mother was visiting her brother in New Mexico, I was allowed to stay in her apartment for four months. Finally, I would like to thank my girl friend Kirsten Oates for her love and understanding. Although she is not a computer scientist, she read my thesis and gave me valuable comments. Kirsten, I love you.

I am relieved to be able to close the long and tedious Ph.D. chapter in my life. It presented many challenges and frustrations. In the end it was my friends that kept me positive. My thanks and gratitude to you all.

Roel,
Edmonton, December 2000.

Chapter 1

Introduction

This thesis introduces a new programming language, *Blossom*, and discusses its design and implementation. There are a plethora of programming languages available today, ranging from general-purpose languages (such as Java) to specialized languages (such as SPSS), from procedural languages (such as FORTRAN) to logic programming (Prolog), and from sequential language constructs (most languages) to parallel constructs (High Performance FORTRAN). *Blossom* is a general-purpose, imperative, parallel programming language.

Why do we need yet another programming language? Despite decades of research, developing high-performance parallel applications is a difficult endeavor. In part this is because of the tight relationship between the application and the parallel architecture that it is to be run on. However, most of the blame lies in the quality of the software tools available for designing, developing, testing, debugging and tuning parallel programs. The *Blossom* language is used to explore new ideas for expressing parallelism that are aimed at simplifying the task of writing high-performance parallel programs. Rather than invent a new language (a mistake if the language is to gain wide-spread acceptance [48]), *Blossom* builds on top of an established, popular, general-purpose, imperative programming language (C++).

1.1 Imperative Programming

Traditionally, programming languages offer a series of commands (or instructions) that the computer has to execute (hence the adjective *imperative*). To this date most programming languages are still imperative, although they have evolved considerably from the early imperative programs of a scant few decades ago.

At the dawn of the computer era, computers were programmed by entering a (possibly long) series of numbers into the computer. Each number corresponded to an instruction (such as `add`), and possibly some arguments (which numbers to `add`). Of course, the computer program was hard to read (and debug) and thus was error prone.

In the mid-1950s, the first programming *language* was born. Assembly language replaced numbers with mnemonic names, resulting in a program that was easier to read and understand and, hence, less error prone. These early languages did not provide many abstractions, and programmers often had to repeat sequences of code that were almost similar (for example, pushing and popping registers to and from the stack).

In the late 1950s, the first of the so-called third generation programming languages appeared: FORTRAN (FORmula TRANslation). The language added control statements (such as `for` loops and `if` statement) and data structures (arrays) to the programmer's repertoire, allowing for a more concise and less error-prone expression of algorithms. FORTRAN gave way to a succession of successful languages, including COBOL, Algol, C, and Pascal.

The evolution from programs as a series of numbers to programs as a series of control and data statements continued. Researchers realized that abstraction was a key idea for writing reusable code. This gave rise to a new generation of languages based on objects, encapsulating data structures and operations (or methods) that operate on that data.

In the early 1960s, the object oriented programming language *CPL* (Combined Programming Language) was designed at Cambridge University and London University. However it would remain a theoretical programming language mainly because the compiler technology was in its infancy. Martin Richards implemented a simplified version of *CPL*, called *Basic CPL* or *BCPL* for short. Objects were typeless and their type was inferred by the operations on them [40].

Object-oriented languages came to the fore in the 1980s with the radical ideas (at the time) embodied in SmallTalk. Many of these ideas were later used for the design and implementation of the popular Java programming language. Object-oriented programming is now standard fare for introductory computing science courses, and Java is the most popular language for introducing students to computing.

The history of the development of programming languages can be clearly seen by following the evolution of one research group's efforts. In 1969 Ken Thompson started implementing an operating system that would later be known as Unix. The first implementation was in assembly language, but quite soon (1970) he realized that he needed a better language. The result was the creation of the language *B*, which was used to write one of the early versions of Unix. *B* was influenced by the *BCPL* language and, like *BCPL*, all data was typeless. Eventually some of the deficiencies in *B* proved to be limiting development of Unix, so Thompson's colleague, Dennis Ritchie, created a new language which he called *C* [28, 29]. Soon afterwards Unix was rewritten in *C*. In the 1980s Bjarne Stroustrup started adding object orientation to *C* and developed *C with classes* which later became *C++*.

The above discussion shows that sequential imperative languages evolved from *typeless* via *typed* to *object oriented*. Each subsequent generation of languages removed the programmer further from the machine level, increased the level of abstraction available to the programmer, increased the sophistication of error detection within the language, and, presumably, increased programmer productivity.

The parallel programming language introduced in this thesis, *Blossom*, is a successor of the *C* version of the parallel programming model *Linda* (*C-Linda*; see Chapter 2). Although *C* is typed, the data structure introduced by *Linda* (the tuple space) is typeless. *Blossom* not only types the tuple space, but also introduces object orientation (see Chapter 4). In effect, *Blossom* is trying to evolve a well-known and successful parallel programming model to use modern programming practices.

1.2 Parallel Programming

Parallel programming languages simplify the task of writing programs that utilize multiple processors. Ideally, a program that runs on N processors can solve a problem in $1/N$th the time of a sequential program. Most parallel programming languages are imperative, and try to achieve the parallelism in one of two ways:

Data Parallelism Data parallel programming languages try to distribute the data over the processes (usually one process per processor). Each process then performs operations on the local data. The main programming difficulty is to find the correct data distribution that allows for good parallelism. Redistributing the data every now and then may be needed. An example of a data parallel language is High Performance FORTRAN (HPF) which can distribute matrices over multiple processes.

A data parallel program can be easy to write, since the programmer specifies how the different data structures have to be distributed and the compiler generates the necessary synchronization and communication.

Task Parallelism Not all applications are suited for data parallelism. Applications with irregular communication patterns are usually better off with task parallelism. A task parallel solution divides the problem into several smaller problems, also known as tasks. Each process performs tasks until all are completed. At that time one process collects all the results and reports back to the user.

A disadvantage of task parallelism is that it puts more burden on the programmer. The programmer has to specify tasks and the synchronization and communication between the processes. The Parallel Virtual Machine (PVM) library is an example of a tool that allows the programmer to create whatever communications pattern is needed.

Most parallel programming languages force the programmer to choose: use data parallelism or use task parallelism. Few programming languages allow the programmer to mix data and task parallelism. Linda and *Blossom* belong to this group. Other languages that have support for both data and task parallelism are Braid [56], Fx [47], Opus [31], and Orca [23].[1] Mixing data and task parallelism illustrates one of the major obstacles to success for any parallel programming tool [6]. In the sequential world, there is a single architectural model that languages compile into (the so-called von Neumann architecture). In the parallel

[1] These latter languages have more support for data parallelism than Linda and *Blossom*.

world, there are different programming models, and the effectiveness of a particular model depends on the architecture of the parallel computer being used. Thus, one cannot discuss the issues of a parallel programming language without knowing the underlying hardware issues. In effect, one has to map the parallel solution to the parallel hardware.

1.2.1 Parallel Architectures

There are roughly three main categories of parallel architectures. Shared memory architectures have multiple processors residing in one machine and sharing a common main memory. When the processors reside in separate machines connected by a network, or when the processors reside in one machine but each have their local memory, this is usually referred to as distributed memory parallelism. The third category of machines, vector processors, is waning in popularity and is not discussed further here.

The main difference between shared and distributed memory architectures is the access time to main memory. On a distributed memory machine (think a network of computers) the local main memory is accessed much faster than the remote main memory (the main memory in another computer). On a shared memory machine, the access time to the main memory is always the same.

The distinction between shared and distributed memory is an important one, since it influences the costs of *communication* (exchanging information between processes) and *synchronization* (ensuring that events happen in the correct order). For a shared memory computer, communication between processes is simple and implicit: processes read from and write to shared memory. However, a parallel program must be aware of the synchronization of memory accesses. It should prevent two processes from writing to the same memory location at the same time, because the outcome may be undetermined. In contrast, both communication and synchronization are explicit in programs running on distributed memory machines. For example, when the program starts the main process has to send data to the other processes (communication). The receiving processes should not start computing before they receive the data (synchronization). The absence of the need to communicate on a shared memory machine, makes programming shared memory machines much easier.

The difference between shared and distributed memory parallelism is not always so clear-cut. For example, the NUMA-CC (Non Uniform Memory Access Cache Coherence) architecture of the SGI Origin 2000 pretends to be shared memory, although the access time to the memory differs depending on what part of the main memory the program wants to access.

There are numerous ways to handle communication and synchronization in shared memory and distributed memory systems. Some of the popular techniques are as follows:

Message Passing

In the message passing model, processes communicate through the exchange of messages. The sending process explicitly sends a message and the receiving process explicitly receives a message. Libraries that provide this basic communication paradigm include Parallel Virtual Machine (PVM) [17] and Message Passing Interface (MPI) [21]. Programming languages based on this model are Communicating Sequential Processes (CSP) [24] and Occam [41]. Actors [1] is also based on this model with the enhancement that every message also contains a continuation that tells the receiving process what it should do next (until the next receipt of a message).

Remote Procedure Call

Since procedure calls are one of the main building blocks of imperative programming languages, many developers like the idea of having inter-process communication conform to the familiar procedure call semantics. Regardless of whether a function resides within the same process or a remote one, the interface to invoke its execution is the same.

When object orientation became fashionable, the *Remote Procedure Call* (RPC) evolved to a *Remote Method Invocation* (RMI). Many parallel object-oriented languages are based on this communication model

Distributed Shared Memory

The distributed shared memory model implements a virtual shared memory on top of a distributed memory model. This allows the programmer to view distributed memory using a shared memory programming model. Non-local memory accesses translate into (hidden) communication.

A distributed shared memory model can be implemented in several ways. The physical memory of the distributed shared memory may reside inside one process's memory, where all other processes have to communicate to this process to read or write data. Although simple to implement, this implementation has a number of problems. The process on which the shared memory resides is very likely to become a communication bottleneck. Furthermore access to the memory should be regulated to prevent inconsistency of the shared memory. Inconsistency may occur when two processes try to access the same address in the shared memory at the same time. For example one process may be reading the data, while another is changing it. What will the first process read? The old value, the new value, or an intermediate value? Obviously the semantics have to be properly defined.

Another approach for implementing distributed shared memory is to replicate the shared memory in each process's memory. Each process can read the shared data locally; however, when writing shared data, the data has to be distributed to the other processes. Keeping the replicated shared memory consistent is again very important.

Shared memory machines have fundamental architectural limitations that restrict the scalability of the architecture. Distributed shared memory gets around the scalability issue, but has a hidden performance cost (communication). There are programming abstractions that can simplify the programmer's task and reduce the performance overhead:

Shared Data In the shared data model the programmer indicates which pieces of data are shared. He may even indicate whether the data should be replicated or on which process the data should be stored in memory. This approach results in shared data distributed over all the processes. The bottleneck of the shared memory on one process is circumvented. The consistency of data is however still a problem that the programmer has to solve. Most of these systems have evolved nowadays to what is called *Shared Objects* (see below).

Shared Pages In the shared pages model of shared memory the programmer puts data on memory pages. Pages migrate between all the processes. If a process wants to write to a page it requests the page and updates it. This approach is simple and fast because the computer hardware provides the essential tools (in the form of the SEGV signal). However dynamic data structures do not really combine well with a fixed page size. The shared page model is used in ADSM [35], Cashmere [45], DSM-Threads [36], JIAJIA [25], Millipede [15], PAMS [37] and Treadmarks [3].

Shared Objects A shared object is an object that is known to potentially many processes, which can invoke its operations. Because the object manages its own data, the consistency of the object's data can easily be obtained by making all of the operations on a shared object atomic. Another advantage of shared objects is that they combine neatly with the object oriented programming paradigm. Examples of systems that use shared objects are Aurora [34], Filaments [33], ICC++ [14], and Orca [5].

Tuple Spaces A different approach to shared memory programming is offered by the Linda programming model. Linda provides an associative shared memory accessible to all processes. Processes put tuples (data) in and retrieve tuples associatively from the tuple space. The consistency of the tuple space can be obtained by making the tuple space operations atomic.

The Linda model can be superposed on any sequential language. Scientific Computing Associates (SCA) has superimposed Linda on C and FORTRAN, resulting in the parallel programming languages C-Linda and FORTRAN-Linda, respectively. The SCA Linda's are considered to be the best available (performance wise). They use a precompiler to transform the, say, C-Linda program to a C program with calls to a Linda C library.

There are several Linda implementations that do not use a precompiler; none of them produce high performance (execution time) executables. This is mainly due to the smart optimizations that a precompiler can do. In principle, embedding library calls into an application would allow for the same optimization opportunities, however the penalty to be paid is too high. Programmers have to specify much extra information in their programs. As a result the programs no longer look like neat and elegant Linda programs.

Linda offers an attractive programming model that can be adapted to run on shared memory and distributed memory architectures. It is a simple but powerful language that has proven itself. Adding object orientation and strong typing makes Linda even more interesting from a programmer's point of view.

1.3 Contributions

The major contributions of this thesis can be briefly summarized as follows:

- *Blossom* extends Linda with multiple, typed tuple spaces and object orientation. Enhancing the power of the language simplifies the programmer's task of expressing a correct parallel program.

- *Blossom* is implemented as a library and not as a precompiler like SCA Linda. This is achieved by using type parameterization in a novel way. This style of implementation allows for efficiency enhancing optimizations.

- Programs written using *Blossom* have extra performance features over programs written in standard Linda.

This dissertation introduces a new attempt to make a high performance Linda language without the use of a precompiler. The resulting language *Blossom* contains more optimizations than the best Lindas that use precompilers do. A full description of *Blossom* is given in Chapter 4.

This is all possible by changing the traditional Linda model slightly. Most optimizations that Linda performs are based on the types of the Linda tuples. However the Linda tuple space is typeless! *Blossom* introduces multiple typed tuple spaces. This means that the programmer has to specify which kind of tuples, the tuple space is allowed to contain. By doing this the compiler can warn the programmer when he tries to insert a tuple in a wrongly typed tuple space. The advantages and disadvantages of the *Blossom* approach are discussed in Chapter 4.

The *Blossom* library makes use of the types specified for the tuple spaces. The library can do so because it is type parameterized. When a programmer declares a tuple space variable, he specifies the types of the tuples that are allowed inside this tuple space as arguments to the tuple space. By smartly using these type arguments the *Blossom* library can perform the same optimizations that the best Linda precompilers do. A more elaborate discussion on how to do this can be found in Chapter 7.

The *Blossom* programmer can specify a tuple space access pattern in the declaration of a tuple space. By doing so it gives *Blossom* valuable information on how to distribute the Linda tuple space over the available machines in case it runs in a distributed environment. If the programmer does not specify a pattern for a tuple space, it will get a default distribution. The introduction of tuple space access patterns is new to Linda and is described in Section 4.3.3.

Blossom also implements some of the features that are introduced in the literature. Among others *Blossom* includes multiple tuple spaces, and a synchronous operation to insert a tuple in a tuple space.

Finally, *Blossom* is built on top of an object-oriented communication library called *Roots*. *Roots* provides active messages and full thread support. It can outperform the traditional communication library Parallel Virtual Machine (PVM).

1.4 Features

The major contributions of this thesis presented in the previous section gloss over some of the important features in the *Blossom* system. While each feature may not itself be a contribution, it contributes to the creation of a new, useful parallel programming tool. In the following, we briefly describe some of the features of *Blossom* and its communication library *Roots*. Both pieces of software are freely available [51, 50].

1.4.1 *Blossom*

Library

- High performance without need for a precompiler.
  The *Blossom* parallel programming language is captured in a library. Traditionally, Linda uses a precompiler to be able to optimize Linda code.

- Type parameterized programming.
  Blossom makes extensive use of type parameterized programming. This is part of the reason why no separate precompiler is needed.

- Type-type and type-value conversions.
  Advanced type-type and type-value conversions are used to improve *Blossom*'s performance.

- No new syntax.
  Because *Blossom* is a library the user of *Blossom* does not have to learn new syntax, but only has to learn some classes, objects and member functions.

- Easy maintainable and extendable.
  A library is easier to maintain and extend than a precompiler.

Programming Model

- Multiple Tuple Spaces.
  Linda provides the user with one global tuple space whereas *Blossom* provides multiple local tuple spaces. A local tuple space can be shared among any number of processes.

- Ease of Use.
  Multiple tuple spaces make it easier for programmers to think and reason about their programs.

- Strongly Typed Tuple Spaces.
  A *Blossom* programmer needs to specify the types of the tuples that are allowed inside the tuple space. In return the programmer gets an error message from the compiler when he tries to insert or retract a tuple with an incorrect type from that tuple space.

- Field Access Patterns.
  A *Blossom* programmer also has to specify the way the fields of tuples may be accessed. In return the programmer gets an error message if he breaks his own specification.

Performance

- High Performance.
  Blossom performs all published optimizations that SCA Linda performs. Nevertheless *Blossom* has some optimizations that Linda is not known to have, namely replication of tuple spaces and asynchronous retrieval of tuples (with futures).

- Strongly Typed Tuple Spaces.
  The strongly typed tuple space allows *Blossom* to partition the tuple space into homogeneous parts. Homogenizing a heterogeneous tuple space mainly reduces the time of searching a tuple space for a matching tuple, it also somewhat reduces the time of inserting a tuple.

- Field Access Patterns.
  The field access pattern allows *Blossom* to optimize the storage of tuples. By providing field access patterns most tuple spaces can be stored in a hash table instead of an ordered list. Hash tables can be distributed over all the processors more easily, allowing better scalability for *Blossom* programs.

- Tuple Space Access Patterns.
  The *Blossom* programmer can specify a tuple space access pattern for a tuple space. By doing so it allows *Blossom* to optimize the location (on which processor) of the tuple space so that the number of communications can be reduced.

1.4.2 *Roots*

The features of the *Roots* communication library are given below.

Communication Network

The communication network provides a (virtually) fully connected network between all the processors that participate in a run. The connections are reliable and one-to-one.

- Portable.
  Roots runs on a multiprocessor machine and on a network of machines. The machines in the network may have different numbers of processors on board as long as they contain the same type of processor and run the same operating system.

- Communication Paradigms.
  Roots supports three kinds of communications: (*i*) shared memory communication, (*ii*) communication through Unix pipes, and (*iii*) interprocessor communication based

on the IP protocol. The user can specify which communication paradigms should be used for a specific run.

Active Messages

An active message contains an action that is performed upon receipt of the active message.

- Implicit Receive.
  Active messages are easier to use for a programmer than traditional messages. A programmer that uses traditional messages has to write code to explicitly send and receive messages. Traditional messages are therefore also called *two-sided communication messages*. A programmer that uses active messages only has to write code to explicitly send messages. The communication package implicitly receives the active messages. Active messages are therefore also called *one-sided communication messages*.

- Implicit Identifier.
  Distributed and parallel programs that make use of traditional message packages (for example PVM and MPI), usually provide all their messages with a unique identifier to be able to distinguish them at receipt. This is no longer needed because *Roots* will implicitly call the correct action belonging to the active message upon receipt. This gives *Roots*' active messages an implicit identifier.

- Implicit Message Recognition.
  An additional advantage for the *Roots* programmer is that he does not have to write the code to distinguish the messages, which is usually done with an ugly, non object-oriented `switch` statement. *Roots* will call the action that corresponds with the received message.

- Implicit Marshalling.
  In most communication packages the programmer has to explicitly marshal all the data. In *Roots* this is done implicitly by calling the constructor of a message object.

- Performance.
  One-sided communication is potentially faster than two-sided communication. A two-sided communication message is received twice: the first time implicitly by the communication package and the second time explicitly by the user program. Both receipts cause context switches, which means that one-sided communication saves the context switches for the explicit receive.

Threads

A thread of control (or simply thread) is a light weight process. Multiple threads can be run inside one process, just as multiple processes can be run on one machine. Whereas the latter allows multi-processing, the former allows multi-tasking.

- Threads and Communication.
  Most communication packages (like PVM and MPI-1) do not support threads. If the programmer still wants to use threads with these packages, all communication from threads in one process to another process have to go through the main thread of the first process. This is not only inefficient but also difficult for a programmer to use. However, *Roots* supports threads so that all threads communicate directly with other processes.

- Remote Spawning.
  Roots allows for threads to be spawned in an arbitrary process that participates in the run.

- Posix compliant.
  All the thread functionality that is defined in the Posix standard 1003.1c pthreads (part of [26]) are also part of *Roots*. However *Roots* has a slightly different interface because it is object oriented.

Synchronization Primitives

Synchronization Primitives are used to allow threads (and processes) to synchronize.

- Different Synchronization Primitives.
  Roots provides the following synchronization primitives: mutexes (for mutual exclusion), condition variables and semaphores.

- Posix compliant.
  All the synchronization primitives that are defined in the Posix standards 1003.1c pthreads and 1003.1b real-time extensions (both part of [26]) are also provided by *Roots*. The *Roots* interface is object oriented and as a consequence differs slightly from the interface provided by the Posix standards.

1.5 Contents

Chapter 2 describes the Linda programming language by introducing tuples, tuple spaces and operations to insert and retrieve tuples in and from the tuple space. Chapter 3 covers the enhancements to the Linda model that have been proposed in the literature. Chapter 4 introduces the new strongly typed programming language *Blossom*. The backbone of the language is based on the Linda model and some of the extensions that can also be found in the literature. Chapter 5 gives a high level description of *Blossom*'s architecture. The intention of this chapter is to provide the reader with an introduction to *Blossom*. Chapter 6 describes the active message communication library and shows some performance numbers for it. Both Chapters 5 and 6 form a introduction to the technical chapter that follows. Chapter 7 uncovers all the details of how *Blossom* (ab)uses C++'s templates in order to avoid writing a precompiler *and* to achieve good performance (similar to commercial Linda implementations that do use precompilers). Chapter 8 shows that the extra features present in *Blossom* allow the *Blossom* programmer to gain extra performance by changing the access pattern of tuple spaces. Chapter 9 reiterates the main points introduces during the thesis and gives directions for future research in this area.

Appendix A shows the syntax of tuple space declarations and *Blossom* variables. Appendix B contains all the experimental data on which Chapter 8 is based.

Chapter 2

Linda

Linda is a shared memory programming model. By superposing the Linda model on top of a sequential programming language, a parallel programming language is born. Linda can be superposed on any sequential programming language and as a result many Linda languages coexist.

For any sequential language *X*, a parallel language *X*-Linda exists. Popular *X*-Lindas include C-Linda, and Fortran-Linda. The Linda examples in this dissertation use C-Linda. So every time I talk about the Linda language, I actually mean the C-Linda programming language. No confusion can exist about the Linda model because there is only one Linda model (though in the next chapter we will see that there are many Linda dialects).

Section 2.1 provides an introduction to the Linda model with examples from the C-Linda language. Section 2.2 shows five elaborate examples of how Linda can be used.

2.1 The Linda Model

Linda is a shared memory programming model that was introduced in 1985 by Gelernter [18]. Although the Linda model was not in its final form, it contained the idea of a shared memory accessible by multiple processes. Linda's shared memory can be seen as a blackboard that processes can write to, read from, and erase portions of. The blackboard can contain a collection of data records, called *tuples*. Hence the shared memory is called the *tuple space*. What tuples are and what tuples look like is explained in Section 2.1.1. Two kind of tuples exist, namely *value tuples* and *query tuples*. Sections 2.1.2 and 2.1.3 explain the differences between the two kinds of tuples. Section 2.1.4 tells us what is needed for the two kinds of tuples to match. Section 2.1.5 shows how processes can insert value tuples in the tuple space. Sections 2.1.6 through 2.1.9 show how processes can get tuples from the tuple space. The matching of query tuples and value tuples is discussed in these sections. Finally Section 2.1.10 introduces how to create processes in Linda.

2.1.1 Tuples

A *tuple* is a record with an arbitrary number of data fields where every data field can be of an arbitrary type. Examples of tuples are `(3, 1)`, `('3', 1, "4")`, `("3.142")`, `(3.1416)` and `()`.

Every tuple has a *prototype* (or *type* for short). The afore mentioned tuples have the prototypes `int` × `int`, `char` × `int` × `string`, `string`, `float` and ϵ (the empty tuple type).

The fields of a tuple are ordered, which means that tuple `(1, 2)` differs from tuple `(2, 1)`. Their prototypes are however equivalent (namely `int` × `int`).

2.1.2 Value Tuples

The tuples we have just encountered are actually all *value tuples*. A value tuple is a tuple that only contains *value fields*. A field is called a value field if it contains a value.

In programming languages values are produced by expressions. Most programming provide the following three kinds of expressions: *(i)* constants, *(ii)* variables and *(iii)* compound expressions. The tuples in the previous section only had constants in their fields. Assuming that `i` is a variable, the tuple `("square", i, i*i)` contains a constant field followed by a variable field and finally a compound expression field.

Depending on the value of integer `i` the tuple has different contents. If `i` has the value 5 the above value tuple is equal to the tuple `("square", 5, 25)` that only consists of constant fields.

Usually tuples start with a constant field. Such a field is called a *tag* and is used to differentiate the tuple from tuples with the same prototype (here for example from tuple `("cube", i, i*i*i)`).

2.1.3 Query Tuples

A *query tuple* is a tuple that can contain value fields as well as query fields. Query tuples are used to query the tuple space for value tuples it contains. A *query field* is a field that contains a variable that can be updated. How the tuple space is queried is described in Sections 2.1.6 through 2.1.9. How query fields are updated is shown in Section 2.1.4.

A variable from the host language can be used as a constant value field as well as an updatable variable field. To distinguish between both meanings a variable that should be treated as an updatable variable is preceded by a question mark (`?`).

Assume that variable `i` is declared as an `int`, variable `c` is declared as a `char` and variable `f` is declared as a `float`. Examples of query tuples are `(3, ?i)`, `(c, ?i, "4")`, `("3.142")`, `(?f)` and `()`.

Query tuples have prototypes just like value tuples. The prototypes of the above query tuples are `int` × `int`, `char` × `int` × `string`, `string`, `float` and ϵ (the empty tuple type). This means that types of the updatable variables determine the prototype of a query tuple.[1]

In traditional Linda terminology a value field is known as an *actual* and a query field is known as a *formal*. The new terminology is introduced because we will encounter another query field in Chapter 4, which results in another meaning for the term formal. In order to prevent confusion we will use the new terminology. In traditional Linda terminology a query tuple is known as an *anti-tuple*. Changing only half the terminology did not seem appropriate.

2.1.4 Matching

A value tuple and a query tuple *are matching* when *(i)* their prototypes are equal, and *(ii)* all value fields in the query tuple have the same value as their corresponding value fields in the value tuple. If such is the case the query tuple can be *matched* with the value tuple, which means that the query fields of the query tuple are updated with the values of the corresponding value fields in the value tuple. Note the subtle difference between being matching and being matched. A query tuple can only be matched with a value tuple when both tuples are matching.

Let's have a look at an example. The following table contains several examples of matchings. The first column contains the query tuples, the second column contains the value tuples and the third column shows whether the tuples from columns one and two match. Assume variable `i` to be an integer.

QUERY TUPLE	VALUE TUPLE	MATCHING
`(3.14159)`	`(3.14159)`	Yes
`("square", 5, ?i)`	`("square", 5, 25)`	Yes, `i == 25`
`(?i)`	`(3.14159)`	No, different prototype
`("square", 6, ?i)`	`("square", 5, 25)`	No, different value fields

Now that the formal foundation has been introduced we can have a close look at the Linda model. The Linda model mainly consists of six operations. They will be introduced in the next six sections.

2.1.5 The Operation `out`

The operation `out` inserts a value tuple *out* of the program into the tuple space (see Figure 2.1). The operation `out` has a value tuple as its argument.[2]

```
out("square", 1, 1);
```

[1] As a consequence the Linda host language should be a programming language that supports types in the sense that each variable has a type associated with it.

[2] The attentive reader will see that this is not completely correct, because only one pair of parentheses is needed. The correct formulation is that the operation `out` takes an arbitrary number of value fields as its arguments, where the value fields together form a value tuple.

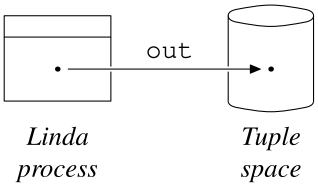


Figure 2.1: The Linda operation `out`
The operation `out` inserts a value tuple *out* of the program into the tuple space.

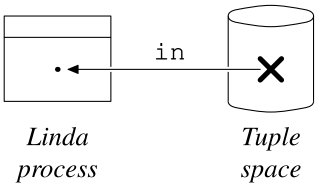


Figure 2.2: The Linda operation `in`
The operation `in` retracts a value tuple from the tuple space *in*to the program by specifying a query tuple. The matching value tuple is removed from the tuple space.

This C++-Linda statement inserts the value tuple `("square", 1, 1)` in the tuple space. Just as in normal programming practice variables can be used.

```
for (int i = 2; i < 5; i++)
   out("square", i, i*i);
```

This piece of code will insert three value tuples `("square", 2, 4)`, `("square", 3, 9)`, and `("square", 4, 16)` in the tuple space.

2.1.6 The Operation **`in`**

Operation `in` takes a query tuple as its argument and tries to find a matching value tuple in the tuple space. Once a matching value tuple is found the query tuple is matched with the value tuple and the value tuple is removed from the tuple space (see Figure 2.2). If multiple value tuples match the query tuple, an arbitrary value tuple is removed from the tuple space and matched with the query tuple.

The program will block if no matching value tuple is available in the tuple space. The operation `in` will then wait until another process inserts a matching value tuple.

Let's start with a little example.

```
in ("square", 1, 1);
```

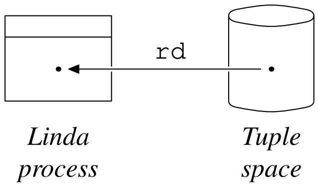


Figure 2.3: The Linda operation `rd`

The operation `rd` *read*s a value tuple from the tuple space into the program by specifying a query tuple. The matching value tuple remains in the tuple space.

Since the query tuple `("square", 1, 1)` is a value tuple as well, this command will retrieve this tuple from the tuple space.

Of course querying the tuple space becomes more interesting when the query tuple contains query fields. Retrieving a known value tuple from the tuple space is only interesting for synchronization with the process that inserted the tuple in the tuple space.

```
int sq;
in("square", 4, ?sq);
```

The `in` operation matches any tuple with three fields, the first being a string with value `"square"`, the second an integer with value 4 and the third an integer with any value. The variable (`sq`) in the query field (`?sq`) will have value `16` after retrieving the tuple `("square", 4, 16)` from the tuple space.

By using query fields processes can exchange information by inserting tuples in and retrieving tuples from the tuple space.

2.1.7 The Operation `rd`

Operation `rd` reads a tuple from the tuple space without removing it from the tuple space. It has a close resemblance to the operation `in`. Operation `rd` will block as well if no matching tuple can be found. It can use updatable variables just like the `in` operation.

An `in` of a tuple immediately[3] followed by an `out` of the same tuple is semantically equivalent to a `rd` of that tuple.

2.1.8 The Operation `inp`

Operation `inp` is a non-blocking variant of `in`, i.e., it will never wait (block) for an other process to `out` a matching tuple. If a matching tuple is found in the tuple space the `inp` operation returns `true` and behaves as if it were an `in` operation. If no matching tuple is found in the tuple space the `inp` operation returns `false` and does not alter the tuple space. Updatable variables will only change value if the operation `inp` returns `true`.

[3]If the operations `in` and `out` are not performed in one atomic action, the non-blocking operations from Sections 2.1.8 and 2.1.9 can exhibit different behavior.

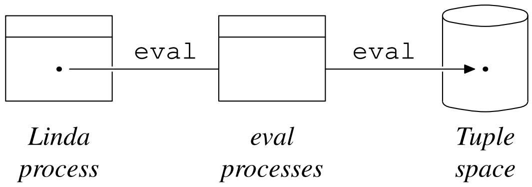


Figure 2.4: The Linda operation `eval`

The operation `eval` *eval*uates a tuple using one or more processes. The tuple is inserted in the tuple space after it is evaluated.

2.1.9 The Operation `rdp`

The operation `rdp` is a non-blocking variant of `rd`. The operation is non-blocking, it will return independent of whether a matching tuple is found in the tuple space or not. If a matching tuple is found the `rdp` operation returns `true` and initializes any query fields. The matching tuple that is found will remain in the tuple space as was the case with the `rd` operation. If no matching tuple is found the `rdp` operation returns `false` and no query field is initialized.

In Linda terminology the non-blocking operations are called predicate operations, hence the suffix `p`.

2.1.10 The Operation `eval`

Linda provides a way to spawn processes that nicely fits in the Linda model. The `eval` operation spawns processes that calculate the values of the fields and eventually inserts the calculated tuple in the tuple space.

Assume the following function is defined.

```
int sqr(int x) {
   return x*x;
}
```

Then the following command

```
eval("square", 5, sqr(5));
```

spawns three processes, each evaluating one of the arguments in parallel with the spawning process. The first process evaluates `"square"`, the second process evaluates `5`, and the third process evaluates `sqr(5)`. When all processes finish their calculations the resulting tuple is inserted in the tuple space (the tuple `("square", 5, 25)` in this case). Examples of how to use `eval` in Linda can be seen in the examples described in Sections 2.2.3, 2.2.4 and 2.2.5.

2.1.11 General Comments

Now that the Linda model has been introduced, it is time for some general comments on the model.

simple The Linda model is simple. An arbitrary sequential programming language can be turned into a parallel language by introducing the six Linda operations.

comprehensive Because the Linda model is simple and only introduces six operations it is comprehensive. Newcomers to the model can start exploiting its full potential within a day.

easy to program It is easy to program with the Linda model. For instance, a sending process does not have to know the receiving process and vice versa.

powerful The Linda model is powerful. Communication between two process only takes two operations: an `out` and an `in` (or `rd`) operation.

However, because the Linda model is so simple it may be hard to get high performance.

2.2 Examples

The following examples illustrate the usage of the concepts introduced in the previous section.

2.2.1 Shared Queue

The first example shows how to implement a shared queue in Linda. In a sequential programming language queues are often implemented using pointers. Compare this with a queue for a counter in a shop. Everybody who joins the queue only has to remember the person directly preceding him. In software terms you have a pointer to the person in front of you.

Parallel languages and pointers do not combine very well. A pointer is usually implemented as an address within one machine and therefore only has a meaning on one machine. On another machine the pointer will probably point to something wrong.

Pointers to Linda tuples do not make much sense either. Linda decides where (on what machine and on what memory address) to put a tuple. A programmer cannot inquire this address; it is hidden in the implementation. It can be even worse, the implementation may decide to move tuples around.

How can dynamic data structures be implemented in Linda? Can a 'pointer' construction be created in Linda? A pointer is mainly used to get a unique object. In Linda it is possible to get a unique tuple by providing tuples with a unique identification, let's say a number. For example if we are looking for the twelfth square, we inquire the tuple space with `rd("square", 12, ?sq)`. We now get a unique tuple from the tuple space because we know that only one value tuple in the tuple space will match the asked query tuple.

By introducing an indirection, here the unique identification, we have obtained a 'pointer' construction in Linda. Such a Linda pointer can be described more appropriately as a Linda `index`.

In the following code example only the tuples with tags `"queue head"` and `"queue tail"` contain Linda indices. The queue elements with tag `"queue data"` do not need an index to the next element, because that index is their own index incremented by one. This is possible because a queue is a relatively regular dynamic structure.

```
void queue_create() {
   out("queue head", 0);
   out("queue tail", 0);
}

void queue_put(int value) {
   int index;
   in("queue head", ?index);
   out("queue head", index+1);
   out("queue data", index, value);
}

int queue_get() {
   int index;
   int value;
   in("queue tail", ?index);
   out("queue tail", index+1);
   in("queue data", index, ?value);
   return value;
}
```

First of all notice that the shared queue uses two counters. To distinguish the two counters the tuples containing the counters get different tags, namely `"queue head"` and `"queue tail"`.

The access to a shared queue has to be coordinated. If several processes change the queue simultaneously the queue may be corrupted. Thus the vital operations (also known as the *critical section*) should be allowed by one process at a time. This can be done by putting *semaphores* around these operations. When a semaphore is obtained by one process it cannot be obtained by another process before the process that has the semaphore releases it.

Semaphore behavior can easily be implemented in Linda. A tuple that is present in the tuple space indicates that the semaphore is available and can be fetched by a process. When the tuple is not present all other processes have to wait until the semaphore is put back in the tuple space before they can obtain the semaphore and start their critical section.

In the shared queue example two semaphores are present, namely the tuples with tags `"queue head"` and `"queue tail"`. They regulate the access to the critical sections of the `queue_put` and `queue_get` operations respectively.

The critical sections in the shared queue example are relatively simple. Let's have a look at the operation `queue_put` first. In between the removal and the reinsertion of the (semaphore) tuple with tag `"queue head"` the only thing that happens is that the counter inside this tuple is incremented by one. The only thing this semaphore does is to make sure that every `queue_put` operation receives a unique index in the shared queue. Similarly the tuple with tag `"queue tail"` makes sure that every location in the queue is only read once.

It is interesting to see what happens when processes try to `queue_get` objects from the shared queue that are not yet available. Every `queue_get` operations gets a unique index and will next block on the operation `in("queue data", index, ?value)` until a matching tuple is inserted by an operation `queue_put`.

2.2.2 Shared Stack

A stack is slightly more difficult than a queue. A queue has FIFO (**f**irst **in** **f**irst **o**ut) semantics and a stack has LIFO (**l**ast **in** **f**irst **o**ut) semantics. In other words objects leave the queue on the opposite side from where they enter the queue, and objects leave the stack on the same side as where they enter the stack.

```
void stack_create() {
   out("stack top", 0);
}

void stack_push(int data) {
   int top;
   in("stack top", ?top);
   out("stack top", top+1);
   out("stack data", top, data);
   out("stack data available");
}

int stack_pop() {
   int top;
   int data;
   in("stack data available");
   in("stack top", ?top);
   in("stack data", top-1, ?data);
   out("stack top", top-1);
   return data;
}
```

We see in this example that both operations `stack_push` and `stack_pop` have to fight over one semaphore tuple with tag `"stack top"`. Furthermore we see that the critical section in the operation `stack_pop` contains the retrieval of a `"stack data"` tuple. This is necessary because indices in a stack are reused. If the retrieval of this tuple is not performed inside the critical section another process might already have called operation `stack_push` again so that two tuples with tag `"stack data"` and the same index are in the tuple space. Whenever this happens the shared stack can no longer guarantee correct semantics.

The tuples `("stack data available")` almost implement a so-called *counting semaphore*. A counting semaphore is more or less a number that indicates how many processes are allowed to enter a critical section. Every time a process enters a critical section it lowers the number by one to indicate that one process less is allowed access to a critical section. When the process has finished its critical section it increments the semaphore again. You can say that a regular semaphore is a counting semaphore with an initial value of one.

Here however, the number of processes that are allowed to perform the `stack_pop` operation is dependent on the number of objects in the shared stack. The counting semaphore is incremented by the operation `stack_push` and decremented by the operation `stack_pop`. If no objects are in the stack anymore the operation `stack_pop` will block until a new object is inserted in the stack again.

2.2.3 Dining Philosophers

The dining philosophers example is often used to see how expressive a parallel language is. Five philosophers are sitting around a round table. In front of every philosopher is a plate. In between two philosophers is one fork (or one chopstick). The philosophers alternate thinking with eating. In order to eat a philosopher needs the forks to the left and to the right of his plate. If one fork is unavailable he continues thinking. While thinking the philosophers put the forks on the table. From the example description we can conclude that at least half of the philosophers will be thinking, or at most half of the philosophers will be eating.

In an implementation of this problem in a parallel language the ultimate solution would be that a philosopher tries to take both forks in one atomic action. Depending on the result of this atomic action he starts eating or remains thinking. If the grabbing of the forks is not performed in one atomic action, it may turn out that every philosopher grabbed the fork to the left of his plate and is waiting for the fork to the right. The situation where every philosopher is waiting for the philosopher to his right to drop his left fork is called a *deadlock*.

```
bool grab_forks(int left, int right) {
   bool result = false;
   in("grab semaphore");
   if (rdp("fork", left) && rdp("fork", right)) {
      in("fork", left);
      in("fork", right);
      result = true;
   }
   out("grab semaphore");
   return result;
}
```

```
void drop_forks(int left, int right) {
   out("fork", left);
   out("fork", right):
}

int philosopher(int no) {
   const int left = no;
   const int right = (no+1) % NUM;
   while (true) {
      think();
      if (grab_forks(left, right)) {
         eat();
         drop_forks(left, right);
      }
   }
   return no;
}

void main() {
   for (int i = 0; i < NUM; i++) {
      out("fork", i);
      eval("philosopher", philosopher(i));
   }
   out("grab semaphore");
}
```

Note the use of the `rdp` operations in the function `grab_forks` to prevent the philosophers to block while getting their forks and to prevent the program from deadlock.

Linda provides a way to combine the getting of both forks in one atomic action as can be seen in function `grab_forks`. The semaphore `("grab semaphore")` combines the grabbing of both forks in one atomic action.

In this example the operation `eval` is only used to create new threads. The tuple creating ability of the operation `eval` is not used because the calculation never terminates. Even if all the philosophers would eat a maximum number of times so the calculation terminates, the resulting `"philosophers"` tuples are of no interest. This gives the impression that the operation `eval` combines two things (creating threads and inserting a tuple in the tuple space) that do not need to be combined.

2.2.4 Matrix Multiplication

Matrix multiplication is often used as a benchmark to compare the speed of parallel and distributed languages. This section shows a simple distributed implementation.

Two matrices (`A` and `B`) are multiplied and the result is stored in matrix `C`. If the multiplication is not allowed due to wrong dimensions of matrices `A` and `B` this Linda program will block. If the dimensions of matrices `A` and `B` are correct, every element in matrix `C` is calculated in a separate process. The `eval` statement nicely creates a process to calculate the value of one element in matrix C. The process will eventually insert the calculated tuple in the tuple space. The matrix multiplication uses mn processes to calculate a matrix "C" of size $m \times n$. This is rather much overhead for a matrix multiplication.

```
double element(int row, int col, int size) {
   double c_value = 0.0;
   for (int i=0; i < size; i++) {
      double a_value;
      rd("matrix element", "A", row, i, ?a_value);
      double b_value;
      rd("matrix element", "B", i, col, ?b_value);
      c_value += a_value*b_value;
   }
   return c_value;
}

void matrix_multiplication() {
   int rows;
   int cols;
   int size;
   rd("matrix size", "A", ?rows, ?size);
   rd("matrix size", "B", size, ?cols);
   out("matrix size", "C", rows, cols);
   for (int row = 0; row < rows; row++)
      for (int col = 0; col < cols; col++)
         eval("matrix element", "C", row, col,
              element(row, col, size));
}
```

The tuples in the matrix multiplication example have two tags, one to indicate that they are matrix related ("matrix size" or "matrix element") and one to indicate a specific matrix ("A", "B" or "C").

2.2.5 Primes

Calculating primes is another popular example. Here is an inefficient implementation. All numbers smaller than a certain upper bound LIMIT are investigated on whether they are a prime. The function is_prime decides whether a number (the first argument) is a prime or not by trying to divide the number by all smaller primes (passed in the tuple space in the second argument).

```
bool is_prime(int number) {
   int limit = sqrt(number);
   for (int i = 2; i <= limit; i++) {
      rd("prime", i, ?prime);
      if (prime && (number%i == 0))
         return false;
```

```
    }
    return true;
}

void main() {
    bool prime;
    for (int i = 2; i < LIMIT; i++)
        eval("prime", i, is_prime(i));
    for (int i = 2; i < LIMIT; i++) {
        rd("prime", i, ?prime);
        if (prime)
            cout << i << endl;
    }
}
```

Chapter 3

Enhancements

The simple and elegant Linda model has attracted much attention by researchers. Several researchers found deficiencies in Linda and proposed enhancements to overcome these deficiencies. As a result several Linda dialects exist today.

This chapter is an overview of enhancements encountered in both the literature and Linda dialects. The enhancements are arranged in three categories, namely *model enhancements* (Section 3.1), *efficiency enhancements* (Section 3.2) and *paradigm enhancements* (Section 3.3). Not all of the enhancements can be put into exactly one category.

For completeness sake a fourth category, *software engineering enhancements* (Section 3.4), will be introduced. This category will be filled in the next chapter where we see that *Blossom* provides several software engineering benefits.

The line of argument for the enhancements will closely follow how they were presented in literature. Nearly all of the enhancements are introduced after stating a Linda deficiency. So what we see in the following sections is a deficiency description followed by the proposed enhancement(s).

3.1 Model Enhancements

Model enhancements enhance deficiencies of the Linda model.

3.1.1 Multiple Tuple Spaces [19]

Deficiency: Abstraction The pure Linda model provides one global tuple space. The tuple space is accessible by every part of the program and thus all tuples inside the tuple space are visible throughout the whole program. When a certain combination of a tuple prototype with a tag is used more than once, the tuples with this prototype/tag combination may collide with each other and introduce unwanted behavior of the program. As a consequence the programmer must know exactly what combinations of tuple prototype and tag(s) are already in use. In a sense the combination of prototype and tag(s) acts like a shared variable listening to the name of the tag. The global tuple space idea entails that these 'variables' are all global, which leads us back to this bad practice known from old programming languages. We saw something similar with the global variables in old programming languages, where the programmer had to know all the variable names that were already in use.

This problem becomes manifest for instance when libraries are used, because a library should provide a list of all used combinations of prototype and tag(s). If two libraries both provide the same combination of prototype and tag(s), the libraries cannot be used together by one program. One of the libraries has to be changed which is not always possible.

Enhancement: Multiple Tuple Spaces By providing multiple tuple spaces the abstraction deficiency can be solved. The contents of two tuple spaces will never collide. By letting a library use its own local tuple spaces a programmer never has to worry about what combinations of prototype and tag(s) are used inside the library.

One could argue that the names of the tuple spaces used in the library should now be known by the programmer. This would be true if the tuple spaces would be global objects, however they are local objects[1] so their names cannot collide with variable names introduced by programmers.

The only thing a programmer has to know about a library is its interface. This interface may contain types, variables or functions. All types, variables and functions that are part of the interface should be known to the programmer for the simple reason that he wants to use the library.

The multiple tuple space enhancement makes it possible to overcome several Linda efficiency deficiencies (see Sections 3.2.6 and 3.2.7). Gelernter describes other reasons to introduce multiple tuple spaces as well [19].

3.1.2 Array Fields [44]

Deficiency: Variable sized fields Most programming languages provide arrays. An array is a data structure that can contain multiple objects of one type. The definition of an array also contains the maximum number of objects that fit into the array. When an array is passed as an argument usually only a pointer to the array is passed. Besides this pointer no further information is passed. So the length of the array is not known to the function. Usually this is not a problem because the function uses the same data space as the program that calls it. However in parallel and distributed programming this is a problem. The function may be executed in a different process, that uses a different data space. A pointer that points to a different data space is not very useful. The contents needs to be in the data space of the process that executes the function. In other words the contents of the array should be copied to the process that will execute the function. To copy the contents of the array the size of the array is important. Without the size the number of elements to copy is unknown.

[1] This could be in the form of static (C/C++) variables.

Enhancement: Array support When a Linda operation is called that inserts a tuple in or retracts a tuple from a tuple space, the contents of this tuple should be sent to the process where the tuple space is located. If arrays may be part of a tuple, the array should be passed. In that case a mechanism has to be found to let the programmer indicate how many objects are inside the array.

The commercial product SCA Linda from Scientific Computing Associates uses a mechanism where the programmer has to indicate how many elements the array contains. This notation not only provides the size of the array but also indicates that this field is an array. The notation contains of a colon (`:`) followed by a number that indicates the size of the array.

```
int a[] = {0, 1, 2, 3, 4, 5, 6};
int b[7];
out(a:4);
rd(?b:4);
in(b:4);
```

The above `out` operation inserts an array of four integers in the tuple space. These are the first four elements of array `a`. The following `rd` operation retrieves an array of any four integers from the tuple space in array `b`. When this operation succeeds the first four integers of array `b` are instantiated with the matching tuple. The final `in` operation removes the just found tuple with a four element array from the tuple space.

3.1.3 Wild Card Fields [44]

Deficiency: No interest in matching value Sometimes the programmer is not interested in all fields of the tuples inside the tuple space. Assume for example that the tuple space contains `"employee"` tuples with four fields, a tag field `"employee"`, an employee identification number, the employee's name and the employee's salary. A programmer who has to calculate the average salary of the company is not interested in the names of the employees. In standard Linda a variable field for the name of the employer has to be introduced to be able to read `"employee"` tuples from the tuple space. This variable will not be used by the program.

```
float sum_salary = 0;
for (int id = 0; id < MAX_ID; id++) {
   string name;
   float salary;
   rd("employee", id, ?name, ?salary);
   sum_salary += salary;
}
cout << "average salary: " << sum_salary / MAX_ID << endl;
```

As can be seen the variable `name` has to be declared, read from the tuple space but it is never used.

Enhancement: Wild card variables The variable `name` cannot be omitted from the operation `rd`. It contains information that is needed to the matching of the query tuple. Remember from Section 2.1.4 that both the number of fields and the types of the fields are important for the matching. By removing the field completely the number of fields would change. Only leaving the question mark (`?`) will omit the type of the field and make matching with the intended tuples impossible.

The solution in SCA Linda [44] is to proceed the question mark with the type of the field, in this case `?string`.

```
float sum_salary = 0;
for (int id = 0; id < MAX_ID; id++) {
   float salary;
   rd("employee", id, ?string, ?salary);
   sum_salary += salary;
}
cout << "average salary: " << sum_salary / MAX_ID << endl;
```

This enhancement is partly an efficiency enhancement as well. However the speed improvement of wild card variables is generally negligible. Though wild card variables are used, still two communications are needed for retrieve operations. The second communication is smaller because no matching value for the wild card variable is needed. Even in case the retrieving tuple only contains wild card variables the second communication is needed to indicate that a matching tuple is found and the retrieve operation can proceed.

3.1.4 Synchronous `out` [44]

Deficiency: Timing The usage of the predicate operations can very easily introduce timing problems [44]. This is due to the fact that most Linda flavors implement the operation `out` asynchronously (for example SCA Linda). Consider the following example.

```
void master() {
   int i;
   for (i = 0; i < TASKS; i++)
      out("task", i);
   out("tasks outed");
   ...
}

void slave() {
   int i;
   rd("tasks outed");
   while (inp("task", ?i))
      do_task(i);
}
```

The worker puts tasks in the tuple space that should be performed by the slaves. The slaves wait until the master has put all tasks in the tuple space indicated by the tuple `("tasks` `outed")`. However the `"task"` tuples are outed but may not yet have arrived in the tuple space because the operation `out` is asynchronous. This means that there exist interleavings of the processes such that not all the tasks are performed.

Enhancement: Synchronous `out` Two solutions are proposed in the literature. The first solution [44] is that the programmer should be aware that this timing deficiency can occur when he is using the predicate operations. He has to prevent this timing deficiency from occurring. The way he should do this is by introducing counters and/or semaphores. This works as long as the total number of tasks is known beforehand. If the number of tasks is unknown the programmer has to implement something like busy waiting. However when can you be sure that all tasks have arrived in the tuple space? This is an unacceptable solution to the timing deficiency.

A far easier way to solve this deficiency is by changing the semantics of the `out` operation so that it becomes synchronous, but this has performance disadvantages.

3.2 Efficiency Enhancements

Efficiency enhancements allow the programmer to improve the speed of Linda programs. Efficiency enhancements vary from making the basic operations faster, removing bottlenecks, and improving parallel performance by reducing communication.

3.2.1 Homogenization of the Tuple Space [38]

Deficiency: Inefficient Search One of the first concerns with Linda is that it is not efficient. Querying a tuple space with a query tuple consists of two comparisons. The first one is the comparison on the prototype of the tuple. If this comparison succeeds, the tuples can be compared on their contents. If the tuple space contains many tuples, the overhead of the first comparison can become quite large.

Enhancement: Partition into Homogeneous Tuple Spaces The solution to this deficiency is to partition the heterogeneous tuple space into several homogeneous tuple spaces. A query operation now only has to look in the tuple space partition that contains the tuples with the same prototype as the query tuple. Once this partition is found the comparison of the tuple prototype is not needed anymore.

This solution can be incorporated in several ways.

- At run time
  Before inserting and retrieving tuples in and from the tuple space the correct partition has to be found. If the correct partition cannot be found, a new partition has to be created. Once the correct partition was found or created the operation can be performed on that part of the tuple space.

- At compile time
  The prototypes of the tuples are all known at compile time. This information can be used to partition the heterogeneous tuple space in homogeneous parts at compile time. This results in an even bigger improvement in performance. For every tuple prototype a separate homogeneous partition of the tuple space is created at compile time. And since the tuple prototype is already known at compile time the operation can immediately be performed on the specific homogeneous partition. This means that the comparison of tuple prototypes is now completely performed at compile time and not at run time anymore. Compilation takes longer; execution of the compiled Linda program takes shorter.

By examining all tuples, further improvements can be made. Tuples with different tags will never match, though their prototype may be the same. Introducing separate partitions for tuples with different tags is thus a good idea. By making these partitions, tags (fields that are always constant) can be removed from the tuples. All these partitions reduce the average number of comparisons, needed to find a matching tuple. This partitioning of the tuple space based on tuple tags can be done at both run time and compile time.

Parallel programming is a discipline where execution speed is most important. Partitioning the heterogeneous tuple space into homogeneous parts at compile time results in faster programs than partitioning at run time. That makes the partitioning at compile time preferable for parallel programming. In traditional Lindas the partitions of the tuple space are detected by a precompiler [13].

3.2.2 Distribution of the Tuple Space [38]

Deficiency: Scalability Another problem is the wide spread misunderstanding that Linda is not scalable. What does this mean? In parallel programming you want to be able to improve the execution time of a program by adding more processors. Ideally, when utilizing n machines, the program has to run in $1/n$ of the time of the sequential program. This linear speed up is not always easily achievable because of the overhead of the communication. However the execution time of a program — according to this idea — should continue to decrease as more processors are added. This will only be the case if the program is scalable.

Using central resources usually leads to implementations that are not scalable. Adding more processors results in more communication to the processor with the central resource and this processor will most likely become a bottleneck. More and more processes will be waiting for the central resource. Linda's global tuple space looks like a central resource, which would make Linda not scalable.

Enhancement: Distribute the Tuple Space The solution to this deficiency lies in removing the source of the problem, which is the centralized tuple space. By decentralizing or distributing the tuple space over all the processes the tuple space could become scalable. However how should the tuple space be distributed over all processors?

Replicating the tuple space on every processor looks attractive: all read operations can be performed locally. However if a tuple is inserted in the tuple space on one of the pro-

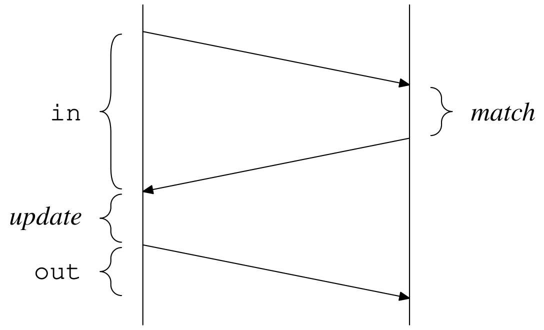


Figure 3.1: Sequence of `in`, update, `out`.

cessors it should appear in the tuple spaces of the other processors as well. If a tuple is removed from the tuple space on one processor, the removal should be performed in all tuple spaces on other processors as well. The communication between all the processes is not scalable either! Every update of the tuple space has to be processed by every tuple space replicate, which makes the distributed, replicated tuple space centralized again (only for fewer operations than before).

Distributing the tuple space over all processors is an alternative idea. You could think of a scheme where all tuples are stored locally. However, this means that when a tuple must be retrieved all tuple spaces must be searched, until a matching tuple is found. Not only should the tuple space be distributed over all processors but also every process should know where (on what processor) to store and retrieve a specific tuple.

This is only possible if more information is available. One solution works as follows. Remember that the tuples that are inserted in the tuple space should contain values in every field. If all the query tuples of a certain prototype always contain values in certain fields, the values of these fields can be used to decide on what processor to store tuples and on what processor to retrieve tuples (of specific prototype). The values in these tuples can be used to “calculate” the processor at which to store or retrieve the tuple.

A precompiler can decide what fields always contain values. These values can be used to decide on which processor to store or retrieve a tuple.

3.2.3 The Operation `in`-update-`out` [38]

Deficiency: `in`, update, `out` sequence In Linda programs the statement sequence `in`, update, `out` occurs quite often. It is comparable to updating a variable in any programming language: get the value of a variable, calculate a new value and store the new value in the variable.

In Linda this `in`-update-`out` sequence takes three communications (see Figure 3.1). The first communication is to request a specific tuple, the second communication sends the requested tuple back to the requesting process, the requesting process updates the tuple and

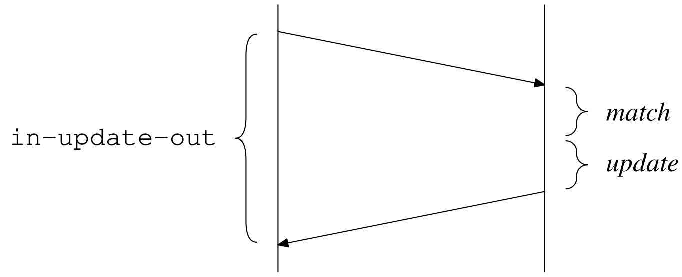


Figure 3.2: Operation `in-update-out`.

the third communication is to insert the updated tuple back in the tuple space again. Three communications is much for something as easy as incrementing one field of a tuple.

Enhancement: Combined `in`-update-`out` operation By combining the three parts of the sequence (the `in`, the update and the `out`) only two communications are needed (see Figure 3.2). This can be achieved by sending the update function with the query tuple to the remote process. When a tuple that matches the query tuple is found, the update function is performed on this matching tuple. Depending on the semantics of the combined `in`-update-`out` operation a copy of the matching tuple before or after the update is returned.

Replacing an `in`-update-`out` sequence by an `in-update-out` operation can be easily done by a precompiler [38].

It is not possible to lower the number of communications to one if the programmer is not interested in the updated tuple after the `out` statement. The combined `in`-update-`out` operation is blocking since it contains the blocking operation `in`. The process that issues the `in`-update-`out` operation still needs to be notified that it can proceed, i.e., the `in` operation has found a matching tuple.

3.2.4 Prefetching [13]

Deficiency: Waiting for `in` and `rd` When querying the tuple space with the blocking operations `in` and `rd` it is possible that the querying process has to wait for another process to insert a matching tuple in the tuple space. During this waiting time the amount of parallelism is reduced by one process. Waiting for a matching tuple can thus waste many processor cycles.

Enhancement: Prefetching The program can be redesigned to make use of the operations `inp` and `rdp`. This does not necessarily improve performance, because the redesigned program usually needs more tries to get a matching tuple, which means extra communication. Since communication is relatively expensive, another solution has to be found.

Assume that it is possible to indicate that in the (near) future a specific tuple is going to be used. In that case a so-called *prefetch* message can be sent to the tuple space. As soon as the prefetch message is sent, the sending process can proceed with work that it still can do before the prefetched tuple is needed. Ideally the prefetched tuple is found and returned before the prefetching process finishes this work. If the prefetched tuple is present, the process can proceed. If no matching tuple is found in time the prefetching process still has to wait for the return of a matching tuple. In this case it has at least filled part of the waiting time by some work.

The following code fragment illustrates prefetching.

```
in(task descriptor);
while(the task descriptor is not a ''poison pill'') {
    do the task ...
    in(task descriptor);
}
```

Right after the condition in the `while` holds, the tuple requested in the `in` operation on line 4 can already be prefetched. Whenever the program reaches the `in` operation on line 4 the tuple is probably already present, so that the program can immediately continue.

It has been proposed [13] to use a precompiler to recognize the possibilities of prefetching. However it is difficult to decide when prefetching is indeed profitable and when it is not profitable. Prefetching is a bit more expensive in time than traditional querying and is thus only interesting if sufficient work can be done in between the prefetching and the update. The decision to use prefetching is further influenced by the knowledge of when a matching tuple will be available in the tuple space. If the matching tuple is not in the tuple space when doing the normal `in`, prefetching only introduces extra overhead. The decision to use or not use prefetching is difficult for a compiler to make. That is probably why it is not implemented in any of the precompiler Lindas.

3.2.5 Tuple Access Patterns [11]

Deficiency: The general Linda communication pattern can be slow When comparing parallel programming in a shared memory model with parallel programming in a distributed memory model several observations can be made. First, shared memory models are easier to program than distributed memory models. Second, distributed memory models can achieve better speed ups than shared memory models. Why is the latter the case?

In a distributed memory model the processors have to communicate via message passing. Message passing is a more basic concept than shared memory communication. Many shared memory models are implemented using message passing. Writing your own communication messages gives the programmer communication patterns that are more suitable to the problem. Shared memory models only use one communication pattern, which gives them a performance handicap.

Enhancement: Access patterns per tuple tag The idea is that by allowing multiple communication patterns in shared memory models, the performance of these shared memory models can be pushed towards distributed memory models [7]. By closely examining communication patterns in distributed memory models nine communication patterns have been distinguished.

An example of a communication pattern is the so-called producer/consumer pattern. In the producer/consumer pattern a producer produces data for a consumer. The data that is produced by the producer is immediately sent to the consumer. In standard Linda where this optimization is not available this takes usually three communications; one for the producer to put the data in the tuple space and two for the consumer to take the date out of the tuple space. And if the operation `out` is synchronous it even takes four communications.

By specifying what communication pattern to use for a given tuple tag, the number of communications can be reduced significantly. Five of the nine communication patterns have also been applied to Linda [11].

3.2.6 The Operation `collect` [9]

Deficiency: Distributed Addition The problem of the distributed addition deficiency is to perform an associative, binary operation on all tuples of a specific type. Every application of the operation reduces the number of tuples (of the specified prototype) by one. An example of an associative, binary operation is addition, hence the name distributed addition deficiency. The application of the operation has to take place $n - 1$ times to accomplish a reduction from n tuples to 1 tuple. This application should be as parallel as possible (see Figure 3.3) independent of the number of tuples. The fastest way this problem can be solved in a distributed fashion is in $\mathcal{O}(\log n)$, where in plain Linda this reduction takes $n - 1$ operations. This distributed $\mathcal{O}(\log n)$ solution is also known as recursive doubling (see Figure 3.3). If the number of tuples (n) is unknown it is very difficult to reach this lower bound $\mathcal{O}(\log n)$.

Enhancement: Operation `collect` The article [9] that tries to solve this deficiency discusses several solutions. Several new operations are proposed. The one that the authors prefers is the `collect` operation. This operation is performed on a tuple space (the first argument) and takes as the following arguments the fields of a query tuple. All the tuples that match the specified prototype in the tuple space that the operation is performed on are moved to the tuple space specified in the first argument. The number of tuples that the operation puts in the argument tuple space is returned by the operation. With this number it is possible to solve the addition optimally in a distributed fashion.

3.2.7 The Operation `copy-collect` [42]

Deficiency: Multiple `rd` The multiple `rd` problem has to do with the non-deterministic behavior of the `rd` operation in Linda. Performing the operation `rd` multiple times after each other may result in the same tuple being retrieved over and over again. Actually, most Lindas implement the operation `rd` this way, because it is simply faster. A disadvantage of this choice is that the operation `rd` becomes unfair.

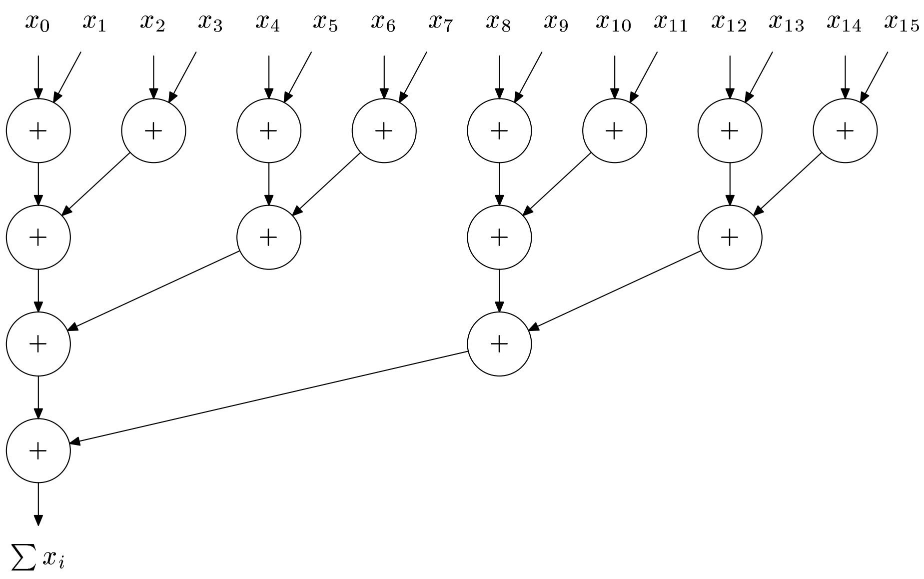


Figure 3.3: Recursive doubling.
The circles form the associative, binary operations. Circles in one column are computed on one processor.

A solution within the Linda model would be to remove all matching tuples and put them in a different tuple space. When no matching tuples are found anymore, all tuples inside the different tuple space are inserted back in the original tuple space again. But if many tuples match the prototype, it will take a long time before the tuple space has obtained its original content again. Another solution is needed.

Enhancement: Operation `copy-collect` This solution should be able to move several tuples together without removing them from the tuple space. A similar operation as the `collect` operation seems obvious. In contrast with the `collect` operation that removes the tuples from the tuple space, the similar operation `copy-collect` does not remove the tuples from the tuple space. The operation `copy-collect` forms the solution for the multiple `rd` problem. The `copy-collect` operation returns the number of copied tuples as well.

3.3 Paradigm Enhancements

Linda is itself a paradigm that enhances an imperative or an object-oriented programming language. Linda can however be extended with other paradigms as well. This section mainly gives an overview of some Linda flavors as they appear in programming languages.

3.3.1 Persistence [8]

Deficiency: Tuple space disappears at program termination Linda does not provide an operation to store and retrieve the tuple space. If the programmer wants the tuple space to outlive the program (execution) the programmer has to store every individual tuple in the tuple space on disk. In the next startup the opposite strategy takes place: the tuples have to be restored from disk and inserted in the tuple space again. One problem with retrieving all the tuples from the tuple space is that it is not always known what the prototypes are of all the tuples in the tuple space and how many tuples with the same prototype are in the tuple space, although the `copy-collect` enhancement (Section 3.2.7) can alleviate this problem.

Enhancement: Persistent tuple spaces It would be much simpler to read a tuple space from disk at start up and write a complete tuple space back to disk at program termination. Special (member) functions for these actions could be provided. This is already a big improvement over loading and saving every individual tuple from and to disk. It would however be much simpler if the storing of the tuple space could be done automatically at termination of the program (for a persistent tuple space).

3.3.2 Fault Tolerance [4]

Deficiency: Hardware failure Hardware failures can happen at any time. Most programs do not survive a hardware failure and have to be started over. When doing a time consuming calculation much computation time can be lost. In parallel computing the chances of a hardware failure are much higher than in sequential computing.

Enhancement: Fault tolerant tuple spaces If a periodic back up is made automatically of the program state, a hardware failure would only result in a partial loss of the calculation. The next calculation can start from a saved program state. The saving of the program state is also known as a synchronization check point (relative to the disk). In fault tolerant computing there is always a trade off to be made. How often is a synchronization point needed? This can't be too often, synchronization takes quite some time due to disk access. But it can't be too seldom either, to minimize the amount of work that has to be redone in case of a failure.

One of the remarkable features of fault tolerant computing is that it hinges on the idea that a program is not supposed to use up all the processors. If one of the processors becomes faulty and the problem is not easily fixed or the fix takes too long, the program has to be restarted. But since one of the processors is faulty there is one less processor available. If all processors were consumed by the initial program the program state consists of all processors and can thus not be restarted on fewer processors. This makes fault tolerant programming only interesting if there are enough processors available. Running a parallel program on three instead of four machines, because you want to have fault tolerance, means a potential extra computation time of 33 percent. This problem can be overcome by the following enhancement.

3.3.3 Openness [30]

Deficiency: Network of workstations as a parallel system Most companies don't have real parallel machines. However they often have a network of machines that can be used as a parallel computer, as the machines are often idle (during off business hours, but even during daytime).

If the parallel applications take longer than the duration of the non-business hours, the tuple spaces should be extended with checkpointing (for fault tolerance) as well.

Enhancement: Open Linda Ideally all computers that are not used at any time could be utilized for some other time consuming calculation job. This means that the number of computers that can participate in the calculation changes continually. An open Linda system can add and remove processors from the calculation pool at all time. Processors should be added to the pool when they are expected to be idle for some time and they should be removed from the pool immediately when a user is about to use the machine. This kind of parallel computing is also called *adaptive parallelism* [12].

3.4 Software Engineering Enhancements

The previous sections showed several disadvantages of the Linda model that were described in the literature. All disadvantages could be solved by enhancing the original Linda model.

Linda has however still some disadvantages that are not described in the literature, namely software engineering disadvantages. Software engineering enhancements are enhancements that help programmers in writing correct and maintainable Linda programs.

For example the Linda tuple space is heterogeneous and can thus contain any tuple prototype that the programmer can make. This can easily lead to unwanted errors. What will happen if the programmer accidentally uses a tuple prototype only once? Assume the programmer uses this unique prototype only to retrieve a tuple. Evidently the statement that retrieves the tuple is blocking. This statement will never finish; it will remain blocked until the process finishes or it might even cause deadlock.

Of course a precompiler could recognize tuples with a unique prototype, and produce a warning upon compilation. But what happens if the programmer uses a prototype twice per accident? There is still a big chance that the program misbehaves, for example due to deadlock. It is also possible that the program gives faulty results.

In the next chapter we will introduce a Linda dialect called *Blossom* that prevents a programmer from making these errors. *Blossom* provides the programmer with several software engineering enhancements besides some of the enhancements described in this chapter (see next chapter).

Chapter 4

Blossom

This chapter is a rewrite of [54].

4.1 Introduction

Traditionally, Linda implementations provide the programmer with a weakly typed tuple space. Tuples of any type can be inserted and retracted from the tuple space. Most of these Linda implementations use either a library approach or a precompiler approach. *Blossom* is implemented as a type parameterized library which allows for strongly typed tuple spaces. This implementation approach is a powerful tool because it remedies many of the drawbacks of traditional Linda encountered in the previous chapter.

Sections 4.1.1 through 4.1.3 describe different implementation approaches for programming languages based on the Linda paradigm. Section 4.1.1 shows the library approach, Section 4.1.2 introduces the precompiler approach, and Section 4.1.3 describes the type parameterized library approach. Section 4.2 introduces the *Blossom* programming language. Section 4.3 goes into the details of how *Blossom* provides strong typing and what the advantages are for the programmer. Section 4.4 compares a Linda code fragment with an equivalent *Blossom* code fragment. Section 4.5 shows what features are implemented *Blossom* (see enhancements in previous chapter). Finally, Section 4.6 compares the features provided by SCA Linda and *Blossom*.

4.1.1 The Library Approach

In the library approach a Linda library is written that can be easily linked to the program (see Figure 4.1). The Linda library only provides six functions, namely one for every Linda operation (`out`, `in`, `rd`, `inp`, `rdp` and `eval`). By doing so all the different tuple prototypes that can be passed to the Linda operations have to be passed through one general tuple type. This general tuple type should be provided by the library as well.

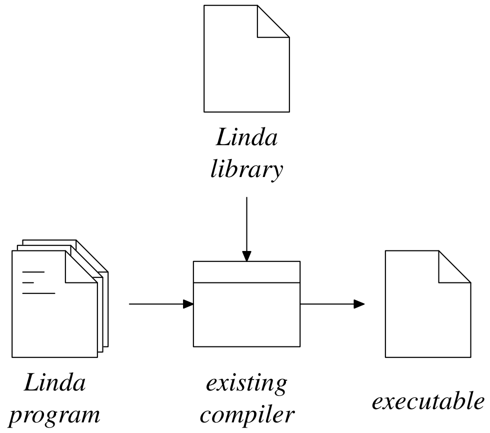


Figure 4.1: The library approach.

The advantage of the library approach is that a Linda library is easily written. This is useful for the programmer of the Linda library, however the programmer of a Linda program using this library will have to do more. He will have to convert all the different tuple types to the general tuple type.

Assume that the general tuple type provided by the Linda library is called `Tuple`. Class `Tuple` provides constructors for tuples with one field, two fields and so on.[1] It does not stop here, because the fields can still have infinitely many different arguments. Furthermore two types of fields exist, namely value fields and variable fields, corresponding to regular arguments and arguments proceeded with a question mark in Linda operations. A Linda library will have to provide these two types (assume `Value` and `Variable`). Both these types should have infinitely many constructors, one for every possible argument. Since this is not possible, let's for the time being limit the number of argument types to include only the basic types (i.e., not composed types). A simple Linda program becomes something like the following.

```
int sq;
for (int i = 0; i < MAX; i++)
   out(Tuple(Value("square"), Value(i), Value(i*i)));
rd(Tuple(Value("square"), Value(5), Variable(sq)));
```

This piece of code is verbose compared to the concise original Linda code. The information on what is actually going on is hidden behind the keywords `Tuple`, `Value` and `Variable`. Such a Linda program is not easy to read and thus not easy to maintain.

[1] A non-object-oriented host language may provide functions `Tuple` with different numbers of arguments. All these functions should, however, return one generic tuple type.

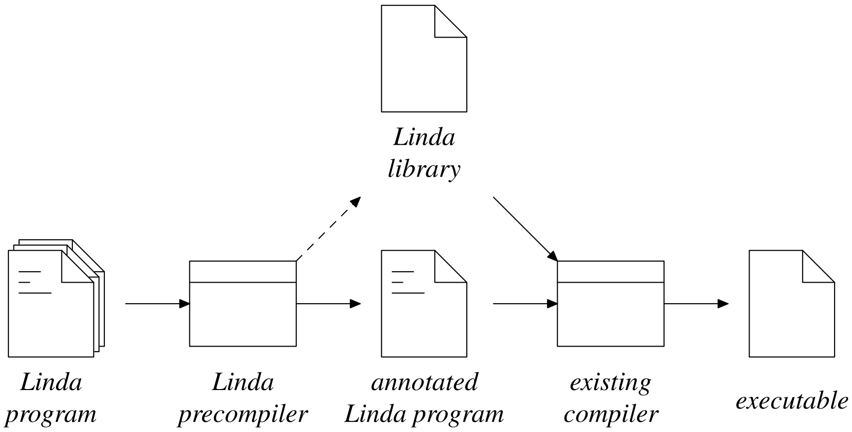


Figure 4.2: The precompiler approach.

Some of the simple library implementations make use of a simple preprocessor to overcome this problem. A preprocessor is a first step in the direction of a precompiler Linda implementation to be discussed in the following section.

If all argument types are allowed in tuples, then we have to map the infinitely many constructor functions `Value` and `Variable` to a finite number. This is possible by passing a pointer to the argument as well as the size of the argument. However, the resulting Linda syntax becomes more cumbersome.

The two partitioning optimizations, homogenization and distribution discussed in the Sections 3.2.1 and 3.2.2 respectively, need new field types and a precise programmer. To homogenize the tuple space, a field type is needed to indicate that this field is constant for all the tuples of this prototype (in other words it is a tag). To distribute the tuple space over all the processors, a field type is needed to indicate that this field will always contain a value. This can then be used to decide on what processor to store or retrieve a tuple. When one of these field types is used, it should be used consistently in all the tuples with the same prototype as well. A mistake may insert a tuple in or retrieve a tuple from a wrong partition of the tuple space.

4.1.2 The Precompiler Approach

Another way to implement Linda is with a precompiler. The precompiler reads the Linda program and translates it to an intermediate program (usually in the Linda host language). This intermediate program is translated by a compiler for the host language into an executable as can be seen in Figure 4.2.

Building a Linda implementation using the precompiler approach is much more laborious than building a simple library. Writing a precompiler that has to accept the host language as well as the Linda extensions entails much work, especially if the host language is a huge language like Ada or C++.

A basic Linda precompiler that only parses the Linda statements (the statements beginning with `out`, `in`, `rd`, `inp`, `rdp` and `eval`) will already be an improvement over a Linda library implementation. The precompiler can insert the code that the programmer had to write when using a library Linda. Programs that are written for an implementation of Linda that uses a precompiler can use the identical syntax as described in the Linda model (Section 2.1). These programs are shorter to write, easier to understand and easier to maintain.

Annotated Linda programs that the precompiler generates (see Figure 4.2) will look similar to the Linda programs of the previous section, where the programmer had to annotate the program with the keywords `Tuple`, `Variable` and `Value`. Since the interface to these annotated Linda programs are not important anymore (to the users), solutions that perform better can be thought of.

It is possible to take the precompiler approach one step further. Such an advanced precompiler will parse (part of) the host language as well. By parsing the host language, the advanced Linda precompiler has more information about the user's program and can make more decisions for the programmer. The most important extra information that an advanced precompiler has are the types of all the fields in the tuples. There are two ways to make use of this information. First the precompiler can warn the programmer for tuple prototypes that are only used once. It is most likely that the programmer made a mistake with this unique prototype.

Second, it can take care of homogenization of the heterogeneous tuple space using type information of the tuple fields. Usually the precompiler detects all the different tuple prototypes. Every unique prototype can be assigned a unique identifier that can be used to partition the tuple space at run time. A "smart" precompiler could even homogenize the tuple space at compile time. A separate homogeneous tuple space partition should be created for every tuple prototype. It is not very likely that current implementations do this.

When an advanced precompiler wants to homogenize the tuple space at compile time, it has to generate separate tuple space definitions for every homogeneous partition. Of the infinitely many homogeneous tuple space definitions that exist, the precompiler only generates those definitions that are used by the Linda program. This means that such a Linda precompiler generates the Linda library itself. This is indicated in Figure 4.2 by the dashed arrow.

4.1.3 The Type Parameterized Library Approach

A third alternative for solving the infinite library problem is an approach that uses a type parameterized Linda library. *Blossom* follows this alternative. A disadvantage of a type parameterized library is that it can only be written for a language that supports type parameterization. Of the main-stream programming languages, only C++ provides type parameterization at the moment. The designers of the Java programming language are considering a type parameterization extension for a future release.

The type parameterized library approach can be positioned between the library approach

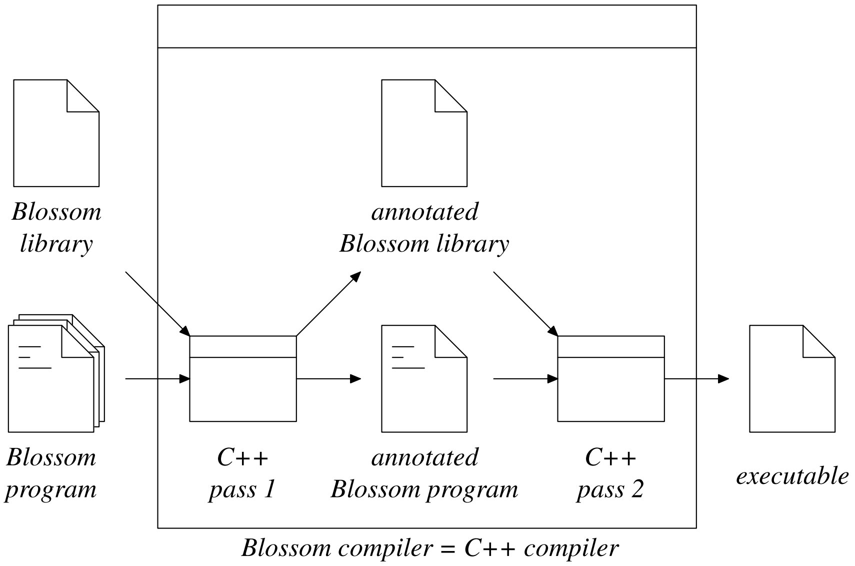


Figure 4.3: The type parameterized library approach.

and the precompiler approach. A type parameterized library can achieve the same performance of an advanced precompiler. It has to achieve this within the syntax of the host language, because a type parameterized library remains a library. *Blossom* is a type parameterized Linda library and thus has some of the disadvantages and advantages of both the library approach and the precompiler approach.

A type parameterized Linda library provides generic definitions of tuple spaces, tuples and operations. Specific instantiations of these generic classes and functions are created for every instance that occurs in a Linda program. This means that the final Linda library can be created from the type parameterized Linda library (see Figure 4.3).

The idea behind type parameterization is to allow definitions of classes and functions that are parameterized with one or more types. Such a type parameterized definition contains several old fashioned, standard definitions. When the compiler (or interpreter) encounters a specific object definition or function call, the appropriate class or function is instantiated from the type parameterized version.

With type parameterization, a programmer can specify an infinite library with a finite number of type parameterized definitions. These functions can span the infinite library. This is exactly what is needed in Linda.

Homogenization of the tuple spaces can be done at compile time (as is done in *Blossom*).

4.2 The Basic *Blossom* Language

This section introduces how to express the basic Linda operations in *Blossom*. There are two reasons for changing the syntax of Linda. First of all, *Blossom* is a type parameterized library which means that the programmer can only write syntax that is accepted by a C++ compiler. Second, *Blossom* provides multiple tuple spaces, one of the model enhancements that is described in Section 3.1.1.

4.2.1 Multiple tuple spaces

One of the proposed enhancements of Linda was multiple tuple spaces (Section 3.1.1). *Blossom* provides multiple tuple spaces. The programmer gives every tuple space a unique name. For example, a variable `ts` that is an instance of the tuple space `TupleSpace` would be declared as follows:

```
TupleSpace ts;
```

Assume that class `TupleSpace` is provided by the library. Later we will see that the declaration of a tuple space uses a somewhat different syntax.

4.2.2 Tuple space operations

Every tuple space operation has to know which tuple space to operate on. The standard way of doing this in an object-oriented language is to specify the tuple space object followed by a dot (`.`), followed by the operation name and arguments. For example, if `ts` is declared as above, then the following code fills tuple space `ts` with two tuples.

```
ts.out(3);
ts.out(3, 1);
```

The first tuple contains one integer with the value `3` and the second tuple contains the two integer fields with values `3` and `1`.

The Linda terminology for the operations is somewhat inappropriate for object-oriented programming. The names of the operations `out`, `in` and `rd` are relative to the program. In object-oriented programming the naming of the member functions is relative to the object they are invoked on.

To overcome this problem I introduced new names for the operations. In *Blossom* `out` becomes `put`, `in` becomes `get`, and `rd` becomes `copy`. Furthermore the predicate operations are renamed from `inp` to `get_nb` and from `rdp` to `copy_nb`. The `_nb` extension stands for "non blocking" which is a more descriptive naming of its functionality.

The operation `eval` is omitted in *Blossom*. Its replacement is the topic of the next section.

4.2.3 Process Creation

Parallelism in Linda is introduced by using the `eval` operation. `Eval` creates processes that evaluate the fields of a tuple. The resulting tuple is inserted in the tuple space as if it were passed to the operation `out`. The combination of creating processes and inserting a tuple in a tuple space is not always suitable. We saw this in the dining philosophers example (Section 2.2.3).

Originally *Blossom* provided an `eval` operation as well. One of the design goals of early *Blossom* was to stay as close as possible to Linda. However, the implementation of `eval` consumed approximately forty percent of the source code and had a huge impact on the design of the *Blossom* library. `Eval` hindered the separation of the orthogonal concepts of the tuple spaces into distinct classes. This is an indication that the `eval` operation is not orthogonal to the rest of the Linda language.

In an attempt to make *Blossom* more orthogonal, the `eval` operation eventually was removed from the language. The tuple inserting ability of `eval` is already present in the *Blossom* model through the `put` operation. The ability to introduce processes was available only through the `eval` operation. Clearly, this capability needs to remain in *Blossom*.

Process creation in *Blossom* is done explicitly. The user has to specify processes and on which machine to spawn them. A process is specified by deriving a class from class `Process`. The activity that should take place in the process should be defined in the member function `main()`.

```
class Master: public Process {
public:
   void main() {
      ...
   }
};
```

The user of a *Blossom* executable has to indicate on which machines the executable may run. A process can be spawned on a specific as well as an arbitrary machine. The machines are numbered (starting at 0). The machine number is used in the spawning operation.

```
Spawn<Master>()(0); // spawn a Master process on machine 0.
Spawn<Master>()();  // spawn a Master process on any machine.
```

The programmer can pass the arguments to the constructor of the `Master` process in between the first pair of brackets. Usually a process needs to share at least one tuple space with another process to be able to communicate. This tuple space should be passed to the process at creation time. An example of how the `Spawn` operation is used can be found in Section 4.4.

4.2.4 Tuple Arguments

In Linda we encountered two types of tuple arguments. Value arguments are fields that contain a value, for example the value of a variable. Variable arguments are fields that contain a variable that may be updated by a retrieving operation. In case of a non-blocking operation, the variable field will only be updated in the event that the predicate operation returns `true`.

In Linda variable fields are indicated with a question mark (`?`). This is not possible in *Blossom* because C++ does not provide a question mark operator. *Blossom* thus needs another way to differentiate variable fields from value fields. What is sometimes done in Linda library implementations is that the question mark is exchanged for an ampersand (`&`). The ampersand operator returns the address of an object. This has as the disadvantage that strings (pointers to character arrays) need to be treated differently from pointers to objects. This irregularity in the language is difficult to sell.

Blossom provides a different solution to this problem. Variables that can be updated by the query operation get a new type, namely type Var<T>, where type `T` is the type of variable that can be updated. The ampersand operator is used to indicate that this variable should be updated. If the variable does not have to be updated, then the ampersand is omitted.

The way these Var<T> variables are used in *Blossom* is shown in the following example. On the left are the statements in *Blossom* and on the right are the corresponding statements in Linda.

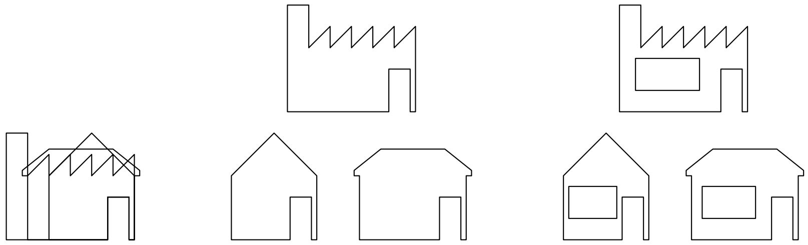
```
TupleSpace square;
for (int i = 0; i < MAX; i++)        for (int i = 0; i < MAX; i++)
   square.put(i, i*i);                  out("square", i, i*i);
Var<int> sq;                         int sq;
square.copy(5, &sq);                 rd("square", 5, ?sq);
cout << sq << endl;                  printf("%d\n", sq);
```

The difference between the programs is not large. In *Blossom* a tuple space has to be defined (Section 4.2.1). This definition is not necessary in Linda because Linda uses one global tuple space. The next difference is that the tuple space should be specified as part of the *Blossom* operations (Section 4.2.1), here `put` and `copy`. The third difference is that tuples in *Blossom* usually contain one field less than their counterparts in Linda. *Blossom* tuples don't need the Linda tag field; this usually becomes the name of the tuple space in *Blossom*. Notice that this tag field can be used by the Linda precompiler to homogenize the tuple space, something that is made explicit in *Blossom*. The fourth difference is the type of the variable `sq` (earlier this section). The last difference is that *Blossom* uses ampersands instead of question marks to indicate variable fields.

When the ampersand is omitted in the `copy` operation, the variable `sq` is passed by value. Thus, where `&sq` forms a variable field, `sq` forms a value field. In this example, variable `sq` was not yet assigned a value, so it has an arbitrary, unknown value.

4.3 Software Engineering Enhancements

The Linda tuple space is a typeless black box. The user can put data of any type in it, and attempt to extract data of any type from it. But the wealth of programming language design



(a) (b) (c)

Figure 4.4: Different kinds of tuple spaces.
(a) Original Linda tuple space: It is a single collection of tuples of arbitrary structure. This implies that the user can try to insert/remove semantically incorrect tuples. The user cannot look inside the tuple space. (b) Strongly typed tuple space: Each tuple space can contain only one type of tuple. Hence a program cannot inadvertently insert/remove a wrongly typed tuple. (c) Strongly typed tuple space with assertions: It provides the user the additional benefit of being able to peek inside the tuple space.

experience suggests that this is a bad idea. The trend in computing today is towards strong typing. We have seen the evolution of BCPL to C to C++, motivated in part by the benefits of strong typing. These benefits are twofold: catching more errors at compile-time, and preventing some classes of run-time errors. Given the inherently difficult nature of parallel programming, any language enhancements that reduce the probability of error are welcome.

The previous chapter described three kinds of enhancements to the Linda model. In this section, four new software engineering enhancements to the Linda model are presented. All of them have to do with either preventing or detecting (parallel) program errors at compile- or run-time. This is achieved by changing the definition of the tuple space: it is given structure, behavior, and is made accessible (see Figure 4.4). The user provides annotations that give the compiler and run-time system information about the expected behavior of the program, and forces it to meet those constraints.

Blossom is a C++ class library. Unlike C-Linda and Fortran-Linda, it does not add new syntax, alter the semantics, or extend the host language, each of which tends to result in programmer confusion [48]. C++ is a modern programming language with a rich set of features. This enables Blossom to achieve many of its objectives (including preventing programming errors) using only C++ syntax, without the need for a separate precompiler. To achieve the same benefits of a Linda precompiler (including error checking and run-time optimizations), the programmers have to specify a small amount of additional detail in their programs. This is reflected in the Blossom examples. They are (slightly) more verbose than their pure-Linda counterparts.

One issue not addressed in this section is how to handle run-time errors generated by the new Blossom enhancements. Since we are using C++, a natural solution is to throw an exception whenever a run-time error occurs. The programmer can (at his discretion) catch the exception and handle it. Although this sounds good, it does have its problems. For example, do child processes throw exceptions in their parents? What if a process' parent is no longer around, but its grandparent is? Right now, all run-time errors result in aborting the application.

4.3.1 Strongly Typed Tuple Spaces

Linda's single tuple space can contain tuples of any type. Although this sounds conceptually simple, it has the disadvantage that it is easy to inadvertently introduce a bug in a program. Accidentally omitting a tuple field, introducing an additional field, changing a field, or reversing two fields is perfectly legal, yet may result in a semantically incorrect program. A sophisticated precompiler may be able to detect some but not all of these errors.

By having multiple tuple spaces, each strongly typed, these problems cannot occur. Each tuple space is created with a specification that gives the number, order, and type of each field of the tuples legally allowed (much like the parameters of a subroutine call). With this specification, many commonly occurring errors can be caught at compile time without the need for a Linda precompiler.

The advantages of strong typing can be summarized as follows:

1. Elimination of Errors.
   A strongly typed tuple space can eliminate many common errors.

2. Performance.
   Although this section emphasizes the software engineering aspects of *Blossom*, it is important to realize that many of the software engineering enhancements combine nicely with efficiency enhancements. In effect, strongly typed tuple spaces combine neatly with homogenizing the tuple space (described in Section 3.2.1). The typing of the tuple space partitions the tuple space into multiple homogeneous address spaces. Hence, given a tuple of a particular type, the system knows exactly in what address space to look for the tuple.

The following example illustrates having multiple, strongly-typed tuple spaces in *Blossom*. The `TupleSpace` declarations each create new named tuple spaces, and specify the type of the tuples that are allowed inside them.

```
#include <cmath>

TupleSpace<Tuple<int, int> > square;
TupleSpace<Tuple<int, float> > sqrroot;

// Put values in the tuple spaces
for (int i = 1; i < 10; i++) {
   square.put(i, i*i);
   sqrroot.put(i, sqrt(i));
}
```

```
// Take values out of the tuple spaces
Var<float> answer;
sqrroot.get(4, &answer);    // Succeeds
square.get(1, 1);           // Succeeds
square.get(2, 2);           // Blocks: no such tuple available
square.get("one", 1);       // Compile-time type error
square.get(1, 1.0);         // Compile-time type error
square.get(1.0, 1);         // Compile-time type error
square.get(1, 1, &answer);  // Compile-time error
                               (too many arguments)
```

The declarations of the tuple spaces `square` and `sqrroot` are different from what we have seen before. The class name consists of the keyword `TupleSpace` followed by one or more `Tuple` declarations in between the signs < and >. The `Tuple` declarations indicate what tuples are allowed inside the tuple spaces. Thus tuple space `square` accepts tuples with prototype `int` × `int` and tuple space `sqrroot` accepts tuples with prototype `int` × `float`. The `Tuple` declarations will be elaborated in the next section, so this example will not compile. This example only illustrates that tuple spaces are strongly typed.

The first three `gets` behave exactly the same as their equivalent `ins` in Linda. The last three `gets` have a different behavior than the equivalent `ins` in Linda. *Blossom* generates compile time errors, whereas the Linda program would block at run time.

4.3.2 Field Access Patterns

The philosophy behind typing a tuple space can be carried even further. A common programming scenario in Linda is to extract tuples using one of the fields as a discriminator. For example, one might have a producer-consumer computation where the tuples produced have to be processed in order. Every tuple contains a unique number which is essentially used as a time stamp. The smaller the number the older the tuple. The consumer `gets` the next tuple by increasing the time stamp field. In this case, this field is not only an integer, but it is always used as a *value*.

Every field in a tuple space declaration contains a declaration as to whether it is legal to use a variable field to retrieve a value or not. In other words, a field is designated as either `Value` or `Any`. `Value` indicates that a tuple field can only be retrieved by specifying a value, while `Any` extends this to include updatable variables (variables preceded by an ampersand).

This additional piece of information offers the user several advantages:

1. Elimination of Errors.
   By providing additional information about the intended behavior of the tuple space, the compiler can check whether the programmer always satisfies the specified requirements.

2. Performance.
   The field access patterns model enhancement combines nicely with the distribution

of the tuple space efficiency enhancement (Section 3.2.2). All the fields that are indicated to be `Value` fields can be used to determine (by hashing) on which machine to store or look for a tuple. An evenly distributed tuple space results in fewer tuples to be searched and removes possible bottlenecks of a big tuple space on one machine.

The following example illustrates field access patterns. The tuple space `square` has two arguments: the first must be a value, while the second can be a value or a variable.

```
TupleSpace<Tuple<Value<int>, Any<int> > > square;
Var<int> sq;
for (int i = 1; i < MAX; i++)
   square.put(i, i*i);
square.get(1, &sq);           // Succeeds
square.get(2, 2);             // Legal, but blocks
square.get(&sq, 4);           // Compile-time error
```

4.3.3 Tuple Space Access Patterns

The object-oriented community has recently embraced the concept of *design patterns* [16]: generic, frequently occurring programming paradigms. Similar ideas have played a prominent role in parallel computing, where common parallel structures have been isolated, such as master-slave and pipelines. Recent studies have shown that there are commonly occurring access patterns[2] on objects [7]. As part of program design, the user knows the intended access patterns on their objects. The first two columns of Table 4.1 gives these access patterns and their description, as defined in the Munin system [7].

The access patterns on objects are easily translated to access patterns on tuples [11]. In *Blossom* the access patterns are applied to tuple spaces instead of tuple tags (as described in Section 3.2.5). There are several reasons for doing this. First, in a good design, every tuple space has a unique purpose. All members of that tuple space are there to fulfill the design goals of the tuple space and should be handled similarly. Tuples with different access patterns in one tuple space are an indication of a poor design. Second, access patterns can be used to influence the (hidden) implementation of the tuple space, affecting program performance. For example, the choice of design pattern can affect the placement of the tuple space, the distribution of the tuple space, and whether data should be replicated.

If a tuple space were annotated with an access pattern specification, then that might impose a constraint on how the tuple space is accessed at run-time (as shown in the third column of Table 4.1). Note that not all access patterns result in constraints on the tuple space. Read-mostly, write-many, and general read-write all allow an arbitrary combination of reads/writes to the tuple space. The only difference is the relative frequency with which the reads/writes occur. Hence these are too general to allow constraints to be imposed.

There are a number of important advantages for having the user declare the access pattern of a tuple space:

[2]The original name is type-specific coherence mechanisms.

Table 4.1: Access patterns for tuple spaces

ACCESS PATTERN	DESCRIPTION	RESTRICTIONS
write-once	written during initialization, afterwards only read	no `puts` after a `get` or `copy`
private	local accesses only	only one process is allowed access
write-many	frequently modified	—
result	not needed until all data is collected	—
synchronization	semaphores	no `copy` or `copy_nb` allowed
migratory	accessed by a single process at a time	—
producer-consumer	written by one process, consumed by other processes	only one process is allowed to `put`
read-mostly	read far more often than written	—
general read-write	general access pattern	—

1. Elimination of Errors.
   If the access pattern of a tuple space does not match the access pattern in the user's code, then warning or error messages can be given at compile- and run-time. For example, if the access pattern is producer-consumer and there are two or more processes `put`ting tuples in the tuple space, the program can detect this at run-time. (Not at compile-time because we do not use a precompiler.) If the program has to run efficiently this run-time check can be turned off.

2. Performance.
   A programmer usually knows the intended use of a tuple space. Depending on these intentions, the programmer can choose a tuple space access pattern that results in the least amount of communication. Less communication usually results in a faster program. We saw something similar in the efficiency enhancement *tuple access patterns* (Section 3.2.5).

3. Design Aid.
   Awareness of access patterns results in better designs. A good design never uses an object for more than one reason. This general rule applies to tuple spaces too. Access patterns can help the programmer distinguish different functionalities more easily.

It is important to realize that access pattern attributes only affect the semantics of a tuple space if a restriction is encountered; if this is not the case access pattern attributes only change the implementation and not the semantics.

The following example illustrates the use of access patterns.

```
TupleSpace<Tuple<Any<int> >, ProducerConsumer> buffer;

// PRODUCER...
while (true) {
   int number = ...;
   buffer.put(number);   // Succeeds
}

// CONSUMER...
while (true) {
   Var<int> number;
   buffer.get(&number);  // Succeeds
   // ...
   buffer.put(16);       // Run-time error
}
```

4.3.4 Tuple Space Assertions

The simplicity of the Linda programming model comes, in part, from the capabilities of the tuple space; it is a black box with all the structure and implementation details hidden from the user. However, when looking for a bug, you want to be able to look inside the black box. This cannot be done in Linda, unless you extract each of the tuples one at a time and then put them back.

Tuple space assertions allow the user to peek into the tuple space. A problem with assertions in parallel programming is that several processes have access to the same tuple space. This means that if you put a tuple in the tuple space and then, for example, immediately assert that the number of tuples in the tuple space is greater than zero, the assertion can fail because another process removed the tuple. So either you have to make the assertion general (i.e., it takes other processes into account) or the assertion should form an atomic action with the other Linda operations. We chose the latter kind of assertion, because it better meets the needs of concurrent debugging.

Assertions act on an entire tuple space, which may be very inefficient if the tuple space is distributed. This is mainly because the whole tuple space should be locked during the evaluation of an assertion; no other processes are allowed to change its state. Assertions on tuple spaces should only be used to debug a program. However after fixing the bug you want to be able to recompile the program without removing the assertions again. Inclusion of another library (a compile-time option) can achieve this.

If the user wants to define an assertion on a tuple space, he has to derive a new class from class `Assertion`. In this class he must define two methods, `foreach` and `exit`. The constructor call initializes the object, the `foreach` call is performed for every tuple in the tuple space, and the `exit` call should return a boolean, indicating whether the assertion succeeded (`true`) or failed (`false`). If the `exit` call returns `false` the program terminates. *Blossom* provides the (hidden) code to ensure that `foreach` and `exit` are called correctly. The `foreach` method has as arguments the fields of the tuples that it inspects.

To perform an assertion on a tuple space, an object of a derived class from class `Assertion` has to be passed as the last argument in any of the operations (`put`, `get`, `get_nb`, `copy` or `copy_nb`). The assertion is performed together with the operation as one atomic action. This is needed because the contents of the tuple space may already have changed because of another process.

Assertions are illustrated in the next section.

4.4 Example

The following example illustrates the usage of the software engineering enhancements introduced in the previous section. The primes example that we already saw in Section 2.2.5 is reused. This time it is written for *Blossom* and a tuple space assertion is included.

The strongly typed tuple space checks at compile-time that all accesses (here `put` and `copy`) have a tuple of the correct type (`put` only allows actual fields and `copy` only allows the specified fields). The `SizeAssertion` checks that the number of tuples in the tuple space is correct after the each insertion of a tuple.

```
#include <iostream>
#include "blossom.h"

typedef TupleSpace<Tuple<Value<int>, Any<bool> > > PrimesTS;

const int LIMIT = 5000;

class SizeAssertion: public Assertion {
   int expected_size;
   int size;
public:
   SizeAssertion(int s): expected_size(s), size(0) {}
   ~SizeAssertion() {}
   void foreach(int, bool) {
      size++;
   }
   bool exit() {
      return size == expected_size;
   }
};

bool is_prime(int number, PrimesTS primes) {
   int limit = sqrt(number);
   for (int i = 2; i < limit; i++) {
      Var<bool> prime;
      primes.copy(i, &prime);
```

```
      if (prime && (number%i == 0))
         return false;
   }
   return true;
}

class Prime: public Process {
   int prime;
   PrimesTS primes;
public:
   Prime(int p, PrimesTS pts): prime(p), primes(pts) {}
   void main() {
      primes.put(prime, is_prime(prime, primes));
   }
};

void main() {
   PrimesTS primes;
   for (int i = 2; i < LIMIT; i++)
#ifdef DEBUG
      primes.put(i, is_prime(i, primes),
            SizeAssertion(i-1));
#else
      Spawn<Prime>(i, primes);
#endif
   Var<bool> prime;
   for (int i = 2; i < LIMIT; i++) {
      primes.copy(i, &prime);
      if (prime)
         cout << i << endl;
   }
}
```

4.5 Current Status of *Blossom*

The intention of the *Blossom* language is to provide a fast and strongly typed Linda implementation. An overview of all the enhancements from the literature (see the previous chapter) that are implemented in *Blossom* follows.

4.5.1 Model Enhancements

The current *Blossom* language contains all the model enhancements discussed in Section 3.1. An overview of the syntax of the model enhancements is described in this section.

Multiple Tuple Spaces

In Section 3.1.1 the idea of multiple tuple spaces was introduced and in Section 4.2.1 the *Blossom* syntax for multiple tuple spaces was introduced. The current *Blossom* language follows the syntax described earlier.

Array Fields

Arrays are first class types in C++, and as such, can be specified as fields in tuples. Usually it is easier to specify the type and length of the array in a separate `typedef`:

```
typedef int Array10[10];

TupleSpace<Tuple<Any<Array10> > ts;
Array10 a;
for (int i = 0; i < 10; i++)
   a[i] = i;
ts.put(a);
Var<Array10> va;
ts.get(&va);
```

It is not possible to omit the size of an array as is sometimes possible in C++. Thus *Blossom*'s arrays are fixed sized *à la* Pascal. This prevents some problems that may occur with the arrays of SCA Linda.

Assume one process runs this code:

```
int a[10];
for (int i = 0; i < 10; i++)
   a[i] = i;
out("array", a:10);
int len;
in("array", ?a:len);
```

and another process runs the following code:

```
int b[20];
out("array", b:20);
```

The former process may accidentally read the twenty element array that the latter process inserted. However array `a` in the first case can only contain up to ten integers.

Wild Card Fields

In Section 3.1.3 we encountered wild card variables. Wild card variables allow the programmer to not specify a variable field in a query tuple that is not used afterwards. *Blossom* provides a special class for wild card behavior, namely class `VarWC`. Class `VarWC` is followed by a type of the field in between angular brackets.

```
TupleSpace<Tuple<Value<int>, Any<int>, Any<int> > > square_cube;
for (int i = 0; i < 10; i++)
   square_cube.put(i, i*i, i*i*i);
Var<int> sq;
square_cube.get(i, &sq, &VarWC<int>);
```

A wild card variable should be preceded with an ampersand to indicate that it is a variable field. This is merely done to keep *Blossom*'s syntax consistent.

Synchronous `out`

Section 3.1.4 showed that an asynchronous `put` causes trouble when used in combination with the non-blocking operations `get_nb` and `copy_nb`. To overcome these troubles *Blossom* provides two versions of `out`: an asynchronous one (`put`) and a synchronous one (`put_s`).

The synchronous `put_s` operation is called in the same way as the asynchronous `put` operation. The only difference is that the synchronous operation `put_s` waits for a confirmation that the tuple is inserted in the tuple space.

4.5.2 Efficiency Enhancements

Homogenization of the Tuple Space

The division of the tuple space into separate homogeneous partitions for each combination of tuple tag(s) and tuple prototype is done in two ways. First, the multiple tuple spaces induce a natural partitioning. Second, heterogeneous tuple spaces are partitioned by *Blossom*. Partitioning the tuple space is one of the most important enhancements to the Linda model.

Distribution of the Tuple Space

The default tuple space access pattern distributes the tuple space over the machines that the *Blossom* program is allowed to use. This is done by using the fields that are indicated by the field access pattern `Value`. These fields are present in all tuples and can be used to calculate (by hashing) a value on which machine to store or look up a tuple. The programmer does not have to do anything to get a distributed tuple space. If the user overrides the default tuple space access pattern, then the tuple space is not necessarily distributed anymore. A Linda precompiler can do something similar, as we saw in Section 3.2.2.

Prefetching

The prefetching enhancement, as described in Section 3.2.4, is available in *Blossom*. The *Blossom* programmer has to implement prefetching himself. Prefetching in *Blossom* is done via Futures [22]. A future is a variable that is updated asynchronously. By providing futures in the fields the operations `get` and `copy` become asynchronous.

```
Var<Task> task;
tasks.get(&task);
while (!task.empty()) {
   Fut<Task> new_task;
   tasks.get(&new_task);
   task.perform();
   task = new_task;
}
```

Class `Fut` followed by a type in between brackets < and > indicates a future of the specified type. The statement `tasks.get(&new_task)` is now performed asynchronously.

The `get` and `copy` operations become asynchronous when the tuple contains at least one future field and no variable fields. Value fields have no influence on the synchronicity or asynchronicity of the operations.

Asynchronicity of a `get` or a `copy` operation means that the program proceeds and does not immediately wait for the receipt of the values with which the futures should be updated. The futures are updated as soon as the new values become available. Ideally this takes place before the program is about to use the new values of the futures. If this is not the case, then the program will have to block and wait for the update of the futures. In the above example this possible block is just before the assignment of the future `new_task` to the variable `task`.

Tuple Access Patterns

Blossom provides tuple space access patterns as described in Section 4.3.3. However these tuple space access patterns are intended as a software engineering enhancement and not as an efficiency enhancement. This means that *Blossom*'s tuple space access patterns serve two goals that clash. Efficiency is not concerned with software engineering tests that check the intended behavior. For this reason separate names for the efficiency tuple space access patterns are used.

At the moment *Blossom* only provides two tuple space access patterns, namely `Distributed` and `Local`. If no tuple space access pattern is defined in the tuple space declaration, the `Distributed` tuple space access pattern is assumed.

Blossom provides the hooks to easily extend the number of tuple space access patterns. Candidates for implementation are the tuple space access patterns `Replicated` and `Migratory`.

The `Replicated` tuple space access pattern replicates the tuple space on every machine so that the `copy` and `copy_nb` operations can be performed locally. The other tuple space operations change the contents of the tuple space and thus need to forward this information to the other machines so that they can update the replicated tuple space. The tuple space access pattern `Replicated` may change the syntax of the tuple space, because it is difficult to keep a replicated tuple space coherent.

The `Migratory` tuple space access pattern divides the tuple space into smaller parts. Whenever a process performs an operation on a tuple space, the machine on which the process is running gets those parts of the tuple space that are of interest to it. This machine temporarily gets exclusive rights to this part. The idea is that processes use a specific part of the tuple space repeatedly in a short period of time. If this is the case, then the communication overhead of migrating parts of the tuple space can be compensated.

4.5.3 Paradigm Enhancements

None of the paradigm enhancements are included in *Blossom*. *Blossom* is intended to be a parallel language and most of the paradigm enhancements are not very interesting for performance.

4.5.4 Software Engineering Enhancements

Though the software engineering enhancements of *Blossom* are novelties for Linda, not all the software engineering enhancements are supported in the current version of *Blossom*. The syntax of the unsupported software engineering enhancements is however well defined in Section 3.4.

Strongly Typed Tuple Spaces

Strongly typed tuple spaces (Section 4.3.1) specifies what tuple prototypes are allowed in tuple spaces. The prototypes specify what arguments are allowed in the tuple space operations. The allowable arguments are further dependent on the operations (see Table 4.2). The operations `put` and `put_s` only allow value arguments. The operations `get` and `copy` allow value arguments and variable arguments as well as future arguments. The operations `get_nb` and `copy_nb` allow value arguments and variable arguments. Of course the tuple prototype in the strongly typed tuple space declaration specifies what type of value argument, variable argument and future arguments are allowed in the tuple space operations.

Field Access Patterns

Field access patterns (Section 4.3.2) specify which arguments are allowed in the query operations. We only encountered two field access patterns, namely `Value` and `Any`. The `Value` field access pattern only allowed value arguments in the query operations, and the `Any` field access pattern allowed any (value as well as variable) arguments in the query operations.

The introduction of futures in *Blossom* introduces another argument type, namely future arguments (variables that can be updated asynchronously). This extra argument type introduces several extra field access patterns, though the field access patterns `Value` and `Any` will remain the most versatile.

For every combination of value, variable and future arguments, a separate field access pattern is introduced (see Table 4.3). Beside the `Value` field access pattern that only allows value arguments, field arguments `Variable` and `Future` are available that only allow variable and future arguments, respectively.

Table 4.2: Tuple space operations and field argument types.
The allowable combinations of tuple space operations and field argument types for a field with prototype `T` are indicated with a $+$ while the combinations that are not allowed are indicated with a $-$. Type `T` is a C++ type (for example `int`).

OPERATION	VALUE
	`T`
	`Var<T>`
	`Fut<T>`
		VARIABLE
		`Var<T>*`
		`VarWC<T>*`
			FUTURE
			`Fut<T>*`
			`FutWC<T>*`

`put`	+	−	−
`put_s`	+	−	−
`get`	+	+	+
`copy`	+	+	+
`get_nb`	+	+	−
`copy_nb`	+	+	−

Table 4.3: Field access patterns and field argument types.
The allowable combinations of field access patterns and field argument types are indicated with a $+$ while the combinations that are not allowed are indicated with a $-$. Type `T` is a C++ type (for example `int`).

FIELD ACCESS PATTERN	VALUE
	`T`
	`Var<T>`
	`Fut<T>`
		VARIABLE
		`Var<T>*`
		`VarWC<T>*`
			FUTURE
			`Fut<T>*`
			`FutWC<T>*`

`Value<T>`	+	−	−
`Variable<T>`	−	+	−
`Future<T>`	−	−	+
`Synchronous<T>`	+	+	−
`Asynchronous<T>`	+	−	+
`Query<T>`	−	+	+
`Any<T>`	+	+	+

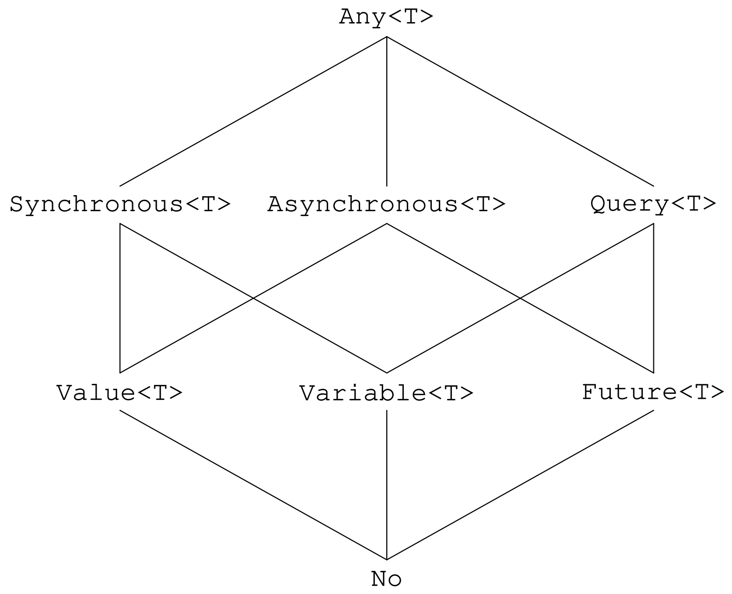


Figure 4.5: Field access patterns and their relationships.
A field access pattern can contain any of the fields that its lower field access patterns can contain.

The field access patterns `Synchronous`, `Asynchronous` and `Query` all allow two argument types in the query operations. `Synchronous` allows value as well as variable arguments, `Asynchronous` allows value as well as future arguments and `Query` allows variable as well as future arguments.

Finally, field access pattern `Any` allows —as the name implies— all three arguments types.

The relation between all seven field access patterns can be shown graphically as well (Figure 4.5). The field access patterns above the field access patterns `Value`, `Variable` and `future` allow all the arguments that they are connected with at a lower level. For completeness sake a "field access pattern" `No` is introduced. The field access pattern `No` does not have any meaning in *Blossom*, since it does not make sense that you cannot query tuples from a tuple space.

Tuple Space Access Patterns

Blossom provides two tuple space access patterns, but none of them provides software engineering advantages. The two tuple space access patterns are mainly for performance reasons.

Table 4.4: Comparison of features between SCA Linda and *Blossom*. *Blossom* provides all the features that SCA Linda provides plus several extra ones.

	SCA LINDA	*Blossom*
MODEL ENHANCEMENTS		
Array Fields	+	+
Wild Card Fields	+	+
Multiple Tuple Spaces	−	+
Synchronous `out`	−	+
EFFICIENCY ENHANCEMENTS		
Homogenization of the Tuple Space	+	+
Distribution of the Tuple Space	+	+
The Operation `in`-update-`out`	−	−
Prefetching	−	+
Tuple Access Patterns	−	−
The Operation `collect`	−	−
The Operation `copy-collect`	−	−
PARADIGM ENHANCEMENTS		
Persistence	−	−
Fault Tolerance	−	−
Openness	−	−
SOFTWARE ENGINEERING ENHANCEMENTS		
Strongly Typed Tuple Spaces	−	+
Field Access Patterns	−	+
Tuple Space Access Patterns	−	+
Tuple Space Assertions	−	−

4.6 Conclusions

Table 4.4 shows which features are available in SCA Linda and *Blossom*. *Blossom* provides all the features SCA Linda does plus several extra ones.

Chapter 5

Blossom Architecture

While Chapter 4 discussed the syntax and (informal) semantics of *Blossom*, this chapter explains the architecture of the *Blossom* implementation. How *Blossom* is implemented is discussed later in Chapter 7. Parallel and distributed computing means communication between two parties. In *Blossom* these parties are called the tuple space front end and the tuple space back end. The tuple space front end forms the interface to the tuple space back end and is the part that is seen by the programmer. The value tuples are actually stored in the back end.

Section 5.1 shows an example of an operation on a tuple space, giving an overview of the requirements and architecture introduced in the remainder of this chapter. Section 5.2 formalizes the requirements of the tuple space front end. In the front end more happens than meets the eye. Section 5.3 gives an overview of the communication sequences that occur between the front end and the back end. Section 5.4 introduces the requirements of the architecture of the back end.

5.1 Example

The following example gives an overview of what the *Blossom* implementation does for a simple `get` operation. The example serves two purposes; (*i*) it illustrates the complexity of the *Blossom* architecture, and (*ii*) it gives the reader some idea of what *Blossom* does behind the scenes.

We start with a simple tuple space declaration.

```
TupleSpace<Tuple<Value<int>, Any<int> >,
           Tuple<Value<int>, Any<int>, Any<int> > > ts;
```

The tuple space `ts` is a heterogeneous tuple space consisting of two homogeneous tuple space partitions. The first partition contains tuples with two integer fields, while the second partition contains tuples with three integer fields. The declaration of tuple space `ts` does not specify a tuple space access pattern. Consequently, the tuple space `ts` gets the default tuple space access pattern `Distributed`. This means that both homogeneous tuple space partitions are distributed over the available processes.

Let's look at an operation on tuple space `ts`.

```
Fut<int> fut;
ts.get(1, 1, &fut);
```

The programmer wants to retract a value tuple from tuple space `ts` that matches the query tuple `(1, 1, &fut)`. A matching value tuple has three integer fields of which the first two should have the value `1` and the last may have any value. The value found for the third field is assigned to the variable `fut`. Without considering the tuple space declaration, two decisions can already be made. First, it can be decided whether the kinds of the fields used in the query tuple are allowed for this operation (summarized in Table 4.2). This is easy, because in principle for a `get` operation all fields are allowed. However the updatable future would not be allowed in case the `get` operation would for example be a `get_nb` or a `put` operation.

Second, there is no need to consider the declaration of tuple space `ts` in deciding that the operation `get` should be performed asynchronously. This is due to the fact that the query tuple contains an updatable future and no updatable variables (Section 3.2.4). In contrast, the following `get` operation should be performed synchronously.

```
Var<int> var;
Fut<int> fut;
ts.get(1, &var, &fut);
```

The updatable variable `var` forces the operation `get` to be synchronous, although the updatable future `fut` allows for asynchronous execution of the operation `get`.

When comparing the fields of the query tuple with the declaration of the tuple space `ts`, it is evident that the tuple fields are allowed by the field access patterns specified in the tuple space. This would not be the case if the tuple space `ts` declaration would use the `Variable` field access patterns instead of the `Any` field access patterns, because the updatable future is not an updatable variable.

Another task that *Blossom* has to perform involves finding the distributed, homogeneous partition (i.e., the machine, or preferably, the process) in which to search for the requested query tuple (`(1, 1, &fut)`). For this task, the declaration of tuple space `ts` is required. For the preceding example this is achieved in the following steps.

Blossom starts to determine the correct homogeneous partition by comparing the tuple prototype of the tuple provided in the `get` operation (`int` × `int` × `int`) with those of the homogeneous partitions of tuple space `ts` (`int` × `int` and `int` × `int` × `int`). From this *Blossom* concludes that the tuple `(1, 1, &fut)` has to be searched in the second homogeneous partition.

The second partition of the tuple space is `Distributed` (just like the whole tuple space). All the fields in the query tuple that have field access pattern `Value` (specified in the tuple space declaration) can be used to calculate on which process the query tuple can be found, as long as the same calculation is used for the value tuples inserted in the tuple space. In the preceding example only the first value `1` can be used to calculate a process from which to retrieve the query tuple. The second field (with value `1`) has field access pattern `Any` (see declaration of tuple space `ts`) and not `Value`. This completes the question of where (which distributed, homogeneous partition) to find matching value tuples for the given query tuple.

Once the correct process has been found on which to look for matching value tuples, a message has to be sent to this process. After sending this message, the front end process proceeds since the `get` operation is asynchronous. Upon receipt of the message by the back end process, the tuple space partition has to be searched for a matching tuple. If no matching tuple resides in the tuple space (partition), the program has to wait for a matching value tuple to be inserted. Once a matching value tuple has been found, a message is sent back. Upon receipt of this message by the tuple space front end, the updatable future is instantiated with its new value. Furthermore, the updatable future is indicated to be up-to-date, such that in case the process was blocked on the updatable future, it can proceed.

5.2 Front End

Most of the activities discussed in the previous section take place in the front end. Activities for which the front end is responsible include:

1. Adding the default tuple space access pattern (`Distributed`) to a tuple space if necessary;
2. Finding the correct homogeneous partition of the tuple space and also checking:
   (a) whether the number and essential types used in the operation and the tuple space declaration match, and
   (b) whether the argument types used in the operation combine with the operation;
3. Checking whether the argument types used in the operation combine with the field access patterns from the tuple space declaration;
4. Sending the correct message to the back end; and
5. Proceeding in case of an asynchronous operation.

Because this list is reasonably long, the front end consists of several classes that oversee several of the above mentioned items.

- `TupleSpace`
  Class `TupleSpace` forms the interface and adds the default tuple space access pattern if no tuple space access pattern is specified (item 1 of the preceding list).
- `TupleSpaceTuples`
  Class `TupleSpaceTuples` partitions the tuple space based on the tuple prototype (item 2a).

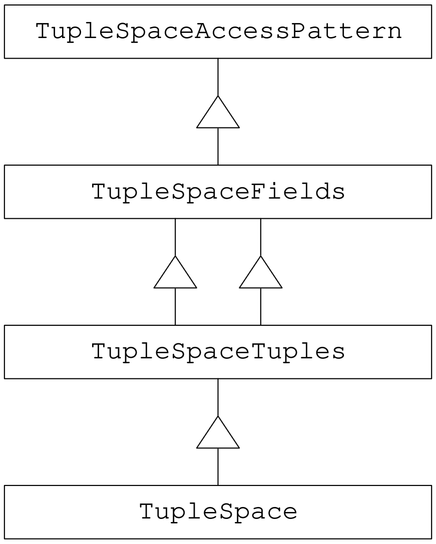


Figure 5.1: The tuple space front end consists of a hierarchy of tuple space classes. All four tuple space classes have a distinct function. Class `TupleSpaceAccessPattern` distinguishes the access patterns, class `TupleSpaceFields` identifies the number of fields, class `TupleSpaceTuples` distinguishes the number of tuples, and class `TupleSpace` gives a nice user interface.

- `TupleSpaceFields`
  Class `TupleSpaceFields` handles everything associated with the fields (i.e., the number of fields, the field access patterns, and the field prototypes). Items 2b and 3 belong to its tasks.

- `TupleSpaceAccessPattern`
  Class `TupleSpaceAccessPattern` handles the actual communication with the back end, thereby sending the correct messages (item 4) and knowing when to wait for arriving messages (item 5).

To take care of all of the items in the preceding list, the tuple space front end classes have to coordinate (see Figure 5.1). Section 7.6 (more specifically Figure 7.1) shows this hierarchy with more implementation details.

5.3 Communication

This section focuses on the different communications that occur between the tuple space front end and back end. The messages being sent depend upon the operation performed on the tuple space.

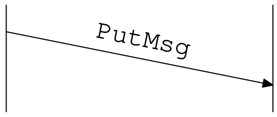


Figure 5.2: Communications due to a `put` operation.
A `put` operation corresponds to a single communication between the tuple space front end on the left and the tuple space back end on the right.

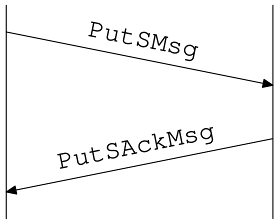


Figure 5.3: Communications due to a `put_s` operation.
A `put_s` operation corresponds to a communication between the tuple space front end on the left and the tuple space back end on the right and a synchronization communication back.

5.3.1 Member Function `put`

The operation `put` inserts a tuple into the tuple space asynchronously, meaning that the program proceeds immediately after the tuple is sent to the tuple space back end. The message with which the value tuple is sent to the back end is called `PutMsg` (see Figure 5.2).

5.3.2 Member Function `put_s`

The operation `put_s` inserts a tuple into the tuple space synchronously, meaning that the program waits after the tuple is sent to the tuple space back end for a confirmation that the tuple is actually inserted in the back end. The message with which the value tuple is sent to the back end is `PutSMsg` and the confirmation message is `PutSAckMsg` (see Figure 5.3).

5.3.3 Member Function `get`

The operation `get` retrieves a tuple from the tuple space synchronously or asynchronously, depending on the field access patterns of its arguments. In both cases two messages are being sent. The first message goes from the tuple space front end to the tuple space back end and contains the query tuple that is being looked for. The return message contains the matching value tuple. In case of a synchronous `get` operation the first message is called `GetMsg`

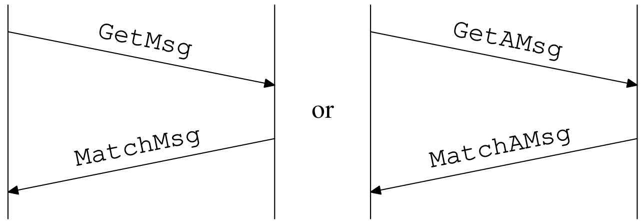


Figure 5.4: Communications due to a `get` operation.
A `get` operation results in two communications between the tuple space front end (left) and back end (right). The first message sends the query tuple to be matched, while the second message sends the matching value tuple.

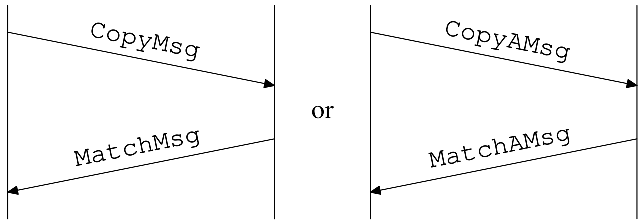


Figure 5.5: Communications due to a `copy` operation.
A `copy` operation results in two communications between the tuple space front end (left) and back end (right). The first message sends the query tuple to be matched and the second message sends the matching value tuple of which a copy remains in the back end.

and the second is called `MatchMsg` (see Figure 5.4). In case of an asynchronous `get` operation the first message is called `GetAMsg` and the second message is called `MatchAMsg` (see Figure 5.4). Of course the `MatchAMsg` updates the future fields asynchronously.

5.3.4 Member Function `copy`

The operation `copy` replicates a tuple from the tuple space. Depending on the field access patterns of its arguments this is done synchronously or asynchronously. In both cases two messages are sent. The first message goes from the tuple space front end to the tuple space back end and contains the query tuple that is being sought. The return message contains the matching value tuple. In a synchronous `copy` operation, the first message is `CopyMsg` and the second is `MatchMsg` (see Figure 5.4). In contrast, for an asynchronous `copy` operation, the first message is called `CopyAMsg` and the second message is called `MatchAMsg` (see Figure 5.4). The `MatchAMsg` updates the future fields asynchronously.

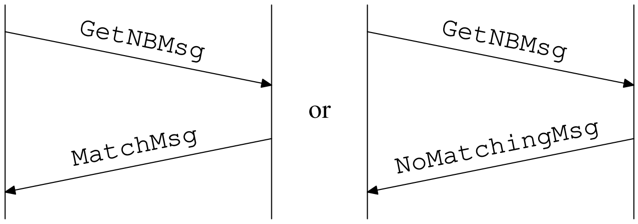


Figure 5.6: Communications due to a `get_nb` operation.
A `get_nb` operation results in two communications between the tuple space front end (left) and back end (right). The first message sends the query tuple to be matched. If a matching value tuple is found at the back end, the second message contains this value tuple. Otherwise, the second message informs the front end that no matching value tuple could be found.

5.3.5 Member Function `get_nb`

The operation `get_nb` tries to retrieve a tuple from the tuple space. Two scenarios exist depending on the presence of a matching value tuple in the tuple space. If a matching value tuple is present in the tuple space back end, the matching tuple is sent back to the tuple space front end. If no matching value tuple is present, the front end is notified. Both scenarios use two messages, where the first (`GetNBMsg`) is the same for both scenarios. If a matching tuple is found, a `MatchMsg` that contains the matching tuple is sent back (see Figure 5.6). Otherwise a notification message called `NoMatchingMsg` is returned.

5.3.6 Member Function `copy_nb`

The operation `copy_nb` tries to replicate a tuple from the tuple space. Two scenarios exist depending on the presence of a matching value tuple in the tuple space. If a matching value tuple is present in the tuple space back end, the matching tuple is sent back to the tuple space front end. If no matching value tuple is present, the front end is notified. Both scenarios use two messages, where the first (`GetNBMsg`) is the same for both scenarios. If a matching tuple is found a `MatchMsg` that contains the matching tuple is sent back. Otherwise a notification message called `NoMatchingMsg` is sent back (see Figure 5.7).

5.4 Back End

The value tuples that are put in the tuple space are kept in the tuple space back end. Depending on the tuple space access pattern, one or more tuple space back ends exist. All tuple space back ends have the same task independent of their number. For ease of comprehension, this section assumes the existence of only one tuple space back end (as if the programmer would only use one homogeneous, local tuple space).

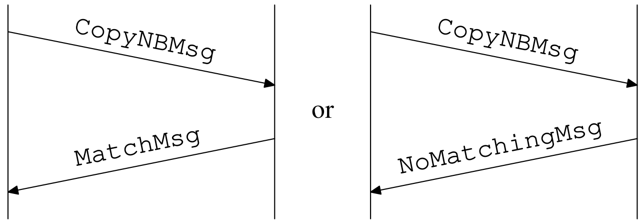


Figure 5.7: Communications due to a `copy_nb` operation.
A `copy_nb` operation results in two communications between the tuple space front end (left) and back end (right). The first message sends the query tuple to be matched. If a matching value tuple is found at the back end, the second message contains this value tuple (leaving a copy of the tuple in the back end). Otherwise, the second message informs the front end that no matching value tuple could be found.

Conceptually a tuple space has been typically perceived to be a storage for value tuples. Therefore, it is very natural to think of a tuple space back end design containing only a storage for value tuples. It appears simply that a `put` operation inserts into this storage a value tuple which is thereafter retrieved by a `get` operation.

However, it is more complex in the case when a `get` operation fails to find a matching value tuple for its query tuple. The `get` operation blocks until a matching value tuple is inserted into the tuple space back end. The `get` operation could relook in the tuple space back end intermittently to see if a matching value tuple has been inserted yet. This process is known as *polling*, resulting in many communications between the tuple space front end and the tuple space back end. This is inefficient. To overcome this performance penalty, the query tuple has to remain in the tuple space back end. As soon as a matching value tuple is found, the matching value tuple should be sent to the process initiating the `get` operation and the query tuple should be instantiated with the value tuple.

The tuple space back end thus has to store value tuples as well as query tuples (with some extra information to find the originating `get` operation). Figure 5.8 shows this design.

The consequence of this design is that the `get` and `put` operations become symmetrical. A query tuple from a `get` operation is being matched against the available value tuples in the value tuple storage. If no matching value tuple is found, the query tuple is stored in the query tuple storage. Similarly, a value tuple from a `put` operation is being matched against the available query tuples in the query tuple storage. If no matching query tuple is found, the value tuple is stored in the value tuple storage.

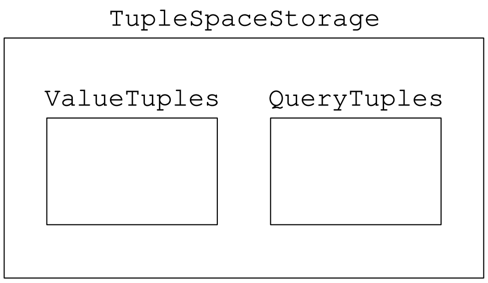


Figure 5.8: The tuple space back end (class `TupleSpaceStorage`).

The operations `put` and `put_s` insert value tuples in the value tuple storage if no matching query tuple is found. The operations `get`, `copy`, `get_nb`, and `copy_nb` insert query tuples in the query tuple storage should no matching value tuples be found.

Chapter 6

Roots

This chapter is a rewrite of [52].

Roots is the communications library on which *Blossom* is built. *Roots*'s design and interface are object-oriented (just like *Blossom*). The *Roots* communications library is based on threads and active messages. In contrast, the PVM and MPI communications libraries are based on processes and peer-to-peer messages.

Even though *Blossom* completely hides *Roots* for the programmer, *Roots* could be used as the foundation to build parallel and distributed programs. That this is a viable solution is shown by *Blossom*. This chapter has a look at the programming interface of *Roots*. Furthermore this chapter shows that *Roots* is significantly faster on both a network of machines as well as on a multi-processor machine.

6.1 Introduction

Given the complexity of parallel programming, tools to simplify the overhead of writing a parallel program are more than welcome. Combined with the trend in programming towards object orientation, this can only lead to one conclusion: There is a need for object-oriented tools for parallel computing.

One such tool is a communications library. The ideal tool should incorporate the following features:

- Speed: A communications library has to perform well, because communication can be a significant overhead in a parallel program.

- Object-Oriented Interface: An object-oriented library is easier to incorporate in an object-oriented program.

- Ease of Learning: The interface of the communications library should be simple so programmers can easily learn and remember it.

- Ease of Usage: A communications library should be easy to use, i.e., it should minimize the amount of code a programmer has to write.

- Versatility: A program developed with a communications library should be able to run on multi-processor machines as well as on a network of workstations.

- Threads: More and more applications are using threads and hence full thread support is mandatory for a communication library.

Several communications packages are available, among the better-known are Parallel Virtual Machine (PVM) [17] and Message Passing Interface (MPI) [21]. Both are widely used but have interfaces that were designed for non-object-oriented programming languages. Another communication library ACE [43] wraps communication system calls in objects. ACE does this in an operating system independent way, so that programs written using the ACE library only need to be recompiled for another platform. However, the ACE interface is low level which makes it hard to use and learn. The parallel language ABC++ [39, 57] is built on top of a smart message communication library. Again this library is reasonably low level which makes the programmer write many lines of code. Finally, the run time system Converse [27] allows for interoperability between code fragments written in different parallel programming paradigms allowing for better modularity and reuse of programs.

All of the above communication packages are reasonably low level. The programmer has to explicitly marshal data into messages, and almost all communication packages (except for ABC++'s smart message library) require explicit receipt by the programmer. This chapter introduces *Roots*, a high performance C++ communications library for parallel object-oriented programs. *Roots* addresses the above concerns. Section 6.2 describes active messages and their use in *Roots*. Section 6.3 shows how to define and spawn threads in *Roots*. Section 6.4 describes how the active messages are implemented in *Roots*. Section 6.5 compares programming with *Roots* and PVM.

6.2 Communication primitives

Roots is a high performance, object-oriented communications package. One of the main reasons for developing *Roots* was the lack of thread support in PVM and MPI. It is possible to use threads in these libraries but their communication primitives can be used only in one thread simultaneously. Either only the main thread can call the PVM and MPI primitives or threads have to acquire a global lock before calling a primitive [20]. This is a disadvantage to programmers using PVM or MPI in conjunction with threads. If a thread wants to use any of the communication primitives, for example, to send a message, it has to put a request in a queue and the main thread has to service the requests in the queue. This scheme is awkward and its implementation will take programming effort and the execution has overhead, i.e., is slow. *Roots* provides full support for threads allowing each thread to use all of *Roots*'s communication primitives.

The main communication primitives found in *Roots* are the subject of the remainder of this section.

6.2.1 Active Messages

Roots provides *active messages* [55]. An active message contains an action that is executed by the receiving process. Consequently, the *Roots* programmer only needs to send an active message to a process; the receipt of an active message is implicitly taken care of by *Roots*. This is in contrast with *peer-to-peer messages* where the programmer explicitly has to receive every sent message (*à la* PVM and MPI).

To define an active message in *Roots*, the programmer has to derive a class from base class `Message` that is provided by *Roots*. For clarity, the definition of class `Message` is shown here.

```
class Message {
public:
   virtual unsigned int size() const = 0;
   virtual void action() const = 0;
};
```

Class `Message` merely provides two pure virtual member functions. A virtual member function (indicated by the `virtual` keyword) can be redefined in a derived class. A pure virtual member function (indicated by "`= 0`") indicates that this member function is not defined in the base class and has to be redefined in a derived class.[1] Thus an active message, which has to be derived from class `Message`, has to provide two member functions. The first member function `size()` should return the size in bytes of an object of the derived active message class. The second member function `action()` should specify the action that is performed by the receiving process.

Consider the following example. Class `HelloMsg` defines a message that writes a message on the screen.

```
class HelloMsg: public Message {
   int from_host;
public:
   HelloMsg(int h): from_host(h) {}
   unsigned int size() const {
      return sizeof(HelloMsg);
   }
   void action() const {
      cout << "Hello from host " << from_host <<
              " on host " << host_id() << endl;
   }
};
```

The definition of class `HelloMsg` contains definitions for the member functions `size()` and `action()`. The member function `size()` simply calls the `sizeof` operator to find

[1] This explanation of *pure* is not completely accurate, but it suffices for our needs.

out the size of an object of class `HelloMsg`. The member function `action()` writes a message to the screen. It calls function `host_id()` which is a *Roots* function that returns the identifier of the current host.[2]

6.2.2 Sending Active Messages

Sending the message `HelloMsg` to the main process on a distributed machine (the process running on the main host with `host_id() == 0`) or to the main thread on a shared memory machine (the thread running on main host) is done as follows.

```
host[0] << HelloMsg(host_id());
```

The variable `host` is an array containing information of where and how to find the other hosts. Since some programmers do not like overloaded C++ operators (operator << in this case), a traditional object-oriented interface is provided as well.

```
host[0].send(HelloMsg(host_id()));
```

The creation of the object of class `HelloMsg` takes care of marshaling the message data that is done explicitly in PVM and MPI. Thus the user code needed to send an active message in *Roots* is less than sending a peer-to-peer message in PVM and MPI.

6.2.3 Receiving Active Messages

The peer-to-peer messages used in PVM and MPI have to be explicitly received by the programmer. Hence, after PVM and MPI receive a message implicitly, the message has to be stored for explicit receipt by the programmer (see PVM column in Table 6.1). In contrast, *Roots* can immediately execute the action (member function `action()`) that belongs to the active message after it is received implicitly (see *Roots* column in Table 6.1). Consequently, the programmer does not have to do anything upon receipt of an active message, i.e., he does not have to write any code for receiving active messages.

The amount of code that a PVM or MPI programmer has to write to receive messages explicitly can be considerable. First, after most PVM calls, the programmer has to check for a possible return of an error. Second, the programmer has to differentiate among all the messages that can be sent. This is usually solved by including a message identifier in the peer-to-peer message. Upon explicit receipt of a message, this identifier specifies which action has to occur. This usually means that a `switch` statement is used to distinguish between all possible values of the message identifier and perform the corresponding action.

From a software engineering point of view, the PVM/MPI `switch` that is needed to differentiate message types is bad programming practice. It separates the construction of the message from the action of the message in the program. In *Roots*, both are specified together in the declaration of the active message class.

Table 6.1 further shows that PVM internally uses an extra copy for receiving messages.[3] This copy copies the message data from the system memory to the user memory and is not

[2] In *Roots* the user specifies the processors (hosts) on which the program will run.

[3] Since MPI uses peer-to-peer messages, it uses this extra copy as well.

Table 6.1: Peer-to-peer messages in PVM versus active messages in *Roots*.

		PVM	ROOTS
SENDING SIDE	USER	pvm_initsend()	host[i] << HelloMsg(...)
		pvm_pack()	
		pvm_pack()	
		pvm_send()	
	SYSTEM	write()	write()
RECEIVING SIDE	SYSTEM	read()	read()
		write()	msg.action()
	USER	pvm_recv()	
		pvm_unpack()	
		pvm_unpack()	
		switch() {...}	

needed when using active messages, which means that *Roots* can potentially run faster than PVM or MPI.

6.2.4 Variable Sized Active Messages

So far active messages are assumed to have a fixed size. However, if a programmer wants to send a variable size string (up to a maximum `MAX_SIZE`), one solution is as follows.

```
class COutMsg: public Message {
   char buffer[MAX_SIZE];
public:
   COutMsg(char* str) {
      strncpy(buffer, str, MAX_SIZE-1);
      buffer[MAX_SIZE-1] = '\0';
   }
   unsigned int size() const {
      return sizeof(COutMsg);
   }
   void action() const {
      cout << buffer << endl;
   }
};
```

In this solution, the active message contains a buffer that is only partially filled with a string. This means that for every string, small or large, the complete buffer is sent across the network. If a programmer wishes to send a short string, many unnecessary characters are sent across the network. *Roots* provides a way to send fewer bytes.

By making the member function `size` not return a constant (as was the case in all the previous examples) but the value of the actual message size, fewer bytes can be sent across the network. However, the message handling agent on the other side also requires the size of the message. The size, thus, should be the first data member sent across the network. As a result all messages, even the fixed sized messages, have a size of (at least) one pointer and one `int`.

```
class COutMsg: public Message {
   unsigned int msg_size;   // first data member!
   char buffer[MAX_SIZE];   // last data member!
public:
   COutMsg(char* str) {
      strncpy(buffer, str, MAX_SIZE);
      buffer[MAX_SIZE-1] = '\0';
      msg_size = align_ceiling(sizeof(int)+strlen(buffer)+1);
   }
   unsigned int size() const {
      return msg_size;
   }
   void action() const {
      cout << buffer << endl;
   }
};
```

If an active message must refer to a remote object that is not global, the address of the remote local object should be contained in the active message. This usually makes *Roots* messages slightly bigger than those of PVM or MPI.

If a programmer wants to obtain data from a remote thread, he has to send an active message that returns an active message with the required data (also known as a completion message). Synchronization mechanisms are required to update the local variable with the remote data. *Roots* provides Posix mutexes (locks), semaphores and condition variables [46].

6.3 Threads

Roots provides an object-oriented interface to Posix threads [32]. This section introduces the main features of *Roots*'s threads.

Roots provides a base class `PThread` from which user-defined threads have to derive. The activities that a thread has to perform are provided in a member function called `main()`.

```
class PThread {
   ...
public:
   virtual void main() = 0;
};
```

The programmer has to define this function `main()` in a derived class. The `main()` function can make use of the data members that are initialized during the construction of the thread object. The following class definition shows a thread with the task to send an active message to the main process or the main thread (running on the host with id `0`).

```
class HelloThr: public PThread {
public:
   HelloThr() {}
   void main() {
      host[0] << HelloMsg(host_id());
   }
};
```

To actually create a thread, it has to be spawned.

```
Spawn<HelloThr>(1)();
```

This line states that an object of class `HelloThr` should be created on the host with id[4] `1` by using the default constructor (hence the empty braces `()` at the end). Since the usage of two pairs of braces may be confusing for programmers unfamiliar with C++, the following two lines have the same effect:

```
Spawn<HelloThr> hello_thr(1);
hello_thr();
```

Threads can send and receive messages, without the limitations of PVM and MPI.

6.4 Implementation

Previous sections introduced us to *Roots* and showed what a programmer has to do when using a communications library based on peer-to-peer messages. In contrast, *Roots* accomplishes most of these tasks for the programmer. How *Roots* takes care of this is the topic of this section.

For *Roots* the identity of an object merely consists of two pieces of information: the size of the message and the action that belongs to the active message. With the `size()` member function, the receiving process knows how many bytes have to be read to receive the complete active message. Once the active message is completely received, the `action()` member function is called.

Roots does not send these two pieces of information together with the active message to the other host explicitly. It does so implicitly. These two pieces of information are glued to an active message. Or, better, they are glued to the type of the active message (the class) and the type is partially preserved when sending an active message to another host. Together with the knowledge that only active messages (objects of classes derived from class `Message`) are sent, the correct type can be reconstructed at the receiving side.

Let's make the previous paragraph more explicit. A C++ object with virtual member functions contains more than just its member data, namely a so-called *virtual function table*

[4] *Roots* numbers all hosts participating in the parallel execution starting with `0`.

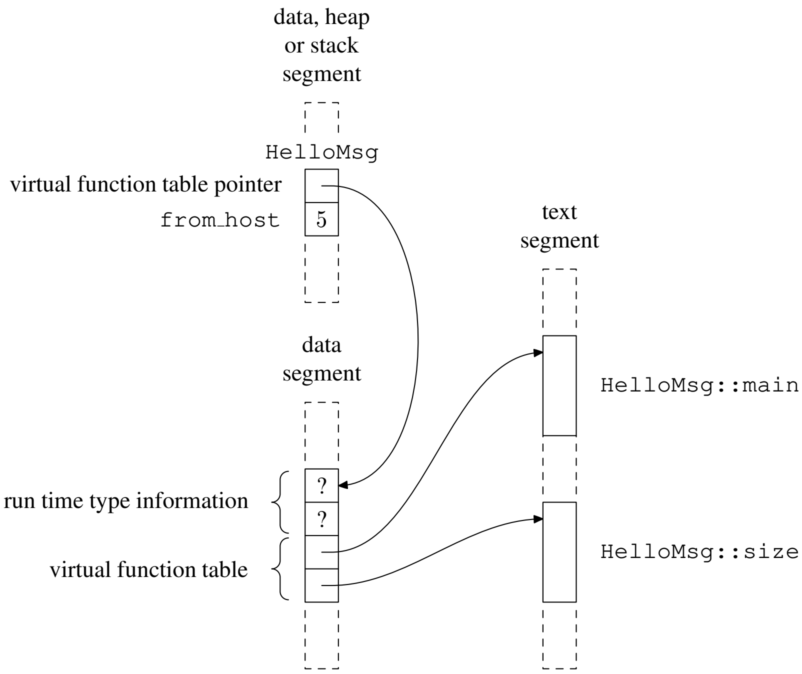


Figure 6.1: Representation of active message `HelloMsg` in memory.

pointer (see Figure 6.1). This virtual function table pointer points to a data structure that contains a virtual function table, which itself contains pointers to virtual member functions. These pointers are dereferenced whenever a virtual function is called.

Thus when sending a C++ object with virtual functions to another host, a programmer sends a pointer (the virtual function table pointer) to the other host as well. Usually this pointer will point to something inappropriate in the other process. However, if the receiving process runs the same executable as the sending process, the pointer will point to the same virtual memory address, at which the correct virtual function table can be found in both processes. All initialized global data (and thus the virtual function tables) are put in the data segment and have fixed virtual addresses in one executable, no matter on which processor you run the executable.

The *Roots* implementation consists of almost 3000 lines of C++ code. *Roots* uses TCP/IP to communicate between two processors on different machines and shared memory to communicate between two processors on the same machine. Instead of sending an active message to a thread on the same machine, the action part of the active message is executed immediately.

6.5 Example: Traveling Salesman Problem

To compare *Roots* with PVM, the traveling salesman problem was implemented using *Roots* (version 1.0) and PVM (version 3.3.11). Both programs solve the same sixteen city traveling salesman problem using the "closest neighbor first" heuristic (also known as greedy TSP). The programs use a work farm model to calculate the shortest cycle. The number of jobs generated is $15 \times 14 \times 13$, which corresponds to partial paths where the first four cities are fixed. This results in at least four communications per job: two communications for getting the job and two for getting the global minimum.[5] Additional messages are sent when a process believes it has found a new global minimum. This generates a substantial amount of communication, which is exactly what is desired because we are mainly interested in the communication. The program however stays CPU bound as demonstrated by the almost linear speedups. The size of the messages used to solve TSP concurrently is small (excluding the header, at most 76 bytes for PVM and 88 bytes for *Roots*).

- Speed: Both implementation of TSP were run on a network of Sun Ultra 10s, and on a shared memory SGI Power Challenge with 8 processors. The resulting timings can be found in Table 6.2. The big difference in timings between the sequential and single processor parallel is caused by the communication overhead. Since this is a comparison between communication packages, the number of communications is artificially increased by a factor of four. Both the timings on the shared memory machine as well as on the distributed memory machine differ considerably. For TSP, *Roots* is on average 24% faster than PVM on a distributed memory machine and on average 18% faster on a shared memory machine. The difference between the timings is explained as follows. First, communication in PVM goes through a PVM daemon, which causes extra overhead. The daemon is needed however to spawn processes, even though in the TSP example no spawning is used. Second, PVM copies the receiving message between the implicit and explicit receipt of a message; *Roots* doesn't do so. Some versions of MPI [49] do not copy the receiving message, however the statement to receive a zero-copy message differs from those in the normal case, which adds to programmers confusion.

  Note that if the size of the messages increases the timing difference on the distributed memory machines will increase because PVM uses an extra copy of the message between implicitly and explicitly receiving a message.

- Object-Oriented Interface: *Roots* has an object-oriented interface, whereas PVM has an imperative interface.

- Ease of Learning: *Roots* uses fewer primitives to communicate than PVM. On the other hand, *Roots* needs synchronization primitives to coordinate its processes. PVM uses only the explicit receipt of messages as its main synchronization primitive.

- Ease of Usage: The number of lines of code in the *Roots* program is 7% less than those in the PVM program. The messages sent in *Roots* are bigger, since active messages

[5] These could be reduced to two communications that simultaneously get the job and the minimum.

Table 6.2: Averaged timings for the Traveling Salesman Problem on a network of Sun Ultra 10s (distributed memory) and on an SGI Power Challenge with 8 processors (shared memory).

	Distributed Memory		Shared Memory	
Processors	PVM	Roots	PVM	Roots
sequential 1	52.5	52.5	127.5	127.5
parallel 1	241.7	183.8	506.0	409.2
2	125.3	94.0	252.8	209.0
3	83.8	63.5	171.5	142.0
4	63.3	48.2	127.5	106.0
5	50.8	38.5	102.3	85.8
6	43.0	32.8	86.8	72.0
7	36.8	28.2	76.7	63.0
8	32.3	24.7	68.3	55.8
9	28.7	22.2	–	–
10	25.8	20.0	–	–

have to contain references to remote objects. Synchronization is easier to implement in PVM than in *Roots*.

- Versatility: Both communications libraries run on shared memory as well as distributed memory machines. In addition, PVM runs on a heterogeneous network of machines, making it more versatile.

- Threads: Only *Roots* supports threads fully. As explained, PVM and MPI can use threads but at the cost of programming comfort.

6.6 Conclusions

Roots is one of the new generation of parallel programming tools. It is object-oriented, has full thread support, and outperforms PVM on a network of machines.

MPI is supposedly a bit faster than PVM; however, MPI copies the incoming message for explicit receipt just like PVM, so it will most likely not outperform *Roots* on a network of machines.

The communication between hosts on different machines uses TCP/IP at present. However since TCP/IP is rather slow for local networks, *Roots* should ideally be built on top of UDP/IP (faster for local networks). However, making reliable communication on top of UDP/IP is laborious.

Chapter 7

Blossom Implementation

Blossom depends heavily on type parameterizations. Type parameterization is a form of polymorphism. Section 7.1 gives a overview of different varieties of polymorphism. Section 7.2 introduces type parameterization in C++ and all its different appearances. Section 7.3 shows how basic type parameterization is used in *Blossom*'s tuples. Section 7.4 describes the implementation of the tuple space back end. Section 7.5 looks at the details of *Blossom*'s communication. Section 7.6 has a closer look at the implementation details of the tuple space front end. In order to keep things simple for the reader some advanced type parameterization tricks remain unspecified in this section. These advanced type parameterization tricks are the topic of Section 7.7.

7.1 Type Parameterization

Type parameterization is one of four varieties of polymorphism [10].

Coercion A coercion is a semantic operation that is needed to convert an argument to the type expected by a function.

Overloading Overloading polymorphism is used to attach several functions to one function name. The context (usually the types of the arguments) decides which function is denoted.

Inclusion Inclusion polymorphism allows objects to belong to multiple classes, namely classes may include each other. An inclusion polymorphic function specifies which type the arguments should have. Objects of all included types are allowed to be passed to this function.

Parametric Parametric polymorphism allows a function to have implicit or explicit type parameters. The types of the arguments for this function are determined for each application of that function.

7.2 Type Parameterization in C++

C++ provides all variations of type parameterization introduced in the previous section.

Coercion Coercion was already introduced in C. For example, an `int`eger is converted to a `float`ing point number whenever necessary.

Overloading Any object oriented language provides overloading polymorphism and so does C++.

Inclusion Inheritance allows objects to be from multiple classes. C++ even allows multiple inheritance, even though this not a requirement for inclusion.

Parametric polymorphism is not found in many languages (at the moment). How C++ provides it is the topic of the remainder of this section.

7.2.1 Traditional Classes

This section shows the shortcomings of programming languages without type parameterization by assuming C++ does not support it. A simple implementation of a hash table is used as an example.

```
class HashTableInt {
   int* elems;
   size_t size;
public:
   HashTableInt(size_t s): size(s), elems(new int[s]) {}
   ~HashTableInt() {
      delete elems;
   }
   void insert(int elem) {
      elems[hash_int(elem) % size] = elem;
   }
   bool find(int elem) {
      return (elems[hash_int(elem) % size] == elem);
   }
   ... // other member function
};
```

A similar class must be written for a hash table of `Objects` (where `Object` is a class that is defined by the user).

```
class HashTableObject {
   Object* elems;
   size_t size;
public:
   HashTableObject(size_t s): size(s), elems(new Object[s]) {}
   ~HashTableObject() {
```

```
      delete elems;
   }
   void insert(const Object& elem) {
      elems[hash_Object(elem) % size] = elem;
   }
   bool find(const Object& elem) {
      return (elems[hash_Object(elem) % size] == elem);
   }
   ... // other member function
};
```

The preceding class definitions look similar (differences are indicated in italics). Most differences are in the names of the classes (`TupleSpaceInt` versus `TupleSpaceObject`), the names of the functions (`hash_int` versus `hash_Object`), the types of the stored elements (`int` versus `Object`) and the types of the function parameters (`int` versus `const Object&`). The last difference requires explanation.

In C++ (and C), arguments to functions can be passed by value or by reference. Arguments passed by value cannot be altered, whereas arguments passed by reference can be changed by the function being called. In C, structures (more or less equivalent to C++ objects) can be passed by reference only. This mainly has to do with performance, because it is costly to copy a whole structure on the stack. Instead, a pointer to the structure is passed to a function (this usually shows up in the code as an address of operator (`&`)). In the function, this pointer needs to be dereferenced again (this usually shows up in the code as a dereference operator (`*` or `->`)). C++ allows objects (and structures) to be passed by value, but most C++ programmers do not use this option because of performance issues. C++ allows programmers an alternative way to pass objects by reference, namely by so called *references*, which is almost equivalent to a pointer. A reference has three main advantages over pointers:

- When passing an object by reference, the programmer does not have to pass the address of it (with operator `&`) because the C++ compiler does this for him.

- When using a referenced object in a function, the programmer does not have to dereference it (with operator `*` or `->`) because the C++ compiler does this for him.

- The C++ compiler ensures that a reference always points to something (an object or a value of native type (`int`, `float`, *etcetera*)). When passing an object by a pointer the programmer always has to check whether the pointer does not point to `0` (`NULL` for C programmers).

In function prototypes, references are indicated with an ampersand (`&`). The ampersands in the member functions `insert` and `find`, indicate references. The keyword `const` indicates that the object (that is passed by reference) is not changed inside the member function.

7.2.2 Templated Classes

The previous two code examples are almost identical, with the only differences being the names of the classes, the names of functions, the types, and the parameters. All differences were indicated in italics.

In case the programmer would like to add a hash table for objects of a class he designed himself, he would end up with almost the same code and similar differences as above. Type parameterization allows him to write this code only once and reuse it for whatever type or class he would like.

Type parameterization is referred to as templating in C++. A type parameterized class is called a templated class in C++. Instead of writing the above hash table code twice or as many times as you have different classes you want to store in a hash table, you have to write the above code only once when using C++ templates.

A templated hash table class parameterizes the type of the object (or native C++ type) so that only one definition is needed for any number of classes. In the following example the parameterized type is called `C` as shown in the first line.

```
template <class C>
class HashTable<C> {
   C* elems;
   size_t size;
public:
   HashTable(size_t s): size(s), elems(new C[s]) {}
   ~HashTable() {}
   void insert(const C& elem) {
      elems[hash(elem) % size] = elem;
   }
   bool find(const C& elem) {
      return (elem[hash(elem) % size] == elem);
   }
   ... // other member functions
};
```

The difference between the two type-specific programs in Section 7.2.1 and the type-independent programs in this section are indicated in italics. Most notable are the changes for the type names into a type parameter name (`C`).

A type parameterized class can be compared to a value parameterized function. For every function call, the appropriate values (all of the same type) are substituted in the parameters of the called function. Instead of writing separate code to square the integer `1`, `2`, `3`, *etcetera*, you write a function to square any integer. A function value parameterizes a certain operation, so you do not have to write similar code repeatedly. Similarly for type parameterization, you write a class to store any type instead of writing separate code to store integers, floats, *etcetera*, in hash tables. This class parameterizes the type you want to store in the hash table.

There is a technical difference between value parameterized functions and type parameterized classes. Functions are implemented by using a stack (of values), which allows every function to exist only once in the compiled program, though it may be called many times. Since a stack of types does not exist, templated classes have to be instantiated for every type you wish to use them. This means that the executable has separate code for every type you wish to insert in a tuple space.

Templated classes are used as follows:

```
HashTable<int> hash_table_1;
HashTable<float> hash_table_2;
HashTable<UserObject> hash_table_3;
```

This program excerpt declares three variables. Variable `hash_table_1` is a hash table that can contain `integer`s, variable `hash_table_2` is a hash table that can contain `float`ing point numbers, and variable `hash_table_3` is a hash table that can contain objects of the user defined class `UserObject`.

More elaborate examples of templated classes are presented later in this chapter.

7.2.3 Templated Functions

Besides templated classes, C++ also allows the programmer to type parameterize functions. The definition of a templated function is similar to that of a templated class.

```
template <class C>
size_t hash(const C&);
```

This program excerpt informs us about a function `hash` that takes one argument of any type and returns an integer (`size_t`). In C++ this is called a templated function prototype.

Elsewhere in the program a function declaration may be found; for example:

```
template <class C>
u32_t hash(const C& elem) {
   size_t value = 0;
   for (size_t s = 0; s < sizeof(C) / sizeof(u32_t); s++) {
      value *= 11;
      value += ((u32_t*) &elem)[s];
   }
   return value;
}
```

This templated function declaration provides declarations for all possible types of arguments. Substitute type parameter `C` with the type of the argument required and the resulting function declares precisely what to do when one argument of the substituted type is encountered (just as was the case in Section 7.2.2 with templated classes).

Using templated functions is even easier than using templated classes. Whereas the programmer has to specify the type in the type parameterized class, he does not have to specify the type for a type parameterized function. The compiler will deduce it for him and automatically choose the corresponding type parameter.

```
int i;
float f;
UserObject user_object;

cout << hash(i);
cout << hash(j);
cout << hash(user_object);
```

The first call to function hash is parameterized with an `int`eger, the second `hash` function call with a `float`ing point number, and the third `hash` function call with a `UserObject`. Further examples on templated functions are presented later in this chapter.

7.2.4 Template Specialization

It is also possible to have different implementations of function `hash` (whose prototype was encountered in Section 7.2.3). A templated function definition of this kind is known as a *template specialization* (or specialization for short).

```
u32_t hash<int>(const int& i) {
   return i;
};

u32_t hash<float>(const float& f) {
   return (u32_t) f;   // assumes sizeof(float) == sizeof(u32_t)
};
```

The preceding `hash` functions specialize the general templated declaration found in Section 7.2.3. Even if the general type parameterized function was omitted, the above function is called a specialization function that specializes the function definition (prototype).

7.2.5 Partial Template Specialization

The last thing to consider about templating at present is called *partial template specialization* (partial specialization for short). This as opposed to *full template specialization* encountered in the previous section. A partial specialized function (or class) specializes part of a type parameter.

```
template <class T>
u32_t hash<T*>(const T* obj) {
   return hash<T>(*obj);
};
```

The preceding definition describes the behavior of function `hash` with a pointer argument. It returns the hash value of the dereferenced pointer.

Another, more familiar variant of partial specialization occurs when specializing one type parameter of a function (or class) that is templated with two or more type parameters.

A templated function (or class) prototype defines infinitely many functions (or classes). A (fully) specialized templated function (or class) declares one function (or class). A partial specialized template function (or class) declares infinitely many functions (or classes), but leaves infinitely many functions (or classes) undeclared.

When deciding which templated function declaration to choose for a given call, C++ selects the most specific (less generic) function that matches the prototype. If a fully specialized declaration matching the calling function exists, it will be chosen. Otherwise, if a partially specialized declaration that matches exists, it will be chosen. If neither exists or matches, the unspecialized declaration is chosen. If no matching specialization exists the C++ compiler will generate an error. Within the class of partially specialized declarations a whole specialization hierarchy may exist (one versus two unspecialized type parameters, *etcetera*).

7.3 Tuples

C++ templates are indicated with the signs < and > of which *Blossom* makes extensive use. The *Blossom* syntax described in Appendix A is full of these < and > signs.

Class `Tuple` is a general structure that can be able to contain many different types. A tuple sometimes has one field, sometimes two fields, sometimes it contains an integer, sometimes a float. Moreover, class `Tuple` should be implemented efficiently with respect to its size, since many `Tuples` should be stored in `TupleSpaces`. If type parameterization would not be available, class `Tuple` can only have a general, i.e., less (code and data) efficient, implementation. Let's have a look at an example.

```
template <class F0, class F1>
class Tuple {
   F0 field0;
   F1 field1;
public:
   Tuple(const F0& f0, const F1& f1):
      field0(f0), field1(f1) {}
};
```

Class `Tuple` is type parameterized (templated) with two type parameters `F0` and `F1`. These parameters can be used anywhere inside the `Tuple` definition where a class name can appear (hence the prefixes `class` in the first line). Data members `field0` and `field1` are defined to have the types of the template parameters `F0` and `F1` respectively. Finally in the constructor the same template parameters are used as constructor parameters to initialize the data members. Note that the types of the constructor arguments are "identical" to the types of the data members. The `const` prefix and the `&` suffix are C++ gibberish and were shortly explained in Section 7.2.1. If these C++ constructs are unfamiliar, it may be assumed that both are omitted.

A programmer can use this `Tuple` definition as follows:

```
Tuple<int, int> tuple1(1, 2);
```

```
Tuple<Tuple<int, int>, int> tuple2(tuple1, 3);
```

Object `tuple1` has two integers as fields that initially contain the values `1` and `2` respectively. Object `tuple2` that has a tuple with two integers and an integer as fields; the tuple is initialized with the value of object `tuple1` and the integer to `3`.

The C++ compiler indicates when the user makes a type error.

```
Tuple<int, int> tuple3(1);
      // ERROR: no instance of constructor
      //        "Tuple<int, int>::Tuple"
      //        matches the argument list
Tuple<Tuple<int, int>, int> tuple4(1, 2, 3);
      // ERROR: no instance of constructor
      //        "Tuple<Tuple<int, int>, int>::Tuple"
      //        matches the argument list
Tuple<int> tuple5(1);
      // ERROR: too few arguments for class template "Tuple"
```

7.3.1 Default Template Arguments

Blossom's syntax description in Appendix A allowed `Tuples` to have any number of template arguments. Yet, the `Tuple` definition in the previous section did not allow that. It needs exactly two type arguments as can be seen by `tuple5` in the last example of the previous section.

It is easy to rename the above class `Tuple` to `Tuple2`, and introduce similar classes `Tuple0` and `Tuple1` with 0 and 1 type parameters respectively. However this is not desired since this would change the *Blossom* syntax considerably; not only every `Tuple` but also every `TupleSpace` needs an additional number (to indicate the number of tuple prototypes that follow) as well. This technique is quite old fashioned.

With C++'s default template arguments we can overcome this problem.

```
class NoField {};

template <class F0 = NoField, class F1 = NoField>
class Tuple {
   F0 field0;
   F1 field1;
public:
   Tuple() {}
   Tuple(const F0& f0):
      field0(f0) {}
   Tuple(const F0& f0, const F1& f1):
      field0(f0), field1(f1) {}
};
```

With this definition of class `Tuple` a programmer can put up to two fields prototypes in between the < and > signs.[1] If the programmer provides less than two type arguments the

[1] The real *Blossom* library provides tuples with more fields.

last parameters are assumed to be of type `NoField` as indicated by the default template arguments after the = sign.

A tuple with less than two template arguments automatically gets member data with type `NoField`. An object of type `NoField` does not have any data, however in C++ the smallest object size is 1 byte. Objects of type `Tuple` with less than two template arguments contain these 1 byte `NoField` objects and are thus a little bigger than required.

With this new definition of class `Tuple`, the following piece of code is now accepted by a C++ compiler.

```
Tuple<> tuple0();
Tuple<int> tuple1(3);
Tuple<int, int> tuple2(3, 1);
Tuple<Tuple<int>, int> tuple3(tuple1, 1);
```

In some cases the C++ compiler gives the expected error messages when the programmer makes mistakes.

```
Tuple<int, int> tuple4(3, 1, 4);
      // ERROR: no instance of constructor
      //        "Tuple<int, int>::Tuple"
      //        matches the argument list
```

The class type of object `tuple4` contains two integers and the constructor should have two integers to initialize such an object. However the above definition of class `Tuple` still gives rise to erroneous programs as can be seen in the following example.

```
Tuple<int, int> tuple5(3);
```

The instantiation of object `tuple5` should not be allowed by the compiler. The next section explains how to overcome this problem.

7.3.2 Template Specialization

C++ allows the programmer to specialize a templated class declaration. The specialized class should be more specific than the original templated class, i.e., one or more template parameters should be instantiated. Assume the following templated class `Tuple` declaration.

```
template <class F0 = NoField, class F1 = NoField>
class Tuple;
```

This declaration indicates that every `Tuple` has two template parameters. If the programmer specifies less the remaining template parameters become `NoField` by default.

The following definition gives the default behavior of the above class `Tuple` declaration.

```
template <class F0, class F1>
class Tuple {
```

```
    F0 field0;
    F1 field1;
public:
    Tuple(const F0& f0, const F1& f1):
        field0(f0), field1(f1) {}
};
```

`Tuple` classes with fewer fields are specializations of the `Tuple` declaration.

```
template <class F0>
class Tuple<F0> {   // Tuple<F0, NoField>
    F0 field0;
public:
    Tuple(const F0& f0): field0(f0) {}
};
```

This definition of class `Tuple` partially specializes the `Tuple` declaration. Since all `Tuple` classes should have two template parameters (as specified in the declaration), the second template parameter gets the default argument `NoField` (specified in the declaration as well). The ability to specify fewer template arguments than needed by the declaration is exactly what is needed.

The specialization for objects of class `Tuple` with no template arguments is as follows:

```
template <>
class Tuple<> {   // Tuple<NoField, NoField>
public:
    Tuple() {}
};
```

After all these definitions, class `Tuple` responds as expected:

```
Tuple<> tuple0();
Tuple<int> tuple1(2);
Tuple<int, int> tuple2(2, 7);
Tuple<int> tuple3();
        // ERROR: no instance of constructor "Tuple<int>::Tuple"
        //        matches the argument list
Tuple<int> tuple4(2, 7);
        // ERROR: no instance of constructor "Tuple<int>::Tuple"
        //        matches the argument list
```

An additional advantage of these `Tuple` definitions is that an object of type `Tuple` is as small as possible, i.e., it does no longer contain data members of type `NoField` that have a minimal size of 1 byte, because the definitions of class `Tuple` that allow fewer template arguments declare fewer data members.

7.4 Tuple Space Back End

The tuple space back end is the part of the tuple space where the tuples are actually stored. In case of a `Local` tuple space the back end consists of one storage. In a `Distributed` tuple space the back end consists of one storage per process. Class `TupleSpaceAccessPattern` creates all storages necessary for the access pattern. The creation of these storages is described in more detail in Section 7.5.

This section examines the tuple space storage for `Local` and `Distributed` tuple spaces. If in the future of the *Blossom* project another tuple space access pattern is added, other tuple space storages may turn out to be convenient.[2]

7.4.1 Class `TupleSpaceStorage`

After putting a tuple in a tuple space, this tuple space should be able to regenerate the tuple for future query calls (`get`, `copy`, `get_nb` and `copy_nb`). This means that tuples that are `put` in the tuple space need to be stored. When a query call is made, the tuple in the tuple storage that matches the query tuple should be found.

Several storage paradigms exist, not all are as suitable.

Unordered List Implementing an unordered list is straightforward. Putting a tuple in a list simply amounts to adding the tuple to the back (or front) of the list. Searching a matching tuple in a list is cumbersome. In the average case half the list has to be searched before a matching tuple is found. Worse, if no matching tuple is present in the list the whole list has to be searched. Unordered lists are not a good solution for tuple spaces, since they are badly scalable.

Ordered List Searching an ordered list is more efficient than searching an unordered list. An ordered list of size n needs at most $\log n$ searches to know whether a matching tuple is available or not. This is a big efficiency win over unordered lists. Storing tuples in an ordered list is however not as easy as storing tuples in an unordered list. By using more space than required, storing tuples in an ordered list becomes faster (for example using B-trees).

Hash Table This is potentially the most efficient store and retrieve storage available. However it is not always possible to use a hashed storage. To be able to use a hashed storage a key needs to be present for every value and query tuple. The key is used to calculate a *hash value*, which is used to find a bucket to store or retrieve the tuple. Ideally every bucket contains at most one tuple, which results in only 1 search for a tuple storage of size n. If buckets contain more than one tuple the buckets can be either unordered lists, ordered lists, or again hash tables (with another hash function). Hash tables are another example of how a program can gain speed by using more memory.

[2]This is the case for the recently added tuple space access pattern `Replicated`.

It should be obvious that hash tables would be the right storage structure for tuple spaces. However can hash tables be used for tuples, i.e., do tuples contain a key that can be used to calculate a hash value?

Fields with field access pattern `Value` have a value in value tuples as well as query tuples. The values of these `Value` fields can be used as a key to calculate a hash value. If two tuples with the same prototype have identical keys, their hash values are equal. This means that query tuples are looking for matching value tuples in the correct bucket.

The query operations `get` and `copy` look in the storage to find a matching value tuple. If no tuple is found, the front end operations have to block. The query tuple has to remain in class `TupleSpaceStorage` until a matching tuple is inserted in the `TupleSpace-Storage`.

This creates a nice kind of symmetry between the `put` operations on the one hand and the query operations on the other hand. A `put` operation first has to look in the storage of query tuples whether there are still outstanding query tuples that match. If no matching query tuple is found, the value tuple can be inserted in the value tuple storage. Query operations first look in the value tuple storage for a matching tuple. If no matching tuple is found the query tuple is stored in the query tuple storage.

Usually the query tuple storage will not be as big as the value tuple storage, mainly because the query storage will usually contain at most as many tuples as there are threads in the application. Only when the program makes use of asynchronous `get` operations, the number of outstanding query tuples per thread can exceed one. The unordered list is the most natural choice for storing a couple of query tuples.

The access to the data structures in class `TupleSpaceStorage` should be coordinated. If one operation is changing (either by storing or by retrieving) data in the value tuple storage or the query tuple storage, it should be impossible for another operation to access that changing storage until after the change is made. This means that among all the member functions in class `TupleSpaceStorage`, only one can be active at a time. This is achieved by a `Mutex` that is `lock`ed at the beginning of every member function and `unlock`ed at the end of every member function.

From where can the concurrency of multiple operations changing the tuple space storage come? All incoming messages are serialized and cannot compete with each other. However operations from within the process may be busy while a message asking to perform an operation from another process may arrive. The concurrency between the local and the remote operations must be coordinated.

7.4.2 Query Tuples

Query tuples are implemented using the `Tuple` definitions from Section 7.3.2. These combined with correct class definitions for the field access patterns `Any`, `Synchronous`, `Asynchronous`, `Query`, `Variable`, `Future` and `Value` (of course templated) allow query tuples to be implemented very efficiently. For a high performance implementation only one type of query tuple can be stored in the query tuple storage. This means that all the query tuples are transformed to the tuple type specified in the TupleSpace declaration. This transformation is performed in the tuple space front end.

Even though query tuples are not stored in a hash table, a hash function has to be defined for them, because query tuples have to find matching value tuples which are stored in a hash table.

```
template <class F0, class F1>
u32_t hash<Tuple<F0, F1> >(const Tuple<F0, F1>& tuple) {
   return hash<F0>(tuple.field0) ^ hash<F1>(tuple.field1);
}
```

The hash function for a query tuple is the *exclusive or* (^) of the hash values of its fields.

```
template <class T>
u32_t hash<Value<T> >(const Value<T>& value_field) {
   return hash<T>(value_field.value);
}
```

Tuple fields with the field access pattern Value are used to calculate a hash value. Fields with other field access patterns are discarded even though they may contain a value:

```
template <class T>
u32_t hash<Any<T> >(const Any<T>& any_field) {
   return 0;
}
```

In principle any number could be returned, but zero (0) combines very nicely with the exclusive or operator ^, because x ^ 0 == x. A good C++ compiler will optimize an exclusive or operation on zero away.

This optimization also allows for only one hash function definition for class Tuple. The default template arguments of class Tuple (class NoField, see Section 7.3.2) should simply have a hash value of zero:

```
template <>
u32_t hash<NoField>(const NoField& no_field) {
   return 0;
}
```

7.4.3 Value Tuples

Section 4.3.2 showed that the tuple prototype in a TupleSpace initialization only indicates the type of the tuples that are allowed in a get or copy operation. The corresponding value tuple prototype can be derived by changing all field access patterns to the Value field access pattern.

Changing all field access patterns to Value makes it impossible to derive from the tuple prototype what fields are part of the key for the hash value. Another value tuple prototype from which it is possible to derive which fields are part of the key, has to be introduced and can be accomplished by changing the name `Tuple` to `ValueTuple` in the tuple prototype of the `TupleSpace` initialization.

For example, assume the following tuple space initialization.

```
TupleSpace<Tuple<Value<int>, Any<int> > > ts;
```

The prototype of the value tuples is `ValueTuple<Value<int>, Any<int> >`. This class has the same data members as class `Tuple<Value<int>, Value<int> >`, namely two fields of type `Value<int>`. However it is much easier for class `ValueTuple` to decide what fields are part of the key, namely the fields with field access pattern `Value`.

```
template <class F0, class F1>
u32_t hash<ValueTuple<F0, F1> >(const ValueTuple<F0, F1>&
      value_tuple) {
   return hash<F0>(value_tuple.field0) ^
         hash<F1>(value_tuple.field1);
}
```

The definitions of the hash function on field access patterns are the same as those in Section 7.4.2.

7.5 Communication

This section shows how *Blossom* uses *Roots*'s active messages.

7.5.1 Member Function `put`

The member function `put` results in a `PutMsg` message to the tuple space back end. The `PutMsg` message contains the value tuple `tuple` to be stored and the tuple space storage `tuple_space_storage` in which the value tuple has to be stored.

```
template <class TSS, class VT>
class PutMsg/*<TSS, VT>*/: public Message {
private:
   TSS* tuple_space_storage;
   VT tuple;
public:
   PutMsg(TSS* tss, const VT& vt):
         tuple_space_storage(tss), tuple(vt) {}
   void action() const {
      tuple_space_storage->put(tuple);
   }
   unsigned int size() const {
      return sizeof(PutMsg<TSS, VT>);
   }
};
```

The PutMsg asynchronously inserts the value tuple in the tuple space storage, hence no message returns to the process/thread that invocated the put operation.

A specialized class PutMsg exists for putting a value tuple with no fields into a tuple space storage. This message mainly reduces the number of bytes of the PutMsg message by removing the empty value tuple from its data members. An empty data member consumes at least one byte. In this case it most likely consumes as many bytes as the pointer does, because multiple PutMsg objects have to align correctly.

7.5.2 Member Function get

The member function get gets a value tuple from the tuple space storage tuple_space-storage by providing a query tuple query_tuple. Assume the get operation is performed synchronously.

```
template <class TSS, class QT>
class GetMsg: public Message {
   TSS* tuple_space_storage;
   QT query_tuple;
   int pid;
   Sem* alarm;
public:
   GetMsg(TSS* tss, const QT& qt, int i, Sem* s):
         tuple_space_storage(tss), query_tuple(qt), pid(i),
         alarm(s) {}
   void action() const {
      tuple_space_storage->get(query_tuple, pid, alarm);
   }
   unsigned int size() const {
      return sizeof(GetMsg<TSS, QT>);
   }
};
```

The operation get is synchronous. A semaphore in the process that initiated the get operation implements the synchronous behavior. Immediately after the GetMsg message is sent to the appropriate tuple space storage process, the sending thread waits on the semaphore. As soon as a matching value tuple is found in the tuple space storage process, a return message MatchMsg will free the semaphore (by calling its post() member function) after initializing the fields of the query tuple with the fields of the value tuple. The process that initiates the get operation sends its process id pid, because the tuple space storage process has to send the matching value tuple back to it.

```
template <class QT, class RT>
class MatchMsg: public Message {
private:
   QT query_tuple;
   RT result_tuple;
```

```
      Sem* alarm;
   public:
      MatchMsg(const QT& qt, const RT& rt, Sem* s):
         query_tuple(qt), result_tuple(rt), alarm(s) {}
      void action() const {
         query_tuple.match(result_tuple);
         alarm->post();
      }
      unsigned int size() const {
         return sizeof(MatchMsg<QT, RT>);
      }
   };
```

7.5.3 Member Function `get_nb`

The member function `get_nb` tries to get a value tuple from the tuple space by specifying a query tuple. If no matching value tuple is found, the operation `get_nb` will return `false`. Otherwise, the found matching value tuple is sent back to the process that called the `get_nb` operation, the query tuple is initialized with the value tuple (if necessary) and the operation `get_nb` returns true.

In order to find a matching value tuple a `GetNBMsg` message is sent to the tuple space back end. The message contains the query tuple `tuple` to look for, and the tuple space back end location `tuple_space_storage` where to look for matching value tuples. Because a message has to be returned to the originating tuple space front end, the `GetNBMsg` further contains the process id `pid` of the originating front end, and pointers to the return value `retval` and semaphore `alarm`.

```
   template <class TSS, class T>
   class GetNBMsg: public Message {
      TSS* tuple_space_storage;
      T tuple;
      int pid;
      bool* retval;
      Sem* alarm;
   public:
      GetNBMsg(TSS* ts, const T& t, int i, bool* b, Sem* s):
            tuple_space_storage(ts), tuple(t), pid(i), retval(b),
            alarm(s) {}
      void action() const {
         tuple_space_storage->get_nb(tuple, pid, retval, alarm);
      }
      unsigned int size() const {
         return sizeof(GetNBMsg<TSS, T>);
      }
   };
```

In case a matching value tuple is found in the tuple space back end, the `MatchMsg` encountered earlier is returned to the tuple space front end. The `MatchMsg` does not have to update a return value, by initializing the return value (`retval`) to `true`. By doing so the `MatchMsg` can be reused for the non blocking operations.

However, in case no matching value tuple is found in the tuple space back end, the return value `retval` has to be set to `false` before the `get_nb` operation returns.

```
class NoMatchingMsg: public Message {
private:
   bool* retval;
   Sem* alarm;
public:
   NoMatchingMsg(bool* b, Sem* s): retval(b), alarm(s) {}
   void action() const {
      *retval = false;
      alarm->post();
   }
   unsigned int size() const {
      return sizeof(NoMatchingMsg);
   }
};
```

7.6 Tuple Space Front End

In Chapter 4 the *Blossom* language was introduced. One of the main characteristics of *Blossom* is its explicit definitions of tuple spaces. Tuple spaces can be defined with any number of tuples, which in turn can have any number of fields, which in turn can have any type. Never the less, both the number of tuples in a `TupleSpace` definition and the number of fields in those tuples are bound by a maximum (and are thus finite). But the number of types is infinite: all C++ types (finite) plus all user defined types (infinite). Having all possible `TupleSpace` definitions in one library leads to an infinite *Blossom* library, which is practically impossible.

A programmer however will use only a finite number of `TupleSpace` definitions in his programs (these programs are finite in size). All the `TupleSpace` definitions that occur in a program should be in the library. Since it is not possible to have an infinite number of predefined `TupleSpace` definitions, another solution has to be found. One where `TupleSpace` definitions are *generated* for all used `TupleSpaces` in the program.

Two ways to generate these `TupleSpace` definitions exist, namely (*i*) with a *precompiler*, which is the traditional Linda approach and described in Section 4.1.2, and (*ii*) with *type parameterization*, as described earlier in this chapter. *Blossom* uses the latter.

Before looking at the usage of type parameterization in *Blossom* an overview of *Blossom*'s class hierarchy is introduced. *Blossom*'s implementation distinguishes three important, orthogonal concepts in a `TupleSpace` definition: (*i*) The number of `Tuple` prototypes in a `TupleSpace` definition, (*ii*) the number of field prototypes in a `Tuple` prototype, and (*iii*) the different access patterns in a `TupleSpace` definition.

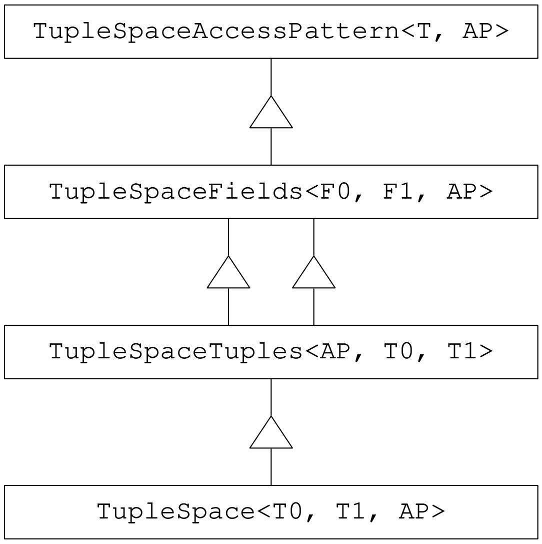


Figure 7.1: A hierarchy of tuple space classes in the front end prevents an exponential code explosion.
All four tuple space classes have a distinct goal. Class TupleSpaceAccessPattern distinguishes the access patterns, class TupleSpaceFields distinguishes the number of fields, class TupleSpaceTuples distinguishes the number of tuples, and class TupleSpace gives a nice user interface.

These three orthogonal concepts are tackled in different classes. Class TupleSpaceAccessPattern<T, AP> defines a tuple space with one tuple prototype (T) and an arbitrary access pattern (AP) and has a different implementation for every access pattern (concept *iii*). Class TupleSpaceFields<F0, F1, AP> defines a tuple space with access pattern AP and with one tuple with an arbitrary number of fields (here at most two; the actual implementation allows more fields). This class has a different implementation for every possible number of fields (concept *ii*). Class TupleSpaceTuples<AP, T0, T1> defines a tuple space with an arbitrary number of tuples (here at most two) and one access pattern AP (concept *i*). Class TupleSpace<T0, T1, AP> shows how a *Blossom* programmer specifies a tuple space (see Chapter 4). This class mainly forms an interface to the three TupleSpace... classes just described.

Figure 7.1 shows how all these classes collaborate. Class TupleSpace inherits from class TupleSpaceTuples which on its turns multiply inherits (once for every tuple prototype) from class TupleSpaceFields which inherits again from class TupleSpaceAccessPattern. More on the inheritance between the classes is discussed in the remainder of this chapter in a top down fashion. This means that we start looking at the class that is visible to the *Blossom* programmer, namely class TupleSpace. Notice that this is the bottom most class in Figure 7.1.

The biggest advantage of this division is that it dramatically reduces the size of code in *Blossom*'s library. For example, assume that the number of fields (at most f) and access patterns (at most a) are tackled simultaneously. Distinguishing every possible combination results in $a(f+1) = o(af)$ definitions, whereas treating them separately, results in $a+f+1$ definitions. Combining the number of tuple prototypes inside a tuple space (at most t) and the number of fields inside a tuple (at most f) results in even more definitions. Distinguishing every possible combination (one tuple with no fields; two tuples, one with one field and the other with two fields, *etcetera*) results in $(f+1)+(f+1)^2+\ldots+(f+1)^t = o(f^t)$ definitions.

7.6.1 Class `TupleSpace`

As discussed, *Blossom* makes extensive use of type parameterization (C++ templates). For example, the declaration of templated class `TupleSpace`, is as follows:

```
template <class T0, class T1 = Distributed,
                    class AP = Distributed>
class TupleSpace;
```

This declaration shows that a `TupleSpace` instantiation can contain up to three template arguments of which one is mandatory. These three template arguments have room for up to two `Tuple` prototypes (`T0` and `T1`) and one tuple space access pattern (`AP`). There should be at least one `Tuple` prototype, since a `TupleSpace` that cannot contain any `Tuples` is useless.

Depending on the number of `Tuple` prototypes in a `TupleSpace` definition, the access pattern is in a different template parameter (`AP` or `T1` in the case of maximal two `Tuples`). This wandering tuple space access pattern with a default makes many specializations of class `TupleSpace` necessary. Template parameter `T1` is sometimes a tuple (`TupleSpace<Tuple<int>, Tuple<float> >`) and sometimes a tuple space access pattern (`TupleSpace<Tuple<int>, Local>`). If the maximum number of `Tuples` in a tuple space is t and the maximum number of tuple space access patterns is a, class `TupleSpace` needs ta specializations.[3] In this dissertation we assume that $t = a = 2$, such that $ta = 4$ `TupleSpace` definitions are needed. All the class `TupleSpace` definitions inherit from class `TupleSpaceTuples` and do not add any member data nor member functions. All the member data and member functions are thus inherited from class `TupleSpaceTuples`.

```
template <class AP, class T0, class T1 = NoTuple1>
class TupleSpaceTuples;

template <class T0, class T1>
class TupleSpace<T0, T1, Local>:
```

[3] It is possible to use $a-1$ specializations less, by combining the a `TupleSpaceTuples` specializations with t tuple template arguments followed by one tuple space access pattern template argument.

```
      public TupleSpaceTuples<Local, T0, T1> {};

   template <class T0, class T1>
   class TupleSpace<T0, T1, Distributed>:
      public TupleSpaceTuples<Distributed, T0, T1> {};

   template <class T0>
   class TupleSpace<T0, Local>:
      public TupleSpaceTuples<Local, T0> {};

   template <class T0>
   class TupleSpace<T0, Distributed>:
      public TupleSpaceTuples<Distributed, T0> {};
```

After looking at these simple single inheritances, it appears that either class `TupleSpace` or class `TupleSpaceTuples` is redundant. The functionality of both classes can be combined, yet both classes serve their purpose. Omission of either class has disadvantages. Class `TupleSpace` gives the *Blossom* programmers a preferable interface, one where no tuple space access pattern needs to be defined. I found this important enough to keep class `TupleSpace` in the design. Note that higher up in the class hierarchy the access pattern parameter is always well defined. Class `TupleSpaceTuples` specifies many member functions. By removing this class, all ta `TupleSpace` definitions need to have an identical copy of all these member functions. This is undesirable from a software engineering point of view (for example, one change in one of the member functions needs to be replicated in ta `TupleSpace` definitions).

7.6.2 Class `TupleSpaceTuples`

Previously, the member functions `put`, `put_s`, `get`, `copy`, `get_nb` and `copy_nb` have not been discussed. These functions have not only the same number of parameters as the number of fields in one of the `Tuple` prototypes in the `TupleSpace` instantiation, but also corresponding types. In the definitions of class `TupleSpace` and class `TupleSpace-Tuples`, the number of fields in the `Tuple` prototypes is known implicitly (as part of one of the template parameters `T0` or `T1`) but not explicitly. The absence of this information makes it difficult to define member functions with the correct number of parameters in class `TupleSpace` or class `TupleSpaceTuples`. Another way for class `TupleSpaceTu-ples` to obtain member functions is by inheriting them from a base class. This strategy is followed in *Blossom* and only makes sense if the base class explicitly knows the number of fields from the `TupleSpace` and `TupleSpaceTuples` definitions.

Assume the base class is called `TupleSpaceFields`, and that it is type parameterized with the fields (`F0` and `F1`) of one of the `Tuple` prototypes from the `TupleSpace` definition and the access pattern (`AP`).

```
   template <class F0, class F1, class AP>
   class TupleSpaceFields;
```

Again, we introduce a simplification with respect to the real *Blossom* implementation: The number of fields in the examples in this chapter is maximal two.

Assume for the moment that a `TupleSpace` contains one `Tuple` prototype, which results in its base class `TupleSpaceTuples` having one `Tuple` prototype. A simple single inheritance derivation from `TupleSpaceFields` will cause class `TupleSpace-Tuples` to inherit all member functions from class `TupleSpaceFields`.

Assume, for example, the programmer wants to use a `TupleSpace` object of type `TupleSpace<Tuple<Value<int>, Any<int> >, Local>`. The corresponding `TupleSpaceTuples` class is class `TupleSpaceTuples<Local, Tuple<Value <int>, Any<int> >` which subsequently inherits from class `TupleSpaceFields< Value<int>, Any<int>, Local>`. Obviously the library has to provide some code to specify this last inheritance.

```
class TupleSpaceTuples<Local, Tuple<Value<int>, Any<int> > >:
   public TupleSpaceFields<Value<int>, Any<int>, Local>;
```

We now encounter an old problem again: the library has to contain infinitely many of these derivations (one for every `TupleSpace` type). Again, the solution is to parameterize.

```
template <class AP, class T0>
class TupleSpaceTuples<AP, T0>:
   public TupleSpaceFields<..., ..., AP> {};
```

Template parameter `T0` should be a `Tuple` prototype, whose field prototypes should be filled in on the dots (`...`). The types of both fields should be extracted from template parameter `T0`. This can be realized by extending the definition of class `Tuple` with the types of its field types, defined as `MyField0` and `MyField1`.

```
template <class F0, class F1>
class Tuple {
   ...
public:
   typedef F0 MyField0;
   typedef F1 MyField1;
   ...
};
```

Using the `MyField` types defined in class `Tuple` the derivation becomes:

```
template <class AP, class T0>
class TupleSpaceTuples<AP, T0>:
   public TupleSpaceFields<T0::MyField0, T0::MyField1, AP> {};
```

What happens if template parameter `T0` is instantiated with a `Tuple` that contains less than two fields? That would make `T0::MyField1` undefined. To overcome this problem specializations of class `Tuple` with less than the maximum number of field prototypes, define the remaining MyField types as class NoField. This puts the responsibility of accepting less fields on class TupleSpaceFields, i.e. this class should recognize NoField arguments and act accordingly. The task of class TupleSpaceFields is to differentiate the number of fields anyway.

So far for the problems with class TupleSpaceTuples with only one Tuple prototype inside. We saw how the TupleSpaceTuples provide member functions with the correct number and type by inheriting from class TupleSpaceFields.

The second type of problem, which occurs when the programmer specifies two or more Tuple prototypes within one TupleSpace. These problems are mainly C++ related. Assume a programmer specifies a TupleSpace with two Tuple prototypes, resulting in a TupleSpaceTuples with two Tuple prototypes. The *Blossom* library cannot simply provide the following declaration.

```
template <class AP, class T0, class T1>
class TupleSpaceTuples:
   public TupleSpaceFields<T0::Myfield0, T0::MyField1, AP>,
   public TupleSpaceFields<T1::MyField0, T1::MyField1, AP> {};
```

In C++ this TupleSpaceTuples definition does not inherit any member functions. The compiler is not smart enough to know what put member function to call, the one from TupleSpaceFields<T0::MyField0, T0::MyField1, AP> or the one from TupleSpaceFields<T1::MyField0, T1::MyField1, AP>. A human can easily see which one to use (deducing this from the number and type of the fields). However the C++ compiler only differentiates them on their function names and they are the same (put) in both base classes. The ANSI C++ standard imposes this, prescribing in our case that member function put is ambiguous in class TupleSpaceTuples with two or more base classes.[4] An ANSI C++ compiler does not know which put function to choose.

The way to let class TupleSpaceTuples have put member functions is by including them directly in the class and let those put functions choose the correct TupleSpaceFields::put function. Member functions put_s, get, copy, get_nb and copy_nb should be included similarly. Since it is not known how many parameters the Tuple prototypes have, the class TupleSpaceTuples needs member functions put with any number (up to an upper bound) of parameters. Three types of the parameters can be passed to the member function put: Type, Var<Type> and Fut<Type> (see table 4.2). If the maximal number of arguments for the member functions is n, the number of put member functions that have to be defined inside class TupleSpaceTuples is $3^0 + 3^1 + 3^2 + \ldots + 3^n = o(3^n)$. The other operations (put_s, get, copy, get_nb and copy_nb) result in exponentially many member functions definitions as well. A good programmer does not want to specify exponentially many member function definitions. A way to circumvent this is by type parameterizing the member functions, as follows:

```
template <class AP, class T0, class T1 = NoTuple1>
```

[4] This has to do with the legacy of C. The exact argument is that both member functions put are defined in different contexts that are not the current context [2].

```
class TupleSpaceTuples:
   public TupleSpaceFields<T0::MyField0, T0::MyField1, AP>,
   public TupleSpaceFields<T1::MyField0, T1::MyField1, AP> {
public:
   void put() {
      TupleSpaceFields<NoField, NoField, AP>::put();
   }
   template <class P0>
   void put(const P0& parm0) {
      typedef ... MyTupleSpaceFields;   // Section 7.7.1
      MyTupleSpaceFields::put(parm0);
   }
   template <class P0, class P1>
   void put(const P0& parm0, const P1& parm1) {
      typedef ... MyTupleSpaceFields;   // Section 7.7.1
      MyTupleSpaceFields::put(parm0, parm1);
   }
   // code for get, copy, get_nb and copy_nb is similar
};
```

The definition of type `MyTupleSpaceFields` in the member functions `put` depends on the template parameters of class `TupleSpaceTuples` (`AP`, `T0` and `T1`) and on the template parameters of the functions themselves (`P0` and `P1`). Because this is not straightforward, this problem is solved in Section 7.7.1. Assume for now that the correct base class can be found based on the template parameters `T0`, `T1`, `P0`, `P1` and `AP`.

The default template argument `NoTuple1` for template parameter `T1` makes one definition of class `TupleSpaceTuples` suffice for any number (in our case up to two, but in the implementation the same idea works for a larger number) of `Tuple` prototypes in class `TupleSpaceTuples`. The definition of class `TupleSpaceTuples` for one `Tuple` prototype earlier in this section can be omitted. This old definition results in another semantics for the arguments in the member functions than the newer and more general definition. The former definition allowed C++ coercion for the arguments, the latter does not due to the fact that coercion is not taken into account in determining the right base class `TupleSpaceFields`. However if coercion of the arguments in `TupleSpace` member functions is preferred, coercion rules can be hard coded in the library.

7.6.3 Class **`TupleSpaceFields`**

`TupleSpace` member functions are allowed to have multiple arguments (see Table 4.2). Member functions `get` and `copy` for example are allowed to have seven different types of arguments (`T`, `Var<T>`, `Fut<T>`, `Var<T>*`, `VarWC<T>*`, `Fut<T>*` and `FutWC<T>*`). Member functions `put` and `put_s` can only have three different types of arguments (`T`, `Var<T>` and `Fut<T>`).

The number of different types of arguments may be restricted by the field access patterns specified in the `TupleSpace` initialization (see Table 4.3).

Class TupleSpaceFields ensures that only arguments of the correct type (according to both tables) are allowed in the member functions, by making a tuple type that can be created out of the arguments if the combination is allowed. How this type is generated for member functions get, copy, get_nb and copy_nb is too elaborate to discuss here and is explained in Section 7.7.2. Significantly, types can be created given the template parameters of the class definition (F0 and F1) and the member definition (P0 and P1).

Class TupleSpaceFields has to decide for the member functions get and copy whether to call the asynchronous or the synchronous version (defined in TupleSpaceAccessPattern). This too has to do with the types of the arguments of these member functions. If Fut<Type>* (or FutWC<Type>*) arguments are present and no Var<Type>* (or VarWC<Type>*) arguments are present the member functions should call the asynchronous (get_a and copy_a), otherwise they should call the synchronous version (get and copy). How this decision is made based on the template parameters is explained in Section 7.7.3.

```
template <class T, class AP>
class TupleSpaceAccessPattern;

template <class F0, class F1, class AP>
class TupleSpaceFields:
      public TupleSpaceAccessPattern<Tuple<F0, F1>, AP> {
   typedef TupleSpaceAccessPattern<Tuple<F0, F1>, AP>
      MyTupleSpaceAccessPattern;
   typedef ValueTuple<F0, F1> MyValueTuple;
public:
   template <class P0, class P1>
   void put(const P0& p0, const P1& p1) {
      MyTupleSpaceAccessPattern::put(MyValueTuple(p0, p1));
   }

   template <class P0, class P1>
   void get(const P0& p0, const P1& p1) {
      typedef ... MyQueryTuple;        // Section 7.7.2
      const bool asynchronous = ...;   // Section 7.7.3
      if (asynchronous)
         MyTupleSpaceAccessPattern::get_a(MyQueryTuple(p0, p1));
      else
         MyTupleSpaceAccessPattern::get(MyQueryTuple(p0, p1));
   }

   // The member function for copy is similar to get.

   template <class P0, class P1>
   bool get_nb(const P0& p0, const P1& p1) {
      typedef ... MyNBQueryTuple;   // Section 7.7.2
      return MyTupleSpaceAccessPattern::
         get_nb(MyNBQueryTuple(p0, p1));
```

```
    }

    // The member function for copy_nb is similar to get_nb.
};
```

This definition of class `TupleSpaceFields` assumes two field prototypes. Specializations of this templated class have to be made for less field prototypes as well.

The assignment to the `asynchronous` boolean in member functions `get` and `copy` does not contain any run time calculations. Whether boolean `asynchronous` becomes `true` or `false` solely depends on compile time calculations. A similar matching is involved as in the determination of the right tuple to be send to the tuple space back end for the operations `get` and `copy` (class `MyQueryTuple`) and the operations `get_nb` and `copy_nb` (class `MyNBQueryTuple`). At compile time this expression evaluates either to the constant `true` or to the constant `false`. This means that this assignment as well as the comparison in the `if-then-else` that follows will be optimized away by a good C++ compiler.

7.6.4 Class `TupleSpaceAccessPattern`

Class `TupleSpaceAccessPattern` is the last part of the front end responsible for the communication with the back end. The communication depends on the tuple space access pattern, which means that a separate implementation of class `TupleSpaceAccessPattern` is needed for every possible access pattern. This is done by using template specialization once more.

The member functions of class `TupleSpaceAccessPattern` perform the main communication functionality. All member functions of the previous `TupleSpace...` classes just form an interface to these member functions. Class `TupleSpaceAccessPattern` provides the following member functions: `put`, `put_s`, `get`, `copy`, `get_a`, `copy_a`, `get_nb` and `copy_nb`. How these calls communicate with the tuple space back end is discussed in Section 7.5.

Remember that until now no `TupleSpace...` class contained any data members. Class `TupleSpaceAccessPattern` contains depending on the access pattern one or more pointers to the back end. All this entails that a `TupleSpace` contains almost no data, which is exactly what was needed by the functional requirements specified in Section 5.2. A `TupleSpace` should be easily copyable to another process.

It is handy to know whether a tuple space is still in use. The back end may consume much memory and it would be a waste to not reuse that memory if it is no longer needed. Consequently, class `TupleSpaceAccessPattern` contains a pointer to a `ReferenceCount`.

The layout of a `Local` `TupleSpace` is shown in Figures 7.2. The back end of a `Local` tuple space is located on one machine (`TupleSpaceStorage`). However multiple processes may be willing to communicate to this back end, thus several front ends need to be able to point to this one back end. The front end has to know on which machine the back end resides. Data member `pid` holds the id of the process that contains the storage.

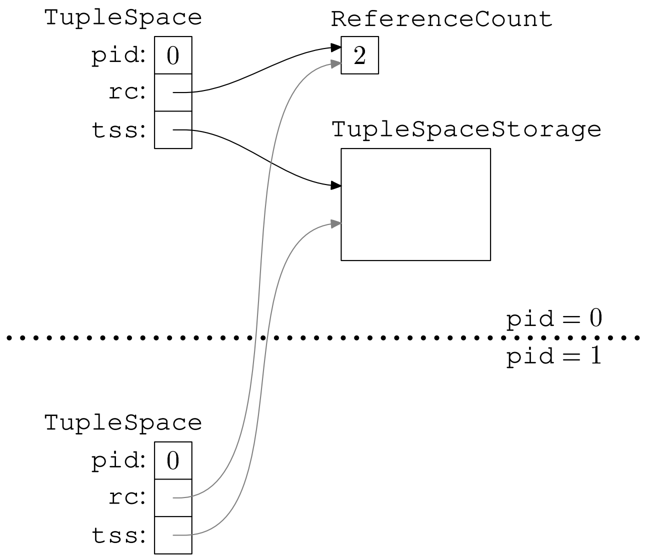


Figure 7.2: `TupleSpace<..., Local>`

This process also maintains the reference count for this storage (`ReferenceCount`). The tuple space front end also contains pointers to the storage and the reference count. These pointers only point to the storage and reference count when you are in the correct process, namely process `pid`.

The lay out of a `Distributed` `TupleSpace` is shown in Figure 7.3. The front end of a `Distributed` tuple space also contains a process id `pid` data member. Here it only indicates on which machine the reference count is located. The tuple space back end consists of several storages, one on every process. Pointers to all the different tuple space storages are maintained in an array. The index of the array is the process id on which the pointer has a meaning and points to a storage. The `rc` data member is a pointer to the reference count that resides on machine `pid`. A distributed tuple space uses hashing to decide in which storage to store or retrieve a tuple.

With regard to performance, the introduction of all these classes that inherit from one another does not influence the performance of the resulting program. A good C++ compiler does not introduce any extra calls or indirections, as long as the member functions are `inline`.

Behind the class hierarchy, the *Blossom* implementation still uses many templates. For performance these templates are all conveniently calculated at compile time. The template reductions even combine nicely with other compile time optimizations. For example, if a template construction reduces to an inline function that just returns `true` or `false`, the compiler will even optimize away a corresponding `if` on this return value (for example end of Section 7.6.4). Similar tactics are used several times in the implementation of *Blossom*.

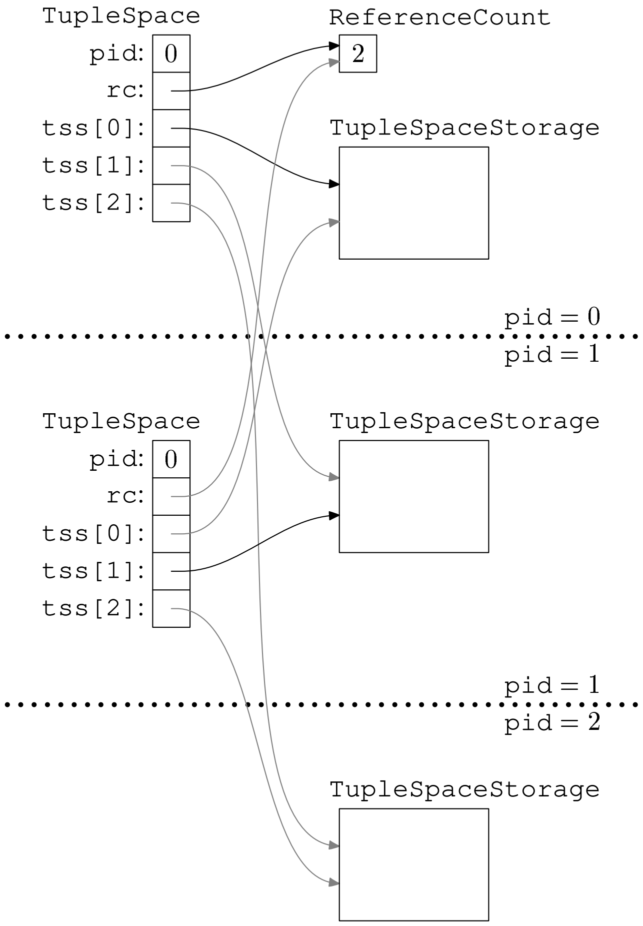


Figure 7.3: `TupleSpace<..., Distributed>`

7.7 Type Conversions

In the description of the tuple space front end (Section 7.6) every now and then holes (`...`) were left in the definitions. All these holes had to do with type manipulation. This section fills in all these holes.

The type manipulations in the following sections are performed in a way that closely resembles the style of logic programming (Prolog).

7.7.1 `TupleSpaceFields` Selection

The first holes in the source code of the *Blossom* library appeared in Section 7.6.2 when dealing with the implementation of class `TupleSpaceTuples`. This piece of code of the *Blossom* library is repeated here.

```
template <class AP, class T0, class T1 = NoTuple1>
class TupleSpaceTuples:
   public TupleSpaceFields<T0::MyField0, T0::MyField1, AP>,
   public TupleSpaceFields<T1::MyField0, T1::MyField1, AP> {
public:
   void put() {
      TupleSpaceFields<NoField, NoField, AP>::put();
   }
   template <class P0>
   void put(const P0& parm0) {
      typedef ... MyTupleSpaceFields;   // Section 7.7.1
      MyTupleSpaceFields::put(parm0);
   }
   template <class P0, class P1>
   void put(const P0& parm0, const P1& parm1) {
      typedef ... MyTupleSpaceFields;   // Section 7.7.1
      MyTupleSpaceFields::put(parm0, parm1);
   }
   // code for put_s, get, copy, get_nb and copy_nb is similar
};
```

Class `TupleSpaceTuples` in general derives from several classes `TupleSpace-Fields`. Every base class `TupleSpaceFields` takes care of a separate `Tuple` prototype specified in a template instantiation of class `TupleSpace`. When a program calls a member function of class `TupleSpace`, it actually calls that member function in class `TupleSpaceTuples`. In addition the member functions in class `TupleSpaceTuples` are delegated to member functions in one of the base classes. In order to call the functions in the correct base class `TupleSpaceFields`, an ingenious type conversion is needed. Based on the template parameters of class `TupleSpaceTuples` (`AP`, `T0` and `T1`) and the template parameters of the called member function (`P0` and `P1`), the correct base class type has to be constructed.

Roughly speaking *Blossom* has to solve the following problem. If the template parameters `P0` and `P1` of the member functions correspond to the template parameter `T0` of `TupleSpaceTuples<AP, T0, T1>` the base class should be `TupleSpaceFields< AP, T0::MyField0, T0::MyField1>`. If the template parameters `P0` and `P1` correspond to the template parameter `T1` the base class should be `TupleSpaceFields< AP, T1:: MyField0, T1::MyField1>`.

Template parameters `T0` and `T1` of class `TupleSpaceTuples` indicate `Tuples`. Remember from Section 7.6.1 that class `Tuple` specifies the types of its fields as `MyField0` and `MyField1`. However the type of a field is a field access pattern (something like `Variable<int>`) and not a base type (something like `int`). So we have to deduce the base type from a field access pattern type. This is established using a trick similar to how we obtained the type of the fields from a tuple definition. A field access pattern specifies the type of its argument as `MyArgument`. Thus the base types of the arguments of tuple `T0` are also known as `T0::MyField0::MyArgument` and `T0::MyField1::MyArgument`.

Template parameters `P0` and `P1` of the member functions can be (under normal usage) one of the following types `A`, `Var<A>`, `Var<A>*`, `Fut<A>`, `Fut<A>*` (where type `A` is called the argument). Of these types only the type of the argument `A` is of interest for the correspondence relation. How can we infer the type of the argument (`A`) from the template parameters `P0` and `P1`?

A similar solution as above, i.e. using a construct like `P0::MyArgument`, does not work because the type `MyArgument` can only be specified in a class. Types `Var<T>*` and `Fut<T>*` are pointers and not classes and can thus not specify type `MyArgument`. Another construction has to be found.

The solution lies in type indirection. By introducing a class that is templated with any of the five possible template argument types the interesting base type can be defined inside this class (by using template specialization). This introduced class is called `ArgumentDB`.

```
template <class A>
class ArgumentDB {
public:
   typedef A MyArgument;
};
```

If not overruled by a specialization this class exports its template parameter by the type `MyArgument`. Four specializations of this general class are defined.

```
template <class A>
class ArgumentDB<Var<A> > {
public:
   typedef A MyArgument;
};

template <class A>
class ArgumentDB<Var<A>*> {
public:
```

```
   typedef A MyArgument;
};

template <class A>
class ArgumentDB<Fut<A> > {
public:
   typedef A MyArgument;
};

template <class A>
class ArgumentDB<Fut<A>*> {
public:
   typedef A MyArgument;
};
```

These specializations export the base type of their template parameters as `MyArgu-ment`, hence, the base type of a general field access pattern type `P0` can now be written as `ArgumentDB<P0>::MyArgument`.

A double type indirection is used to find the correct base class `TupleSpaceFields` for a specific *Blossom* tuple space operation. In the member functions the definition of `MyTupleSpaceFields` can now be filled in. For the member functions with one field argument, the definition becomes

```
typedef TupleSpaceFieldsDB<T0, T1, AP, P0>::MyTupleSpaceFields
        MyTupleSpaceFields;
```

For the member functions with two field arguments the definition becomes

```
typedef TupleSpaceFieldsDB<T0, T1, AP, P0, P1>::
        MyTupleSpaceFields MyTupleSpaceFields;
```

The templated class `TupleSpaceFieldsDB` performs the two conversions described above, before it matches the prototypes of the tuples specified in the tuple space declaration and the *Blossom* operation.

```
template <class T0, class T1, class AP,
          class P0, class P1 = NoParameter>
class TupleSpaceFieldsDB {
   typedef TupleSpaceFields<T0::MyField0, T0::MyField1, AP>
         TSF0;
   typedef TupleSpaceFields<T1::MyField0, T1::MyField1, AP>
         TSF1;
   typedef T0::MyField0::MyArgument T0A0;
   typedef T0::MyField1::MyArgument T0A1;
   typedef ArgumentDB<P0>::MyArgument A0;
   typedef ArgumentDB<P1>::MyArgument A1;
public:
   typedef TupleSpaceFieldsDB2<TSF0, TSF1, T0A0, T0A1, A0, A1>
```

```
        ::MyTupleSpaceFields MyTupleSpaceFields;
};
```

Class `TupleSpaceFieldsDB` needs an extra class `TupleSpaceFieldsDB2` to find the correct base class `TupleSpaceFields`. Class `TupleSpaceFieldsDB` "unpacks" its template parameters, transforming them to better usable template arguments for class `TupleSpaceFieldsDB2`. Class `TupleSpaceFieldsDB2` is templated with the different base classes, the argument types of the first base class and the argument types of the member function call.

If the argument types of the first `TupleSpaceFields` match the argument types, the first `TupleSpaceFields` is the correct base class.

```
template <class TSF0, class TSF1, class A0, class A1>
class TupleSpaceFieldsDB2<TSF0, TSF1, A0, A1, A0, A1> {
public:
   typedef TSF0 MyTupleSpaceFields;
};
```

Note that this is a specialization of the templated definition to be discussed now. The general case occurs when the argument types of the first `TupleSpaceFields` do not match the argument types of the member function call. In that case we shift the `Tuple-SpaceFields` one place to the left and fill a dummy class (`NoTupleSpaceFields`) on the empty spot. Furthermore, the argument types of the first `TupleSpaceFields` have to change since the first `TupleSpaceFields` changed. A `TupleSpaceFields` definition exports the types of its first and second argument under the names of `MyArgument0` and `MyArgument1`.

```
template <class TSF0, class TSF1, class TSF0A0, class TSF0A1,
          class A0, class A1>
class TupleSpaceFieldsDB2 {
public:
   typedef TupleSpaceFieldsDB2<TSF1, NoTupleSpaceFields,
      TSF1::MyArgument0, TSF1::MyArgument1, A0, A1>::
            MyTupleSpaceFields MyTupleSpaceFields;
};
```

If the argument types of the member function do not match any of the argument types of the base classes `TupleSpaceFields` the search can stop. Notice that this definition again is a specialization of the one given above.

```
template <class TST0A0, class TST0A1, class A0, class A1>
class TupleSpaceFieldsDB2<NoTupleSpaceFields,
      NoTupleSpaceFields, TST0A0, TST0A1, A0, A1> {
public:
   typedef MemberFunctionArgumentsDoNotMatchTupleSpaceDefinition
      MyTupleSpaceFields;
};
```

7.7.2 Query Tuple Selection

The next place in the description where holes were encountered was in the code for class `TupleSpaceFields` templated with two fields (`F0` and `F1`) and a tuple space access pattern (`AP`). This code is repeated below.

```
template <class F0, class F1, class AP>
class TupleSpaceFields:
      public TupleSpaceAccessPattern<Tuple<F0, F1>, AP> {
   typedef TupleSpaceAccessPattern<Tuple<F0, F1>, AP>
      MyTupleSpaceAccessPattern;
   typedef ValueTuple<F0, F1> MyValueTuple;
public:
   template <class P0, class P1>
   void put(const P0& p0, const P1& p1) {
      MyTupleSpaceAccessPattern::put(MyValueTuple(p0, p1));
   }

   template <class P0, class P1>
   void get(const P0& p0, const P1& p1) {
      typedef ... MyQueryTuple;        // Section 7.7.2
      const bool asynchronous = ...;   // Section 7.7.3
      if (asynchronous)
         MyTupleSpaceAccessPattern::get_a(MyQueryTuple(p0, p1));
      else
         MyTupleSpaceAccessPattern::get(MyQueryTuple(p0, p1));
   }

   // The member function for copy is similar to get.

   template <class P0, class P1>
   bool get_nb(const P0& p0, const P1& p1) {
      typedef ... MyNBQueryTuple;   // Section 7.7.2
      return MyTupleSpaceAccessPattern::
         get_nb(MyNBQueryTuple(p0, p1));
   }

   // The member function for copy_nb is similar to get_nb.
};
```

Actually the definition of class `TupleSpaceFields` contains two kinds of holes. The first kind is labeled with `Section 7.7.2`, the other is labeled with `Section 7.7.3`.

Class `TupleSpaceTuples` is instantiated with one or more tuple prototypes. For every tuple prototype class `TupleSpaceTuples` inherits member functions from a different base class `TupleSpaceFields` (instantiated with different template arguments). Class `TupleSpaceFields` is templated with the template parameters `F0` and `F1` which are the first and second field definition of one of the tuple arguments in class `TupleSpaceTuples`.

The member functions of class `TupleSpaceFields` accept all the possible arguments. Hence the member functions are templated with template parameters `P0` and `P1`. When the field arguments of the member functions can be converted to the field arguments of the class `TupleSpaceTuples` the member function arguments are of an allowed type.

Class `TupleSpaceFields` decides on questions where the types of the fields are important. Once in class `TupleSpaceFields` we know that the argument types of tuples passed to the member functions matches One of these questions is what `Tuple` field to send to the tuple space back end. This `Tuple` should be as small (in bytes) as possible.

```
template <class F0, class F1, class P0, class P1 = NoParameter>
class QueryTupleDB {
public:
   typedef QueryFieldDB<F0, P0>::MyQueryField MyQueryField0;
   typedef QueryFieldDB<F1, P1>::MyQueryField MyQueryField1;
   typedef Tuple<MyQueryField0, MyQueryField1> MyQueryTuple;
};

template <class F, class P>
class QueryFieldDB;

template <class F>
class QueryFieldDB<F, NoParameter> {
public:
   typedef No MyQueryField;
};

// ...

template <class T>
class QueryFieldDB<Synchronous<T>, T> {
public:
   typedef Value<T> MyQueryField;
};

template <class T>
class QueryFieldDB<Synchronous<T>, Var<T> > {
public:
   typedef Value<T> MyQueryField;
};

template <class T>
class QueryFieldDB<Synchronous<T>, Fut<T> > {
public:
   typedef Value<T> MyQueryField;
};

template <class T>
```

```
class QueryFieldDB<Synchronous<T>, Var<T>*> {
public:
   typedef Variable<T> MyQueryField;
};

template <class T>
class QueryFieldDB<Synchronous<T>, VarWC<T>*> {
public:
   typedef Variable<T> MyQueryField;
};

// The following field types are not allowed by the field
// access pattern Synchronous.

template <class T>
class QueryFieldDB<Synchronous<T>, Fut<T>*> {
public:
   typedef No MyQueryField;
};

template <class T>
class QueryFieldDB<Synchronous<T>, FutWC<T>*> {
public:
   typedef No MyQueryField;
};

// Similar specializations for all other field access patterns
// (Value, Variable, Future, Asynchronous, Query, Any)
```

7.7.3 Synchronous or Asynchronous

Member functions `get` and `copy` in class `TupleSpaceFields` have to decide whether the corresponding synchronous or asynchronous member function in its base class `TupleSpaceAccessPattern` has to be called. This decision is based on the template arguments of the functions. The lines

```
const bool asynchronous = ...;   // Section 7.7.3
```

in these member functions become

```
const bool asynchronous = Asynchronous<P0, P1>::result();
```

If the arguments contain futures and no variables the corresponding asynchronous version should be called.

```
template <class P0, class P1 = NoParameter>
class Asynchronous {
public:
```

```
    inline static bool result() {
       const bool future_fields =
          IsFutureField<P0>::result() ||
          IsFutureField<P1>::result();
       const bool variable_fields =
          IsVariableField<P0>::result() ||
          IsVariableField<P1>::result();
       return future_fields && !variable_fields;
    }
 };
```

Classes `IsFutureField` and `IsVariableField` define whether argument types are futures or fields, respectively.

```
 template <class P>
 class IsFutureField {
 public:
    inline static bool result() {
       return false;
    }
 };

 template <class A>
 class IsFutureField<Future<A>*> {
 public:
    inline static bool result() {
       return true;
    }
 };

 template <class P>
 class IsVariableField {
 public:
    inline static bool result() {
       return false;
    }
 };

 template <class A>
 class IsVariableField<Variable<A>*> {
 public:
    inline static bool result() {
       return true;
    }
 };
```

Chapter 8

Experiments

This chapter compares *Blossom* programs based on default Linda behavior with *Blossom* programs that utilize the *Blossom* extensions (from Chapter 4). The extensions studied are the `Replicated` tuple space access pattern in Section 8.2, the `Local` tuple space access pattern in Section 8.3, combining `Local` and `Replicated` tuple spaces in Section 8.4, and futures in Section 8.5. These sections will assess performance by comparing the execution time and number of communications performed. Section 8.1 forms a common prelude to all experiments. The experiments are meant to be illustrative of the potential of *Blossom*. Applications have been chosen to show that having the flexibility that *Blossom* provides can result in performance improvements.

8.1 Experiments Design

The experiments performed in this chapter are designed to stress communication, not to necessarily achieve the best speedups. This is an important observation, since achieving good speedups for the simple problems in this chapter, reduces the chances to show improvements of the *Blossom* extensions.

The remaining of this section gives an overview of the technical specifications of the hardware, operating system and compiler. It also looks at the statistical aspects of the experiments.

Hardware The experiments are run on a Beowulf cluster of twelve dual Pentium III 550 MHz computers. Every computer has 512 Mbyte of RAM on board. The machines are completely idle, except for the occasional daemon process. All computers are connected through a fully switched 100 BaseT network.

Operating System The Beowulf cluster is running Linux 2.14-15 from the Mandrake 7.0 Linux distribution.

Compiler All programs are compiled with $g{+}{+}$ version 2.95.2. The compile time options used are `-O3`, `-DCPU=i686`, `-funroll-loops`, `-fexpensive-optimizations`, `-mpentiumpro`, `-ffast-math`, `-fno-strength-reduce`, and `-fomit-frame-pointer`. In other words, all speed optimizations are turned on.

Number of Runs Every experiment is repeated ten times to reduce the influence of daemons on the experiments. The results of all experiments can be found in Appendix B. This chapter uses the average timings and number of communications of the ten runs.

Averaging Results As you can see from the experiments, the results are quite reproduceable. Hence all results are simply averaged over the number of runs.

8.2 Experiment: `Replicated` Tuple Space

Thesis A replicated tuple space can outperform a distributed tuple space.

Methodology A *Blossom* program is run twice. In the first run a tuple space has the default Linda tuple space access pattern of `Distributed`, and in the second run that same tuple space is given a *Blossom* `Replicated` access pattern. The program with the `Replicated` tuple space should use less communications and/or outperform the program with the `Distributed` tuple space.

More precisely, the experimental program is the Traveling Salesman Problem (TSP) based on the *closest neighbor first* heuristic. The TSP problem instance is divided into several jobs that can be searched independently of each other by multiple processes. Jobs are maintained in sorted order so as to ensure the closest neighbor first heuristic is preserved. The TSP implementation in *Blossom* utilizes five tuple spaces.

- A tuple space containing the index of the job to be calculated next,
- A tuple space containing all the jobs,
- A tuple space with the minimal tour distance found so far,
- A tuple space that contains the maximum job index from which the processes can decide when all work is done, and
- A tuple space in which the processes can report that they are finished, and from which can be concluded that the problem is solved.

The experiment involves changing the access pattern of the tuple space containing the minimal tour distance from `Distributed` to `Replicated`. Since the tuple space containing the minimal tour distance will only contain *one tuple* it is easier to think of it as a `Local` tuple space.

CPUs	Distributed minimum	Replicated minimum
1	1792.91 s	1793.46 s
2	1000.91 s	984.67 s
3	699.38 s	681.76 s
4	543.56 s	527.92 s
5	450.16 s	436.63 s
6	388.54 s	375.41 s
7	344.61 s	332.60 s
8	312.87 s	300.48 s
9	284.49 s	275.69 s
10	264.40 s	255.87 s
11	248.78 s	239.72 s
12	235.74 s	227.39 s

Table 8.1: Average timings for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the minimum traveling distance is `Distributed` for the results in the second column and `Replicated` for the results in the third column.

To reduce the overhead on the tuple space containing the minimum tour length, every thread gets a local copy at the beginning of a new piece of work. And only when an improved minimum (over the possible 'old' local minimum) is found, an updated minimum is requested.

Results A sixteen city TSP problem instance is used in the calculations and the number of jobs generated is $15 \times 14 \times 13 \times 12 \times 11$. The number of jobs is chosen this high to show differences in the timings. A smaller number of jobs will show perfect linear speedups, which are difficult to improve upon.

Table 8.1 shows average timings of two TSP implementations in *Blossom*. One implementation uses the Linda tuple space access patterns (`Distributed` minimum) and the other uses the *Blossom* `Replicated` tuple space access pattern for the tuple space containing the minimal travel distance.

Table 8.2 shows the number of inter-process communications that take place for both implementations.

Analysis The timings in Table 8.1 for both implementations are almost the same. The difference is approximately 5%, which does not seem to be all that much. This is confirmed in the speedup graph in Figure 8.1.

The better performance of the TSP with the replicated tuple space containing the minimum can mainly be explained by looking at the difference in the number of communications (see Figure 8.2). The `Replicated` tuple space allows each process to look up the current tour minimum locally. Of course once a lower tour minimum is found all processes have to

CPUS	DISTRIBUTED MINIMUM	REPLICATED MINIMUM
1	0	0
2	1,305,642	1,100,610
3	1,790,958	1,474,698
4	2,074,670	1,665,347
5	2,243,772	1,768,598
6	2,367,797	1,836,378
7	2,436,261	1,885,195
8	2,493,835	1,919,888
9	2,536,547	1,931,795
10	2,573,314	1,969,394
11	2,603,222	1,986,590
12	2,641,772	2,003,885

Table 8.2: Average number of communications for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the minimum traveling distance is `Distributed` for the results in the second column and `Replicated` for the results in the third column.

be notified which causes more communications. But the number of updates of the minimum tour length is relatively small, so that the total number of communications drops.

8.3 Experiment: `Local` Tuple Space

Thesis A local tuple space can out perform a distributed tuple space.

Methodology A *Blossom* program is run twice. In the first run the tuple space that contains the jobs has the default tuple space access pattern of `Distributed`, and in the second run that same tuple space is given a `Local` access pattern. The program with the `Local` tuple space should use less communications and/or outperform the program with the `Distributed` tuple space.

The TSP program used in the `Replicated` experiment is repeated, but this time with a distributed and local jobs tuple space.

Results The same sixteen city TSP problem instance of the previous experiment is used in the calculations. Again the number of jobs generated is $15 \times 14 \times 13 \times 12 \times 11$ to show the differences in timings more clearly.

Table 8.3 shows the average timings for both TSP implementations. Table 8.4 shows the average number of communications for both TSP implementations.

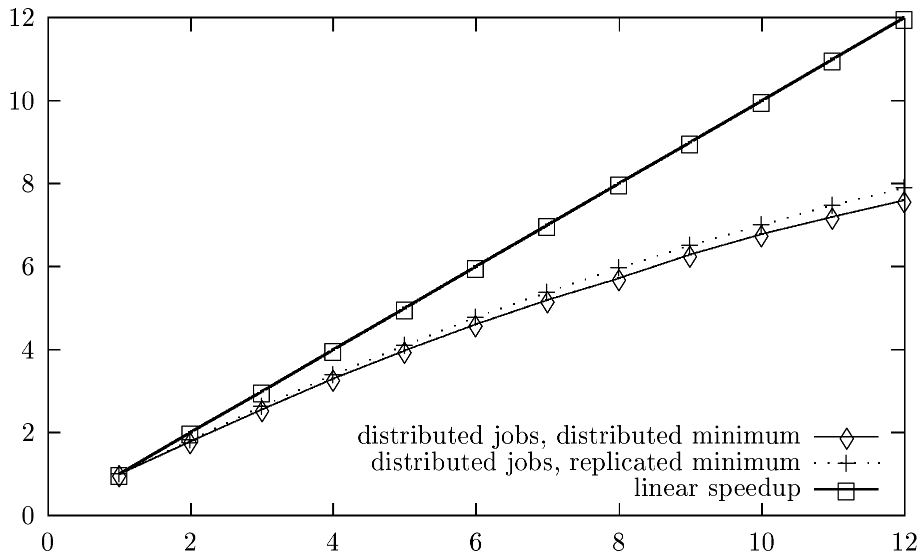


Figure 8.1: Speedups of the Traveling Salesman Problem with `Distributed` tuple space access patterns and with a `Replicated` tuple space that contains the minimal tour distance.

CPUs	DISTRIBUTED JOBS	LOCAL JOBS
1	1792.91 s	1793.11 s
2	1000.91 s	963.51 s
3	699.38 s	669.88 s
4	543.56 s	527.00 s
5	450.16 s	439.03 s
6	388.54 s	379.96 s
7	344.61 s	337.48 s
8	312.87 s	305.65 s
9	284.49 s	281.27 s
10	264.40 s	262.26 s
11	248.78 s	249.30 s
12	235.74 s	234.71 s

Table 8.3: Average timings (in seconds) for the Traveling Salesman Problem on a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Distributed` for the results in the second column and `Local` for the results in the third column.

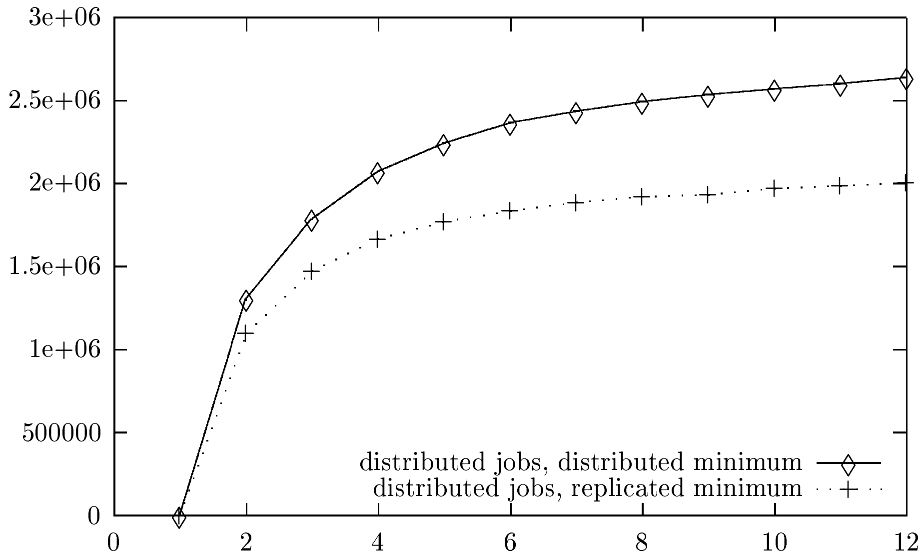


Figure 8.2: Average number of communications in the Traveling Salesman Problem with `Distributed` tuple space access patterns and with a `Replicated` tuple space that contains the minimal tour distance.

CPUs	DISTRIBUTED JOBS	LOCAL JOBS
1	0	0
2	1,305,642	769,571
3	1,790,958	1,198,401
4	2,074,670	1,603,939
5	2,243,772	1,843,049
6	2,367,797	1,989,520
7	2,436,261	2,084,055
8	2,493,835	2,135,379
9	2,536,547	2,187,547
10	2,573,314	2,230,254
11	2,603,222	2,280,133
12	2,641,772	2,310,950

Table 8.4: Average number of communications for the Traveling Salesman Problem on a different number of machines. The access pattern of the tuple space that contains the jobs is `Distributed` for the results in the second column and `Local` for the results in the third column.

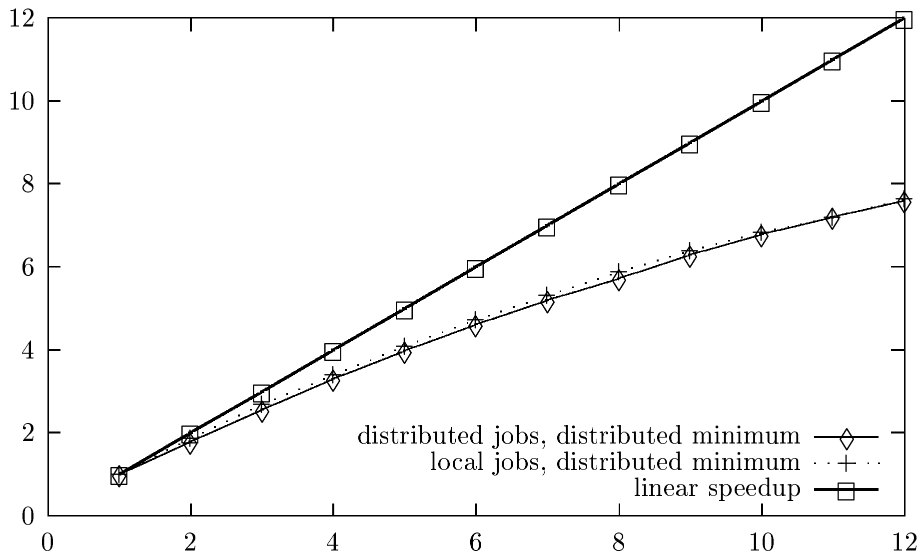


Figure 8.3: Speedups of the Traveling Salesman Problem with `Distributed` access patterns and with a `Local` tuple space that contains the jobs to be calculated.

Analysis Table 8.3 shows that the `Local` job tuple space is about 4% faster than the `Distributed` job tuple space for a small number of CPUs. For higher number of CPUs the advantage goes away completely. Figure 8.3 shows hardly any difference between both TSP implementations, even though the difference in number of communications is considerable (see Figure 8.4).

The lower number of communication for the `Local` jobs tuple is caused by the removal of the communications in the production of the jobs because all jobs are stored in a tuple space that is local to the producer of the jobs.

The smaller performance gain for higher number of processors is unexpected. A possible explanation is that the `Local` jobs tuple space becomes a bottleneck (many communications to this one cpu).

8.4 Experiment: `Local` and `Replicated` Tuple Spaces

Thesis By using both a `Local` tuple space for the jobs (see experiment in Section 8.3) and a `Replicated` tuple space for the minimal traveling distance found so far (see experiment in Section 8.2), the timings of the Traveling Salesman Problem should decrease further.

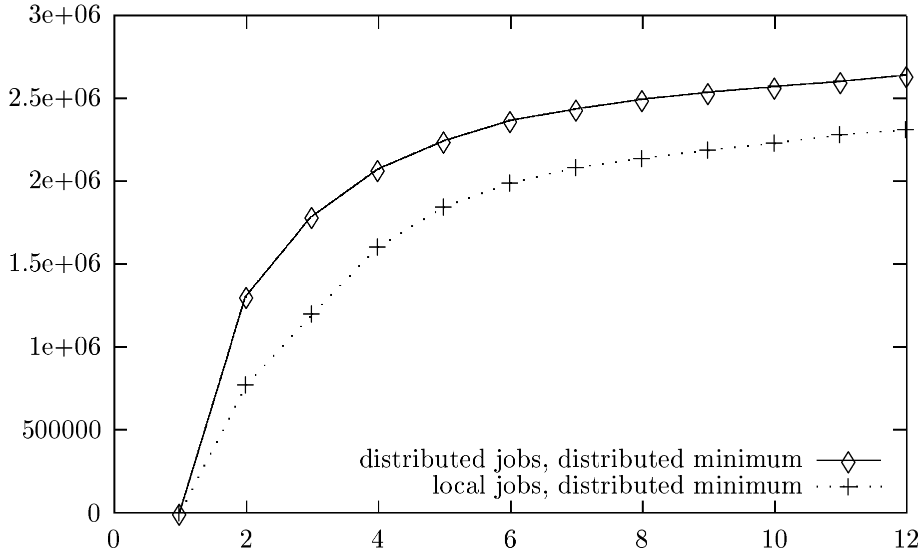


Figure 8.4: Average number of communications of the Traveling Salesman Problem with `Distributed` access patterns and with a `Local` tuple space that contains the jobs to be calculated.

CPUs	Distributed jobs, Distributed minimum	Local jobs, Replicated minimum
1	1792.91 s	1793.61 s
2	1000.91 s	948.89 s
3	699.38 s	663.40 s
4	543.56 s	515.04 s
5	450.16 s	425.51 s
6	388.54 s	366.65 s
7	344.61 s	323.93 s
8	312.87 s	292.42 s
9	284.49 s	268.10 s
10	264.40 s	248.79 s
11	248.78 s	233.26 s
12	235.74 s	221.07 s

Table 8.5: Average timings (in seconds) for the Traveling Salesman Problem on a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Distributed` for the results in the second column and `Local` for the results in the third column. The access pattern of the tuple space that contains the minimal travel distance is `Distributed` for the results in the second column and `Replicated` for the results in the third column.

Experiment Run the same TSP instance with the optimizations introduced in the previous two sections and view the timings and number of communications found.

Results The same sixteen city TSP problem instance of the previous two experiments is used in this experiment. Again the number of jobs generated is a $15 \times 14 \times 13 \times 12 \times 11$ to show the differences in timings more clearly.

Table 8.5 shows the average timings for both TSP implementations. Table 8.6 shows the average number of communications for both TSP implementations.

Analysis The average timings for the optimized TSP are 5 to 6% faster than the Linda version (see Table 8.5). The difference also shows up nicely in the speedup graphs in Figure 8.5. The difference can be explained by the large difference in the numbers of communications as shown in Figure 8.6.

8.5 Experiment: Futures

Thesis Getting tuples asynchronously from the tuple space by using futures can improve the performance of *Blossom* programs.

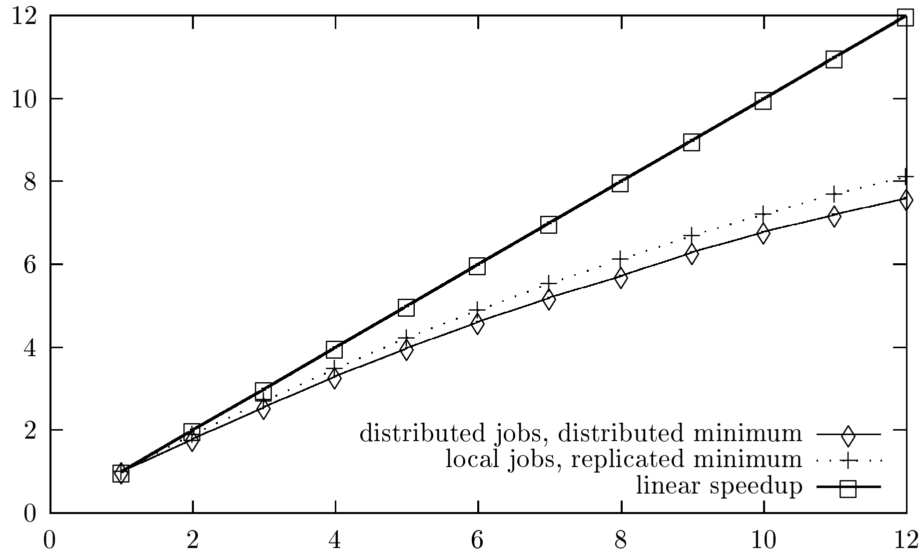


Figure 8.5: Speedup of the Traveling Salesman Problem with a `Distributed` tuple space that contains the jobs and a `Distributed` tuple space that contains the minimal tour, compared with the speedup of the Traveling Salesman Problem with a `Local` tuple space that contains the jobs and a `Replicated` tuple space that contains the minimal tour.

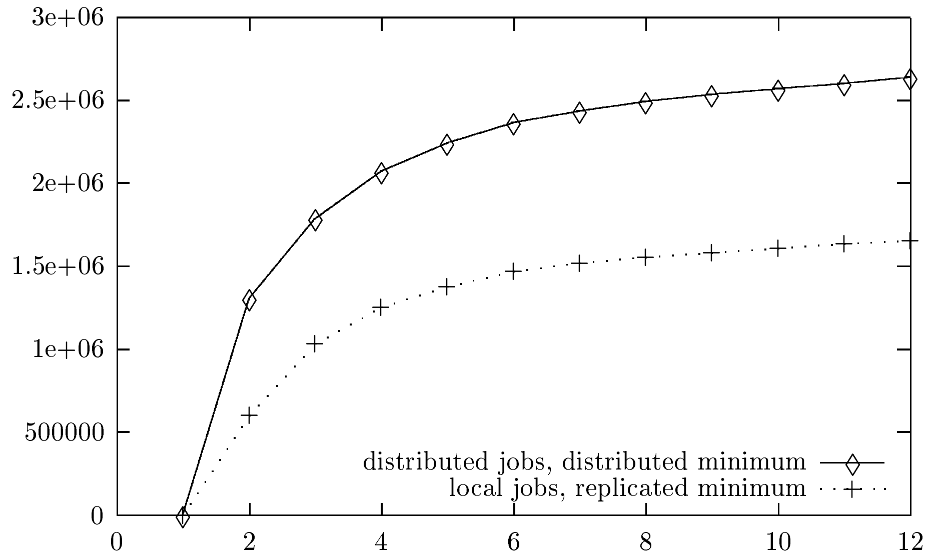


Figure 8.6: Average number of communications of the Traveling Salesman Problem with a `Distributed` tuple space that contains the jobs and a `Distributed` tuple space that contains the minimal tour, compared with the speedup of the Traveling Salesman Problem with a `Local` tuple space that contains the jobs and a `Replicated` tuple space that contains the minimal tour.

CPUs	Distributed jobs, Distributed minimum	Local jobs, Replicated minimum
1	0	0
2	1,305,642	602,112
3	1,790,958	1,034,072
4	2,074,670	1,251,638
5	2,243,772	1,375,343
6	2,367,797	1,469,192
7	2,436,261	1,517,813
8	2,493,835	1,553,024
9	2,536,547	1,582,470
10	2,573,314	1,607,971
11	2,603,222	1,635,919
12	2,641,772	1,654,900

Table 8.6: Average number of communications for the Traveling Salesman Problem on a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Distributed` for the results in the second column and `Local` for the results in the third column. The access pattern of the tuple space that contains the minimal travel distance is `Distributed` for the results in the second column and `Replicated` for the results in the third column.

Methodology A *Blossom* program is run once with synchronous `get` operations (Linda default) and once with asynchronous `get` operations (by using futures, a *Blossom* feature). The second program should run faster than the first one.

The program that demonstrates this behavior is called *Reaction-Diffusion*. The reaction-diffusion algorithm calculates textures based on the simulation of a process of local nonlinear interaction. This process has been proposed as a model of biological pattern formation [58].

Results The reaction-diffusion takes place in a two dimensional array. The parallel version distributes blocks of the array (multiple consecutive columns) over the available CPUs. Every iteration the boundary conditions of the block have to be communicated with the neighbors. The non-future version inserts the boundary conditions in a tuple space before the algorithm calculates the interior of the block. After the interior is calculated the boundary conditions from the neighbor are read from the tuple space. The version that utilizes futures sends off the boundary conditions to the tuple space and immediately sends off a request for the boundary conditions of the neighbor. While calculating the interior the neighbor's boundary conditions should arrive.

In the distributed version of the algorithm without futures, getting the neighbor's boundary data is synchronous, blocking the process from doing anything else than wait for the reply. By introducing futures this time can be spent doing calculations because the future

CPUs	WITHOUT FUTURES	WITH FUTURES
sequential 1	177.86 s	177.86 s
parallel 1	211.97 s	188.60 s
2	106.72 s	94.60 s
3	71.56 s	63.26 s
4	53.86 s	47.56 s
5	43.31 s	38.16 s
6	36.27 s	31.90 s
7	32.67 s	28.81 s
8	27.49 s	24.07 s
9	25.68 s	21.48 s
10	22.23 s	19.39 s
11	20.32 s	18.59 s
12	19.63 s	17.17 s

Table 8.7: Average timings (in seconds) for Reaction-Diffusion using a different number of processors (in separate machines). The results of the program without futures are in the second column and the results of the program with futures are in the third column.

overlaps the communication (getting the boundary data) with the computation (calculating a new iteration of data).

Table 8.7 shows the average timings for the sequential algorithm, and the distributed algorithms with and without futures. The one processor parallel version of the programs uses the tuple space to communicate the boundary conditions to itself, hence the difference in timings with the sequential version.

Analysis The version with futures is about 12% faster than the version without futures. Figure 8.7 shows the speedups of both the version with and without futures (relative to the sequential version). The distance between linear speedup and reaction-diffusion is halved by introducing futures in the algorithm.

8.6 Conclusions

This chapter has shown that *Blossom*'s features (tuple space access patterns and futures) can improve performance. The features are easy to use by simple changes to your program to get different behavior. The user can easily experiment with all these features to find the combination that gives the best performance.

Blossom offers more flexibility than Linda, which results in better performance.

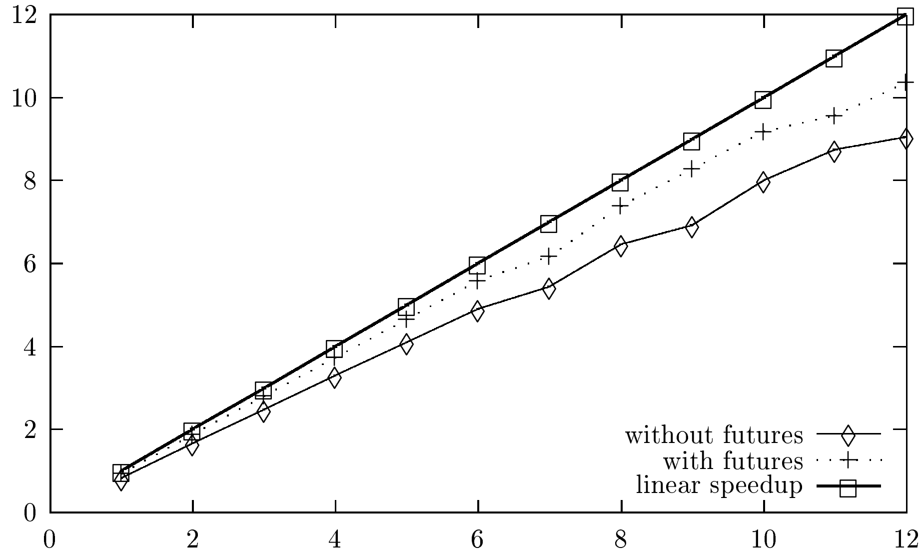


Figure 8.7: Speedups of Reaction-Diffusion with and without futures.

Chapter 9

Conclusions and Future Work

This thesis introduced *Blossom*, an object oriented coordination language based on Linda. The benefits of *Blossom* over Linda are:

- *Blossom* extends Linda with multiple, strongly typed tuple spaces and object orientation. Enhancing the power of the language simplifies the programmer's task of expressing a correct parallel program.
- *Blossom* is implemented as a library and not as a precompiler like SCA Linda. This is achieved by using type parameterization in a novel way to avoid rewriting a C++ parser. This style of implementation allows for efficiency enhancing optimizations.
- Programs written using *Blossom* have extra performance features over programs written in standard Linda.

Blossom could be improved by the following extensions.

Compiler Error Messages If the programmer tries to insert a tuple with an incorrect type in a tuple space the C++ compiler gives an error message during compilation. Although the template related error messages of C++ compilers have improved in clarity, it would be nice to have error messages that are more to the point. It would be nice if the C++ compiler would give the programmer a one line error message telling that the tuple does not correspond to the tuple space definition. Instead C++ compilers give an error message "No member function with prototype ...," where the prototype that follows can easily be five lines long in the case of *Blossom*. Extending *Blossom* with shorter, easier to understand error messages would make it easier to use.

Code Bloat C++ templates provide the programmer with a very powerful tool. It comes however with one big disadvantage called *code bloat*. Code bloat is caused by the generation of new code for every different instantiation of a templated class. The programmer has to actively prevent code bloat from happening by introducing indirections in the data structures. The indirections prevent the C++ compiler from generating code for every specific instantiation; instead generic code is used. Usually the specific code is faster than the generic code, so strategically introducing these indirections is important.

Comparison Some initial comparison tests between SCA Linda and *Blossom* indicate that *Blossom* programs run faster than SCA Linda programs. More elaborate testing is needed to make this a hard claim.

Tuple Space Access Patterns Additional Tuple Space Access Patterns can be implemented to further improve *Blossom*'s performance.

Blossom is an attempt to modernize the Linda programming model by modernizing the language, increasing its performance, and making it more accessible. Ultimately, the real test is to establish a user community. The programming community is very slow to adopt new language technologies. Hopefully *Blossom* that is freely available can attract more interest in this clean and easy programming model.

Appendix A

Blossom Syntax

A.1 Tuple Spaces

TupleSpace	::=	`TupleSpace` < Tuples >
	|	`TupleSpace` < Tuples , TupleSpaceAccessPattern >

A tuple space definition is recognized by the word `TupleSpace` followed by the opening sign < several tuple prototypes, possibly a tuple space access pattern and the closing sign >. If no tuple space access pattern is specified a default tuple space access pattern is assumed.

Tuples	::=	Tuple
	|	Tuple , Tuples

Inside a tuple space definitions several tuple prototypes can be specified. Tuple prototypes are separated by a comma (`,`).

Tuple	::=	`Tuple` < Fields >

The tuple prototype in a tuple space definition is recognized by the word `Tuple` followed by a pair of opening sign < and closing sign >. In between the signs the field types are specified.

Fields	::=	Field
	|	Field , Fields

Inside a tuple prototype one or more fields can be specified. Field types are separated by a comma (`,`).

```
Field  ::=  FieldAccessPattern < Type >
```

A field type consists of a field access pattern followed by the opening sign < the actual type of the field which should have a fixed size and the closing sign >.

```
FieldAccessPattern  ::=  Value
                      |  Variable
                      |  Future
                      |  Synchronous
                      |  Asynchronous
                      |  Query
                      |  Any
```

Seven field access patterns exist. They specify the allowable field arguments of the retrieving operations on a tuple space.

```
Type  ::=  ...
```

In C++ a fixed size type is anything that is a primary type. This means that arrays can in principle not be specified.

```
TupleSpaceAccessPattern  ::=  Local
                           |  Distributed
                           |  Replicated
                           |  ...
```

At the moment only three tuple space access patterns are provided by *Blossom*. Another tuple space access patterns of interest is `Migrating`. A tuple space with this access pattern is migrated to the process that wants to change the tuple space (more or less *à la* Treadmarks [3]).

A.2 Arguments

```
ArugmentType  ::=  Type
               |  BlossomArugmentType < Type >
```

$$
\begin{array}{lcl}
\mathsf{BlossomArugmentType} & ::= & \texttt{Var} \\
& | & \texttt{Fut} \\
& | & \texttt{VarWC} \\
& | & \texttt{FutWC}
\end{array}
$$

Appendix B

Results

This Appendix shows the results of the experiments performed in Chapter 8. For more information on the actual experiments read Chapter 8. All experiments are repeated ten times and this appendix shows all the timings and (if necessary) the number of communications measured of these ten runs. This appendix provides the averages of the timings and the number of communications as well.

B.1 Traveling Salesman Problem

B.1.1 `Distributed` jobs and `Local` minimum

CPUS	TIMINGS					AVERAGE
1	1792.75 s	1793.00 s	1792.88 s	1792.93 s	1793.12 s	
	1792.98 s	1792.93 s	1792.65 s	1792.89 s	1792.95 s	1792.91 s
2	1001.44 s	1000.23 s	1001.17 s	1000.50 s	1000.70 s	
	1002.28 s	1001.46 s	1000.90 s	1000.49 s	999.95 s	1000.91 s
3	699.58 s	699.31 s	699.73 s	698.84 s	699.53 s	
	699.46 s	699.25 s	699.38 s	699.11 s	699.65 s	699.38 s
4	543.49 s	542.77 s	544.36 s	543.60 s	543.20 s	
	543.83 s	543.86 s	543.73 s	543.31 s	543.47 s	543.56 s
5	450.21 s	450.56 s	450.32 s	450.45 s	449.88 s	
	450.52 s	448.99 s	450.60 s	450.25 s	449.81 s	450.16 s
6	388.60 s	388.36 s	388.58 s	389.23 s	388.27 s	
	388.47 s	388.58 s	388.72 s	388.52 s	388.11 s	388.54 s
7	344.40 s	346.25 s	343.92 s	344.93 s	344.00 s	
	344.32 s	344.48 s	344.42 s	344.71 s	344.65 s	344.61 s
8	311.67 s	330.51 s	311.52 s	309.55 s	311.35 s	
	310.80 s	311.57 s	311.28 s	311.35 s	309.11 s	312.87 s
9	281.96 s	282.62 s	286.81 s	286.10 s	285.13 s	
	285.98 s	280.99 s	283.59 s	284.74 s	286.95 s	284.49 s
10	266.17 s	259.65 s	264.37 s	264.63 s	260.46 s	
	265.18 s	265.65 s	265.77 s	266.45 s	265.65 s	264.40 s
11	249.99 s	243.29 s	250.15 s	250.51 s	246.86 s	
	249.29 s	249.84 s	249.79 s	249.85 s	248.21 s	248.78 s
12	237.29 s	236.98 s	237.28 s	237.43 s	230.19 s	
	236.51 s	236.68 s	237.22 s	230.53 s	237.32 s	235.74 s

Table B.1: Timings for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Distributed` and the access pattern of the tuple space that contains the minimum traveling distance is `Local`.

CPUS	COMMUNICATIONS					AVERAGE
1	0	0	0	0	0	
	0	0	0	0	0	0
2	1,305,965	1,306,025	1,306,427	1,305,444	1,305,375	
	1,304,897	1,305,003	1,305,660	1,305,431	1,306,202	1,305,642
3	1,791,561	1,791,261	1,790,371	1,790,531	1,790,070	
	1,790,307	1,792,768	1,791,535	1,790,239	1,790,944	1,790,958
4	2,073,978	2,075,451	2,075,733	2,074,314	2,073,475	
	2,074,348	2,075,342	2,074,740	2,074,882	2,074,440	2,074,670
5	2,245,473	2,243,603	2,243,649	2,244,966	2,243,724	
	2,242,963	2,241,790	2,244,022	2,244,379	2,243,155	2,243,772
6	2,367,891	2,367,907	2,366,583	2,368,418	2,369,031	
	2,367,403	2,367,323	2,368,656	2,366,644	2,368,115	2,367,797
7	2,436,324	2,437,142	2,435,851	2,435,458	2,437,297	
	2,437,290	2,436,590	2,435,505	2,436,244	2,434,912	2,436,261
8	2,493,350	2,496,366	2,493,692	2,494,072	2,493,634	
	2,492,907	2,492,327	2,494,250	2,493,855	2,493,903	2,493,835
9	2,534,280	2,536,507	2,537,429	2,536,116	2,535,101	
	2,536,360	2,539,002	2,536,910	2,537,483	2,536,291	2,536,547
10	2,572,385	2,573,969	2,574,649	2,571,041	2,574,232	
	2,573,344	2,572,674	2,573,620	2,572,543	2,574,687	2,573,314
11	2,601,467	2,602,966	2,602,811	2,603,852	2,603,280	
	2,603,595	2,603,570	2,603,182	2,603,169	2,604,329	2,603,222
12	2,642,692	2,642,072	2,641,355	2,641,928	2,639,945	
	2,642,950	2,641,112	2,640,848	2,642,996	2,641,823	2,641,772

Table B.2: Communications for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Distributed` and the access pattern of the tuple space that contains the minimum traveling distance is `Local`.

B.1.2 `Distributed` jobs and `Replicated` minimum

CPUS	TIMINGS					AVERAGE
1	1792.92 s	1793.67 s	1793.46 s	1793.29 s	1793.54 s	
	1793.53 s	1793.62 s	1793.58 s	1793.39 s	1793.64 s	1793.46 s
2	984.21 s	985.35 s	985.92 s	985.35 s	983.28 s	
	984.43 s	983.90 s	984.97 s	984.48 s	984.84 s	984.67 s
3	681.16 s	681.68 s	680.70 s	681.84 s	684.68 s	
	683.26 s	681.25 s	680.66 s	681.27 s	681.15 s	681.76 s
4	527.63 s	528.15 s	527.55 s	527.81 s	527.80 s	
	527.71 s	527.82 s	527.96 s	528.21 s	528.51 s	527.92 s
5	436.65 s	436.75 s	436.37 s	436.87 s	436.80 s	
	436.30 s	436.63 s	436.60 s	436.62 s	436.74 s	436.63 s
6	375.43 s	375.79 s	375.15 s	375.47 s	374.86 s	
	375.42 s	375.62 s	375.36 s	375.23 s	375.72 s	375.41 s
7	332.53 s	332.48 s	332.25 s	332.67 s	332.56 s	
	332.73 s	332.44 s	332.86 s	332.56 s	332.88 s	332.60 s
8	300.56 s	300.43 s	299.60 s	300.38 s	300.53 s	
	300.12 s	300.85 s	300.15 s	300.85 s	301.37 s	300.48 s
9	275.26 s	276.31 s	275.44 s	276.13 s	275.73 s	
	276.26 s	274.57 s	274.22 s	274.66 s	278.32 s	275.69 s
10	255.81 s	256.14 s	255.85 s	255.78 s	255.99 s	
	256.26 s	255.32 s	255.55 s	255.89 s	256.16 s	255.87 s
11	238.44 s	239.96 s	239.80 s	240.12 s	239.94 s	
	238.84 s	240.83 s	239.64 s	239.84 s	239.78 s	239.72 s
12	226.60 s	228.15 s	226.99 s	227.41 s	227.41 s	
	227.19 s	227.63 s	227.68 s	227.60 s	227.25 s	227.39 s

Table B.3: Timings for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Distributed` and the access pattern of the tuple space that contains the minimum traveling distance is `Replicated`.

CPUS	COMMUNICATIONS					AVERAGE
1	0	0	0	0	0	
	0	0	0	0	0	0
2	1,100,132	1,104,533	1,100,169	1,107,051	1,094,659	
	1,099,290	1,098,328	1,101,432	1,099,003	1,101,506	1,100,610
3	1,475,805	1,475,835	1,473,895	1,474,786	1,479,272	
	1,473,641	1,471,817	1,471,807	1,475,853	1,474,274	1,474,698
4	1,664,895	1,665,665	1,665,182	1,664,474	1,665,013	
	1,665,195	1,665,449	1,665,392	1,665,752	1,666,453	1,665,347
5	1,768,240	1,768,907	1,768,409	1,768,882	1,769,250	
	1,768,794	1,768,513	1,768,109	1,768,719	1,768,161	1,768,598
6	1,836,259	1,836,513	1,836,577	1,836,665	1,837,895	
	1,836,109	1,836,487	1,835,632	1,835,081	1,836,568	1,836,378
7	1,885,890	1,885,476	1,885,300	1,885,323	1,883,668	
	1,884,873	1,884,761	1,885,396	1,885,850	1,885,422	1,885,195
8	1,918,608	1,920,671	1,920,390	1,920,167	1,920,030	
	1,919,953	1,919,533	1,919,408	1,919,444	1,920,681	1,919,888
9	1,947,304	1,946,183	1,946,929	1,947,787	1,947,558	
	1,947,356	1,947,548	1,947,365	1,947,383	1,792,539	1,931,795
10	1,969,646	1,969,249	1,968,854	1,969,874	1,969,423	
	1,969,160	1,969,556	1,969,342	1,969,516	1,969,321	1,969,394
11	1,986,228	1,986,919	1,985,837	1,986,279	1,986,538	
	1,987,058	1,986,580	1,986,938	1,986,487	1,987,041	1,986,590
12	2,003,784	2,004,010	2,004,157	2,003,946	2,003,795	
	2,003,649	2,003,798	2,003,784	2,004,124	2,003,805	2,003,885

Table B.4: Communications for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Distributed` and the access pattern of the tuple space that contains the minimum traveling distance is `Replicated`.

B.1.3 `Local` jobs and `Local` minimum

CPUS	TIMINGS					AVERAGE
1	1792.78 s	1792.87 s	1792.96 s	1791.98 s	1793.66 s	
	1792.95 s	1793.95 s	1792.97 s	1792.78 s	1794.15 s	1793.11 s
2	963.77 s	963.44 s	963.22 s	963.68 s	963.71 s	
	963.77 s	963.24 s	963.67 s	963.49 s	963.12 s	963.51 s
3	669.61 s	670.14 s	669.38 s	669.40 s	670.27 s	
	670.26 s	669.70 s	670.30 s	670.15 s	669.63 s	669.88 s
4	526.57 s	527.37 s	526.75 s	526.78 s	527.32 s	
	527.45 s	526.72 s	526.98 s	526.96 s	527.15 s	527.00 s
5	438.88 s	439.40 s	438.75 s	438.71 s	438.79 s	
	439.29 s	438.97 s	438.93 s	439.41 s	439.13 s	439.03 s
6	378.96 s	379.93 s	379.29 s	379.21 s	379.75 s	
	379.49 s	379.78 s	382.80 s	379.56 s	380.80 s	379.96 s
7	337.21 s	337.69 s	337.82 s	337.21 s	337.33 s	
	337.57 s	337.55 s	337.46 s	337.56 s	337.41 s	337.48 s
8	305.29 s	305.73 s	305.30 s	305.77 s	305.33 s	
	305.49 s	305.72 s	305.44 s	306.88 s	305.51 s	305.65 s
9	280.76 s	281.18 s	280.93 s	281.34 s	280.95 s	
	281.78 s	281.62 s	281.37 s	281.32 s	281.45 s	281.27 s
10	261.69 s	262.29 s	263.52 s	261.84 s	262.20 s	
	262.18 s	261.95 s	262.30 s	262.24 s	262.42 s	262.26 s
11	247.27 s	246.80 s	269.79 s	246.73 s	246.87 s	
	246.89 s	247.20 s	247.67 s	246.86 s	246.88 s	249.30 s
12	237.17 s	234.27 s	234.38 s	233.61 s	234.16 s	
	233.83 s	237.40 s	234.24 s	233.84 s	234.16 s	234.71 s

Table B.5: Timings for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Local` and the access pattern of the tuple space that contains the minimum traveling distance is `Local`.

CPUS	COMMUNICATIONS					AVERAGE
1	0	0	0	0	0	
	0	0	0	0	0	0
2	768,366	770,991	769,990	768,205	768,618	
	769,878	769,514	770,081	770,361	769,710	769,571
3	1,197,594	1,197,784	1,200,254	1,198,672	1,197,174	
	1,200,226	1,197,503	1,197,363	1,198,238	1,199,211	1,198,401
4	1,602,752	1,605,924	1,601,835	1,604,573	1,605,413	
	1,603,659	1,605,482	1,602,226	1,602,921	1,604,608	1,603,939
5	1,841,839	1,842,204	1,842,504	1,843,897	1,844,443	
	1,843,897	1,841,405	1,843,512	1,841,663	1,845,129	1,843,049
6	1,989,179	1,990,658	1,987,991	1,988,361	1,990,040	
	1,989,208	1,991,853	1,988,025	1,988,767	1,991,126	1,989,520
7	2,083,547	2,084,914	2,084,918	2,083,455	2,083,794	
	2,085,055	2,083,045	2,083,753	2,084,032	2,084,043	2,084,055
8	2,135,061	2,135,064	2,135,232	2,135,219	2,135,148	
	2,135,169	2,137,056	2,135,169	2,135,078	2,135,596	2,135,379
9	2,186,628	2,186,789	2,187,972	2,187,335	2,187,647	
	2,187,563	2,188,595	2,187,129	2,187,430	2,188,391	2,187,547
10	2,229,971	2,229,729	2,229,295	2,230,118	2,230,125	
	2,231,003	2,230,597	2,230,534	2,231,568	2,229,603	2,230,254
11	2,281,223	2,280,082	2,274,424	2,280,950	2,280,366	
	2,281,055	2,280,079	2,281,090	2,281,178	2,280,884	2,280,133
12	2,311,321	2,310,502	2,311,199	2,309,862	2,310,408	
	2,310,184	2,311,486	2,311,741	2,311,181	2,311,619	2,310,950

Table B.6: Communications for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Local` and the access pattern of the tuple space that contains the minimum traveling distance is `Local`.

B.1.4 `Local` jobs and `Replicated` minimum

CPUS	TIMINGS					AVERAGE
1	1793.66 s	1793.59 s	1793.64 s	1793.78 s	1793.54 s	
	1793.61 s	1793.61 s	1793.54 s	1793.50 s	1793.63 s	1793.61 s
2	948.11 s	948.89 s	948.91 s	948.89 s	948.88 s	
	948.92 s	949.79 s	948.84 s	948.69 s	948.94 s	948.89 s
3	663.20 s	663.97 s	663.58 s	663.15 s	663.22 s	
	663.49 s	663.27 s	663.24 s	663.43 s	663.45 s	663.40 s
4	514.78 s	514.85 s	515.32 s	514.79 s	514.72 s	
	515.38 s	515.13 s	514.62 s	514.97 s	515.88 s	515.04 s
5	426.41 s	425.45 s	425.52 s	425.64 s	425.50 s	
	425.35 s	425.45 s	425.22 s	425.21 s	425.34 s	425.51 s
6	366.21 s	366.17 s	366.43 s	366.52 s	366.39 s	
	366.42 s	366.51 s	369.11 s	366.28 s	366.43 s	366.65 s
7	323.48 s	324.19 s	323.95 s	323.69 s	323.92 s	
	323.76 s	323.83 s	324.88 s	323.98 s	323.60 s	323.93 s
8	291.94 s	292.92 s	291.84 s	292.33 s	292.72 s	
	292.62 s	292.18 s	292.71 s	292.53 s	292.38 s	292.42 s
9	267.62 s	267.97 s	267.88 s	267.90 s	268.24 s	
	268.30 s	267.90 s	268.84 s	268.11 s	268.24 s	268.10 s
10	248.55 s	248.79 s	248.25 s	248.62 s	248.66 s	
	250.93 s	248.25 s	248.61 s	248.74 s	248.46 s	248.79 s
11	233.37 s	233.38 s	232.66 s	233.63 s	233.16 s	
	233.70 s	233.14 s	233.24 s	233.17 s	233.12 s	233.26 s
12	220.31 s	220.94 s	220.60 s	220.58 s	220.95 s	
	220.69 s	224.36 s	220.88 s	220.88 s	220.46 s	221.07 s

Table B.7: Timings for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Local` and the access pattern of the tuple space that contains the minimum traveling distance is `Replicated`.

CPUS	COMMUNICATIONS					AVERAGE
1	0	0	0	0	0	
	0	0	0	0	0	0
2	601,932	602,087	602,107	601,572	602,187	
	602,082	602,537	602,597	601,797	602,227	602,112
3	1,034,608	1,030,783	1,035,352	1,034,493	1,032,593	
	1,034,643	1,035,242	1,035,097	1,034,393	1,033,518	1,034,072
4	1,253,106	1,249,027	1,253,103	1,250,404	1,250,443	
	1,251,242	1,252,419	1,251,003	1,252,962	1,252,671	1,251,638
5	1,374,505	1,375,599	1,375,728	1,375,938	1,375,259	
	1,375,388	1,376,429	1,374,280	1,375,569	1,374,744	1,375,343
6	1,469,767	1,468,296	1,469,388	1,468,782	1,470,255	
	1,469,229	1,469,156	1,467,843	1,469,125	1,470,082	1,469,192
7	1,517,521	1,517,512	1,518,498	1,518,242	1,518,573	
	1,517,918	1,517,047	1,517,764	1,517,999	1,517,063	1,517,813
8	1,552,330	1,552,097	1,553,572	1,552,322	1,553,612	
	1,554,531	1,552,747	1,553,597	1,552,387	1,553,051	1,553,024
9	1,582,054	1,583,399	1,581,973	1,582,719	1,582,295	
	1,582,244	1,582,764	1,582,604	1,582,478	1,582,174	1,582,470
10	1,608,130	1,607,781	1,607,610	1,608,021	1,607,411	
	1,607,872	1,608,711	1,608,226	1,607,385	1,608,565	1,607,971
11	1,636,470	1,636,188	1,636,613	1,635,870	1,634,952	
	1,635,765	1,635,330	1,636,150	1,635,955	1,635,902	1,635,919
12	1,654,752	1,654,753	1,654,737	1,655,101	1,654,963	
	1,655,267	1,654,461	1,655,441	1,654,616	1,654,917	1,654,900

Table B.8: Communications for the Traveling Salesman Problem (TSP) using a different number of processors (in separate machines). The access pattern of the tuple space that contains the jobs is `Local` and the access pattern of the tuple space that contains the minimum traveling distance is `Replicated`.

B.1.5 Sequential

CPUS	TIMINGS					AVERAGE
1	1781.56 s	1785.70 s	1783.28 s	1782.24 s	1785.55 s	
	1785.16 s	1784.53 s	1785.72 s	1786.42 s	1782.26 s	1784.24 s

Table B.9: Timings for the sequential Traveling Salesman Problem (TSP).

B.2 Reaction-Diffusion

B.2.1 Without Futures

CPUS	TIMINGS					AVERAGE
1	212.08 s	211.88 s	212.03 s	211.96 s	211.99 s	
	211.97 s	211.85 s	211.98 s	211.93 s	211.98 s	211.97 s
2	106.78 s	106.78 s	106.67 s	106.65 s	106.67 s	
	106.74 s	106.72 s	106.74 s	106.59 s	106.81 s	106.72 s
3	71.66 s	71.45 s	71.61 s	71.65 s	71.59 s	
	71.57 s	71.56 s	71.55 s	71.47 s	71.54 s	71.56 s
4	53.95 s	53.79 s	53.91 s	53.87 s	53.89 s	
	53.88 s	53.86 s	53.82 s	53.83 s	53.79 s	53.86 s
5	43.28 s	43.31 s	43.30 s	43.32 s	43.28 s	
	43.27 s	43.35 s	43.32 s	43.32 s	43.33 s	43.31 s
6	36.27 s	36.25 s	36.24 s	36.28 s	36.29 s	
	36.30 s	36.29 s	36.26 s	36.29 s	36.23 s	36.27 s
7	32.69 s	32.68 s	32.68 s	32.68 s	32.65 s	
	32.67 s	32.66 s	32.65 s	32.65 s	32.65 s	32.67 s
8	27.48 s	27.48 s	27.46 s	27.48 s	27.50 s	
	27.47 s	27.49 s	27.54 s	27.49 s	27.47 s	27.49 s
9	25.67 s	25.66 s	25.68 s	25.68 s	25.67 s	
	25.67 s	25.69 s	25.68 s	25.70 s	25.68 s	25.68 s
10	22.24 s	22.23 s	22.25 s	22.23 s	22.22 s	
	22.23 s	22.23 s	22.22 s	22.22 s	22.22 s	22.23 s
11	20.30 s	20.32 s	20.33 s	20.32 s	20.32 s	
	20.30 s	20.32 s	20.32 s	20.32 s	20.32 s	20.32 s
12	19.78 s	19.59 s	19.59 s	19.61 s	19.58 s	
	19.58 s	19.81 s	19.59 s	19.58 s	19.59 s	19.63 s

Table B.10: Timings for Reaction-Diffusion using a different number of processors (in separate machines) where the program does not utilize futures.

B.2.2 With Futures

CPUS	TIMINGS					AVERAGE
1	188.60 s	188.56 s	188.66 s	188.50 s	188.52 s	
	188.48 s	188.67 s	188.69 s	188.66 s	188.62 s	188.60 s
2	94.58 s	94.60 s	94.56 s	94.61 s	94.58 s	
	94.62 s	94.61 s	94.62 s	94.62 s	94.61 s	94.60 s
3	63.28 s	63.25 s	63.25 s	63.24 s	63.29 s	
	63.25 s	63.24 s	63.29 s	63.26 s	63.28 s	63.26 s
4	47.54 s	47.55 s	47.53 s	47.61 s	47.56 s	
	47.58 s	47.53 s	47.58 s	47.60 s	47.56 s	47.56 s
5	38.18 s	38.14 s	38.19 s	38.14 s	38.14 s	
	38.15 s	38.16 s	38.15 s	38.18 s	38.16 s	38.16 s
6	31.88 s	31.89 s	31.89 s	31.92 s	31.89 s	
	31.89 s	31.91 s	31.92 s	31.90 s	31.90 s	31.90 s
7	28.82 s	28.82 s	28.80 s	28.83 s	28.80 s	
	28.81 s	28.80 s	28.79 s	28.83 s	28.80 s	28.81 s
8	24.08 s	24.06 s	24.07 s	24.07 s	24.08 s	
	24.09 s	24.08 s	24.07 s	24.08 s	24.07 s	24.07 s
9	21.47 s	21.48 s	21.48 s	21.48 s	21.49 s	
	21.48 s	21.48 s	21.48 s	21.50 s	21.49 s	21.48 s
10	19.39 s	19.38 s	19.41 s	19.39 s	19.41 s	
	19.38 s	19.39 s	19.41 s	19.40 s	19.38 s	19.39 s
11	18.59 s	18.59 s	18.60 s	18.60 s	18.60 s	
	18.59 s	18.58 s	18.58 s	18.58 s	18.59 s	18.59 s
12	17.13 s	17.13 s	17.13 s	17.50 s	17.23 s	
	17.13 s	17.13 s	17.11 s	17.11 s	17.12 s	17.17 s

Table B.11: Timings for Reaction-Diffusion using a different number of processors (in separate machines) where the program utilizes futures.

B.2.3 Sequential

CPUS	TIMINGS					AVERAGE
1	177.90 s	177.88 s	177.88 s	177.85 s	177.81 s	
	177.95 s	177.91 s	177.83 s	177.86 s	177.81 s	177.87 s

Table B.12: Timings for sequential Reaction-Diffusion.

Summary

In the early 1980s, David Gelernter came up with a new parallel programming paradigm, which he called Linda. Linda provides the programmer with a shared memory abstraction called the tuple space. Processes in a Linda program communicate with each other by reading from and writing to the tuple space. Reading tuples from the tuple space is generally done by providing a partially specified tuple: some fields are specified (constants), others are unspecified (variables). The variables are instantiated with the values found in a matching tuple. Since variables are typed, tuples are typed, however the Linda tuple space is typeless!

This dissertation introduces a new parallel programming language, called *Blossom*, that is similar to Linda, however it has typed tuples *and* typed tuple spaces. *Blossom* is implemented in C++. By providing typing, the Linda tuple space is divided into multiple *Blossom* tuple spaces. This division of the tuple space has several advantages. First, it allows the programmer to specify a distribution per tuple space, resulting in higher performance of parallel programs. Second, it allows for better software engineering. For example, since tuple spaces are variables, the programmer can have local tuple spaces or pass tuple spaces as arguments to functions. Several other advantages are discussed in the dissertation.

Whereas traditional Linda implementations use a precompiler approach, *Blossom* uses a library approach. The *Blossom* library is based on type parameterization, which has several advantages. First, a library is easier to develop and maintain than a precompiler, because the precompiler needs to parse the underlying programming language. Second, a type parameterized library allows potentially more performance enhancements than a precompiler. The programmer has to pay a small price by declaring the tuple spaces in his *Blossom* programs. Third, the extra programmer's declarations provide additional compile time support.

To make *Blossom* programs run as fast as Linda programs, the limits of type parameterization have to be pushed. This dissertation shows how you can use C++'s templates for important tasks such as minimize the memory usage of tuples and tuple spaces, and it also shows how you can achieve difficult tasks as type-type conversions and type-value conversions.

Underneath the high-level abstraction of tuple spaces, low-level communication takes place. Ordinary communication libraries provide basic point-to-point communication. To overcome that problem a new communications library *Roots* was designed and implemented. *Roots* allows for communication through active messages, which provide a higher abstraction of (point-to-point) communication. *Roots* is implemented in C++ and it outperforms

the traditional communication package Parallel Virtual Machine (PVM).

Finally, *Blossom* programs that utilize *Blossom*'s specific enhancements are shown to outperform *Blossom* programs that only utilize Linda specific enhancements. The former programs use less communication or utilize better parallelism than the latter programs.

Samenvatting

In het begin van de tachtiger jaren verzon David Gelernter een nieuw paradigma voor parallel programmeren, genaamd Linda. Linda geeft de programmeur een gemeenschappelijk geheugen abstractie genaamd de tuple ruimte. Processen in een Linda programma communiceren met elkaar door middel van het schrijven en lezen in de tuple ruimte. Het lezen van tuples van de tuple ruimte wordt in het algemeen gedaan met een partieel gespecificeerd tuple: enkele velden zijn gespecificeerd (constanten) en enkele zijn ongespecificeerd (variabelen). De variabele velden worden geïnstantieerd met de waardes gevonden in een overeenkomstig tuple. Aangezien variabelen zijn getypeerd, zijn tuples getypeerd. Echter de Linda tuple ruimte is ongetypeerd!

Dit proefschrift introduceert een nieuwe parallelle programmeertaal, *Blossom*, vegelijkbaar met Linda, maar met getypeerde tuples *en* tuple ruimtes. *Blossom* is geïmplementeerd in C++. De typering splits de Linda tuple ruimte in meerdere *Blossom* tuple ruimtes. Deze splitsing van de tuple ruimte heeft verschillende voordelen. Ten eerste, dit geeft de programmeur de mogelijkheid om een distributie te specificeren per tuple ruimte, wat resulteerd in snellere parallelle programma's. Ten tweede, dit maakt betere software engineering mogelijk. Bijvoorbeeld, aangezien tuple ruimtes variabelen zijn, kan de programmeur lokale tuple ruimtes hebben of tuple ruimtes doorgeven als argumenten aan functies. Andere voordelen worden in dit proefschrift besproken.

Waar traditionele Linda implementaties een voorvertaler benadering kiezen, gebruikt *Blossom* een bibliotheek benadering. De *Blossom* bibliotheek is gebaseerd op type parameterisatie, wat een aantal voordelen heeft. Ten eerste, een bibliotheek is eenvoudiger te ontwikkelen en onderhouden dan een voorvertaler, omdat de voorvertaler de onderliggende programmeertaal moet kunnen analyseren. Ten tweede, een type geparameteriseerde bibliotheek staat potentieel meer efficiëntie verbeteringen toe dan een voorvertaler. De programmeur betaald een kleine prijs door het declareren van de types van de tuple ruimtes. Ten derde, deze extra declaraties van de programmeur geven hem meer help ten tijde van het vertalen.

Om *Blossom* programma's even snel als Linda programma's te laten draaien, moeten de limieten van type parameterisatie worden gezocht. Dit proefschrift legt uit hoe men gebruik kan maken van C++ templates voor belangrijke zaken als het minimalizeren van geheugen gebruik, en hoe men moeilijke zaken als type-type conversies en type-waarde conversies kan doen.

Onder de abstractie van tuple ruimtes, vindt communicatie plaats. Normale communicatie bibliotheken bieden eenvoudige punt-naar-punt communicatie. Om dit probleem te omzeilen, is een nieuwe communicatie bibliotheek, *Roots*, ontworpen en geïmplementeerd. *Roots* communiceert door middel van actieve boodschappen, die een hoger niveau abstractie bieden dan punt-naar-punt communicatie. *Roots* is geïmplementeerd in C++ en presteerd beter dan de traditionele communicatie bibliotheek *Parallel Virtual Machine (PVM)*.

Tot slot, *Blossom* programmas die gebruik maken van specifieke *Blossom* extensies presteren beter dan *Blossom* programma's die alleen gebruik maken van extensies die aanwezig zijn in Linda implementaties. De eerst genoemde programma's communiceren minder of parallelizeren beter dan de laatst genoemde programma's.

Bibliography

[1] AGHA, G. A. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, University of Michigan, Computer and Communication Science, 1985. also MIT AI Laboratory Technical Report 844.

[2] AMERICAN NATIONAL STANDARD (ADOPTED ISO/IEC STANDARD). *Programming Languages – C++*, Sept. 1998. Document Number: ANSI/ISO/IEC 14882.

[3] AMZA, C., COX, A. L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer 29*, 2 (Feb. 1996), 18–28.

[4] BAKKEN, D. E., AND SCHLICHTING, R. D. Supporting Fault-Tolerant Parallel Programming in Linda. *IEEE Transactions on Parallel and Distributed Systems 6*, 3 (1995), 287–302.

[5] BAL, H. E. *Programming Distributed Systems*. Prentice Hall International, 1991.

[6] BAL, H. E., AND HAINES, M. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency 6*, 3 (1998), 74–84.

[7] BENNETT, J. K., CARTER, J. B., AND ZWAENEPOEL, W. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)* (1990), pp. 168–176.

[8] BROWN, T., JEONG, K., LI, B., TALLA, S., WYCKOFF, P., AND SHASHA, D. *PLinda User Manual*. http://merv.cs.nyu.edu:8001/∼binli/plinda/manual.ps, 1997.

[9] BUTCHER, P., WOOD, A., AND ATKINS, M. Global Synchronisation in Linda. *Concurrency: Practice and Experience 6*, 6 (Sept. 1994), 505–516.

[10] CARDELLI, L., AND WEGNER, P. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys 17*, 4 (Dec. 1988), 471–522.

[11] CARREIRA, J., SILVA, L., AND SILVA, J. G. On the design of Eileen: a Linda-like library for MPI. In *Proceedings of the 1994 Scalable Parallel Libraries Conference* (1995), pp. 175–184.

[12] CARRIERO, N., FREEMAN, E., GELERNTER, D., AND KAMINSKY, D. Adaptive Parallelism and Piranha. *IEEE Computer 28*, 1 (Jan. 1995), 40–49.

[13] CARRIERO, N., AND GELERNTER, D. *Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1990, ch. 7, pp. 114–125.

[14] CHIEN, A. A., AND DOLBY, J. T. *ICC++*. In Wilson and Lu [57], 1996, ch. 9, pp. 343–382.

[15] FRIEDMAN, R., GOLDIN, M., ITZKOVITZ, A., AND SCHUSTER, A. Millipede: Easy Parallel Programming in Available Distributed Environments. *Software: Practice and Experience 27*, 8 (Aug. 1997), 929–965.

[16] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns*. Addison-Wesley, 1995.

[17] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. *PVM Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[18] GELERNTER, D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (Jan. 1985), 80–112.

[19] GELERNTER, D. Multiple Tuple Spaces in Linda. In *Proceedings PARLE'89: Parallel Architectures and Languages Europe* (June 1989), pp. 20–27.

[20] GROPP, W., LUSK, E., DOSS, N., AND SKJELLUM, A. A high-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing 22*, 6 (Sept. 1996), 789–828.

[21] GROPP, W., LUSK, E., AND SKJELUM, A. *Using MPI*. MIT Press, 1994.

[22] HALSTEAD, A. R. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems 7*, 4 (1985), 501–538.

[23] HASSEN, S. B., BAL, H. E., AND JACOBS, C. J. H. A Task- and Data-Parallel Programming Language based on Shared Objects. *ACM Transactions on Programming Languages and Systems 20*, 6 (Nov. 1998), 1131–1170.

[24] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1984.

[25] HU, W., SHI, W., AND TANG, Z. JIAJIA: An SVM System Based on A new Cache Coherence Protocol. In *Proceedings of the High Performance Computing and Networking (HPCN'99), LNCS 1593* (Amsterdam, The Netherlands, Apr. 1999), Springer, pp. 463–472.

[26] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*, IEEE Std 1003.1 ed. Piscataway, N.J., 1996.

[27] KALE, L. V., BHANDARKAR, M., JAGATHESAN, N., KRISHNAN, S., AND YELON, J. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the International Parallel Processing Symposium* (Honolulu, Hawaii, Apr. 1996).

[28] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[29] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[30] KIELMANN, T. Designing a Coordination Model for Open Systems. In *Coordination Languages and Models, First International Conference COORDINATION '96* (1996), P. Ciancarini and C. Hankin, Eds., pp. 267–284.

[31] LAURE, E., MEHROTRA, P., AND ZIMA, H. Opus: Heterogeneous Computing With Data Parallel Tasks. *IEEE Concurrency 6*, 3 (1998), 74–84.

[32] LEWIS, B., AND BERG, D. J. *Multithreaded Programming with Pthreads*. Sun Microsystems Press, 1998.

[33] LOWENTHAL, D. K., FREEH, V. W., AND ANDREWS, G. R. Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing. *Journal of Parallel and Distributed Computing 37* (1996), 41–54.

[34] LU, P. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proc. 11th International Parallel Processing Symposium* (Geneva, Switzerland, Apr. 1997), pp. 467–473. Available at http://www.cs.ualberta.ca/~paullu/.

[35] MONNERAT, L. R., AND BIANCHINI, R. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th IEEE International Symposium on High-Performance Computer Architecture (HPCA'98)* (Feb. 1998).

[36] MUELLER, F. On the Design and Implementation of DSM-Threads. In *International Conference on Parallel and Distributed Processing Techniques and Applications* (June 1997), pp. 315–324.

[37] MYRIAS COMPUTER TECHNOLOGIES INC. *The PAMS User's Guide*, 1996.

[38] NICHOLAS JOHN CARRIERO, J. *Implementation of Tuple Space Machines*. PhD thesis, Yale University, 1987. also Research Report YALEU/DCS/RR-567.

[39] O'FARRELL, W. G., EIGLER, F. C., PULLARA, S. D., AND WILSON, G. V. *ABC++*. In Wilson and Lu [57], 1996, ch. 1, pp. 1–42.

[40] RICHARDS, M., AND WHITBY-STREVANS, C. *BCPL The Language and its Compiler*. Cambridge University Press, Cambridge, 1979.

[41] ROSCOE, A. W., AND HOARE, C. A. R. The Laws of Occam Programming. *Theoretical Computer Science 60*, 2 (Sept. 1988), 177–229.

[42] ROWSTRON, A., AND WOOD, A. Solving the Linda Multiple `rd` Problem. In *Coordination Languages and Models, First International Conference COORDINATION '96* (1996), P. Ciancarini and C. Hankin, Eds., pp. 357–367.

[43] SCHMIDT, D. C., HARRISON, T., AND AL-SHAER, E. Object-Oriented Components for High-speed Network Programming. In *Proceedings of the 1st Conference on Object-Oriented Technologies and Systems* (June 1995), pp. 21–38.

[44] SCIENTIFIC COMPUTING ASSOCIATES, INC. *Linda User's Guide & Reference Manual*, 1995.

[45] STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., AND SCOTT, M. CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France, Oct. 1997), pp. 170–183.

[46] STEVENS, W. R. *UNIX Network Programming, volume 2: Interprocess Communications*. Prentice Hall, 1999.

[47] SUBHLOK, J., AND YANG, B. A New Model for Integrated Nested Task and Data Parallel Programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, June 1997), pp. 145–156.

[48] SZAFRON, D., AND SCHAEFFER, J. An Experiment to Measure the Usability of Parallel Programming Systems. *Concurrency: Practice and Experience 8*, 2 (1996), 147–166.

[49] TEZUKA, H., O'CARRELL, F., HORI, A., AND ISHIKAWA, Y. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS) and the 9th Symposium on Parallel and Distributed Processing (SPDP)* (Apr. 1998), pp. 308–314.

[50] VAN DER GOOT, R. *Blossom*. http://www.cs.ualberta.ca/~roel/blossom/, 1998.

[51] VAN DER GOOT, R. *Roots*. http://www.cs.ualberta.ca/~roel/roots/, 1998.

[52] VAN DER GOOT, R. Roots: An Object-Oriented Active Message Library. In *High Performance Computer Systems 1999 (HPCS'99)* (1999), A. Pollard, D. J. K. Mewhort, and D. F. Weaver, Eds.

[53] VAN DER GOOT, R., AND DE BRUIN, A. Syntax and Semantics of PROCOL. In *Mathematical Foundations of Computer Science 1995 (MFCS'95), LNCS 969* (1995), J. Wiedermann and P. Hajek, Eds., Springer Verlag, pp. 509–518.

[54] VAN DER GOOT, R., SCHAEFFER, J., AND WILSON, G. V. Safer Tuple Spaces. In *Coordination Languages and Models* (1997), D. Garlan and D. L. Metayer, Eds., Spinger-Verlag, pp. 289–301.

[55] VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSSER, K. E. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)* (May 1992), pp. 256–266.

[56] WEST, E. A., AND GRIMSHAW, A. S. Braid: Integrating Task and Data Parallelism. In *Symposium on the Frontiers of Massively Parallel Computation* (McLean, Virginia, Feb. 1995), pp. 211–219.

[57] WILSON, G. V., AND LU, P., Eds. *Parallel Programming Using C++*. MIT Press, 1996.

[58] WITKIN, A., AND KASS, M. Reaction-Diffusion Textures. *Computer Graphics (SIGGRAPH '91 Proceedings) 25*, 4 (1991), 299–308.

Curriculum Vitae

In September 1993 Roel van der Goot received his Master's degree in Computer Engineering at Delft University of Technology, The Netherlands, with honors. As part of his Ph.D. studies he was invited in 1996 to the Center of Advanced Studies at IBM Toronto Labs and to the Computing Science Department at the University of Alberta in Edmonton (both Canada). Due to the great research environment at the latter institute he would frequent it in the years to follow. In 1999 he became an SGI Origin 2000 programmer analyst/system administrator for MACI (Multimedia and Advanced Computing Institute) at the University of Alberta. He was promoted in 2000 to become the MACI systems manager. Later that year he joined the Edmonton based startup YottaYotta as a systems developer.

The Tinbergen Institute is the Institute for Economic Research, which was founded in 1987 by the Faculties of Economics and Econometrics of the Erasmus Universiteit Rotterdam, Universiteit van Amsterdam and Vrije Universiteit Amsterdam. The Institute is named after the late Professor Jan Tinbergen, Dutch Nobel Prize laureate in economics in 1969. The Tinbergen Institute is located in Amsterdam and Rotterdam. The following books recently appeared in the Tinbergen Institute Research Series:

184. N. van Giersbergen, *Bootstrapping dynamic econometric models.*
185. K.M. Schilstra, *Industrial relations and human resource management: A network approach.*
186. H. Supratikno, *Competitive strategy, production organization and inter-firm collaboration. A case study of subcontracting arrangements in three manufacturing firms in Indonesia.*
187. J. Gouwens, *Interne consistentie in personeelsmanagement: Naar een typering van HRM in grote bedrijven in Nederland.*
188. F. Pot, *Continuity and change of human resource management: A comparative analysis of the impact of global change and cultural continuity on the management of labour between the Netherlands and the United States.*
189. M.J. Kleijn, *Demand differentiation in inventory systems.*
190. J.F.M. Sweegers, *Coordination, cooperation and institutions.*
191. P.A. de Hek, *Economic growth and fluctuations: A theoretical enquiry.*
192. J. Meijaard, *Decision making in research and development: A comparative study of multinational companies in the Netherlands and the United States.*
193. K.P.B. Oldenkamp, *Derivatives in portfolio management.*
194. C.M.M.P. Wetzels, *Squeezing birth into working life. Household panel data analyses comparing Germany, Great Brittain, Sweden and the Netherlands.*
195. J.M. van Leeuwen, *Armoede en koopkrachtongelijkheid; theoretische beschouwingen over het effect van inkomensafhankelijke prijzen.*
196. P. Frijters, *Explorations of welfare and well-being.*
197. J.A. Cabral Vieira, *The evolution of wage structures in Portugal 1982-1992.*
198. M.J. Smits, *Technology choice and the role of institutions. The case of wheat and the input market in Pakistan.*
199. M.T. Kögel, *Services and economic growth.*
200. D.J.C. van Dijk, *Smooth transition models: Extensions and outlier robust inference.*

201. E. Kaper, *Panel effects in consumer research. Statistical models for underreporting.*

202. K. Sadiraj, *Albania: Transition to a market economy.*

203. A. Ranasinghe, *A pearl of great price: The free education system in Sri Lanka.*

204. P.J. van der Sluis, *Estimation and inference with the efficient method of moments: With applications to stochastic volatility models and option pricing.*

205. P.G.W. Alders, *Family ties in an aging world. Causes and consequences of fertility shifts.*

206. P. van Hasselt, *Dynamics of price formation in financial markets.*

207. K. Verweire, *Performance consequences of financial conglomeration with an empirical analysis in Belgium and the Netherlands.*

208. P.W.T. Ghijsen, *Labour and technology in Japan. An analysis of labour adjustment and technological change.*

209. E. Drissen, *Government decisions on income redistribution and public production. A political-economic general equilibrium approach.*

210. J. Spreeuw, *Heterogeneity of hazard rates in insurance.*

211. G.T. Post, *Finding the frontier: Methodological advances in data envelopment analysis.*

212. L.D. Meijers, *Ruimtelijke netwerken van de zakelijke dienstverlening.*

213. R.P. Plasmeijer, *Maintenance optimisation techniques for the preservation of highways.*

214. J. Tuinstra, *Price dynamics in equilibrium models.*

215. P.A. Groenendijk, *Essays on exchange rate dynamics.*

216. S.M. de Bruyn, *Economic growth and the environment: An empirical analysis.*

217. Y. Schipper, *Market structure and environmental costs in aviation. A welfare analysis of European air transport reform.*

218. E. van Gameren, *The internal economics of firms. An investigation into the labour mobility within firms.*

219. A.J. Dur, *Political institutions and economic policy choice.*

220. B.E. Baarsma, *Monetary valuation of environmental goods: Alternatives to contingent valuation.*

221. E.C. van der Sluis-den Dikken, *Management learning and development: The role of learning opportunities and learning behavior in management development and career success.*

222. A.J.H. Pels, *Airport economics and policy: Efficiency, competition and interaction with airlines.*

223. B. van der Klaauw, *Unemployment duration determinants and policy evaluation.*

224. F. Medda, *The assembled city: Analyses of multiple urban dimensions.*

225. A.F. Tieman, *Evolutionary game theory and equilibrium selection.*

226. R.R.P. Kouwenberg, *Dynamic asset liability management.*

227. J.S. Sidhu, *Organization mission, business domain orientation and performance: A conceptual and empirical inquiry.*

228. G. Romijn, *Economic dynamics of Dutch construction.*

229. M.C. Versantvoort, *Analysing labour supply in a life style perspective.*

230. J.J.J. Groen, *Testing multi-country exchange rate models.*

231. C.F.A. van Wesenbeeck, *How to deal with imperfect competition: introducing game-theoretical concepts in general equilibrium model of international trade.*

232. M.L. Ndoen, *Migrants and entrepreneurial activities in peripheral Indonesia. A socioeconomic model of profit-seeking behaviour.*

233. L.A. Grogan, *Labour market transitions of individuals in eastern and western Europe.*

234. E.G. Van de Mortel, *An institutional approach to transition processes.*

235. P.H. van Oijen, *Essays on corporate governance.*

236. H.M.M. van Goor, *Banken en industriefinanciering in de 19e eeuw. De relatie tussen Mees en Stork, Van den Bergh gaat naar Engeland.*

237. F.R.M. Portrait, *Long-term care services for the Dutch elderly. An investigation into the process of utilization.*

238. M. van de Velde, *Topics in correspondence analysis.*

239. G. Draisma, *Parametric and semi-parametric methods in extreme value theory.*

240. I.F.C. Mulder, *Soil degradation in Benin: Farmers' perceptions and responses.*

241. A.W. Saly, *Corporate entrepreneurship. Antecedents and consequences of entrepreneurship in large established firms.*

242. S. van Velzen, *Supplements to the economics of household behavior.*