Institute of Informatics of the University of Fribourg (Switzerland)

Parallel Optimizations

Advanced Constructs and Compiler Optimizations for a Parallel, Object Oriented, Shared Memory Language running on a distributed System

Thesis

submitted to the Faculty of Science of the University of Fribourg (Switzerland)
in conformity with the requirements for the degree of
Doctor scienticarum informaticarum

by

Claudio Fleiner

from Aarau, Switzerland

Thesis #1148

Imprimerie St. Paul, Fribourg, 1997

Accepted by the Faculty of Science of the University of Fribourg (Switzerland) on the proposal of Professors Dr. Béat HIRSBRUNNER (University of Fribourg, Switzerland), Dr. Jerôme FELDMAN (University of Berkeley, CA and Director of the International Computer Science Institute) and Dr. Rolf INGOLD (University of Fribourg, Switzerland).

Fribourg, April 11th, 1997

The Dean:

Prof. Dr. Jean-Paul BERRUT

This thesis is available online at `http://www.icsi.berkeley.edu/~fleiner/thesis`
The author can be reached at `fleiner@acm.org`

Abstract

Today's processors provide more and more processing power, yet there are many applications whose processing demand cannot be met by a single processor in the near future, besides, the demand for more processing power seems to increase at least as fast as the speed of new processors and the only way to complete such calculation-intensive programs is to execute them on many processors at once. The history of parallel computers of the last several years suggests that the distributed, parallel computer model will gain widespread acceptance as the most important one. In this model a computer consists of several node, each with its own processors and memory, As such a computer does not offer one global memory space, but rather a separate memory per node (distributed memory), it is no longer possible to directly use the shared memory programming paradigm. However, as it is generally easier to program with shared memory rather than using message based communications, several new languages and language extensions that simulate shared memory have been suggested.

Such a parallel, distributed language has not only to provide special support for managing parallelism and synchronization, the specification and implementation of the language has to address the issue of distributed memory as well. One of the most important issues is the selection of the memory consistency model, which defines when writes of one node are observed by the other nodes of the distributed computer. Many vital optimizations used by compilers for serial languages are often not possible if the memory model is too restrictive, but a weaker memory model makes the language harder to use.

This thesis discusses several problems and solutions for such languages. It uses the language pSather, an object oriented, parallel language developed at the International Computer Science Institute in Berkeley as an example. A very flexible synchronization construct including different implementations of it, is introduced that allows the user to define new synchronization primitives, and avoids deadlocks and starvations in many common cases.

Several memory consistency models and their implications for programmers and the compiler, especially regarding optimizations, are discussed. The effect of several optimizations (adaptations of optimizations used in serial compilers and special parallel optimizations) and their implementation will be shown. The effect of those optimizations will be measured by using test programs written in pSather. The results clearly indicate that a weaker memory model is necessary to achieve the desired efficiency and speedup, even though usage of the language becomes less convenient. However, pSather offers some constructs that solve some of the problems.

The thesis concludes with a summary of the results and some ideas for additional work.

Zusammenfassung

Die heutigen Prozessoren werden zwar immer schneller, trotzdem gibt es nach wie vor viele Anwendungen, deren Verlangen nach Rechenleistung wesentlich grösser ist als die Prozessoren, die in naher Zukunft verfügbar sein werden. Ausserdem steigt der Bedarf nach mehr Rechenleistung mindestens so schnell wie die Leistung der neuen Prozessoren. Die einzige Möglichkeit, um solche rechenintensiven Anwendungen zu implementieren sind parallele Rechner. Die Entwicklung der parallelen Computer der letzten Jahre deutet darauf hin, dass verteilte, aus mehreren Knoten mit eigenem Speicher und einem oder mehreren Prozessoren bestehende Parallelrechner die grösste Verbreitung haben werden. Solche Rechner bieten allerdings nicht mehr einen globalen Speicher an, sondern jeder Knoten des Rechners hat seinen eigenen, privaten Speicherbereich (distributed memory). so dass das gemeinsame Speicherbenutzungsmodel (shared memory) nicht mehr zutrifft. Nun ist es aber schwieriger, Programme zu schreiben, deren Teilstücke nur mit Hilfe von Meldungen, die verschickt werden, Daten austauschen können. Aus diesem Grund wurden verschiedene neue Sprachen und Spracherweiterungen vorgeschlagen, die einen gemeinsamen Speicher simulieren.

Eine parallele, verteile Sprache muss es nicht nur erlauben, die einzelnen Teile zu verwalten und zu synchronisieren, sondern sowohl in der Definition der Sprache wie auch in der Implementierung muss die Simulation des gemeinsamen Speichers klar definiert sein. Einer der wichtigsten Punkte hier ist die Wahl und Definition des Speichermodels (memory consistency model), das u.a. festhält, wann ein Knoten die Änderungen, die ein anderer Konten im Speicher gemacht hat, sieht. Viele der in seriellen Sprachen häufig eingesetzten Optimierungen sind nicht möglich, wenn das Speichermodel zu restriktiv ist, anderseits sind laxere Modelle schwieriger zu programmieren.

Diese Dissertation diskutiert verschiedene Probleme und Lösungsmöglichkeiten für parallele, verteilte Sprachen. Diese werden an der Sprache pSather erläutert. Diese objekt-orientiert, parallele Sprache wurde am International Computer Science Institute in Berkeley entwickelt. U.a. wird ein sehr flexibler Synchronisationbefehl eingeführt, der es dem Benutzer erlaubt, neue Synchronizationsmodelle zu implementieren und der gleichzeitig in vielen Fällen sowohl “Deadlock” wie auch “Starvation” verhindern kann.

Verschiedene Speichermodelle und deren Auswirkungen für den Programmierer und den Übersetzer, speziell im Bezug auf Optimierungen, werden vorgestellt und diskutiert. Der Effekt der verschiedenen Optimierungen (sowohl von Optimierungen die von seriellen Sprachen bekannt sind und angepasst wurden wie auch spezielle parallele Optimierungen) und deren Implementierungen werden vorgestellt. Die Auswirkungen der Optimierungen werden mit verschiedenen Testprogrammen, die in pSather geschrieben sind, gemessen. Die Resultate zeigen eindeutig dass laxere Speichermodelle notwendig sind, um die gewünschte Geschwindigkeit zu erreichen und das dies den Nachteil einer etwas komplexeren Programmierung klar aufwiegt.

Keywords

Parallel, object oriented language,
parallel compiler optimizations,
synchronization,
deadlock avoidance
memory consistency models,
shared memory simulation,
distributed computing.

Acknowledgments

I would like to thank my advisors, Prof. Dr. Béat Hirsbrunner and Prof. Dr. Jerôme Feldman, for their advice, encouragement and support while writing this thesis at the University of Fribourg in Switzerland and the International Computer Science Institute in Berkeley.

This thesis would have not been possible without the financial support of the Swiss National Science Foundation, received as a scholarship from the University of Fribourg in Switzerland and which allowed me to work during two years at the International Computer Science Institute under the supervision of Prof. Dr. Jerôme Feldman in the Sather/pSather group.

Without the many people that contributed to the Sather/pSather project at the International Computer Science Institute (ICSI) this thesis would not exist. During my two year stay at the ICSI I had the pleasure to work together with Ben Gomes, Holger Klawitter, Michael Philippsen, David Stoutamire, Boris Weissman, Cliff Draper and Arno Jacobsen, which are just some of the persons that have shaped the current Sather/pSather compiler.

I also have to thank The National Research Supercomputer Center (NERSC) for the computer time and staff support. Most of the distributed examples were running on their Meiko CM-5.

Boris Weissman, Ben Gomes, David Stoutamire and Luke O'Connor read different draft versions of the thesis, and their comments and remarks were very helpful.

I have also to thank my wife for her support, as without her this thesis would not have been possible!

Table Of Contents

List of Tables

List of Figures

1. Introduction

Today Processors run at 300Mhz and faster, can read and write 64- bits at the same time, use several levels of cache memory and get faster every year yet are still too slow to solve many important problems in a reasonable time. Weather simulation, for example, is not yet able to accurately predict the weather over a long time or in small areas like in mountains and valleys. Routinely, owners of Swiss mountain restaurants are angry because the television announces nice weather for the Weekend, so they buy a lot of meat and on Saturday they find themselves within a cold cloud and too many sausages, or, vice versa, a lot of hungry customers but nothing to sell as the weather turned out much better than predicted.

The only way to run such calculation-intensive programs is to execute them on many processors at once. This problem will not change in the near future, as the need for more processing power seems to increase at least as fast as the processing power of the newest computers available. Several different parallel computer models have been proposed and implemented, with some of the main differences beeing:

- **Power and number of the processors**: either a few but powerful processors are used, as in the Meiko CS-2 with up to four Sparc processors per node, or many, but less powerful ones as in transputers.
- **The connection between processors**: how fast is the communication, can each processor address all other processors or only a few neighbors?
- **Single instruction (SIMD) versus multiple instruction (MIMD):** can each processor execute its own program or will all processors execute the same program step per cycle as in the PRAM model?
- **Memory**: each processor can have its own memory (a small one in the case of neuronal networks, a lot in the case of the Meiko CS-2 or the Gigabooster) or all processors share the same memory (as in the multi-processor Sparcs).

The history of parallel computers of the last several years suggest that the distributed, parallel computer model will gain widespread acceptance except for specialized applications [Feldman 94]. In this model we connect many standard “off the shelf” computers or computer boards, each with one or several CPUs, its own cache and memory, and sometimes even its own disc system with a high speed network as shown in figure 1-1. This model is seen in large multiprocessors like Meiko’s CS-2 [Meiko 95] or the Gigabooster developed by the ETH in Zurich and Supercomputing Systems SA [Gunzinger 92], and also networks of workstations connected with high speed networks, such as the Myrinet used in the NOW project at UC Berkeley [Anderson 95b].

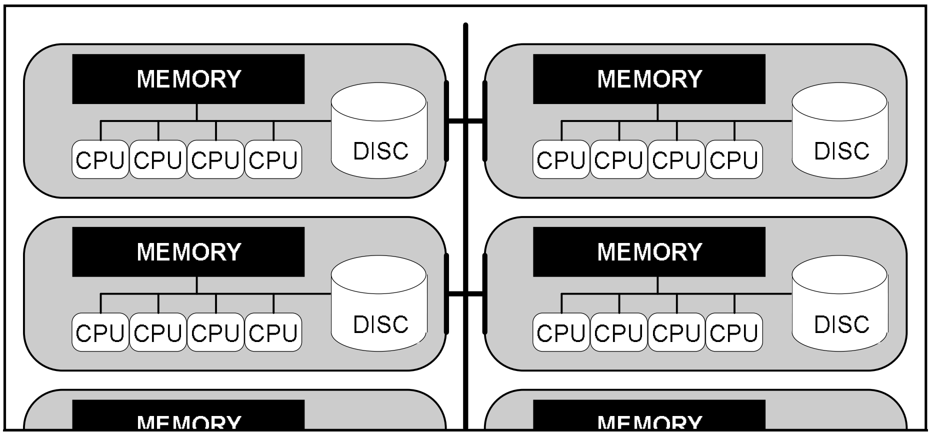

Figure 1-1: A Distributed Computer System with 4 CPU's per node

Obviously, the languages used to program serial, one processor systems are no longer adequate for parallel, distributed systems, which in turn are not useful for neuronal networks or SIMD computers. There are several options for programmers that write programs for distributed computer systems:

- **use a parallelizing compiler**: such a compiler translates a serial program into one that runs in parallel on a distributed computer system. This approach is quite limited though, as it is often not possible to parallelize a serial algorithm and get good results, regardless of the efforts of the compiler. The differential evolution program described in appendix C.3.3 uses, for example, a slightly different algorithm for the parallel and serial versions. The serial version uses just one random number generator, while the parallel one uses one per thread, but has to make sure that the random number generater get initialized correctly to avoid that they yield the same numbers to each thread. If it used just one generator, all accesses to it would have to be serialized. Many common serial algorithms, such as various sorting algorithms, Alpha/Beta search, finding the n-th largest number in a distributed array, matrix multiplication and others, cannot be automatically parallelized. On the other hand, the work invested by the programmer is minimal and all programs and libraries already written can use at least some of the additional processing power. This approach is used by high performance FORTRAN compiler where a lot of work has been done for loop optimizations.
- **use parallel libraries**: this approach is especially useful for numerical work, where the programmer uses a library for most calculations. A parallel version of the widely used FORTRAN BLAS library [Lawson 79] used for matrix calculation, for example, would speed up all serial programs that use this library by simply relinking those programs[1].
- **use a communication library**: in this case the programmer writes a program in any of the standard serial languages for each node of the distributed computer (often the same program runs on each node) and these communicate with each other by exchanging messages. By using one of the communication standards currently

---

[1] If the system the program is running on uses shared or dynamically linked libraries it may not even be necessary to relink the program, as the system will automatically use the parallelized version of the library if it is available.

available (for example PVM, MPI, AM, which are described in more detail in chapter 2.2), it is possible to write portable programs that run on a variety of computer systems. This programming model clearly gives the user more power than those outlined above, as he has more control over the parallelism and can implement algorithms tailored for the underlying architecture.

- **use a parallel language**: many parallel languages and parallel language extensions of serial languages have been developed for distributed computer systems. They have different programming paradigms (object oriented, shared memory model, active objects, data parallel or threaded), expressiveness (synchronization possibilities, thread creation, thread management), and other special properties (fault tolerance, special hardware support). Unfortunately most run on only a very limited number of computer systems and programs written in such languages are not very portable yet. However, these languages have some important advantages over communication libraries:
  - the shared memory model is generally easier to use,
  - specialized constructions for thread creation and synchronization make programming easier and more secure, as the compiler can check certain properties since it is aware of the parallelism involved.
  - compiler support allows for better error checking, debugging facilities and, most important, optimizations of the parallel constructs.

This thesis will discuss some of the problems and implementations of the last two advantages, namely a flexible synchronization mechanism and compiler optimizations for parallel, object oriented languages. While synchronization is described in detail in chapter 3, we will now discuss some of the special problems that occur when optimizing parallel programs.

Compilers for high level languages are supposed to create machine code that is as good as the hand coded version of the same program. Unfortunately, there is no compiler yet that produces such code. However, many serial compilers use optimizations to either increase the speed of the code they generate or to get smaller programs[2]. Such optimizing compilers are widely used for serial languages. In fact, even freely available compilers like the GNU C/C++ compiler optimize code in different ways. As shown in chapter 5 simple optimizations can easily make a program an order of magnitude faster.

Obviously, we would like to use optimizations in compilers for parallel languages as well, especially as normally only programs that use a lot of CPU cycles and run for hours are parallelized and therefore already small optimizations are measurable and welcome.

Unfortunately, some of the most useful optimizations used in compilers for serial languages cannot be applied to many parallel languages, as the compiler can rely on the single execution thread in serial languages, which makes those languages deterministic in most cases. This allows the compiler to know exactly which portions of the code may change which memory locations. With this knowledge, it is possible to rear-

[2] Smaller programs (binaries) are still important, as instruction cache faults are less frequent for smaller programs, and hence they tend to be executed faster.

range code to get higher speed without affecting other parts of the program. Examples of such optimizations include common subexpression eliminations and loop invariant hoisting (see chapter 5.6 for a description of those optimizations).

Many parallel languages are non-deterministic and may have several execution threads that may run on different processors at the same time or share the same processor. To prove that a memory location cannot be altered by some random thread is in most cases impossible for a compiler, even when analyzing the complete program.

Figure 1-2 shows a simple Sather example where an obvious optimization (loop invariant hoisting) is only possible in a serial language but not in a parallel one.

```
loop
    -- for each iteration another array index is assigned to i.
    i::=0.upto!(array.size-1);
    a:=a+array[i]*b*size;
end;

-- the same loop after hoisting the loop invariant outside the loop
tmp::=b*size;
loop
    i::=0.upto!(array.size-1);
    a:=a+array[i]*tmp;
end;
```

Figure 1-2: A simple Sather loop and its optimization

If this code is executed as part of a parallel program, and neither `b` nor `size` are local variables but attributes of an object (`self` in this case), these variables are no longer loop invariants as another thread could well change their values, given access to the same object `self`. It could even be the case that the call of `array.elt!` uses some synchronization and waits until some thread has changed the value of `b` (actually, this is often quite easy to detect if the program uses standard synchronization calls, but not if it uses busy loops like in figure 1-3). Although busy loops are not generally useful in parallel programs, as they use CPU cycles without producing anything, they can be used if it is certain that they will only run for a few microseconds. In such cases it would be more expensive to switch to another thread. This works of course only on multiprocessor systems with shared memory.

```
-- wait until a thread signals that it updated variable b by setting w to true
loop until!(w); end;
```

Figure 1-3: Synchronization using busy waits

We will now take a look at different optimization techniques used in modern compilers after defining two important characteristics of optimizations:

- **correctness**: an optimized program must still produce a sensible and, as far as the language specification is concerned, correct result. Parallel programs may no longer produce the same result though, as the speedup of threads changes their scheduling, and any result that relies on a specific scheduling will necessarily change when optimizations are used. However, such scheduling dependencies are either irrelevant, bad programming style, or, even worse, coding errors that are only exposed when optimizing the program.

A frequent error in pSather programs for example is to assume that the result of another thread will be immediately visible. This is not the case for the memory consistency model used in pSather. Figure 1-4 shows an example that actually runs with the current compiler, unless optimizations are turned on. As pSather does not guarantee that a thread sees changes made by another thread before executing an `import`[3], `obj.found` is, as far as the compiler is concerned, a loop invariant and therefore the test can safely be evaluated before the loop. pSather offers several different possibilities to write this code in a much cleaner and more elegant way by using one of the synchronization classes provided by the pSather library (see appendix A for an introduction to pSather synchronization primitives).

```
    obj.found:=false;
    -- start a search thread on each cluster
    parloop do@clusters!
        loop
            -- end the thread if the result has been found
            until!(obj.found);
            -- do some more searching
            if found_result then break!; end;
        end;
        obj.found:=true;
    end;

--  optimizations replace the loop with
        tmp::=obj.found;   -- obj.found is a loop invariant
        loop
            until!(tmp);
            -- do some more searching
            if found_result then break!; end;
        end;

--  To make the example work, we have to introduce an import:
        loop
            -- make sure we get the newest version of obj.found
            import;
            until!(obj.found);
            -- do some more searching
            if found_result then break!; end;
        end;
```

Figure 1-4: Erroneous pSather code

[3] A pSather thread checks for changes of the global memory only when executing an import. See chapter 4.4 for a description of the Release Consistency Model.

- **speed up**: an optimization that does not speed up a program seems to be a contradiction. However, most compilers make assumptions about the code and optimize it in a way that yields speed up for the standard case. A compiler can, for example, safely assume that a loop is normally executed more than once and therefore it is faster to evaluate loop invariant expressions before entering the loop. If, on the other hand, the loop is frequently left during the first iteration it may well be that invariant expressions have been evaluated for nothing. If this is the standard case, as shown for example In figure 1-5, the compiler (or, more precisely, the programmer and user) loses.

```
loop
    -- random_number returns a number between 0 and 1
    if random_number>0.01 then break!; end;
    -- none of the expressions following this line should be hoisted,
    -- as they are only evaluated about once in  every 100 times this loop
    -- is executed.
end;
```

Figure 1-5: Unusual loop

Technique	Example[4]
reusing intermediate results (CSE)	• eliminating duplicate evaluations of the same expression. • storing global values in registers for multiple use.
moving instructions	• hoisting loop invariant expressions out of loops so that they are evaluated once and not on every iteration. • changing the order of instructions to use pipelined processors more efficiently.
replacing instructions	• inlining to avoid function calling, which is relatively expensive and to enable other optimizations like CSE. • loop unrolling, where the code of body of the loop is repeated several times to avoid the jump and test at the end or beginning of a loop • replacing slow instructions with faster ones (for example multiplication with bit shifting).
deleting instructions	• dead code elimination • elimination of useless code (double assignments to the same variable, pushing an exception frame onto an exception stack which is never used)

Table 1-1: Serial optimization technique

[4] Most of those optimizations are described in chapter 5.

Technique	Example[5]
caching values	• caching: values read from another processor can be stored in a software cache, so other threads running on the same processor can reuse those values (runtime optimization). • or the values can be stored in local variables (compile time optimization, this is similar to register usage in serial languages).
scheduling	• threads can be moved and rescheduled to achieve good load balancing (runtime and/or compile time optimizations). • It may sometimes be faster to execute the code of several threads within one thread to reduce the number of context switches.
moving data	• by moving data to the correct place prior to its use it is possible to omit unnecessary waiting time (runtime and/or compile time optimizations).
cutting down on communication	• message pipelining combines several messages into one to make better use of network hardware.

Table 1-2: Parallel optimization techniques

Most of the optimizations used in serial compilers use one of the techniques described in table 1-1. All of those optimizations are done at compile time. A shared memory, distributed[6], parallel language adds some more techniques, some of which are outlined in table 1-2. Those optimizations are no longer compile time optimizations only, but also optimizations implemented in the runtime library, which may need support from the compiler to execute them.

To apply some of those techniques to parallel languages we have to define semantics for global reads and writes that allow the compiler to apply those optimizations. Chapter 4 describes several of those so-called Memory Consistency Models and their implications for the programmer, the compiler and the runtime.

Most of the optimizations described in table 1-1 are done at one of two levels inside the compiler, either by working on the intermediate code or when generating assembly or machine code. Parallel languages offer at least one more place for optimizations, namely the runtime of the parallel system. The runtime system is responsible for the communication between different threads, the code necessary to simulate shared memory on distributed systems and the synchronization of threads. It consists partly of a library that is linked with the pSather program and code fragments inserted by the compiler in the generated C code. The structure of the pSather compiler developed at the ICSI and the different opportunities for optimizations are shown in figure 1-6. As the pSather compiler generates C code it relies to some extent on a good C compiler. In fact, it needs a C compiler that is at least able to make sensible use of registers for

---

[5] Most of those optimizations are described in chapter 6 and chapter 7.

[6] Even though pSather uses the shared memory model, it runs on a distributed system by simulating shared memory

local variables, as the pSather compiler uses a lot of local variables without reusing them and assumes that the C compiler will take care of reusing the same memory positions for different locals if possible.

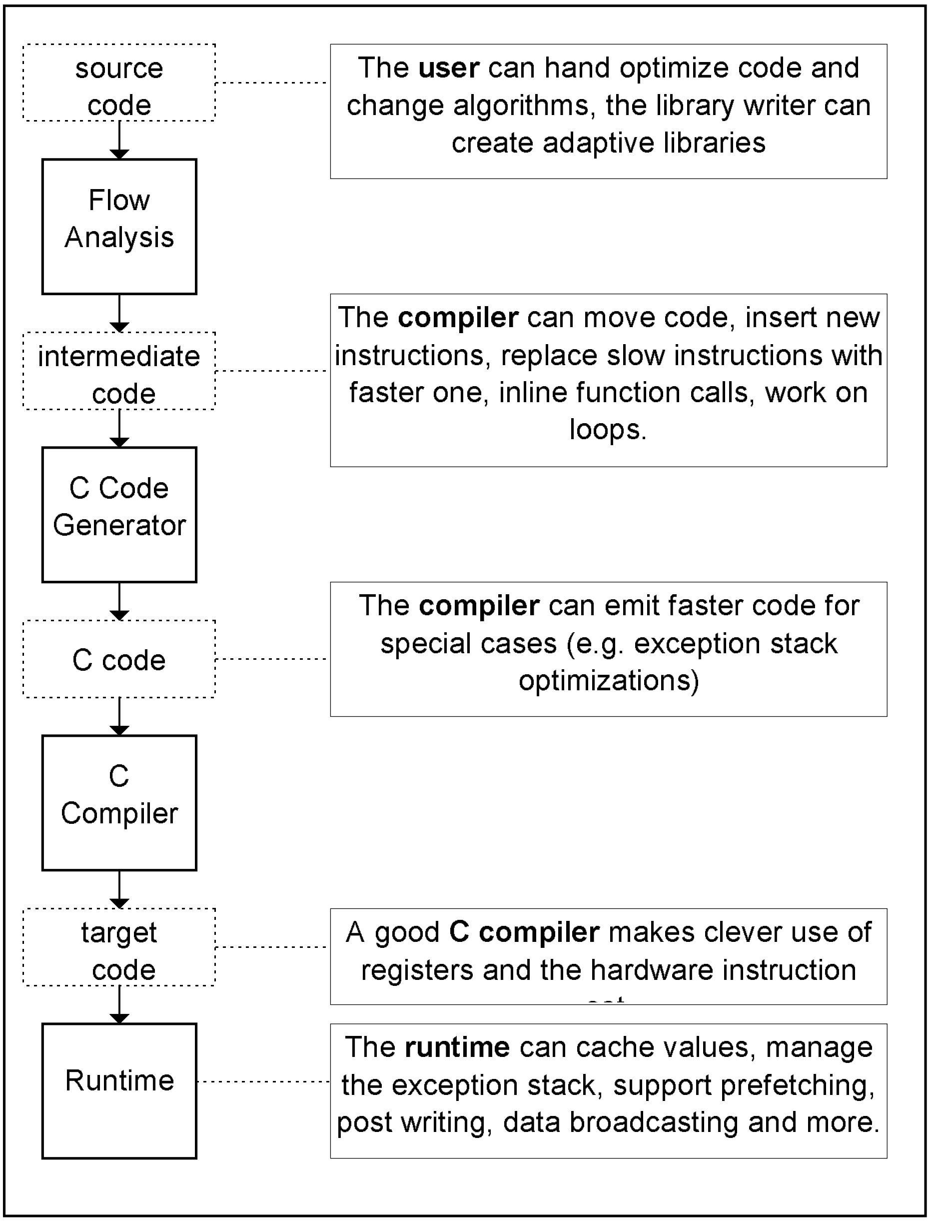

Figure 1-6: Places for optimizations in the pSather compiler

For the rest of this thesis we distinguish between three different types of optimizations:

1) **Serial Optimizations**: adaptation of serial optimizations for parallel, distributed languages. Those optimizations are all done during compilation. Examples are common subexpression elimination and inlining.

2) **Parallel Optimizations**: special optimizations for parallel, distributed languages regarding memory distribution and thread scheduling, some of them are compiler op-

timizations, other runtime optimizations. Examples include caching of remote objects and prefetching of objects.

3) **Synchronization Optimizations**: pSather supports a flexible and easy-to-use synchronization mechanism, which replaces locks, semaphores and regions. A naive implementation of this mechanism is relatively slow, but several optimizations (runtime and compile time) allow for an acceptable speed.

The remainder of this thesis is organized as follows:

The next chapter describes related work in the field of high performance computing on different architectures with different languages and optimizations. Chapter 3 describes some advanced and powerful synchronization constructs for an object oriented language and possible implementations. Chapter 4 discusses different memory models used in parallel, shared memory languages and why it is important to choose an adequate memory model for an efficient parallel language. Chapter 5 is an introduction of several well-known optimization strategies for serial languages and how they can be used in parallel languages while chapter 6 describes the parallel optimizations implemented in pSather. Chapter 7 has an introduction regarding the synchronization mechanism optimizations of pSather. The last chapter summarizes the results.

A short introduction to Sather and pSather can be found in Appendix A, while Appendix B describes the pSather runtime and its implementation. Timings and speed ups of optimizations are measured with several programs described in Appendix C and the optimization options of the current compiler are described in Appendix D.

[7] An introduction to the Sather language and its extension pSather can be found in appendix A. The complete Compiler including the language specification, tutorials and the language specification is available online at `http://www.icsi.berkeley.edu/~sather`.

2. Related Work

2.1 Distributed Computers

In this chapter we briefly examine several distributed computers. A distributed computer is a system that consists of several nodes[8] connected via a low latency network. Each node may have one or more CPU's that share memory among each other. The main difference between the systems described here is the fact that some use "off-the-shelf-products", while other use special customized hardware and/or software. While systems with custom built hardware are generally slightly faster, they are more expensive to build and upgrade, and generally use hardware that is one to two years behind the fastest serial computers.

[Krishnamurthy 96] describes several more large computers, namely Thinking Machines CM-5, the Intel Paragon and the Cray T3D and their networks and how they can be used to efficiently implement shared memory.

2.1.1 NOW

The goal of the NOW (***N***etwork ***O***f ***W***orkstations) [Anderson 95b] project of the University of California at Berkeley is to build support for using a network of workstation as a building-wide distributed supercomputer, as off-the-shelf workstations offer much better price / performance than dedicated supercomputers.

NOW defines an interface for distributed applications running on network of workstations connected over a high speed network, namely the Myrinet [Boden 95]. A Myrinet LAN is composed of point-to-point links that connect hosts and switches as shown in figure 2-1. The one way speed between two Sparc20 workstations connected with a Myrinet Switch is about 136 Mbits/sec (the complete performance data is available online from `http://www.myri.com/myrinet/performance`).

The goal of the NOW group is to create a distributed UNIX called Glunix which will, among others, offer the following features:

- fast communication and support for parallel programs [Mainwaring 95]
- Network Ram [Anderson 95]. To access a memory page on a remote system is faster than accessing it on the local disc, provided that there is a fast communication system in place. Therefore it is better to use the memory of idle workstations in the LAN as virtual memory rather than a local or remote disk.

---

[8] pSather uses clusters as a synonym for nodes.

- serverless, distributed file system [Anderson 95], that uses the memory of idle machines as cache.

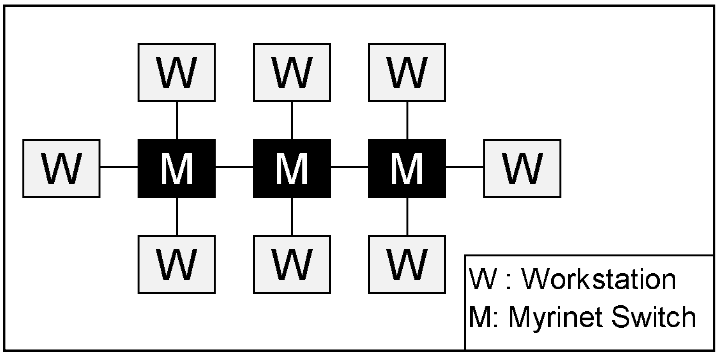

Figure 2-1: A Myrinet LAN with 8 Workstations

The NOW group has already realized several applications on their network, the most prominent one being the parallel web search engine Inktomi accessible at `http://inktomi.berkeley.edu`.

2.1.2 Meiko CS-2

The CS-2 produced by Meiko [Meiko 95] is a distributed computer, where each node is a Sparc processor running the Solaris operating system. Meiko offers different options regarding the nodes, namely

- the Sparc/IO Board, which is a full-fledged UNIX workstation, including up to two processors and I/O interfaces.
- the Multi-Sparc board, with up to 4 processors but no I/O capabilities. Such a node is configured as a disc-less UNIX system.
- VPU board with one Sparc CPU and 2 Fujitsu Micro Vector co-processors.

Each node has a dedicated “Elan” network processor with a dedicated connection to the network, a fat tree. The main processors communicate over memory with the network processor. The network processor has its own TLB and can therefore execute arbitrary used code.

The pSather language has been ported to the Meiko CS-2 and tested on a system with up to 40 Nodes consisting of 2 Sparc Processors each running at 66 MHz[9]. The CS-2 uses a logarithmic network [Meiko 93] constructed from 8 way crosspoint switches and bi-directional links.

---

[9] This system, owned by the Lawrence Livermoore National Laboratory, has been recently upgraded to 90 MHz processors.

2.1.3 WEB Computer

Several massively parallel programs have been run by using the protocols used by the World Wide Web, namely HTTP [Berners-Lee 96] and standard eMail. An example is the factoring of a 130 digit integer published by RSA, described in [Cooperating 96]. A new effort to factorize an even larger number has been proposed at the Supercomputing '95 in San Diego [Bhatt 95]. Several new problems arise with this kind of widespread computer system:

- **Network Reliability**: as the network is not reliable, connections between different parts of the computer may shut down at any time, and therefore only very loosely coupled algorithms are useful in this environment. Ideally each computer gets one message with the data to work on, and after a lengthy calculations, in the order of 10 minutes to days, sends back the answer.
- **Architectures**: many different computers with different architectures and operating systems run the program, which has to be extremely portable and cannot be tuned for a special architecture.
- **Security**: as the computers that run this program belong to different people and corporations, they have to trust that the program is not a threat to their system by either consuming an unreasonable amount of CPU time, being vulnerable to an attack (most of those programs need network connections that could be exploited by hackers) or installing back doors for future attacks.
- **Controlling**: the program is executed by many different people in different places. It is very easy for a mischievous person to return wrong results. Either the results have to be easily verifiable or the program has to be tolerant against such attacks (the latter was the case in the factoring program described above).

2.2 Message Passing Systems

All programs running on distributed computer systems like networks of workstations need a message passing facility. This chapter describes several different standards used on many different systems and for large applications. To use a distributed memory system to simulate shared memory seems to be strange, as it would probably be better to use hardware to create a real shared memory system. [Kranz 93] argues that a modern computer should support both, a message passing interface and shared memory, as, depending on the problem, either one or the other is necessary to get an efficient implementation. [Larus 93] however showed that compiler implemented shared memory has the potential to exploit more effectively the resources in a distributed system.

pSather uses the last one described, the Active Message system developed at the University of California at Berkeley in the context of the NOW project [Anderson 95b].

2.2.1 PVM

PVM (***P***arallel ***V***irtual ***M***achine) has over 8 years of history, as outlined in [Geist 94]:

> *"The PVM project began in the summer of 1989 at Oak Ridge National Laboratory. The prototype system, PVM 1.0, was constructed by Vaidy Sunderam and Al Geist; this version of the system was used internally at the Lab and was not released to the outside. Version 2 of PVM was written at the University of Tennessee and released in March 1991. During the following year, PVM began to be used in many scientific applications. After user feedback and a number of changes (PVM 2.1 - 2.4), a complete rewrite was undertaken, and version 3 was completed in February 1993."*

The current version of PVM is 3.3, and work is under way to create 3.4. PVM does not support active messages and is not thread safe, as it uses global buffers to create a message, and then the same buffer to send it out. However, three PVM versions that support threads are available, namely PT-PVM by [Krone 95], TPVM, see [Ferrari 94] and [Ferrari 95] and LPVM described in [Zhou 95].

PT-PVM lets several thread run inside a PVM process. Local threads communicate by using shared memory. Remote communication however is channeled through a special process on each node for outgoing messages, while incoming messages are received by one thread, the system communication thread and then distributed to the correct receiver.

TPVM lets several threads run inside a PVM process, although the threads may not exchange data among each other over the address space they share (this is not enforced by either the compiler or the runtime system). As the system is built on top of the PVM system it cannot send out more than one message at any point (it uses explicit locks around the non thread safe PVM calls).

LPVM on the other hand is used for communications among threads of one process. An LPVM system cannot have several processes communicating among each other. It is meant as a programming environment for SMP systems that support global memory and threads. To make LPVM thread safe, most calls had to get an additional argument, so programs written for a PVM system are not compatible with LPVM.

One of the major drawbacks of the PVM system is its speed and the inherent non-threadssafe interface. However, many applications have been written for PVM, as it was the first message passing interface that was available on many modern supercomputers, and the next version of PVM should be redesigned to improve thread safety, active message kind of communication will however not be available in the near future.

2.2.2 MPI

MPI stands for ***M***essage ***P***assing ***I***nterface. Like PVM, it is a standard used on many different systems to exchange messages among different nodes. A preliminary report was available in November 1992 and a revised version in February 1993 [Dongara 93]. It had only point-to-point communication, but no collective communication. Also, it was, like PVM, not thread-safe. Later versions of MPI however, were thread-safe, a major advantage over PVM.

Currently work is under way to define the new MPI-2 standard [MPI-FORUM], which should include among others dynamic process management, active messages (one-sided communications), parallel I/O, language bindings and real-time extensions.

MPI is available on most high performance computer and used for the same kind of applications as PVM. For more information about MPI check the [MPI-FAQ].

2.2.3 AM / GAM / UAM

Active Messages have been developed by the NOW team in Berkeley as an efficient way to communicate over high speed networks [von Eicken 92]. An active message is composed of a function pointer and up to four arguments. The receiving node executes the function and passes in the arguments received. Active Message are therefore similar to RPC; however, they are more restricted since an active message handler is supposed to execute quickly, it may not suspend[10] and cannot send another active message out, unless it replies to the node that started the handler in the first place. This last requirement ensures that deadlocks because of buffer overruns are avoided.

The original Active Message interface described in [Culler 94] has been replaced by a new version, which, among other things, supports threads [Mainwaring 95]. pSather still uses the old version, but extended it with thread creation routines as there were no implementations available that supported the new interface.

The main advantage of using Active Messages for the implementation of the runtime of pSather (besides speed) is the fact that most of the messages used do not have a receiving thread but are just used to read or write some memory on the remote node. PVM and MPI (in their current form) would actually need a thread that receives those messages, checks what functions should be executed and then executes those functions.

pSather and Split-C [Culler 93] (another parallel language that runs on distributed computer systems) have the property that most of their message handlers are few in numbers and very small. Therefore, it should be possible to use communications co-processors available in many of the distributed computers to execute the handlers instead of using the main processors. This approach has been studied in [Schauser 96]. It turns out that, as the co-processor is slower and has less computing power than the main processor, it executes even reads and writes of the memory about 30-40% slower than the main processor. However, as the co-processor is normally more responsive than the main processor, some programs do run a little faster, while others slow down. pSather does not use such techniques yet.

[10] Actually, the AM implementation used by pSather allows a message handler to suspend by acquiring a lock. However, this lock must be released before the message handler returns, and no thread may ever send out a message as long as it holds this lock. This possibility had to be added to allow message handlers to execute code that has to run atomically.

2.3 Distributed Locks

Every distributed system needs a way to synchronize access to shared resources and several different solutions have been proposed to solve this problem. Some of the solutions require the process that needs exclusive access to acquire a token. Such algorithms have been implemented mostly in software, a very well known version however is implemented in hardware, namely in the Token Ring Network [Göhring 93]. A free token is constantly passed from one card to the next. If a card needs to send a message, it removes the token and replaces it with a busy token. The message directly follows the token and the receiving card removes the busy token and the message and restarts a free token.

A software protocol has been proposed by [Lamport 78], where a processor that needs the token broadcasts requests and as soon as it received an answer from all processors it implicitly got the token. This protocol was enhanced by [Ricart 81] and [Carvalho 83] by reducing the numbers of messages sent.

A distributed priority lock algorithm has been developed by [Goscinski 90] and a similar one, although faster and with less overhead, was also proposed in [Johnson 94]. However, neither of those algorithms addresses the problem of deadlocks when acquiring more than one lock at the same time, although they do have a first-come-first-serve policy. The problem here is that to avoid deadlocks, each process has to acquire all the locks it needs in a total system order, which may lead to a bad overall usage of resources. Suppose for example that a process needs two locks, l_1 and l_2 and that l_2 is frequently blocked for a long time, while l_1 usually only for short times. This process will now block l_1 until it has a chance to get l_2 and no other process will be able to lock l_1 even for a short amount of time. This results in a less than optimal usage of resources and can even starve some threads completely.

Distributed Reader/Writer locks have also been proposed, mostly to access shared memory pages in a distributed system that simulates shared memory by copying the pages from one processor to another. One, called Available Copies, was proposed in [Bernstein 84] and another one, Weighted Voting in [Gifford 79]. This one works as follows: each copy of the data is assigned a weight; to read the object, the process must collect as many copies as necessary so that the sum of all weights is larger than a predefined value, called the read quorum. The same happens if a process wants to write the data, but this time it has to collect enough copies to satisfy the write quorum. This works only if the sum of the read and the write quorum is larger than the sum of all weights available in the system. Whenever a process writes a new version of the data, it increments a version counter, which is needed by processes that reads the data to decide which of the different copies they get is the most recent one. [van Renesse 88] made this algorithm fault tolerant by adding ghosts in the event of container or network crashes. All those algorithm suffer from the same problems as the ones described above.

The VAX / VMS cluster system [Fox 87] uses a more sophisticated distributed lock system described in [Snaman 87] that allows a lock to be locked in one of several different ways, namely for concurrent reading or writing, protected reading or writing or as an exclusive lock. The locks themselves are organized in a tree like structure that allows tasks to lock complete subtrees in one operation. The system is fault tolerant and has a deadlock detection mechanism. It is used for a large set of different tasks in a VMS cluster, like the Record Management System and distributed file system. Fairness is not addressed in this system, and deadlocks are detected when they arise, but not prevented.

An implementation of a distributed lock system that did address deadlock problems and the problem of using resources as much as possible was done for an earlier implementation of pSather on a CM-5 by [Lim 93]. It did not guarantee fairness though and had no support for the disjunctive lock statement.

2.4 Optimizations for Parallel Languages

An excellent overview of the different optimizations (inlining, common subexpression elimination, loop invariant hoisting, loop unrolling and others) used in today's Fortran and C compilers including several optimizations to parallelize High Performance Fortran [Richardson 96] (vectorization) is available as a technical report [Bacon 93]. Therefore this section will only describe some optimizations for parallel languages. For an overview of most of the currently available parallel languages see [Philippsen 95].

Several synchronization optimizations have been implemented for data parallel languages. [Philippsen 95b] removes synchronizations in FORTRAN's `FORALLs` since many loops do not need all the synchronization required by the language and [Tseng 95] presents algorithms to remove normal barrier synchronizations. A similar problem is described in [Cytron 90], where programs using the `doall` (similar to the `FORALL`) are converted to execute in the SPMD model using `dopar/endpar` and `serbegin/serend`. The author argues that by eliminating the forks used in normal `FORALL` and reusing threads the program executes faster.

Another area of research for data parallel languages is the alignment of arrays on massively parallel computers like the MasPar. Several algorithms for the optimal evaluation of array expressions in Fortran 90 are presented in [Chatterjee 95]. While Fortran D [Hirandani 92] allows the user to specify the correct alignment, [Gupta 92] and [Philippsen 93] among others showed that in fact in many cases the compiler is able to do the correct alignment and distribution of the data without user annotations. [Anderson 93] describes an algorithm to automatically find computation and data decomposition for dense matrix calculations that optimize parallelism and locality. A similar problem, namely analyzing array accesses in nested loops, is solved in [Amarasinghe 93].

In [Hirandani 94] several different optimization techniques for Fortran D are outlined, among others combining messages, message pipelining (similar to prefetching and postwriting described in chapter 6) and different pipelining optimizations.

Scheduling of threads and distributing threads to different processors automatically has been extensively studied. In many cases this done by using libraries, not by changing the compiler or the runtime system. [Stoffel 94] for example tried successfully to use fuzzy logic and a neuronal network to achieve load balancing on a MIMD System. [Diderich 95] describes two algorithms for dynamic load balancing after showing that for many problems it is not possible to make statically decisions regarding load balancing. One algorithm runs locally on each node, while the other makes global decisions. The decision about which one to use depends on the parallel program. Another load balancing problem is studied in [Keckler 92], where the compiler tries to schedule different threads such that they utilize 4 high performance ALUs controlled by one CPU as much as possible by exploiting instruction-level and inter-thread parallelism.

The compiler algorithm described by [Mowry 92] introduces prefetching calls in dense matrix operations (in [Mowry 94] the algorithm is extended to sparse matrix code) to avoid latency due to cache misses, while [Chen 92] describes a more general prefetching algorithm. Both work however on singe processors to reduce the waiting time during cache misses, but not for distributed memory systems that simulate shared memory as does the pSather language.

[Krishnamurthy 94] describes how optimizations can be applied to a program using the sequential consistency memory model by detecting which memory access would result in a violation of the memory model if moved.

A parallelizing compiler that uses only information available at compile time is described in [Hall 95]. Not surprisingly it turns out that the compiler can parallelize some programs, but others are impossible to parallelize either because information only available at runtime is needed, or because the analysis is too complex.

2.5 History of Sather / pSather

The detailed history of the Sather compiler is available in [Stoutamire 96]. The first Sather Compiler (Version 0) was written in Sather and bootstrapped by hand translating it to C over the Summer 1990, and the first available Sather version 0.1 was publicly available in June 1991. The first pSather compiler was implemented by Chu-Cheow Lim on the Sequent Symmetry, workstations and the CM-5 [Lim 93]. This version of pSather included among others the lock statement (but not the disjunctive lock statement), typed and untyped gates (first introduced as monitors in [Feldman 91]) as builtin classes.

Sather 1.0 was a major language change that introduced several new features (among others iterators, bound routines, separation of type and code inclusion, exception and a new library design). It was released in the Summer of 1994.

The current version of the language and compiler (available since the Fall of 1996) is 1.1. This includes for the first time the pSather extension used in this thesis, running on Workstations, Network of Workstations connected with either Myrinet or TCP/IP and the Meiko CS-2 Computer.

A group at the University of Karlsruhe created a compiler for Sather 0.1. The language their compiler supports, Sather-K [Goos 95], diverged from the ICSI specification when Sather 1.0 was released.

Sather/pSather has its own home on the World Wide Web at `http://www.icsi.berkeley.edu/~sather`, and also its own newsgroup named `comp.lang.sather`. The compiler and many related files are available at the ftp directory in `ftp://ftp.icsi.berkeley.edu/pub/sather`. The WWW page features the current language specifications, many published papers that describe various features of Sather and pSather, several tutorials and the frequently asked question list. It is also possible to view the library and the compiler source code by using the HTML interface of the browser.

The current release includes the compiler that runs on most UNIX systems, the pSather runtime for Networks of Workstations running Solaris and connected though Myrinet or TCP/IP, the shared memory pSather for Solaris and pSather for the Meiko CS-2. It includes support for debugging, a source code browser, an interface to TK/TCL and a large library with Sather and pSather classes.

3. Advanced Synchronization Constructs

The first versions of pSather already had a sophisticated synchronization statement, the lock statement. It allowed the locking of several locks at the same time (atomically) and as the lock statement is a sytactic block, the system ensured that the locks were properly released at the end of the lock statement. Gates, which implemented a queue with a blocking dequeue, were also introduced at the same time. It turned out, however, that this lock statement was not enough in some cases, and an even more flexible construct was needed. We introduced the disjunctive lock statement, an extension of the standard lock statement.

This chapter is organized as follows: after introducing the disjunctive lock statement and its syntax, we show one of the most import reasons why the disjunctive lock was introduced and then an example how it is used to create thread-safe container classes. The next part of the chapter shows how the user can create new synchronization objects and the rest of the chapter deals with implementation issues.

3.1 The Disjunctive Lock Statement

We will now define the disjunctive lock statement and its exact semantics. The disjunctive lock statement in pSather replaces the standard synchronization mechanisms found in other languages with a flexible and more secure one. It replaces and combines locks, semaphores, regions, monitors (see [Silberschatz 88] for a description of those constructs), the select statement in Ada [ADA 95] and the `ALT` statement in OCCAM [Barret 92].

It allows the implementation of standard mutual exclusion, reader/writer locks, rendezvous and much more. It also allows the user to implement new synchronization mechanisms by providing an interface to the lock manager[11]. Actually, all the synchronization classes of the standard pSather library are written purely in pSather.

The disjunctive lock statement (figure 3-1) is evaluated as follows [pSather Manual]:

- The optional guard condition for each `when` branch is evaluated. If it evaluates to false, this branch is ignored for the rest of the evaluation of the lock statement.
- The lock expressions of all remaining branches are evaluated in order. They must return an object of type `$LOCK`. An object of type `$LOCK` can be acquired by a thread and released later. The exact meaning of acquire and release depends on

[11] The lock manager is responsible for blocking and releasing threads, see page 45 for a more detailed description of it.

the object itself. For an object of type MUTEX, for example, only one thread can acquire it at any time.

- If all $LOCK objects of a branch x can be acquired by the thread, it will do so and execute the corresponding $\texttt{statements}_x$.
- If no branch satisfies this condition and there is an else branch, the $\texttt{statements}_{else}$ are executed. If the else branch is missing, the thread will block until it can acquire all $LOCK objects of one of the branches not rejected in point 1.

```
lock
   [guard cond₁] when lock₁,₁ [, lock₁,₂ ..., lock₁,ₙ] then
      statements₁
   [[guard cond₂] when lock₂,₁ [, lock₂,₂ ..., lock₂,ₙ] then
      statements₂
   ...
   [[guard condₘ] when lockₘ,₁ [, lockₘ,₂ ..., lockₘ,ₙ] then
      statementsₘ]
   [else
      statementsₑₗₛₑ]
end;
```

Figure 3-1: Syntax of the disjunctive lock statement

The system guarantees that no thread will starve to death, that is, if a thread can eventually run, it will do so [pSather spec]. Similarly, it guarantees that no branch will be indefinitely chosen over another one of the same lock statement. Also, all locks of a branch are acquired atomically, and therefore it is not possible to have a deadlock because several lock statements are executed simultaneously. It is, however, still possible to get a deadlock by nesting lock statements dynamically.

Figure 3-2 shows how a dining philosophers[12] class can be implemented in pSather. Unlike in other languages one can use the straightforward implementation and just lock the left and right fork, as the system guarantees that there will be neither deadlock nor starvation.

Another important point is the fact that all $LOCK objects acquired in a lock statement will be released when the thread leaves the lock statement either by executing the next statement after the lock statement, a return, quit or an exception. This is a very important feature, as it ensures that no thread “forgets“ to release a lock. However, sometimes it is necessary to release a lock before the end of the statement block, which can be done with the unlock statement. Each lock can be unlocked only as many times as it has been locked, and only inside the syntactic block of the lock statement, but not in functions called within the lock statement.

[12] The dining philosophers is a well known problem used to demonstrate deadlocks and starvation. The program simulates several philosophers sitting around a table with a plate of spaghettis on the middle as shown in figure 3-3. They either think or eat. To eat they have to grab the fork on their left and right. See [Silberschatz 88] for a more detailed description of this problem.

```
class DINING_PHILOSOPHER is
   readonly attr left_fork,right_fork:MUTEX;
   create(left,right:MUTEX) is
      r::=new;
      r.left_fork:=left;
      r.right_fork:=fork;
      return r;
   end;

   life is
      loop
         -- thinking
         thinking;
         -- eating: the philosopher needs its left and right fork.
         -- (the system guarantees that no deadlock happens and
         -- that no philosopher will starve)
         lock left_fork, right_fork then eat; end;
      end;
   end;
end;

class MAIN is
   main is
      -- create 4 philosophers
      ph:ARRAY{DINING_PHILOSOPHER}:=#(4);
      -- create first philosopher and the two first forks
      ph[0]:=#(#MUTEX,#MUTEX);
      -- create remaining philosophers except the last one
      loop i::=1.upto!(ph.size-2)
         ph[i]:=#(ph[i-1].right,#MUTEX);
      end;
      -- create last philosopher
      ph[ph.size-1]:=#(ph[size-2].right,ph[0].left);

      -- let philosophers do their work
      parloop p::=ph.elt! do
         p.life;
      end;
   end;
end;
```

Figure 3-2: The Dining Philosopher Class

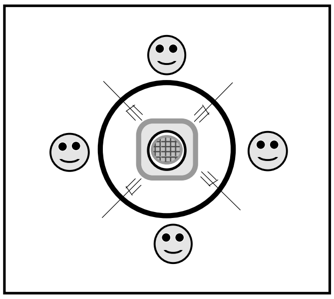

Figure 3-3: Four Dining Philosophers

An implementation of the disjunctive lock statement has to provide the following features:

- efficient locking and unlocking, especially in the cases of simple mutual exclusion locks.
- implementing the fairness and starvation-free guarantees of the language.
- a simple interface to allow the creation of new synchronization objects (this is not a real requirement, but a nice additional feature if it can be provided).

The reminder of this chapter describes how the lock system is used to create reentrant classes, why we removed the possibility to kill threads in pSather and how the missing functionality can be replaced with the disjunctive lock statement. The user interface of the synchronization mechanism and how synchronization classes are defined is described too, followed by a discussion of the fairness requirement of the pSather language specification and some implementation details.

3.2 Reentrant Classes

The Sather library includes many useful classes that we would like to use in pSather too. However, most of those classes are not reentrant (or not thread safe) and cannot be used by several threads at the same time. A class can be thread safe on one of three levels:

- **not thread safe**: such a class can only be used by one thread at a time, and pSather programs have to use some synchronization mechanism to make sure that only one thread uses objects of this class at any time. Nearly all classes that use shared attributes[13] fall under this category. However, shared attributes are used very seldom.
- **class thread safe**: this class can be used by several threads at the same time, provided that every thread access its own object of this class. It is not possible to have several threads calling functions of the same object at the same time. Most of the classes written for Sather are already class thread safe, provided that they do not use shared attributes or objects of classes that are not class thread safe.

---

[13] Shared attributes in Sather are the equivalent of static attributes in C++ and Java and are used as global variables.

- **object thread safe**: several threads can safely call functions in the same object at the same time. All immutable classes that do not use shared attributes are already object thread safe, reference classes have to be modified to include synchronization statements as explained below.

We identified three different types of methods in classes that are not yet object thread safe:

- **mutator**: a mutator is a method that changes the structure of an object (like inserting a new element in a linked list) and puts the object temporarily in an unsafe state during the execution of the method, meaning that certain properties of the object that are normally always guaranteed to hold will be broken. Therefore, only one mutator can be active in an object at any time, and no visitor can run at the same time.
- **visitor**: a visitor is a method that relies during execution that all propreties normally guaranteed by the class interface hold, but it does never break those properties (iterating over the elements of a list is clearly a visitor). Several visitors can run at the same time, but no mutator may run concurrently.
- **status checks**: such functions return a simple status check and do not rely on a specific property on the object (most calls that simply return the value of an attribute are in this category) and are already thread safe. Any function that that reads more than one attribute cannot be a simple status check, unless one of the attributes is a constant that was set during the creation of the object, like the size of an array.

We implement the protocol with a reader/writer lock (see [Fleiner 96] for a more detailed description of the visitor/mutator protocol). While designing a library, one has to identify the different visitors and mutators and add the corresponding lock statements. Visitors lock the reader/writer lock as readers, while mutators lock it as writers and status checks use no lock at all. The example in figure 3-4 shows some of the methods used in a stack implemented with a fixed sized array.

However, such a protocol is only feasible if the implementation of the lock statement is very fast, at least for these special cases where only one special lock is needed (besides the reader/writer lock used in this example we need a fast implementation of the MUTEX lock too). Additionally, as Sather/pSather have not only function methods, but also iterator methods, we need a way to write iterators that observe this model (see page 116 for a description on how iterators work). Originally, we disallowed `yield`'s inside lock statements, as we thought that we had no real use for it. Since iterators have no finalization clauses, we could not get around that limitation. The only solutions were to either not use iterators at all, require special calling conventions (locks acquired outside of the loop, or similar inconvenient and impractical requirements) or to allow `yields` inside locks.

We implemented the last option with the following semantics: all locks acquired inside an iterator will be released when either the statement after the lock is executed or the loop that called this iterator quits. The iterator in figure 3-5 yields all elements of the stack defined in figure 3-4 and observes the visitor/mutator protocol:

```
class ARRAY_STACK{T} is
   include ARRAY{T};
   rw:RW_LOCK;
   size:INT;

   -- status checks
   full:INT is return size=asize; end;

   -- visitors
   top:T
   is
      lock rw.reader then
         return [size-1];
      end;
   end;

   -- mutator
   push(t:T)
   is
      lock rw.writer then
         [size]:=t;
         size:=size+1;
      end;
   end;
end;
```

Figure 3-4: The visitor/mutator protocol

However, this solution has a potential deadlock problem. Consider the function in figure 3-6 which doubles the entries of an array. If this array is unbounded, a `set` call may change the structure of the array when a element is assigned to an index that was never used before and therefore has to be a mutator, while `elt!` is a visitor. As `elt!` is evaluated before `set!` and because a visitor cannot upgrade to a mutator we have instant deadlock. The deadlock detection scheme of the current compiler will point out the problem, but only after it has already happened.

```
   elt!:T is
      lock rw.visitor then
         loop
            yield self[0.upto!(size-1)]14;
         end;
      end;
   end;
```

Figure 3-5: An iterator as visitor

[14] Note that `self` is not strictly necessary in this line, as the [] operator calls by default the `aget` function of `self`, so this line is equivalent to **`yield`** `[0.upto!(size-1)];`

```
-- the following code deadlocks if set! is a mutator and elt! a visitor.
a:UNBOUND_ARRAY{INT};
loop
   a.set!(a.elt!*2);
end;

-- the following loop does not have this problem, as the visitor (aget)
-- releases the lock before the mutator (aset) starts.
-- However, if some other thread changes the size of the array while the loop
-- runs one may get unexpected results.
loop
   a[a.size.times!]:=a[a.size.times!]*2;
end;

-- to get the correct semantics, the mutator has to be locked before the loop
-- this works because a mutator can still call visitor functions.
lock a.mutator then
   loop
      a.set!(a.elt!*2);
   end;
end;
```

Figure 3-6: Potential deadlock problem with reentrant iterators

Unfortunately there is no easy solution to this problem. The compiler could try to detect potential deadlocks for those special calls, but it will not be able to warn against the erroneous loop that uses the `times!` iterator. Note that allowing a thread to upgrade from a visitor to a mutator is not a solution, as two threads that are visitors and try to upgrade to be mutators at the same time will instantly deadlock, as none of them can become a mutator as long as the other one is still a visitor.

Another problem with this protocol (if it is used as described here) is the fact that pre- and post-conditions are no longer useful. The push function in figure 3-4 for example could be written as shown in figure 3-7. However, as the pre- and post-conditions are evaluated outside the lock statement they may use the wrong information, as the stack may not be full, for example, when the pre-condition is evaluated, but may well be by the time the thread acquires the lock.

```
push(t:T)
pre not full
post size=initial(size)+1
is
   lock rw.writer then
      [size]:=t;
      size:=size+1;
   end;
end;
```

Figure 3-7: Pre- and post conditions and the visitor/mutator protocol

There are three possible solutions to this problem:

- Pre- and post-conditions are not used in classes that use the visitor/mutator protocol.
- Instead of pre- and post-conditions, asserts are used as shown in the first example in figure 3-8. This is very inconvenient for post conditions, as the special expression `initial()` does not work in assertions, although it would be easy to add.
- the compiler knows how to implement the visitor/mutator protocol and the functions are annotated by the programmer as shown in the second example in figure 3-8. The compiler will insert the code for the pre- and post-condition inside the lock statement.

```
-- using assertions for pre- and post-conditions
push(t:T)
is
    lock rw.writer then
        assert not full;           -- pre condition
        initial_size::=size;       -- save initial value for post cond.
        [size]:=t;
        size:=size+1;
        assert size=initial_size+1;   -- post condition
    end;
end;

-- using annotations for the visitor/mutator protocol
mutator push(t:T)
pre not full
post size=initial(size)+1
is
    [size]:=t;
    size:=size+1;
end;
```

Figure 3-8: Fixing pre- and post conditions

The last option, namely the annotation of the functions seems to be the cleanest way. This also simplifies compiler optimizations as the protocol can now be implemented with special code that does not use the standard reader/writer lock and does not make use of the lock manager at all. However, without additional support it is also no longer possible to lock the visitor or mutator lock outside of the class.

3.3 Killing Threads

We will now describe the reasons why the possibility to kill threads has been removed from the language and how it can be replaced with the disjunctive lock statement (this was actually the main motivation to introduce the disjunctive lock statement into the language.

The first version of the pSather 1.0 language specification included the possibility to kill a thread. However, we convinced ourselves that it was neither practical nor efficient to support the kill operation in a secure and efficient object-oriented parallel language like pSather. Originally, the thread termination was designed such that an exception was thrown in each thread killed. A thread could either catch the exception and resume or defer the exception until after some critical section (a thread could “shield” itself against exception). We identified several problems:

- **calling conventions:** the user has to know if a function was supposed to be called while the thread was shielded or not, some functions could handle kill exceptions, while other could not. Making all functions “kill”-safe was not an option either, as the code became very clumsy and slower, shielding code does not come for free.
- **code reuse:** a similar problem exists with code reuse: the programmer has to know if a function has to shield itself or not against the kill exception.
- **bound routines:** the caller of a bound routine[15] has to know if the routine called has to be shielded or not. This cannot be a compile time check unless all functions and bound objects are properly annotated.
- **runtime implementation:** the runtime has to shield itself too against the kill exception in many places, and was slower and in some places nearly unreadable.
- **precision:** exceptions raised by the user are precise in the sense that the programmer knows exactly where they happened, whereas a termination exception is imprecise and there is no way to know how far even a simple assignment has been executed. Figure 3-9 shows the problem.
- **optimizations:** some optimizations, especially those regarding the exception stack (see chapter 7.2) and those that reorder code are no longer possible, as an exception could happen any time (see figure 3-10 for an example).

```
protect
    f:=#INPUT_FILE("passwd");
when TERMINATION then
    -- we don't know if the file is open or not, or if f is a valid file variable.
end;
```

Figure 3-9: Imprecise termination exception

A more detailed description of the problems with the termination exception can be found in [Fleiner 96]. We tried several different designs and implementations of the termination of threads, and finally removed this feature from the language specification of pSather 1.0.

However, we had to replace this feature with another one to be able to get a thread out of a locked state without using busy waiting as in figure 3-11.

[15] Bound routine are the Sather equivalent of closures or function pointers.

```
loop_started::=false;
protect
   loop
      ... -- some code
      -- this assignment cannot be hoisted, even if loop_started is never used
      -- inside the loop, as the value of the variable would be wrong if a termination
      -- exception happens before the assignment.
      loop_started:=true;
      ... -- rest of loop body
   end;
when TERMINATION then
   if loop_started then ....
end;
```

Figure 3-10: Imprecise termination exception

```
   -- lock some_lock, but stop trying if end_thread is true
   done:=false;
   loop until!(end_thread);
      lock some_lock then
         -- critical section
         done:=true;
         break!;
      else end;
   end;
   if end_thread and not done then return; end;
```

Figure 3-11: Getting a thread out of a standard lock without termination.

We added the disjunctive lock statement, an extension of the standard lock statement already present in the language. It was inspired by the select statement in Ada [ADA 95] and the `alt` statement in OCCAM [Barret 92], although they are both used to synchronize two threads, while the disjunctive lock can be used for many different tasks. The new statement allows a thread to try to lock one of several synchronization objects[16], and execute some code, depending on the locks it got. Figure 3-12 shows how the code of figure 3-11 can be written in a much simpler and more readable way by using the disjunctive lock statement[17]:

[16] A synchronization object can be used in lock statements. The exact meaning of locking a synchronization object depends on the object. The current pSather library offers MUTEX, Reader/Writer locks, DOOR locks, RENDEZVOUS and more. See page 32 for a more detailed description and how new synchronization objects can be defined.

[17] See page 21 for the syntax of the disjunctive lock statement.

```
lock
    when some_lock then
        -- critical section
    when end_thread then
        return;
end;
```

Figure 3-12: Simple example using the disjunctive lock statement

Note that `end_thread` is not a boolean variable anymore, but a synchronization object, more precisely a DOOR lock[18], which cannot be locked unless the door is opened.

This new statement proved to be very flexible and easy to use. Figure 3-13 shows how a simple consumer can be written. This consumer gets its input from the gate[19] `comm` and terminates as soon as the gate is empty and there are no more producers around. Several of those consumers can run in parallel. The code assumes that all producers are attached[20] to the gate `prod`.

```
consumer(prod:GATE,comm:GATE{T}) is
    t:T;
    loop
        lock
            when comm.not_empty then
                t:=comm.dequeue;
            when comm.empty, prod.no_threads then
                return;
            end;
        end;
        -- consume t
    end;
end;
```

Figure 3-13: Simple consumer using the disjunctive lock.

Neither `comm.empty` nor `prod.no_threads` are booleans. Both are special synchronization objects that can be locked only under the condition that the communication gate is empty or that all threads attached to `prod` have finished. See the next chapter for a description of how this feature is implemented.

[18] A DOOR synchronization object has two routines, named `open` and `close`. As long as the door is closed, no thread can lock this object, but as soon as a thread opens the door, all threads can lock it until some thread closes the DOOR object again.

[19] A gate is used in this example as a queue. It is possible for a thread to wait until a gate is empty or not. A dequeue blocks if a gate is empty until a value has been enqueued.

[20] See appendix A for a description on how threads are attached to a gate.

3.4 User Defined Synchronization Objects

This chapter describes the interface that synchronization objects have to provide in order to subtype from the type `$LOCK` and be usable in lock statements. At the end of this description we will provide two examples, namely the MUTEX implementation and the skeleton of the READER/WRITER implementation. Other examples and descriptions of synchronization objects are available online in the pSather library code.

User synchronization classes are useful to implement all of the widely used synchornization primitives like standard locks, semaphores and barrier locks, but the most important use is to implement locks that can be locked under certain conditions. This can be used for example to implement a stack or a queue that can only be locked if it is not empty. This makes it possible to implement a dequeue or pop such that it always returns a value by blocking until some other thread enqueues or pushes data.

A synchronization object has an internal state that defines which threads may acquire[21] this object. The internal state may only change

- when the lock manager calls some of the functions defined below
- after a thread has acquired this object and before it is released again. Such a state change may be visible to other threads only after the object has been unlocked.

Figure 3-14 shows the interface of the class `$LOCK`, the superclass for all synchronization objects.

```
abstract class $LOCK is
   primary:$LOCK;
   reservable(tid:THREAD_ID):BOOL;
   reserve(tid:THREAD_ID);
   free(tid:THREAD_ID);
   request_reservation(tid:THREAD_ID);
   cancel_reservation(tid:THREAD_ID);
   combinations:ARRAY{ARRAY{$LOCK}};
   wait_for(tid:THREAD_ID):ARRAY{THREAD_ID};
end;
```

Figure 3-14: The abstract class `$LOCK`

Those functions have to respect some special properties:

- None of those functions may block, use the `lock` statement or raise an exception.
- the functions must be 'class thread safe', but do not have to be 'object thread safe', that is, the same function may be called on different objects at the same time, but the system guarantees that only one function is called per object at any time.

---

[21] We use the term acquiring a synchronization object as a synonym to locking an object to avoid confusion, as locking an object has already a predefined meaning, which does not apply to all the synchronization objects defined in the pSather library, like RENDEZVOUS and BARRIER locks.

- The state of a synchronization object may only be changed within a `lock ... end` block or within one of the functions defined by the `$LOCK` interface.
- The functions should not have side effects outside their lock objects.

The THREAD_ID class used in this example is a standard pSather class with the interface shown in figure 3-15. A thread-id is, as far as the programmer is concerned, just an opaque value that cannot be used for anything else. The interface provided allows one to use thread-id's in hash tables and to sort them, and, for debugging purposes, it is also possible to print an id. However, there is no guarantee about any special format of it, and the user should not depend on either the current size of the id or a particular format or order. It does, for example, not guarantee that threads created later get an id that is larger than threads created earlier. The only way to create thread-id's for a thread is to ask for its own id, or the get a nil id which is guaranteed to be different from all other id's.

```
immutable class THREAD_ID
            < $IS_LT{THREAD_ID}, $HASH, $NIL, $STR is

   nil:SAME;         -- returns the nil id, which is different from all other thread id's.
   me:SAME;          -- returns the id of the calling thread.

   is_nil:BOOL; --  returns true if self is the nil id.
   is_eq(e:THREAD_ID):BOOL; -- true if e and self are the same id.
   is_lt(e:THREAD_ID):BOOL; -- true if self  is smaller than  e.

   hash:INT;         -- returns a hash value useful for hash tables
   str:STR;          -- returns a string, useful for debugging
end;
```

Figure 3-15: The THREAD_ID class interface

We will now describe the individual functions of the $LOCK class and how they can be used to create the different synchronization classes available in the pSather library.

3.4.1 Reservable, Reserve and Free

These three functions are the most important ones and must be defined by all synchronization objects. All three functions have one parameter, namely the ID of the thread that tries to acquire this lock. `THREAD_ID`'s can be compared, the class also defines a hash value and a `str:STR` function useful for debugging.

`Reservable` returns true if the object can be acquired or locked by the thread passed as ID, while `reserve` actually acquires the object. `Free` will release the lock again.

Those three functions are already enough to define the MUTEX class as shown in figure 3-16.

```
-- all synchronization objects that behave like a MUTEX (only one thread can acquire
-- such an object, and it can do so multiple times) are supposed to subtype from
-- $MUTEX
class $MUTEX < $LOCK is end;

class MUTEX < $MUTEX is
   -- ID of the thread that currently locks this MUTEX lock
   attr locked_by:THREAD_ID;

   -- number of times that the thread stored in locked_by has locked this MUTEX
   attr locked:INT;

   -- returns true if either this MUTEX is not locked yet or already locked by the
   -- same thread that tries to lock it again
   reservable(tid:THREAD_ID):BOOL is
      return locked=0 or locked_by=tid;
   end;

   -- locks this MUTEX for the thread tid
   reserve(tid:THREAD_ID)
   pre locked=0 or locked_by=tid
   is
      locked:=locked+1;
      locked_by:=tid;
   end;

   -- frees the lock, only the thread that locked it can unlock it.
   free(tid:THREAD_ID)
   pre locked>0 and locked_by=tid
   is
      locked:=locked-1;
      -- not really necessary, but makes the code cleaner
      if locked=0 then
         locked_by:=THREAD_ID::nil;
      end;
   end;
end;
```

Figure 3-16: Class MUTEX

3.4.2 Primary

With the exception of the simple locks like MUTEX, it is often necessary to have different lock objects that work on the same lock, like the reader and the writer of a reader/writer lock which form a lock family. The system has to know which lock objects work together in this way. `Primary` is used by the system to get the "main" lock object of a family of lock objects. For all family members the method `primary` has to return the same object. Figure 3-18 shows the implementations of the reader/writer lock.

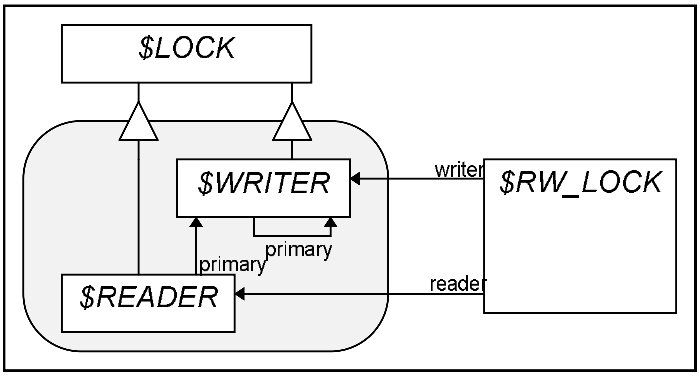

Figure 3-17: The reader/writer lock family

```
--  the $RW_LOCK defines the interface for read/writer types of locks. Such a lock
--  is not itself under $LOCK.
class $RW_LOCK is
   reader:$READER;
   writer:$WRITER;
end;

--  the class  $READER has no special interface, but all locks that behave
--  like readers are supposed to subtype from this class instead of $LOCK
class $READER < $LOCK is end;

--  the same is true for the class $WRITER, although this class subtypes from
--  $MUTEX, as it behaves like a mutex lock.
class $WRITER < $MUTEX is end;

--  The fair reader/writer lock. The two attributes are just used to store the
--  reader and the writer part respectively.
class FRW_LOCK < $RW_LOCK is
   readonly attr reader:$READER;
   readonly attr writer:$WRITER;

   create:SAME is
      r::=new;
      r.writer:=#FRW_WRITER;
      r.reader:=#FRW_READER(r.writer);
      return r;
   end;
end;
```

Figure 3-18: The Reader/Writer class

```
class FRW_WRITER < $WRITER is
   private attr writer_id:THREAD_ID;
   private attr write_locks,read_locks:INT;

   create:SAME is return new; end;
   primary:$LOCK is return self; end;

   -- the next three functions are used when working on the writer part of the lock
   -- They work exactly the same way as in the MUTEX class, with the only
   -- exception that reservable has to make the additional check that there is
   -- no reader that has acquired this lock
   reservable(tid:THREAD_ID):BOOL is
      return (read_locks=0 and write_locks=0)
             or writer_id=tid;
   end;

   reserve(tid:THREAD_ID) is
      write_locks:=write_locks+1;
      writer_id:=tid;
   end;

   free(tid:THREAD_ID) is
      write_locks:=write_locks-1;
      if write_locks=0 then
         writer_id:=THREAD_ID::nil;
      end;
   end;

   -- the next three functions are used when working on the reader part of the lock.
   -- A reader can acquire the lock unless there is already another writer
   -- (note that the same thread can acquire first the writer and then the reader,
   -- but not the other way around).
   r_reservable(tid:THREAD_ID):BOOL is
      return write_locks=0 or writer_id=tid;
   end;

   r_reserve(tid:THREAD_ID) is
      read_locks:=read_locks+1;
   end;

   r_free(tid:THREAD_ID) is
      read_locks:=read_locks-1;
   end;
```

Figure 3-19: The fair writer class

```
class FRW_READER < $READER is
    -- the reader delegates all calls to the writer. This way the code is concentrated
    -- in one class to make maintenance easier.
    private attr w:$WRITER;
    primary:$LOCK is return w; end;

    create(wr:$WRITER):SAME is
        r::=new;
        r.w:=wr;
        return r;
    end;

    reservable(tid:THREAD_ID):BOOL is
        return w.r_reservable(tid); end;
    reserve(tid:THREAD_ID) is w.r_reserve(tid); end;
    free(tid:THREAD_ID) is w.r_free(tid); end;
end;
```

Figure 3-20: The Reader class

3.4.3 Request_reservation, Cancel_reservation

Each time a thread waits for a lock, the system calls the function `request_reservation`, and, as soon as the thread continues, it will call `cancel_reservation` for all locks, regardless of whether the thread acquired some, all or none of the locks. Those functions are used as shown in figure 3-21 to implement reader/writer locks with a priority for readers or writers, that is, as soon as a thread requests the reader lock, no thread will be able to get the writer lock.

3.4.4 Combinations

This function defines which locks of a lock family have to be locked together, a feature used for rendezvous locks. Rendezvous locks are used whenever two threads have to synchronize at some point and need to exchange some data at that point (see page 123 for more information and an example). Figure 3-22 shows how the rendezvous class defined in the pSather library uses this function to define that the rendezvous main lock `self` can either be locked by itself, or the locks `r1` and `r2` have to be locked simultaneously by one or two threads.

```
class WR_WRITER < $WRITER is
   include FRW_WRITER r_reservable->,
                      request_reservation->;

   private attr writers_waiting:FSET{THREAD_ID};

   request_reservation(tid:THREAD_ID) is
      writers_waiting:=writers_waiting.insert(tid);
   end;

   cancel_reservation(tid:THREAD_ID) is
      writers_waiting:=writers_waiting.delete(tid);
   end;

   -- the reservable function does not change for writers, but readers can now only
   -- reserve the lock if no writer is waiting. There is also the special case where the
   -- same thread waits for a reader and a writer lock: in this case the reader
   -- can actually reserve the lock. This happens for code like
   --         lock rw.reader,rw.writer then ... end;
   -- and
   --         lock when rw.reader then ...
   --              when rw.writer then ...
   --         end;
   r_reservable(tid:THREAD_ID):BOOL is
      return
          (write_locks=0
           and (writers_waiting.size=0
                or (writers_waiting.size=1
                    and writers_waiting.first_elt=tid)))
          or writer_id=tid;
   end;
end;
```

Figure 3-21: The unfair writer class

```
   combinations:ARRAY{ARRAY{$LOCK}} is
      return ||self|,|r1,r2||;
   end;
```

Figure 3-22: Combination lock

3.4.5 Wait_for

This function is used for deadlock detection and should return the list of threads that have to release this lock before the thread passed as argument can eventually acquire it. The list of threads is returned as an array of `THREAD_ID`'s. Figure 3-23 shows the definition of this functions the way it could be used in the `MUTEX` class defined in figure 3-16.

```
wait_for(tid:THREAD_ID):ARRAY{THREAD_ID} is
   if locked>0 and tid/=locked_by then
      return |locked_by|;
   end;
   return void;
end;
```

Figure 3-23: Wait_for as defined in the class MUTEX

3.4.6 Summary

The table 3-1 lists all functions and shows when they are called by the lock manager and whether they may change the state of the lock object or not.

function	**description**[22]	**may change internal state**[23]	**call pattern**
`reservable`	returns true if the thread may acquire this lock		called whenever the object may have changed its state
`reserve`	acquires this lock for the given thread	✓	once to acquire a lock
`free`	releases a lock	✓	once to release a lock
`request_reservation`	used to prioritize certain locks inside a lock family	✓	once for each lock object when a thread enters a lock statement
`cancel_reservation`	used together with request_reservation	✓	once for each lock object when a thread got the locks or executes the else part.
`combinations`	returns which locks of the lock family have to be locked together		once
`wait_for`	returns an array of threads that have to release the lock before the thread can get it		occasionally, but only if deadlock detection is enabled

Table 3-1: The $LOCK interface

[22] All functions, with the exception of `primary` have a thread id as argument. This is the thread mentioned in the column "description", which is not necessarily the same as the thread that calls those functions.

[23] The internal state of a lock object may also be changed by other functions, as long as this happens only inside a `lock ... end` block where this lock object has been locked.

3.5 Fairness

pSather guarantees fairness and starvation freeness for threads that acquire locks. The specification of the language defines this as follows [Stoutamire 96]:

> *"The pSather implementation of `lock` statements ensures that threads that can run will eventually do so; no thread will face starvation because of the operation of the locking and scheduling implementation. Similarly, no `when` clause will be repeatedly chosen over another such that a clause starves."*

We will now describe a model that allows us to implement the fairness requirements by a centralized lock manager facility, and a simple extension to create a decentralized facility.

In our simple model, we will represent `when` branches by discs (arrays) that are speared on a giant stick (priority queue) as shown in figure 3-25, which shows the stick when all threads in figure 3-24 enter their respective lock statement. Discs to the right side of the stick have a higher priority to get their locks. Whenever some locks are free, the disc with the highest priority that can acquire all its locks will get them, unless there is another disc with a higher priority that waits for one or more of the same locks. In figure 3-25 for example, thread 1 can get lock `L1` but not lock `L3` as long as the disc owned by thread 3 is at the top of the priority queue.

```
-- thread 1
   lock
   when L1 then
      ...
   when L2 then
      ...
   when L3 then
      ...
   end;
```

```
-- thread 2
lock L1, L2, L3 then
   ...
end;

-- thread 3
lock
when L1, L2 then
   ...
when L1, L3 then
   ...
end;
```

Figure 3-24: Threads competing for the same locks

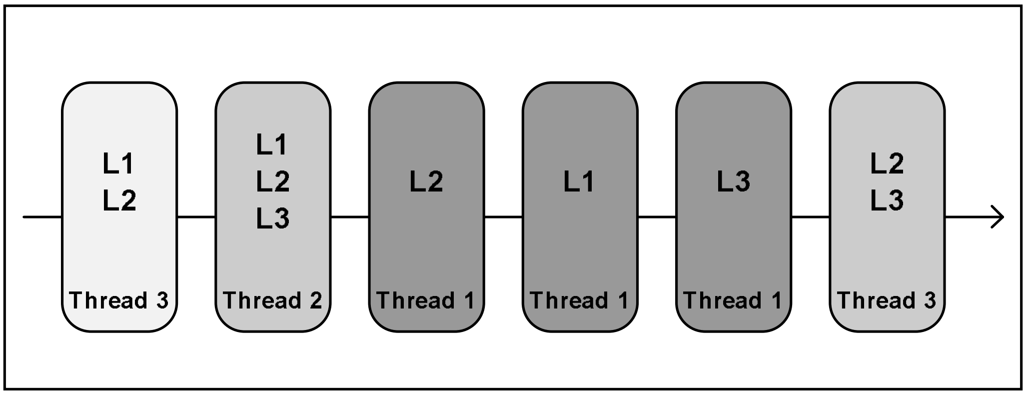

Figure 3-25: The lock priority queue

However, each disc has an internal timer, and as soon as it goes off, the disc will be moved to the end of the stick. The timer is then reset to a time proportional to the squared age of the disc. This simple trick ensures that no thread will starve to death, provided that there is a constant time period L such that all the locks the thread waits for are never locked longer than L.

In the following claims, D denotes a disc, D_{senior} the oldest disc on the stick and D_{junior} the second oldest disc on the stick. A_D , $A_{Dsenior}$ and $A_{Djunior}$ the respective ages of the discs, L the maximum time a lock is locked during the execution of the program and w the factor used to calculate the time until the timer will go off.

Claim 1: if a disc D, which waits for n locks stays at the front of the priority list longer than the time interval L, it will get the locks it needs.

> **Proof**: All locks that this disc waits for will be free after at most L seconds. No other disc can acquire those locks, as the disc D is the first one and has the highest priority.

Claim 2: D_{senior} will stay for more than L seconds at the front of the priority list as soon as $w \cdot (A^2_{Dsenior} - A^2_{Djunior}) > L$

> **Proof**: All discs in front of D_{senior} have the same age as D_{junior} or are younger. Therefore, they will all either get their locks or jump behind D_{senior} after at most $w \cdot A^2_{Djunior}$ seconds. At this point, D_{senior} will still stay for at least $w \cdot A^2_{Dsenior} - w \cdot A^2_{Djunior}$ seconds at the head of the list. As this number is larger than L, D_{senior} will stay for more than L seconds at the front of the priority list.

Those two claims prove that our model is actually fair, and that all threads will get their locks sooner or later. Obviously, the proof will not work if some thread acquires a lock and does not release it for an arbitrary amount of time. Note that instead of squaring the age of the discs one can use any function *f()* such that $f(A_{Dsenior}) - f(A_{Djunior})$ gets larger as time passes by.

This centralized model can be easily extended to a more decentralized version by replacing the one giant stick with many smaller sticks, one per lock. Each disc is now speared over all the locks it needs. The priority queue in figure 3-25 will be split in three different queues under this model as shown in graphic 3-26. This model is used in the decentralized implementation, were discs that use disjoint sets of sticks can be managed separately.

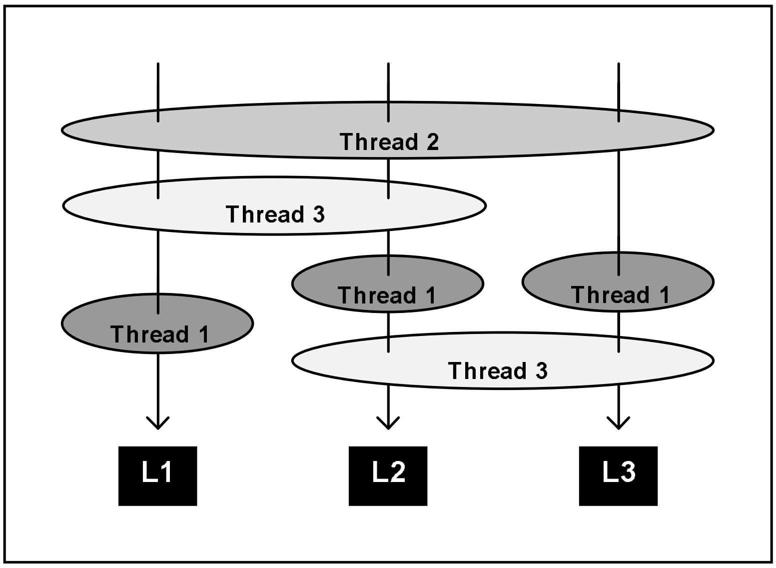

Figure 3-26: The distributed lock priority queue

Note that the fairness definition above does not define when the `else` part of a lock statement should be executed. The problem is that the else statement should be executed when the thread cannot get all the locks of one of the when branches, however, the term “cannot get the locks” must be defined more exactly:

block-else:

In this case the thread waits until all branches wait for at least one lock that is actually locked. If a branch cannot get a lock because another thread has a higher priority it will wait until it can either lock all the corresponding locks or one of the locks is locked by another thread. Only in the latter case it will execute the `else` branch of the `lock` statement.

This version is callled the block_else, as the thread may wait for an arbitrary amount of time before executing the else branch.

weak-else:

Here the thread will just check if one of the `when` branches can get all the necessary locks and if this is not the case it will immediatly execute the `else` branch. Even if the locks of all `when` branches are lockable by this thread, but some other threads are waiting for the same locks with a higher priority, it will execute the else branch.

The current pSather implementation uses the block-else version.

3.6 The Exception Stack

Both, the Sather and pSather runtimes create an exception stack used to jump back on the stack frame whenever an exception is raised. The pSather stack however also creates an entry for every lock statement. This entry is used to unlock all locks whenever an exception is raised. The same entries are also used when lock statements are left, for example because of a return or quit statement.

The Sather implementation uses a global variable to store the top of the exception stack as shown in figure 3-27. As only `protect` statements generate elements stored on the exception stack we get a very efficient implementation that does not slow down most of the program, as only protect statement will add a small overhead to the code.

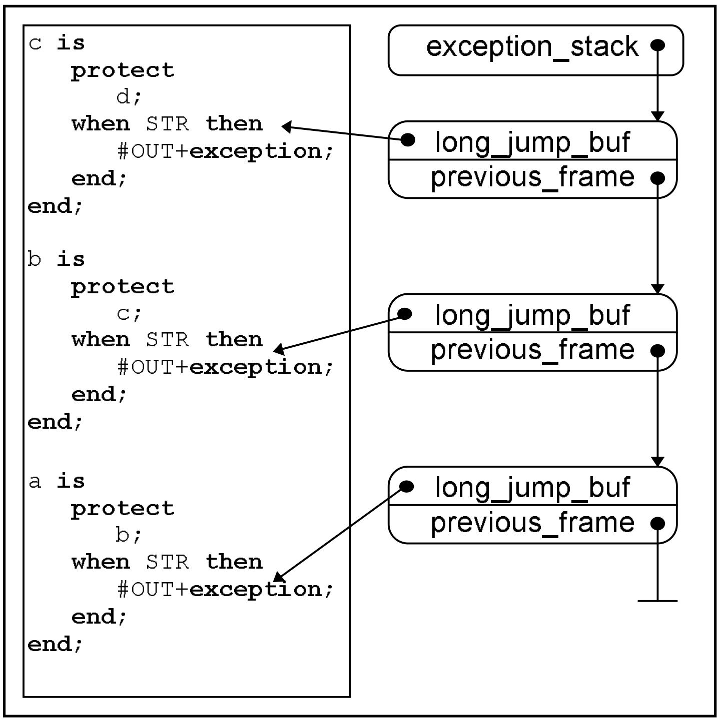

Figure 3-27: Sather Exception Stack

pSather however is very different for several reasons:

- It is not possible to use a global variable, as every thread needs its own exception stack. We could either reserve one register to hold the top of the stack, or we need some thread specific[24] memory. The first solution blocks one register for the entire program and slows it down even if no exceptions are used at all, as fewer registers are available for optimizations, while the second one costs on most architectures at least a function call, sometimes even a system call (this depends on the thread library used). On our a Solaris Sparc system it takes about 0.7µsec. to access the thread local memory.
- Threads can migrate from one processor to another, and therefore it is possible to have a stack that has elements stored on different processors.
- Locks have entries on the exception stack too, as all locks have to be unlocked in the case of an exception.

---

[24] Also called thread local memory. Usually implemented as a function that returns a pointer to some part of the memory, and depending on which thread calls this function, another pointer is returned.

- Many loops need an entry too, at least if they use iterators that yield inside a lock, as those locks have to be released in the case of an exception. Loops add also another dimension to the stack, as every iterator with a yield inside a lock needs a new entry in this loop element. We have no longer a stack, but a tree where the root replaces the bottom of the exception stack.

The graphic in figure 3-28 shows the exception stack when executing the function `X()` in figure 3-29 the first time.

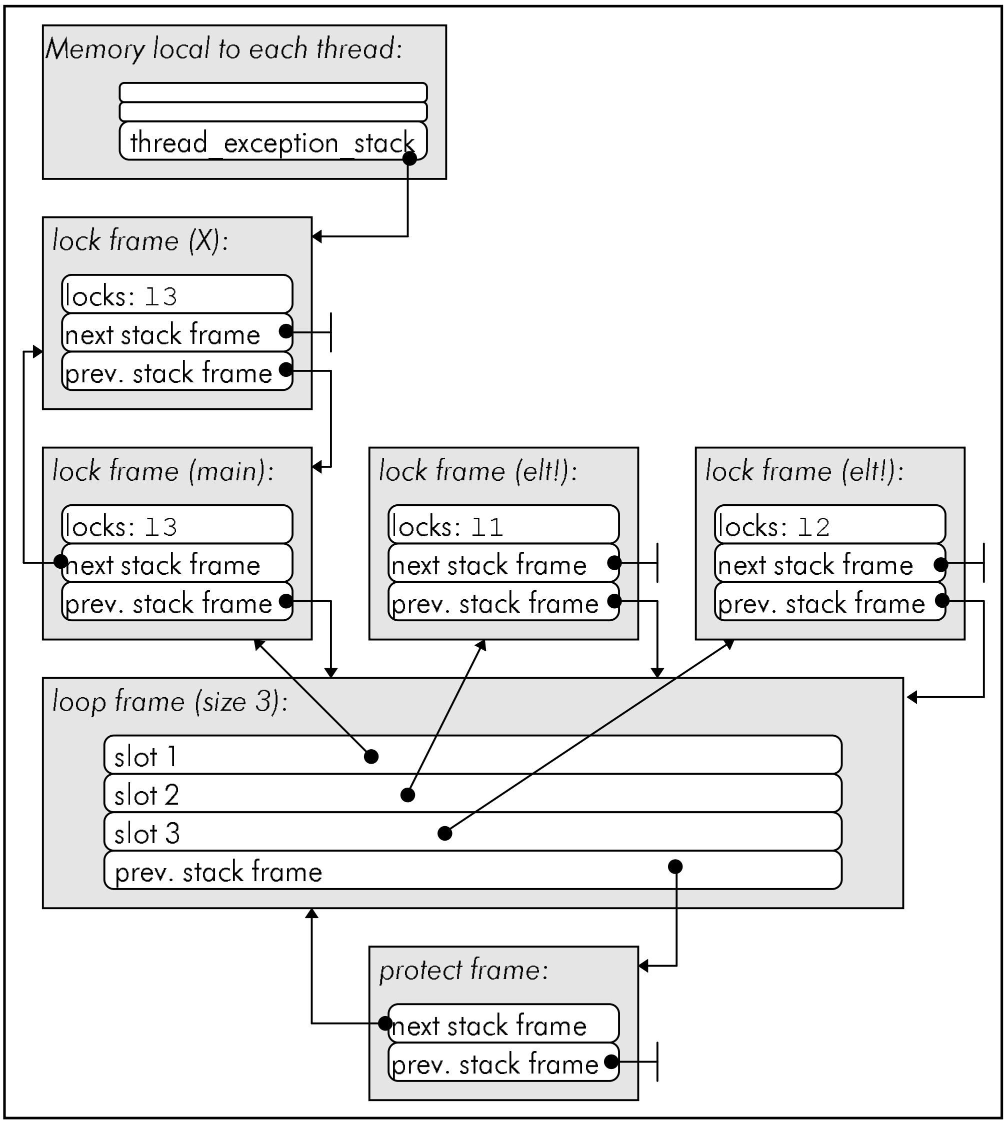

Figure 3-28: pSather Exception Stack (with respect to figure 3-29)

The naïve implementation of the exception stack in pSather leads to slow and inefficient code. Chapter 7.2 describes some of the possible optimizations and their impact.

```
elt!(l:$LOCK):INT is
   lock l then
      loop yield 5.times!; end;
   end;
end;

x(l:$LOCK):INT is
   lock l then return 5; end;
end;

main is
   l1::=#MUTEX;l2::=#MUTEX;
   protect
      loop
         lock l1 then
            b::=elt!(l1)+elt!(l2)+x(l1);
         end;
      end;
   when STR then
      #OUT+exception+"\n";
   end;
end;
```

Figure 3-29: Using locks inside iters.

3.7 Implementations

3.7.1 Centralized Lock Manager

Our model leads to a fairly straightforward implementation, where the priority list is managed by a single thread on one cluster. Each thread that executes a `lock` statement sends all the necessary information (the locks used in the different `when` branches and whether there is an `else` branch or not) to the lock manager thread and waits for the response as shown in figure 3-30. The lock manager thread enqueues those requests, calls the necessary functions of the lock objects to decide if some or all locks can be acquired by some of the discs, and sends the correct response back as shown in figure 3-31.

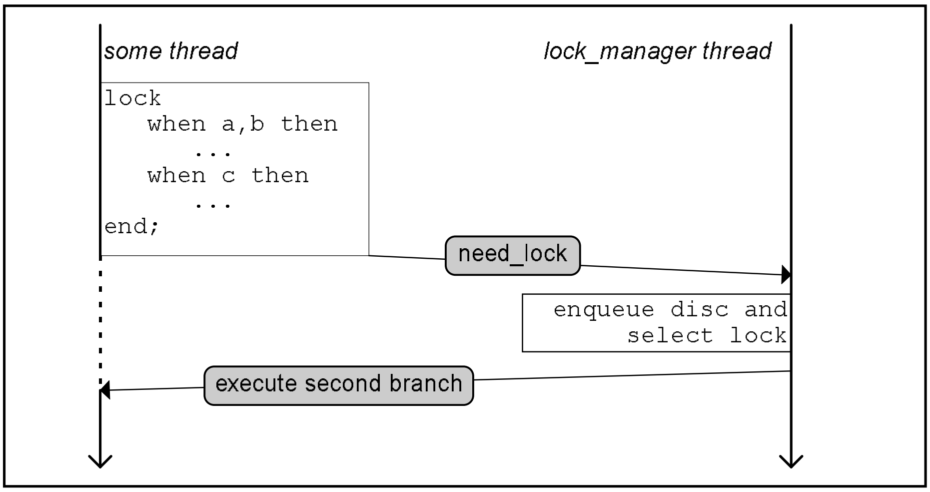

Figure 3-30: The centralized lock manager

The advantage of this implementation is its simplicity and the fact that all the information about locks is stored in a centralized place. This helps for debugging facilities, and deadlock detection is also much easier. However, this model does obviously not scale very well and is not very useful for a large number of clusters.

An additional problem is the fact that each thread that has to acquire a lock has to send a message to the lock manager thread and wait for the response. It would be nice if these messages could be avoided at least for simple cases (locking one `MUTEX` or reader/writer lock). Chapter 7 will show an extension of the lock interface described on page 32 that allows a thread in some cases to decide by itself if it can acquire a lock or not.

The current version of pSather uses a centralized lock manager plus the optimizations described later.

3.7.2 Passive Lock Manager

Instead of assigning the job of managing the locks to one thread, all threads that have to acquire some locks can use a protocol to decide which thread gets the locks and who has to wait. The advantage over the centralized lock manager is the fact that threads which compete for disjoint sets of locks do not interact with each other. This ensures higher throughput as we get real parallelism inside the lock manager. However, the protocol used to synchronize the threads works by using several locks and semaphores, and the time won with parallelism may be lost again because of the additional synchronization requirements.

Figure 3-32 shows the pseudocode used to select the thread that gets its lock. The code uses a list for each lock. Every thread that waits for a lock enqueues its ID in the list owned by this lock. Only the thread that is at the top of this queue can actually acquire this lock. The code in figure 3-32 assumes that each thread waits for exactly one “when” branch (or disc). To make the code work for additional branches the thread just needs to loop over all discs each time. To ensure that the thread does not try over and over again to get a blocked lock, each thread will stop if it cannot get the locks it needs, and wait for either a timer message (of the timer it sets itself), a move message generated by a thread that moved itself from the top of the message queue to the end or by an unlock message sent by a thread that unlocks a lock. Figure 3-33 shows the unlock function.

```
lock_manager is
    loop
        wait_for timer_event, lock_request or unlock_message;
        create_discs_for_new_lock_request;
        work_on_unlock_ messages;

        -- unmark all locks.
        loop all_locks.elt!.marked:=false; end;

        loop
            d::=discs.next!;
            got_locks::=true;
            loop
                l::=d.locks!;
                -- make sure that there is no other plate in front that
                -- has higher priority
                got_locks:=got_locks and l.reservable(d.tid)
                                and ~l.marked
                -- make sure no other plate with lower priority gets this lock
                l.marked:=true;
            end;
            if got_locks then
                loop d.locks!.reserve(d.tid); end;
                send_lock_message
                discs.delete_discs_of(d.tid);
            elsif d.has_else_part and is_last_disc_for(d.tid) then
                send_execute_else_part_message
            elsif disc_has_to_move then
                discs.move_disc_back;
            else
                -- calculate when this plate has to move
                timer:=timer.min(d.age*d.age*w);
            end;
        end;
        set_timer(timer);
    end;
end;
```

Figure 3-31: Pseudo code for centralized lock manager

```
-- whenever a thread needs some locks it enters this function. This code assumes that
-- the lock has only one when branch. Returns true if the locks have been acquired
-- and false if the else part should be executed
select_locks(d:LOCKS,has_else:BOOL):BOOL is
    -- enqueue the lock id on each lock queue
    loop d.locks!.enqueue(thread_id_object); end;
    me::=THREAD_ID::me;
    loop
        -- get an internal lock on all locks needed (ensures mutual exclusion
        -- while testing the locks). This has to be done in a total order to
        -- avoid deadlocks.
        loop d.locks!.lock_internal_lock; end;
        got_locks::=true;
        loop l::=d.locks!;
            got_locks:=got_locks and l.reservable(me) and
                                l.highest_priority_thread=me;
        end;
        if got_locks then
            loop l::=d.locks!;
                l.reserve(d.tid);
                l.remove(me);
                l.unlock_internal_lock;
            end;
            return true;
        elsif has_else then
            loop d.locks!.unlock_internal_lock; end;
            return false;
        elsif disc_has_to_move then
            loop l::=d.locks!
                l.move_to_back(me);
                l.send_move_message_to_thread_in_front;
            end;
        else
            -- calculate when this plate has to move
            timer:=timer.min(d.age*d.age*w);
        end;
        loop d.locks!.unlock_internal_lock; end;
        wait_for_timer, move or unlock message
    end;
end;
```

Figure 3-32: The select_lock function for the passive lock manager

```
-- function called when unlocking a lock
unlock_one_lock(l:LOCK) is
   l.lock_internal_lock;
   l.free(THREAD_ID::me);
   l.send_unlock_message_to_thread_in_front;
   l.unlock_internal_lock;
end;
```

Figure 3-33: The unlock function for the passive lock manager

The first implementation of pSather 1.0 used a passive lock manager and it worked quite well, mostly because this version of pSather was not yet a distributed version, but ran on a single cluster only. A multiple cluster version of such a lock manager would have to either fork a new thread for each `lock` statement on one of the clusters which would then proceed to acquire the necessary locks, or otherwise lock and release many remote locks to get the necessary synchronization. This was the reason why later versions of the runtime used a centralized lock manager.

3.7.3 Distributed Lock Manager

Instead of using one centralized lock manager, or one thread per lock statement as in the passive implementation we can also use one lock manager per cluster. This lock manager is responsible for managing all locks used exclusively on this cluster and a protocol among all those lock managers ensures that threads that lock remote objects will still get a fair chance to acquire those locks.

This implementation is actually quite effective for all lock statements that use only one lock, especially if this lock is stored on the same cluster. However, if a thread tries to lock several locks that are stored on different clusters, it is no longer as easy to guarantee the fairness specified by the pSather manual as it is in the centralized version. There are two ways to deal with such a situation:

- All locks requested together with some other locks are managed by one thread on one cluster, which effectively implements the centralized lock manager for all those threads.
- The different lock managers use a protocol similar to the passive lock manager protocol to select which thread will get the locks needed. Such a protocol is only feasible if the network connections are extremely fast, as the number of messages sent between the different lock managers is quite high. The lock manager will send at least three messages to all clusters that manage at least some of the locks it needs and will have to wait for their acknowledgment:
  1. get the internal lock[25]
  2. check if the locks are reservable
  3. reserve the locks and release the internal lock

[25] The internal lock is a special lock inside each lock object. It is used to serialize the access of the different lock managers to the lock object.

Chapter 7 will show an optimization that allows threads to lock simple lock statements by themselves without using the lock manager or sending messages to other clusters. As this optimization works for the centralized version of the lock manager too, the decentralized version, which has an efficient implementation for exactly those special cases, would probably not be much faster than the centralized version, unless the number of clusters is very high. Therefore we did not implement a decentralized lock manager yet, and kept the centralized for the time being.

4. Memory Consistency Models

Each shared memory, parallel language has to define a precise semantics for memory reads and writes on different processors, the memory consistency model. The consistency model is crucial for the programmer, as it is needed to reason about the correctness of programs. The compiler and the runtime need information about the memory model too, as many optimizations depend on a particular memory model and cannot be used with others. Normally, one expects that a read of a memory location returns the value of the last write to this location. For serial languages the last write is clearly identifiable, and even for multiprocessor, shared memory systems it is possible to use the sequential memory consistency model described below. However, the Sun Multiprocessor Systems do not use it as it would compromise the efficiency [Weaver 93], pp. 117-129. Also, although this model is very convenient to use, it prevents many important optimizations widely used for serial languages as they violate the requirements of this memory model.

Many different memory consistency models have been defined and used in different languages. This chapter describes some of the most common consistency models. pSather could use any of them, the only difference as far as the user is concerned are convenience and speed. The weaker memory models force the user to cleary synchronize the access to shared variables. Code that works correctly "by accident" when using the sequential consistency model will probably break under the weak memory model. A more detailed introduction to the memory models described here and several others can be found in [Adve 95].

4.1 Sequential Consistency

[Lamport 79] defines sequential consistency as follows:

> *"A system is sequentially consistent if the result of execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."*

or, in other words: all threads observe all writes in the same order. Figure 4-1 shows the strict requirements imposed by this memory model. This is the most convenient memory model for the programmer, but all optimizations that rely on the possibility of reordering expressions and instructions cannot be used.

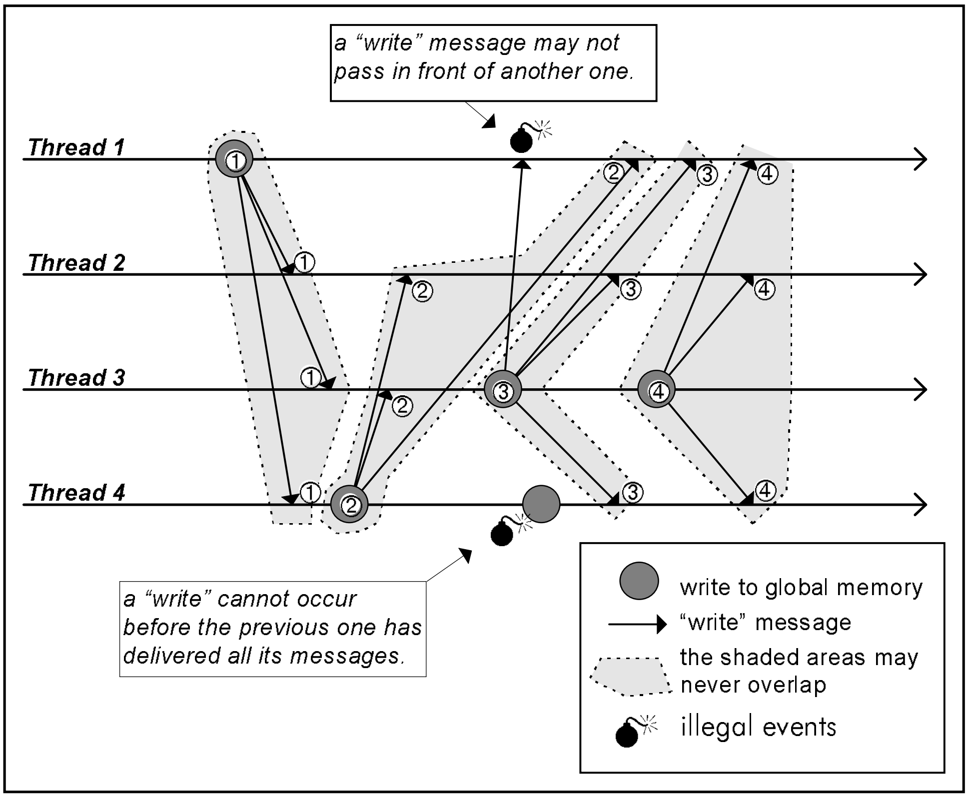

Figure 4-1: Sequential consistency

```
-- Code written by the programmer
obj.a:=b+c;
obj.d:=e;
obj.a:=obj.a+f;

-- optimized version of above code
obj.d:=e;
obj.a:=b+c+f;
```

Figure 4-2: Simple optimization

Suppose that a thread executes the optimized version of the code in Figure 4-2. If, for some reason, another thread reads `obj` while its attribute `d` has been updated but before `a` has been updated, it will see an inconsistent state of the object. Note that such an optimization is perfectly legal for single threaded, serial code, as the program has no way to find out about this change, unless it uses a memory tracing facility, interrupts or some other hardware facilities.

In some cases it is possible for the compiler to prove that it may safely reorder code. This happens if it can prove that the memory positions involved are only known to the current thread, for example because they have just been allocated and the pointer has only been stored in local variables. However, this requires complex inter procedural analysis and cannot be done in many cases.

4.2 Processor Consistency

[Goodman 89] introduced processor consistency.

> Writes of any processors are to be observed in the same order by any other processor, while a processor may observe writes issued by two different processors in different orders.

This model was proposed to relax the restrictions of the sequential consistent memory model and [Gharachorloo 91] shows that it performs significantly better than sequential consistency on shared memory multiprocessors.

Processor consistency does not work much better for distributed systems as each processor executes several threads that use the sequential memory consistency model among themselves. The same arguments that work against optimizations that reorder memory reads and writes in the sequential consistency model apply here too. Figure 4-3 shows the implications of this model.

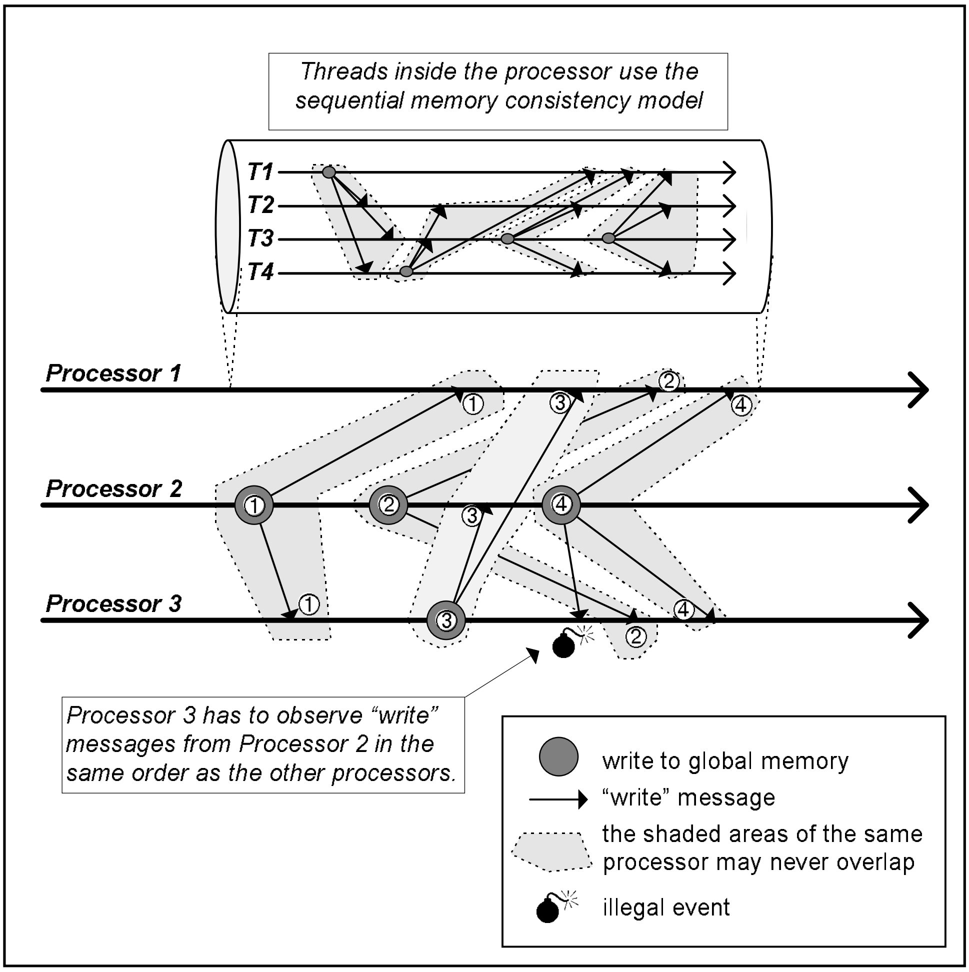

Figure 4-3: Processor consistency

4.3 Thread Consistency

We define thread consistency by substituting “processor” with “thread” in the above definition of processor consistency:

> Writes of any thread are to be observed in the same order by any other thread, while a thread may observe writes issued by two different threads in different orders.

Processor consistency would not be very meaningful for a language that allows thread migration like pSather does, and, as shown above, has the same problems as serial consistency. The thread consistency memory model works quite well and is easy to implement. One drawback however is that all messages sent to other processors regarding new values for objects have to arrive in the same order as they are sent, and they have to be sent in the same order as those values are stored in local variables and the local memory. This last condition ensures that local threads observe writes in the same orders as do threads on other processors. Figure 4-5 shows how this model works.

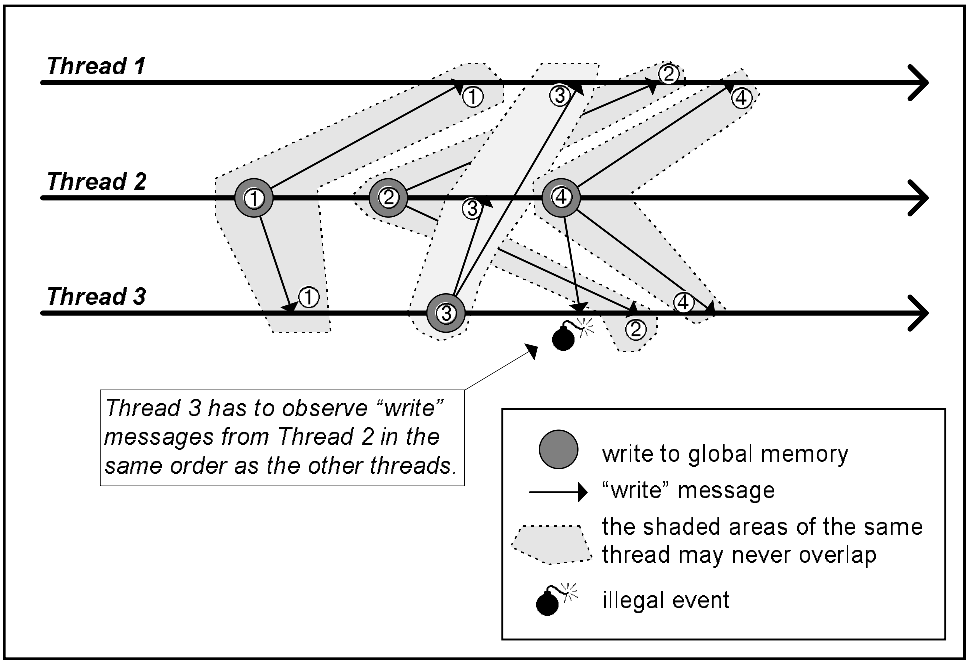

Figure 4-5: Thread consistency

Unfortunately, many of the high speed networks used by pSather implementation do not guarantee the order of messages sent, partly because of hardware limitations, partly because of the particular software implementation. The following active message libraries used for the pSather do not guarantee ordering (note that the generic active message definition [Culler 94] does not guarantee the ordering anyway):

- shared memory active message library by [Fleiner 95],
- Myrinet Active Message Library, by [Martin 95],
- Meiko CS-2 Active Message Library, by [Yoshikawa 95].

Only the TCP/IP Active Message library (TAM) by [Liu 94] guarantees the ordering, because the software and the underlying communication library (TCP) respect message ordering, however, as this library relies on TCP/IP it is too slow for most pSather programs.

Additionally, the same argument against instruction reordering is still valid for this memory model. In fact, the example shown for the serial consistency memory model works here too, as this system too guarantees an ordering among all writes

4.4 Weak Consistency

What we really need is a memory model that allows a thread to read and change the global memory for some time without interaction from other threads. Only at some memory synchronization points there will be an exchange of the global memory information. A model that supports this is the weak memory model described here and the release consistency model described below.

The weak memory model introduced by [Dubois 90] defines that memory writes are observed by other threads at latest at synchronization points as shown in figure 4-6.

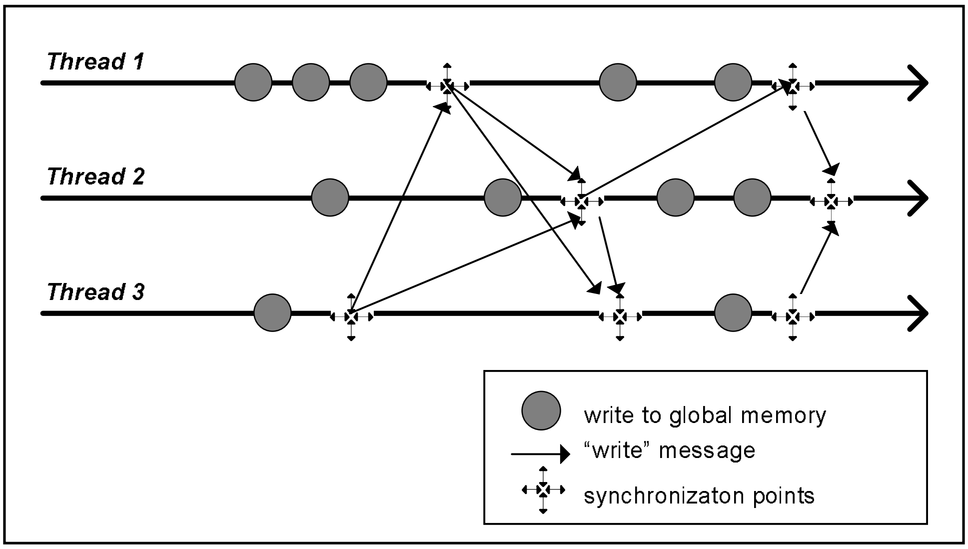

Figure 4-6: Weak consistency

Within this model all optimizations can be applied, as it no longer guarantees that any thread sees writes from some other thread in a given order.

4.5 Release Consistency

The release consistency model was introduced by [Gharachorloo 90] as an extension of the weak consistency model. It distinguishes two kinds of synchronization points, namely acquire and release synchronization points. Acquire synchronization points are used to gain access to a critical section of the program, while release synchronization points are used when leaving a critical section.

When entering a critical section, a thread has to know the last values written to the variables by some other threads that entered this critical section, hence the thread has to “import” (or acquire) all the writes done by all threads that left this critical section. Similarly, at the end of the critical section, the thread has to make sure that all subsequent threads that enter this critical sections see the changed it made by “exporting” (or releasing) all writes it did.

This is the model used in pSather, as it is the most flexible one with respect to implementation, optimizations and efficiency. pSather knows about two explicit synchronizations points, namely the `export` and `import` statements. If a thread executes an `export` statement, it makes all writes done since the previous export known to all other threads. Similarly, during an `import`, it will check all writes done by other threads. Figure 4-7 shows the relationship between imports and exports.

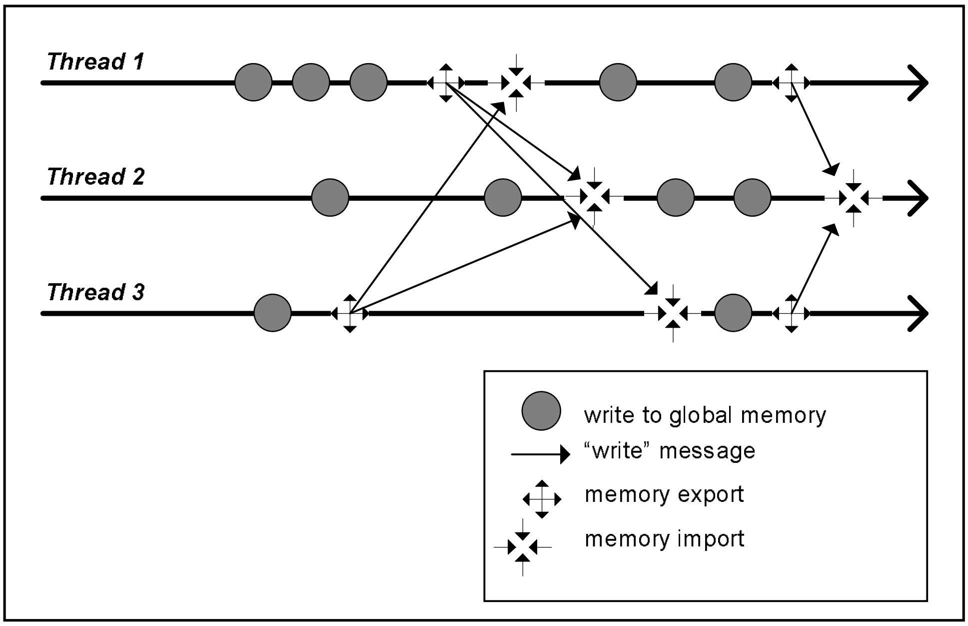

Figure 4-7: Release consistency

One important note: export and import are only the *latest* points used to propagate writes or reading other threads memory. The weak memory model allows this to happen earlier. In the pSather implementation on the multiprocessor, shared memory architecture Sparc10 most writes are visible to other threads almost immediately, although the programmer cannot rely on this. In fact, Solaris does not guarantee this, but instead uses a very similar memory model, where writes are usually only visible after an explicit synchronization of threads [Weaver 93]. However, pSather does not guarantee that writes that are visible before an import stay visible until the import, it may even be that the value of a variable flips between the new and the old value until an import is executed by this thread[26].

The lock statement executes an implicit import and export as required by the release consistency memory model as shown in figure 4-8. pSather executes an implicit import before synchronizing, and an export before releasing the locks. This eliminates virtually all explicit imports and exports. For example all programs used in this thesis do use neither an explicit import nor an export.

```
lock l1, l2 then
    import; -- implicit import
    ...
    export; -- implicit export
end;
```

Figure 4-8: Implicit import and export[27]

This feature allows one to use the release consistency memory model the same way one would use the sequential memory model, if every access to the global memory space is enclosed in an appropriate lock statement. This synchronizes all threads in a way that each thread will see the latest data written by other threads to a particular global memory position, at the same time this eliminates all race conditions. This works the other way around too: by eliminating all race conditions one gets the equivalent of a sequential consistent memory model, as all write accesses to the global memory are properly serialized.

Note that at synchronization points the complete global memory is “imported” and “exported”, not just the values read or written within the critical section. Without this global import/export many programs would no longer work. Consider for example the insertion operation of a list of some complex object as shown in figure 4-9. The new element of the list, namely `t`, has been constructed outside the critical section and would not be exported to other threads if only values changed inside the critical sections would have been exported at the end of the lock statement.

The implementation of the import/export command in pSather is obviously very machine dependent. All the current Solaris machines do it as follows (at least if caching and post writing (see chapter 6 for a more detailed description of those optimizations) are used.

`import`: flush the cache that is used to store far values for the thread that executes the import (other threads may still see older values though). To flush the cache in the current implementation requires just an incrementation of one counter and is therefore very cheap.

[26] This can happen for example with common subexpression elimination, when some accesses to an attribute are replaced with locals, but some remain. If the attribute changes value before an export, all expression that use the local variable see the old value, while the other expressions see the new value.

[27] The lock statement used in this example is described below in chapter 3.1

`export`: wait for all writes that have been done since the last export to finish (each cluster that is the target of a remote write has to send an acknowledge back).

```
class LIST{T} is
   insert(t:T) is
      l::=#LIST_ELEMENT{T}(t);
      -- make sure that only one thread inserts an element at a time
      lock mutex then
         if size=0 then
            head:=l;
         else
            tail.next:=l;
         end;
         tail:=l;
         size:=size+1;
      end;
   end;
end;
```

Figure 4-9: Inserting a list element

4.6 Summary

This table summarizes the advantages and problems of the different memory models. The details, especially regarding the different optimizations, are described in the next two chapters.

Memory Model	***Sequential Consistency***	***Processor Consistency***	***Thread Consistency***	***Weak Consistency***	***Release Consistency***
Serial Optimizations					
• Inlining	✓	✓	✓	✓	✓
• Loop Unrolling	✓	✓	✓	✓	✓
• CSE	local[28]	local	local	✓	✓
• Loop Invariant Hoisting	local	local	local	✓	✓
• Iter Initialization	local	local	local	✓	✓
Parallel Optimizations					
• Caching	☠	☠	☠	✓	✓
• Prefetching	☠	☠	☠	✓	✓
• Post Writing	☠	☠	☠	✓	✓
• Parloops	✓	✓	✓	✓	✓
• Local Execution	✓	✓	✓	✓	✓
• Remote Execution	✓	✓	✓	✓	✓

Table 4-1: Memory Models

[28] Local memory is the memory accessible only by the current thread, i.e. local variables

5. Serial Optimizations

This chapter describes some of the well known optimization techniques for serial languages after a short introduction that describes how the Sather compiler collects the data it needs to implement the optimizations.

The results of all optimizations are studied on several Sather and pSather programs described in Appendix C. The full results can be found on the world wide web at `http://www.icsi.berkeley.edu/~fleiner/benchmarks`. The results are presented as a speedup table and graphic. The tables show, for each program, two speed up numbers calculated as follows for a given optimization (the calculations are done for the inlining option):

n_1 := time to execute the program compiled with no optimizations.

n_2 := time to execute the program compiled with inlining.

n_3 := time to execute the program compiled with all optimizations except inlining.

n_4 := time to execute the program compiled with all optimizations.

$$Speedup_1 := \frac{n_1}{n_2}$$

$$Speedup_2 := \frac{n_3}{n_4}$$

$Speedup_1$ therefor measures the speedup of an optimizations by itself, while $Speedup_2$ shows the impact of the optimization in the context of all other optimizations.

All benchmarks were executed on a Sparc 10 66MHz system with 128MB RAM running Solaris 2.5 with the Sather Compiler version 1.1b. For each timing the program was run until the five fastest measurements had a standard error of less than 1%. This was necessary to eliminate fluctuations in the measurements stemming from some other programs running on the same computer.

A table with the complete results can be found in chapter 8.

5.1 Introduction

All optimizations described in this chapter are done wihtin the context of one function (also known as intraprocedural or global optimizations). However, the compiler needs detailed information about all the functions called from within the function it is working on. The current pSather compiler maintains for each function a large list of characteristics. This list is created for each function while it is compiled. The table 5-1 lists all the information available on a per function basis[29].

	Type	Description
attribute read	List of Attributes	list of class attributes that may be read by this function or any function called by it. Example: for the function in fig 5-1 we get "LIST::size"[30].
array read	List of Classes	List of all array classes, if an array element of such a class may be read.
attribute write	List of Attributes	list of class attributes that may be written while executing the function. For the function in fig 5-1 we get "LIST::head", "LIST::tail", "LIST::size" and "LIST_ELEMENT::next".
array write	List of Classes	List of all array classes, if an array element of such a call may be changed by this function.
exception	Boolean	true if the function may raise an exception.
import	Boolean	true if the function may execute an explicit or implicit[31] import. A function with an import is similar to a function that "reads everything".
export	Boolean	true if the function may execute an explicit or implicit[31] export.
fatal error	Boolean	true if the function may abort the program, for example because of an attribute access of void. Used to evaluate expressions even if at this point the compiler is not sure that the value will ever be used. See chapter 5.3 on page 67 for a more detailed description of this problem.
arithmetic error	Boolean	true if the function may abort the program because of an arithmetic check. Note that the current pSather compiler has an option to turn arithmetic checks off. Used for the same optimizations as "**fatal error**"
yield inside lock	Boolean	only meaningful for iterators, and true if there is a yield inside a lock statement. Used for synchronization optimizations described in chapter 7.2.

[29] The first version of the algorithm that collected a part of this information in the Sather compiler has been implemented by Trevor Pering and was later extended by the author to add the information needed for the more complex optimizations and for pSather optimizations.

[30] The current compiler uses an upper limit for the number of attribute reads and writes per function. If the function reads or writes more attributes, it is considered to "read everything" or "write everything".

[31] The lock statement executes an implicit import/export, see page 57 for more details.

	Type	Description
blocking	Boolean	true if the function can block because of a lock statement or by calling some system or library function that may block. Used primarily to execute the work of several threads by one thread, as described in chapter 6.5.
forks	Boolean	true if the function forks a new thread.
unsafe	Boolean	true if the function is generally unsafe and cannot be considered for most optimizations. This is notably the case of functions with inlined C and all external functions.
creates	Boolean	true if the function or one of the functions it calls create a new reference object. Used to decide if functions can be executed remotely.

Table 5-1: Information gathered for each function

```
class LIST is
   append(e:LIST_ELEMENT) is
      if size=0 then
         head:=e;
         tail:=e;
      else
         tail.next:=e;
         tail:=e;
      end;
      size:=size+1;
end;
```

Figure 5-1: Appending an element to a linked list

The information gathering process is relatively straightforward, as the pSather compiler compiles functions in a bottom up way, that is, whenever a function is compiled all the functions it calls are already compiled (with the exceptions of recursive functions of course). Figure 5-2 shows the pseudo code for the recursive function used in the compiler to accomplish this. Note that all builtin functions[32] are properly annotated such that the compiler knows their side effects too.

[32] builtin functions are the functions of the standard library which are not implemented in Sather but for which the compiler knows the C code.

```
compile_function(f:FUNCTION) is
   loop
      c::=statement!;
      if c.is_function_call then
         if c.not_yet_compiled then
            compile_function(c);
         end;
      end;
      -- compile statement c
   end;
end;
```

Figure 5-2: Simple compile function inside the compiler

The memory and time requirement for this kind of dataflow mechanism are proportional to the number of functions and the size of the program, respectively. However, in some cases the information available through this mechanism is not detailed enough, forcing the compiler to make conservative decisions. Take for example the list copy function in figure 5-3, where the information available to the compiler does not show the fact that the list passed as argument will not change at all. As both lists are of the same type, the compiler assumes that either one could change, as it knows only that this function changes all attributes of a list, but not which list will change. The compiler cannot even assume that only the lists passed as arguments will change, but any list could change.

```
class LIST is
   copy(l:SAME) is
      clear;
      loop append(l.elt!); end;
   end;
end;
```

Figure 5-3: Copying a list

5.2 Inlining

To call a routine or function on most computer architectures today uses several CPU cycles, namely to store the return address, save some of the registers on the stack and to set up the new frame for local variables. The cost of calling a function is not very important as long as all functions called are relatively large. Object oriented languages however tend to define many small member functions, which in other languages like C would have been defined as macros. A natural solution that has been adopted in several languages is to inline small functions. Figure 5-4 shows the effect of inlining.

```
class ARRAY2{T} is
    -- return the element of a 2 dimensional array (Sather knows only about
    -- 1 dimensional arrays, higher dimensions are simulated)
    aget(x,y:INT) is return self[x+y*cols]; end;
end;

    loop y::=r.cols.times!;
        loop x::=r.rows.times!;
            loop z::=a.rows.times!;
                -- the main line of a matrix multiplication routine
                r[x,y]:=r[x,y]+a[x,z]*b[z,y];
            end;
        end;
    end;

    -- the main line after inlining
    r[x+r.cols*y]:= r[x+r.cols*y]+
                          a[x+b.cols*z]*b[z+b.cols*y]
```

Figure 5-4: Inlining

However, by inlining such code we can win twice, as not only do we eliminate a function call, but in the example in figure 5-4 we can also remove several loop invariants, as many subexpressions used to calculate array indices are constant, at least inside the innermost loop. Chapter 5.3 shows the win when applying loop invariant hoisting after inlining. Note that inlining works for serial programs as well as for parallel programs.

For most programs inlining is a very effective optimization strategy. The main reason is that object-oriented programs tend to have lots and lots of small routines, like the `aget` and `aset` routine of arrays and matrices. Additionally, inlining is important for some other optimizations, as those optimizations work on one function only, and not on function calls. By inlining, many function calls are turned into standard Sather statements that can be further optimized as discussed below.

Note that some programs do not benefit at all from inlining. One of those programs (Queen) uses a recursive algorithm, while the other one (Differential Evolution) has an inner loop which cannot be inlined by the Sather inliner (it uses builtin iters and functions, and those are inlined in any case). A program can get slower when inlining functions because functions get larger, which may results in less cache hits for the code cache, and those functions tend to use more locals (register pressure), which may result in less cache hits for the data cache.

For some other programs (Matrix Multiplication) inlining by itself gives already a speedup, however, combined with other optimizations the program gets more than twice as fast as without inlining. The reason is that some other optimizations (in this case loop invariant hoisting) can do a much better job after inlining.

The inliner of the current compiler was implemented by Boris Weissman and Ilia Vinarsky [Weissman 95]. It inlines all function calls that have at most one return at the end of the function. The iterator inliner is much more restricted, as it is only able to inline iterators of the form shown in figure 5-6. Moreover, most of the iterator it inlined in the 1.0.6 Sather compiler have been converted to “builtin” iterators[33]. Builtin iterators are always inlined, even if the inliner is not used at all.

Program[34]	inlining only	inlining including all serial optimizations
Heat	1.37	1.48
Parallel Heat	1.23	1.31
Matrix Multiplication	1.54	6.79
Parallel Matrix Multiplication	1.50	4.57
Queen	0.97	0.98
Parallel Queen	1.03	0.99
Differential Evolution	1.06	1.01
Parallel Differential Evolution	0.92	0.98
Photo Filter	1.26	1.26
Photo Filter in Parallel	1.16	1.18
Compiler	1.21	1.16
Compiler with Optimizations	1.20	1.24

Table 5-2: Speedup of -O_inline

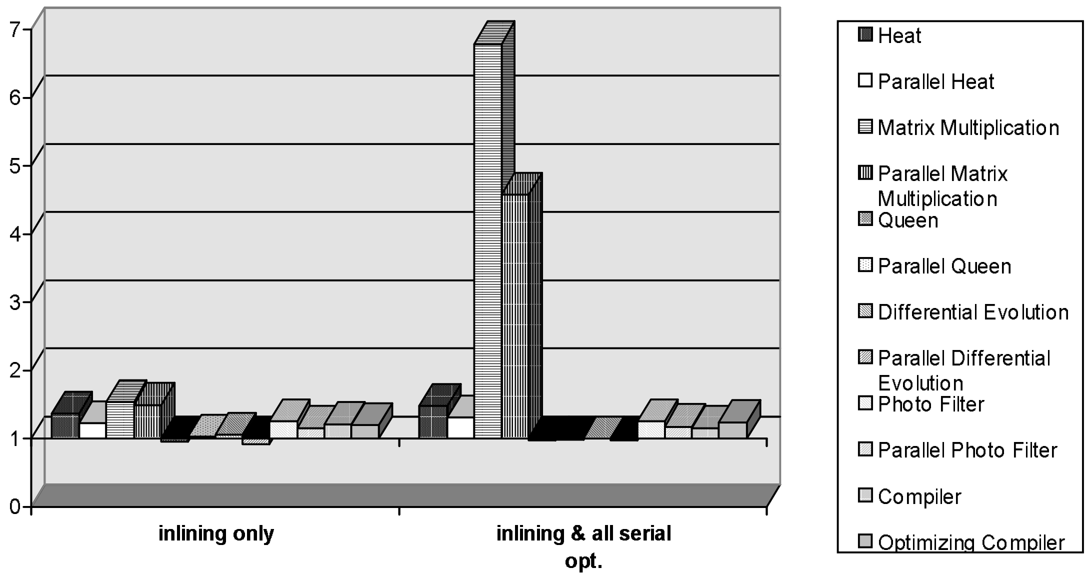

Figure 5-5: Speedup of -O_inline

[33] Builtin iterators and functions are part of the standard library, but rather than being implemented in standard Sather code, they are implemented directly in C. The compiler then replaces every call to a builtin function or iterator with the appropriate C code.

[34] The programs used as benchmarks are described in Appendix C

```
iter!(args):optional_return_type
is
    -- initialization
    loop
        -- before yielding code
        yield optional_return_value;
        -- after yielding code
    end;
end;
```

Figure 5-6: Skeleton of iterators that are currently inlined

5.3 Loop Invariant Hoisting

Many loops have loop invariant expressions that are evaluated in each iteration. This is unnecessary work and the compiler should evaluate all invariant expressions at the beginning of the loop, store the result in a local variable and use this result instead of reevaluating the same expression over and over again.

This technique is especially useful for small loops. Figure 5-8 shows the inner loop of a matrix multiplication routine and the code after inlining and loop invariant hoisting. This example also shows that a strongly typed language has a much better optimization potential than a weakly typed one. The same code written in C++ (or C, for that matter) will not be optimized as far as the Sather code. The reason is that the line `r[x,y]:=...` changes an element of the global memory, namely the matrix element. Unfortunately, the number of columns and rows of the matrix is also stored in the global memory and the compiler has no way to know if an assignment to `r[x,y]` may change one of those sizes of the other matrix at the same time. Therefore none of the attribute accesses are loop invariants as far as the C or C++ compiler is concerned.

Matrix Multiplication (300x300)	non optimized	compiler optimized	hand optimized
Sather	96.9 sec.	12.3 sec.	n/a
C	87.5 sec.	58.3 sec.	9.9 sec.
C++	91.2 sec.	57.6 sec.	12.7 sec.
pSather (400x400)	130.2 sec.	14.0 sec.	n/a
threaded C (400x400)	60.3 sec.	42.0 sec.	13.9 sec.

Table 5-3: Speed of Sather versus C / C++ program

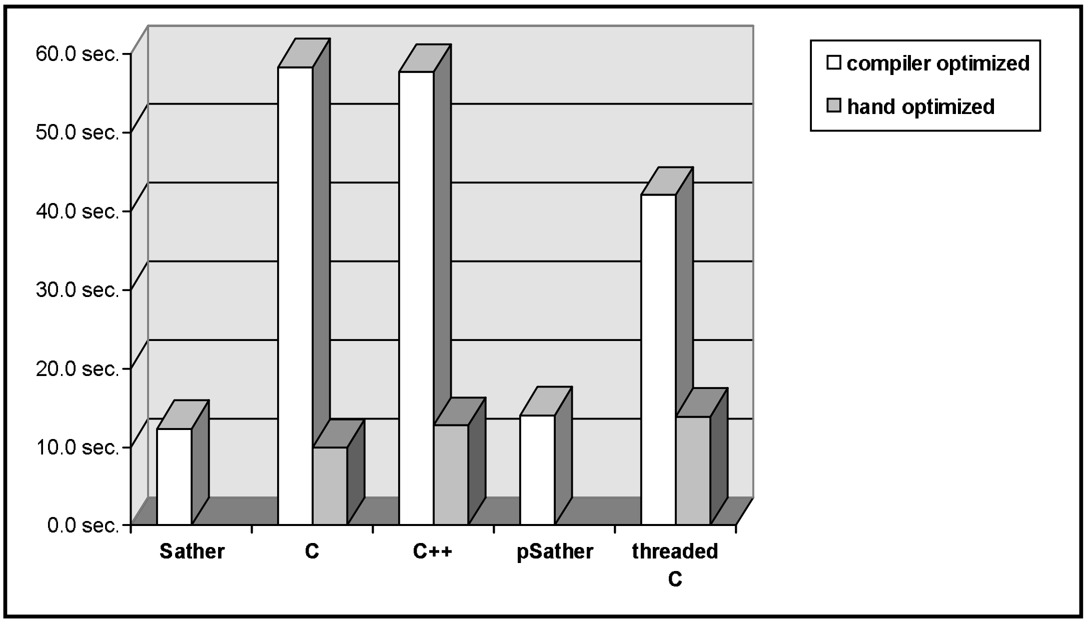

Figure 5-7: Speed of Sather versus C / C++ programs

The results in table 5-3 show the time of the optimized C and C++ versions, and of hand optimized C and C++ versions. The Sather compiler is able to generate code that is faster than C++ and nearly as fast as hand optimized C code (the difference comes from the fact that the C code moves the break of the loop to the end of the loop and therefore has a slightly smaller loop. A good C compiler however could easily create the same code for Sather as the hand optimized C version[35]).

Loop invariant hoisting consists not just of identifying expressions that are invariant during the evaluation of a loop; the compiler can only move expressions that have no side effects and cannot throw exceptions. Additionally, the compiler has to make sure that either the expression is evaluated in any case at least once (for Sather/pSather the compiler has to check that it is evaluated before the first iterator or the first function that could raise an exception that would end the loop), or that it cannot crash the program by accessing a void attribute or executing illegal instructions that cause a program termination like “division by zero”. Figure 5-9 shows an example where the invariant expression `a.size` cannot be hoisted without precautions, as it is only evaluated as long as the array `b` is not empty. If, in this case, `a` is void, the hoisted expression would end the program with a fatal error.

[35] The C code generated by the Sather compiler has been compiled with Gnu C Version 7.1.

```
mult(a,b:MATRIX{T}):MATRIX{T} pre cols=m.rows is
    r::=#SAME(a.rows,a.rows);
    loop y::=r.cols.times!;
        loop x::=r.rows.times!;
            loop z::=a.rows.times!;
                -- the main line of a matrix multiplication routine
                r[x,y]:=r[x,y]+a[x,z]*b[z,y];
            end;
        end;
    end;
    return r;
end;

    -- the main line after inlining (expressions that can be hoisted are underlined
    r[x+r.cols*y]:= r[x+r.cols*y]+
                            a[x+b.cols*z]*b[z+b.cols*y]

-- the same routine as above, after loop invariant hoisting
mult(a,b:MATRIX{T}):MATRIX{T} pre cols=m.rows is
    r::=#SAME(a.rows,a.rows);
    tmp_r_cols::=r.cols; -- attr access are slower than locals
    tmp_r_rows::=r.rows;
    tmp_a_cols::=a.cols;
    tmp_a_rows::=a.rows;
    tmp_b_cols::=b.cols;
    loop y::=tmp_r_cols.times!;
        tmp1::=tmp_r_cols*y;
        tmp2::=tmp_b_cols*y;
        loop x::=tmp_r_rows.times!;
            tmp3::=x+tmp1;
            loop z::=tmp_a_rows.times!;
                -- the main line of a matrix multiplication routine
                r[tmp3]:=r[tmp3]
                            +a[x+tmp_b_cols*z]*b[z+tmp2];
            end;
        end;
    end;
    return r;
end;
```

Figure 5-8: Inlining and invariant hoisting

```
loop
   sum:=sum+b.ind!*a.size;
end;

-- the following code is NOT equivalent to the one above, as it will crash
-- with an attribute access of void in the first line,
-- if a and b are both void, while the one above will not crash in this case
tmp::=a.size;
loop
   sum:=sum+b.ind!*tmp;
end;

-- to make it work, we can add an if construct (the current compiler uses the
-- conditional C construct :? for those cases).
tmp::=0;
if ~void(a) then tmp:=a.size; end;
loop
   sum:=sum+b.ind!*tmp;
end;
```

Figure 5-9: Hoisting of attribute accesses

Invariant expressions can be hoisted under the following conditions in pSather:

- the expression is truly invariant during evaluation of the loop (this check has to take into account the effect of an import inside the loop) and has no side effects.
- the expression is guaranteed to be executed at least once[36], or it has no "bad" side effects, i.e. no attribute access of void or illegal instructions (the expression may be rewritten by the compiler to avoid such instructions, although this makes the code slightly larger and slower to execute. The current Sather compiler does use such techniques to avoid dereferencing void variables.).

Some comments: the innermost loop of the matrix multiplication program has no constant expressions to hoist, unless the inliner actually inlines all accesses to the matrices. As soon as this happens, the speed of the program is about 3 to 5 times faster than before. The same is true for the Parallel Queen version. In the case of Differential Evolution however, the option "-O_hoist_loop_invariants" hoists exactly the same expressions as the option to hoist iterator initialization (see below) and there is nothing left to hoist.

[36] The expression has to be before the first iterator call and code that could raise an exception, and it may not be inside conditional code (i.e. in an if expression).

Program	Hoisting loop invariants only	Hoisting Loop invariants including all serial optimizations
Heat	1.00	1.09
Parallel Heat	1.00	1.05
Matrix Multiplication	1.00	4.44
Parallel Matrix Multiplication	1.00	3.03
Queen	0.97	0.99
Parallel Queen	0.99	1.19
Differential Evolution	1.14	1.00
Parallel Differential Evolution	1.06	0.97
Photo Filter	0.98	1.00
Photo Filter in Parallel	1.00	1.02
Compiler	0.98	0.90
Compiler with Optimizations	1.01	1.01

Table 5-4: Speedup of -O_loop_const

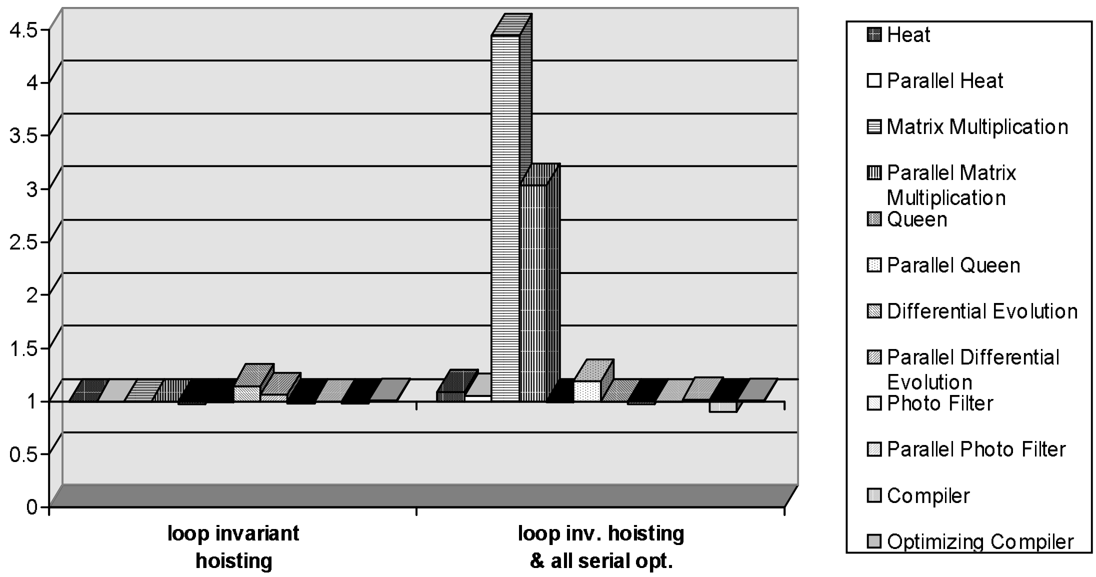

Figure 5-10: Speedup of loop invariant hoisting

5.4 Iterator Initialization

Iterators are a very special construct in Sather/pSather. They are defined like normal functions, but instead of returning a value they "`yield`" a value and are resumed later. Their arguments are special too, as some arguments are evaluated only the first time the iterator is called and others each time the iterator is called. Figure 5-11 shows two often used iterators.

```
immutable class INT is
    -- yield all integers, starting with self up to and including the argument
    upto!(once to:INT):INT is
        i::=self;
        loop
            yield i;
            if i=to then quit; end;
            i:=i+1;
        end;
    end;
end;

class ARRAY{T} is
    -- set successive elements of the array
    set!(t:T) is
        loop
            self[0.upto!(asize-1)]:=t;
            yield;
        end;
    end;
end;

-- using the set! iterator to set all elements of an array to successive integers
    loop
        array.set!(100.upto!(array.asize+100));
    end;
```

Figure 5-11: Two widely used iterators of the standard library

When calling an iterator for the first time in a loop, the system has to execute some special code which will not be executed on subsequent iterations of the loop. This initialization code evaluates all once arguments and stores them in local variables (note that `self` itself is a once argument). Figure 5-12 shows parts of the C code generated for the loop in figure 5-11.

```
-- generated C code (not optimized)
int array_setb_first=1;
int int_uptob_first=1;
while(1) {
    /* initialize INT::upto! frame */
    if(int_uptob_first) {
        int_uptob_first=0;
        int_uptob_self=100;
        int_uptob_arg1=array->asize+100;
    }
    tmp=INT_uptob(int_uptob_self,int_uptob_arg1);
    /* initialize ARRAY::set! frame */
    if(array_setb_first) {
        array_setb_first=0;
        array_setb_self=array;
    }
    ARRAY_setb(array_setb_self,tmp);
}

-- generated C code ( optimized)
/* initialize INT::upto! frame */
int_uptob_self=100;
int_uptob_arg1=array->asize+100;
/* initialize ARRAY::set! frame */
array_setb_self=array;
array_setb_self=array;
while(1) {
    tmp=INT_uptob(int_uptob_self,int_uptob_arg1);
    ARRAY_setb(array_setb_self,tmp);
}
```

Figure 5-12: Iterator initialization code

Although the generated C code works fine, one would certainly prefer to move both tests out of the loop, and in this case this is possible, as all expressions evaluated inside the if expressions are not affected by the code inside the loop that is executed prior the their evaluation. The second part of figure 5-12 shows the optimized version of the C code.

Hoisting iterator initialization code is very similar to hoisting loop invariants, the only difference is that the initialization expressions have to be invariant between the beginning of the loop and the first time they are evaluated, as opposed to loop invariants that have to be constants during the whole loop. Often, hoisting loop invariants results in nearly the same code as this optimization, the only difference being the `if()` statement which loop invariant hoisting will not remove.

Program	Hoisting iter. initialization only	Hoisting iter. initialization including all serial optimizations
Heat	1.01	1.01
Parallel Heat	1.00	1.02
Matrix Multiplication	1.02	1.07
Parallel Matrix Multiplication	1.01	1.03
Queen	1.18	1.13
Parallel Queen	1.16	1.17
Differential Evolution	1.27	1.12
Parallel Differential Evolution	1.18	1.07
Photo Filter	1.03	1.00
Photo Filter in Parallel	1.00	0.99
Compiler	0.99	0.88
Compiler with Optimizations	1.02	1.00

Table 5-5: Speedup of option -O_hoist_iter_init

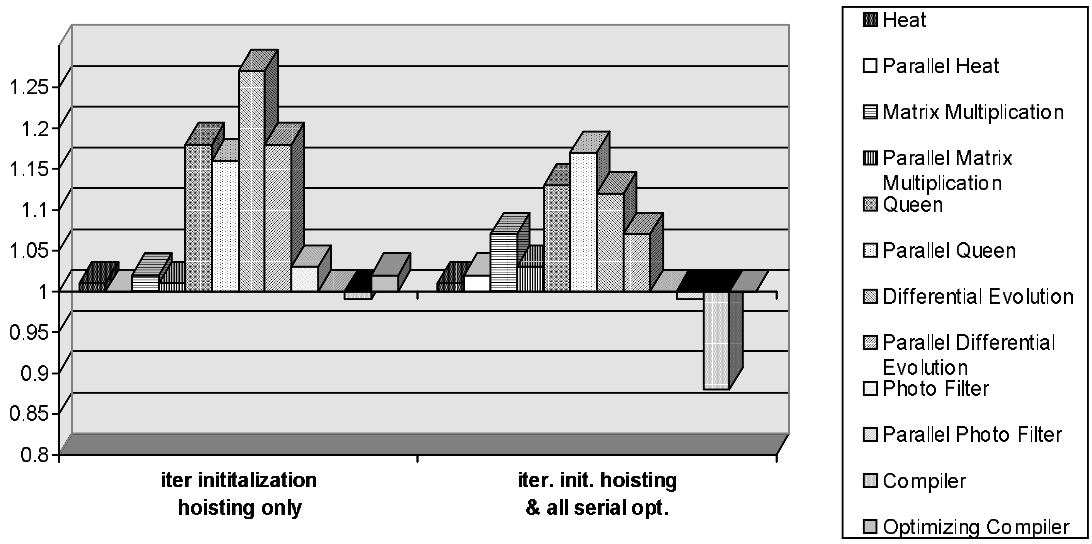

Figure 5-13: Speedup of iterator initialization hoisting

Note that for the differential evolution program we have the same effect for this option as for the option “hoist loop invariants”, as those two options optimize the same expressions in the program. As the option discussed here is also able to remove some superfluous `if()` statements, it does a better job in these cases than hoisting loop invariants only.

5.5 Loop Unrolling

Loop unrolling describes a loop optimization technique where the body of a loop is repeated several times to reduce the number of tests to check if the loop should be terminated. This works best if the number of times the loop has to be executed is a multiple of the number of loop body repetitions, as otherwise one has to take care of the remaining iterations. Figure 5-14 shows a strange way to solve this problem in C.

```
register n = (count + 7) / 8;      /* count > 0 assumed */
switch (count % 8)
{
case 0:        do { *to = *from++;
case 7:              *to = *from++;
case 6:              *to = *from++;
case 5:              *to = *from++;
case 4:              *to = *from++;
case 3:              *to = *from++;
case 2:              *to = *from++;
case 1:              *to = *from++;
                   } while (--n > 0);
}
```

Figure 5-14: Duff's device, unrolling a loop [Raymond 91]

The Sather compiler however does only some very limited loop unrolling (which may not even qualify as loop unrolling) by moving initial `while!` or `until!` iterators to the end and adding an `if` statement before the loop. Figure 5-15 shows how this is done. There are two reasons for this optimization:

- when we move an iterator to the end of the loop, loop invariants and iterator initializations can more easily be hoisted. We have seen that one of the conditions of moving a loop invariant is that it appears either before the first iterator, or has no "bad" side effects. In the example in figure 5-15 we can safely hoist the initialization of the `set!` iterator, which will read the size of the array, something that cannot be done if the array is void. This could not have been done in the original loop, because if both `list.head` and `array` are void, the `set!` iterator would have never been evaluated.
- if the loop terminates the first time the test is executed, all loop invariants and iterator initializations hoisted out of the loop would have been evaluated for nothing. In figure 5-15 we avoid the initialization of the `set!` iterator if the list is empty and the iterator would have never been called anyway.

```
-- copy the elements of a linked list to an array
   list_element:=list.head;
   loop
      until!(void(list_element));
      array.set!(list_element.value);
      list_element:=list_element.next;
   end;

-- the same function after unrolling
   list_element:=list.head;
   if ~void(list_element) then
      loop
         array.set!(list_element.value);
         list_element:=list_element.next;
         until!(void(list_element));
      end;
   end;
```

Figure 5-15: Loop unrolling in the Sather compiler

Of course it is not possible to apply this optimization if the condition inside the `while!` or `until!` iterator itself has a call to an iterator, as in this case the condition could not be duplicated and there cannot be an iterator call outside the loop.

Program	Loop unrolling only	Loop unrolling including all serial optimizations
Heat	1.00	1.01
Parallel Heat	1.01	1.01
Matrix Multiplication	1.01	0.96
Parallel Matrix Multiplication	1.00	1.01
Queen	1.05	1.08
Parallel Queen	1.12	1.18
Differential Evolution	1.07	0.99
Parallel Differential Evolution	0.98	0.96
Photo Filter	1.00	1.00
Photo Filter in Parallel	0.98	1.06
Compiler	0.97	0.88
Compiler with Optimizations	1.01	1.00

Table 5-6: Speedup of loop unrolling

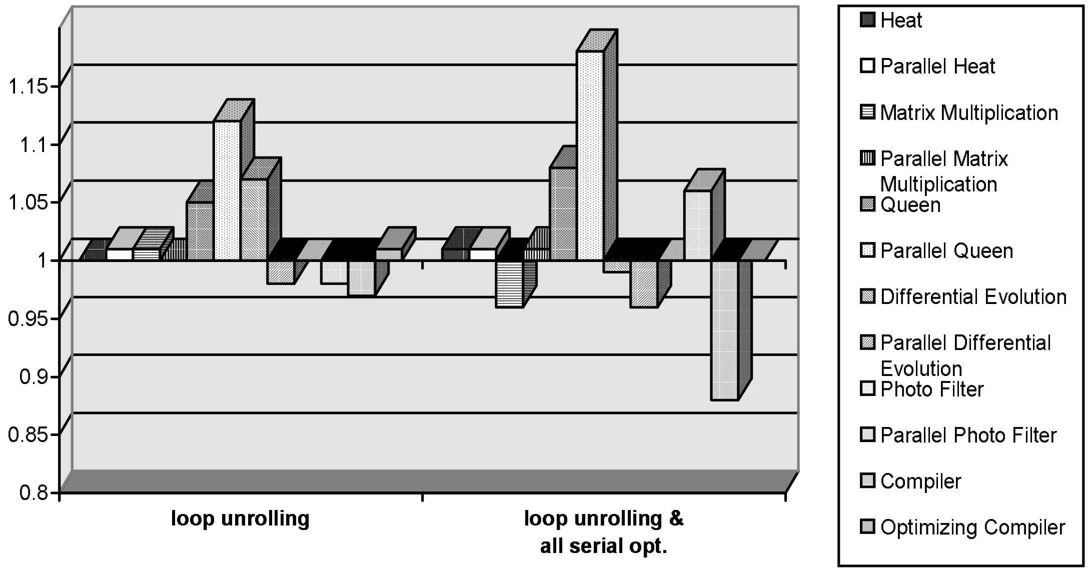

Figure 5-16: Speedup of loop unrolling

Normally one would expect that this option used by itself would not have any effect on the program. However, moving the test from the beginning of the loop to the end may help in reusing registers and the new code may fit a compiler pattern of the C level optimizer. Besides, by executing the test before initializing the data structures used for the loop, it is possible to save additional CPU cycles in the case when the test fails the first time.

5.6 Common Subexpression Elimination

Common subexpression elimination (CSE) allows programmers to write clearer code that repeats the same expression several times[37]. The compiler recognizes those repetitions, executes the calculations once and stores the result for future use. Obviously, this works only if the compiler can prove that the result of the calculation cannot change from one instance of the expression to the next as shown in figure 5-17 and that the expression has no side effects, although the expression may throw an exception or crash the program, for example by accessing attributes of a void object.

```
-- non optimized version
   f[x*width+y]:=g[x*width+y]+h[x*width+y];

-- optimized version
   tmp::=x*width+y;
   f[tmp]:=g[tmp]+h[tmp];
```

Figure 5-17: Common Subexpression Elimination

[37] This is only true for small expressions, especially for attribute accesses. Large expression should be assigned to local variables, as this is often clearer.

Unfortunately, this optimization is often not very useful by itself, especially for Sather and pSather, were the C Compiler will elimiate most of the common subexpressions anyway. Figure 5-19 shows the result of common subexpression elimination when used by itself on the test programs.

However, together with inlining we sometimes get better results, as it frequently happens that inlining exposes many common subexpression. An example is matrix code, where each access to an element of the matrix needs the width of the matrix. As the width is stored in an object, it is faster to store it for as long as possible in a local variable, especially if the matrix is a far object in pSather. Figure 5-18 shows an example where inlining and CSE work together.

```
-- square all elements of a matrix
loop
   x::=0.upto!(m.size_x-1);
   loop
      y::=0.upto!(m.size_y-1);
      m[x,y]:=m[x,y]*m[x,y]; -- ***
   end;
end;

-- the line *** after inlining
      m[x*m.size_x+y]:=m[x*m.size_x+y]*m[x*m.size_x+y];

-- the same code after common subexpression elimination
      tmp1::=x*m.size_x+y;
      tmp2::=m[tmp1];
      m[tmp1]:=tmp2*tmp2;
```

Figure 5-18: Inlining and CSE

Even though CSE is not a big win for serial Sather programs, we get better speedups in distributed pSather, as it sometimes allows the elimination of far reads by eliminating attribute accesses.

Before eliminating a subexpression, the compiler has to make sure that

- the subexpression has no side effects, as those side effects would no longer be visible on the second call,
- if the expression can raise an exception, the compiler has to make sure that either the second expression cannot be evaluated after the first one raised an exception, or it has to insert special code that raises the same exception when the second expression is evaluated. The current Sather/pSather compiler checks this by making sure that both expressions are in the same `protect` block.
- the evaluation of the subexpression yields the same value in both cases, that is, none of code between the first and second evaluation of the subexpression changes any of the values used in those subexpressions, not even because another thread could write a new value to those objects, which can be easily checked

in pSather by making sure that there is no implicit or explicit import between the two expressions.

Program	CSE only	CSE including all serial optimizations
Heat	1.00	1.00
Parallel Heat	1.01	1.01
Matrix Multiplication	1.00	0.97
Parallel Matrix Multiplication	1.00	1.00
Queen	1.02	1.00
Parallel Queen	0.96	1.06
Differential Evolution	1.00	1.00
Parallel Differential Evolution	0.98	0.98
Photo Filter	1.00	1.00
Photo Filter in Parallel	0.99	0.99
Compiler	1.05	0.90
Compiler with Optimizations	1.03	0.99

Table 5-7: Speedup of -O_cse

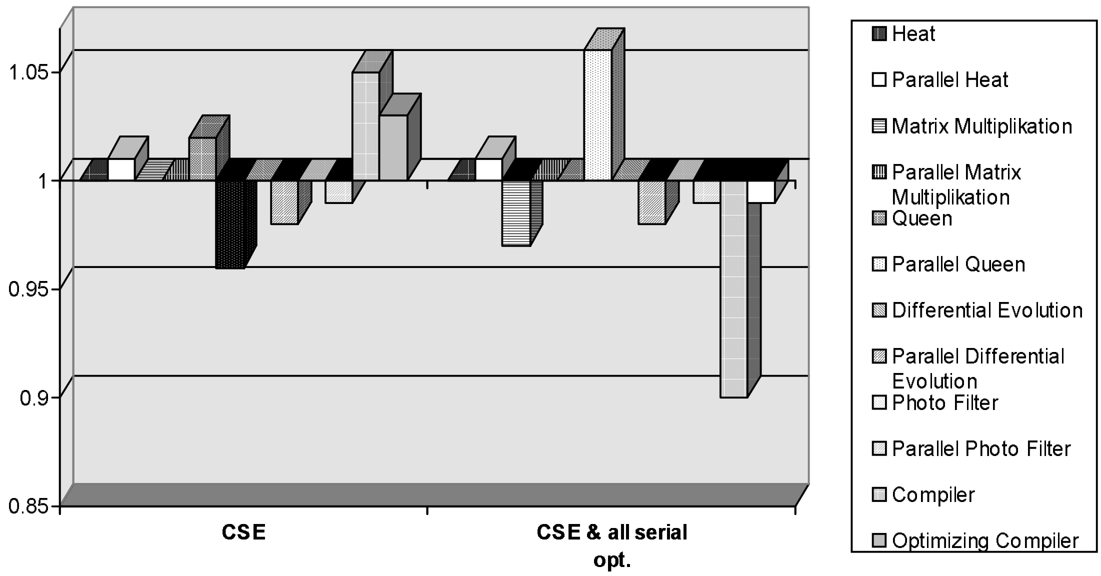

Figure 5-19: Speedup of common subexpression elimination

The reason for the bad results for the option -O_cse comes from the fact that two other optimization options remove most of the common subexpressions in a program (the only ones that are not removed are the ones that are frequently eliminated by the C compiler or that are outside loops, where the elimination has nearly no impact). Those two optimizations are loop invariant hoisting and once argument hoisting. Both options move constant expressions out of loops, and the first one also eliminates double expressions while doing so. Even for distributed pSather there is an option that nearly eliminates the use of CSE, namely caching. Accessing the same attribute several times does not result in several far reads, but in the use of the local cache (see chapter 6.2).

A first version of the CSE algorithm was implemented by Trevor Pering. He did this by annotating the intermediate representation of the program. As all other optimizations knew nothing about those annotations, it had the unfortunate side effect that CSE had to be done as the last step of the optimization process. Some of the parallel optimizations described in chapter 6 however should be done after CSE. Prefetching remote values for example should be done only once for the same attribute access. By first eliminating all common subexpressions, the prefetching algorithm is simpler and yields better results. To achieve this the CSE module had to be completely reimplemented to not annotate the intermediate representation of the program, but to perform a code transformation and to assign the expression to a local variable and use this variable. This allowed other optimizations to work on the code generated by the common subexpression algorithm.

6. Parallel Optimizations

6.1 Introduction

The parallel optimizations described in this chapter are quite different from the serial ones described before, as they tend to increase code size, are partly runtime optimizations rather than compile time optimizations and some replace high level pSather language constructs by other, more efficient ones. While the serial optimizations were used to increase speed and sometimes to reduce the code size, the parallel optimizations have some more goals, namely

- **reduce network traffic**: the network that connects different clusters has a limited bandwidth, if the program needs to send more data over the network than it can the network can handle, it will slow down. Additionally, each node that receives a message has to steal valuable CPU cycles from the threads running on it.
- **reduce number of messages sent**: as it is more expensive to send two messages out than one larger (in the very least the program has to call the same function twice instead of once), the compiler and the runtime should try to reduce the number of messages sent around by combining them. Note that some hardware or software implementations may put an upper bound on how much time can be won by this technique. The active message library used for pSather for example supports only very tiny messages (about 20 Bytes per message) and therefor this technique is not used in the current pSather implementation.
- **reduce number of threads running**: if on particular clusters there are more threads running than the number of processors, some threads will have to wait, which is not a problem per se, but those threads have to be created, memory has to be allocated and rescheduled. If it is possible for one thread to execute the tasks we are able to save the time needed to create and manage threads.
- **reduce waiting times**: a thread that needs some information from another cluster will send out a message and wait for the answer. However, if the thread could send ahead of time, the answer could arrive before the thread needs it, and reduce or eliminate the waiting time.

Table 6-1 shows the goal of the different parallel optimizations described here.

Most optimizations described here are only meaningful for the weak memory consistency model and use the fact that a thread can change its view of the global memory without affecting any of the other threads.

	reduce network traffic	reduce messages sent	reduce threads	reduce waiting
Caching	✓	✓		
Prefetching	✓			✓
Post Writing	✓	✓		✓
Combining Parloops			✓	
Local Execution	*speeds up code the same way serial optimizations do*			
Remote Execution	✓	✓		

Table 6-1: Effect of parallel optimizations

The speedup shown in the following tables is, unless otherwise noted, the mean time between three runs on a Meiko CS-2 with up to 40 nodes with 2 processors per node running at 66 MHz each. Runs with a deviation of more than 10% were ignored. The numbers shown for cache hits / prefetches are the numbers from one particular run. All programs were compiled with all serial optimizations as described in the previous chapter on, including the synchronization optimizations described in the next chapter. The second part of the speedup tables shows the speedup when all parallel optimizations are on, namely caching, dynamic local calls (but not remote calls) and postwriting. Remote calling was not turned on as this option may change the semantics of a program and prefetching was also ignored for the reasons outlined below in section 6.3.

The timings represented in this chapter have to be taken with a grain of salt. The Meiko CS-2 used for the measurements was running as a multi-user machine and frequently other programs were running on it (although never on the same nodes that were used to run the benchmark programs). The spread of the distribution of the measurements was frequently higher than 10%, and, in some rare cases some single runs ran more than twice as long as the average. While it is not too much of a surprise to get such measurements for nondeterministic, irregular problems like the Queen program (where the total time depends on how the load gets distributed), it also happened for the very regular Photo Retouching program. There are several reasons for this behavior:

- **cache anomalies**: due to the nondeterminism of a parallel program the data will not be allocated in the same order and in the same place. If two data structures that are frequently used happen to share the same cache lines, the program will run noticeably slower. This can also happen to the code: by adding a function somewhere, a loop may be moved so that it suddenly lies completely inside the cache, or it may now share the cache with some system routine that executes very often. In the first case the program runs faster, in the second one it slows down.
- **network anomalies**: if the network operates just barely below saturation, a small change in the execution of the program may suddenly overload the network. Like a traffic jam this delay may spread out over all nodes and slow them down for some time. The same problem arises when the network is suddenly overloaded by some transmissions of an unrelated program running on some other nodes.

6.2 Caching

Every time a thread needs some attributes of an object stored on another cluster it will have to send a message there and wait until it gets the answer as shown in figure 6-1. If the same thread or some other thread of the same cluster needs the same attribute again, another message will be sent to read it again. This is clearly a waste of time and bandwidth, as the second read could use the same value as the previous one, provided that there was no import between the two reads.

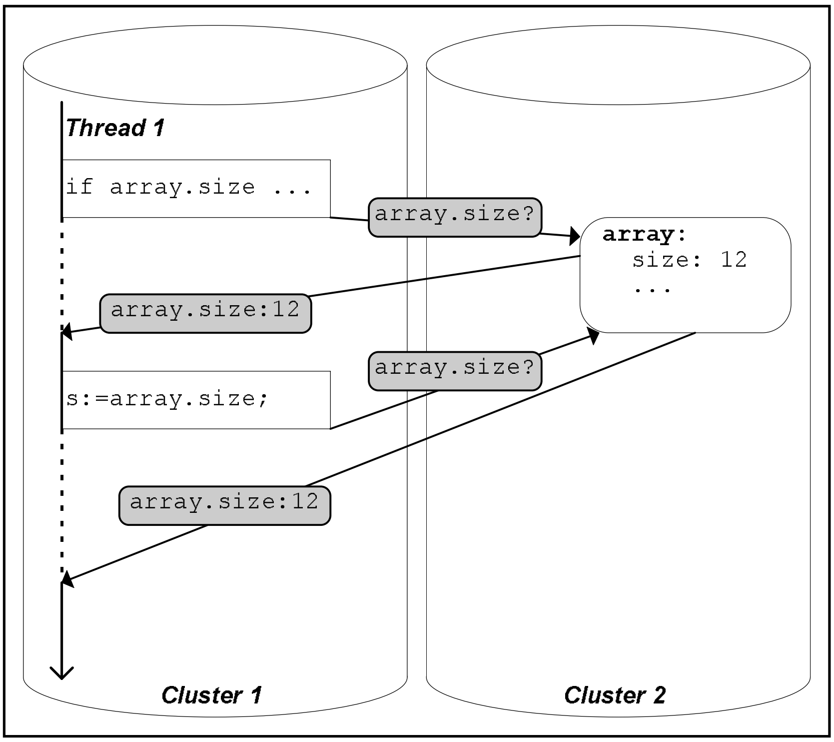

Figure 6-1: Reading an attribute from a far object

If the runtime is able to store the values of attributes or complete objects or even parts of an array, it would be possible to decrease the number of messages sent and speed up the program this way as shown in figure 6-2. Such a cache needs the following properties:

- **preserve memory model semantics**: the runtime has to make sure that the value returned for a thread that needs data from a far cluster is at least as recent as the last import executed by that thread.
- **simple implementation**: if the time overhead added by the cache is too high the time won by sending less messages around would be lost by the access time and administration of the cache.
- **consistency**: if a thread needs the same data twice the system must make sure that the thread gets either the same value or a newer one when it requests the same data again.
- **correct flushing**: the system has to make sure that the cache is correctly flushed to avoid the reuse of obsolete values:

- whenever a thread executes an import the cache has to be flushed, at least with respect to all reads from this thread (other threads may still use older values)
- whenever an object is destroyed (either by the garbage collector or by an explicit destroy) all cached attributes of this object on all clusters have to be flushed. Obviously, the garbage collector can do this without a problem. The `SYS::destroy()` provided by pSather however is a real problem, especially if there is no garbage collector at all (see figure 6-3 for an example). The following solutions exist:
  - The `SYS::destroy()` is ignored whenever the cache is on,
  - whenever a `SYS::destroy()` is executed, all caches on all systems are flushed,
  - all `SYS::destroy()` calls are queued and executed in regular intervals (if the garbage collector is running, they may be executed during garbage collection).

  All three options are undesirable for a program that depends on `SYS:destroy()` to limit its memory consumption. Such programs should probably not use the caching mechanism at all.

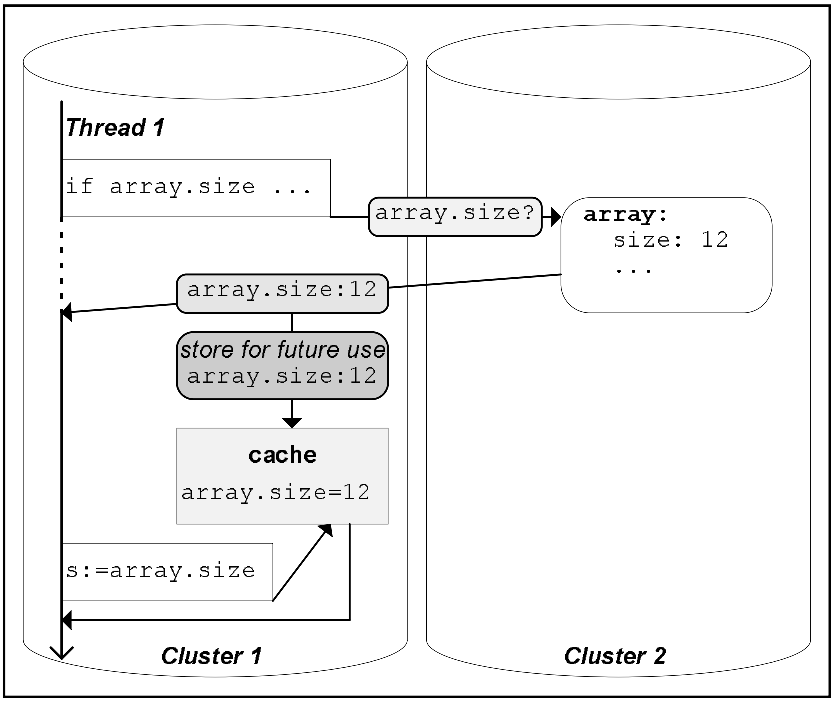

Figure 6-2: Attribute cache: the second access of `array.size` is resolved locally

The current pSather runtime implements an attribute and array element cache, that is, it caches individual attributes and array elements of reference objects with a size of at most 16 bytes, the size of one cache slot. This minimum size was chosen such that all immutable classes of the Sather and pSather library are cached. Larger objects are not cached at all. Each attribute fetched from another cluster will be stored in one cache slot, which is selected depending on its address by masking out the highest bits.

```
class X is
   attr a:INT;
end;

class Y is
   attr b:INT;
end;
```

thread 1 on cluster 1 executes:

```
   x:X:=get_some_x();
   i::=x.a; -- assume where(x)=2
   SYS::destroy(x);


   -- returns the newly created Y. This
   -- can happen even if this thread did
   -- non execute an import.
   y::=get_some_y();
   -- as the cache does not know that y is
   -- another object than x, even though it
   -- has the same address, it returns the
   -- cached value x.a
   j::=y.a
```

thread 2 on cluster 2 executes:

```
-- by chance the new Y uses the same
-- memory position as the freed X
store_some_y(#Y);
```

Figure 6-3: Cache and destroy inconsistency

To ensure that each thread gets the correct value when accessing a far object and to decide when an object has to be refetched, the following protocol is used:

- **Import Counter** `I`: each cluster maintains an import counter, which is incremented each time one of the threads running on this cluster executes an import.
- **Thread Import Counter** I_t: each thread knows the value the import counter `I` had when the thread executed its last import.
- **Cache Slot Import Counter** I_c: each cache slot knows the value the import counter `I` had when the object stored in it was fetched.

Whenever a thread needs an attribute or array element of an object stored on a far cluster, it will first check if the object stored in this attribute is available in the cache. If this is the case, and $I_t \leq I_c$ the cached object is used, otherwise the object is re-fetched from the far cluster. The structure of the cache and the code used to access it are shown in figure 6-4 and the speed of a remote read versus a cached read is in table 6-2. Also note that writes to far objects have to be handled differently when the cache is used, as the cache has to be updated whenever an object is written to an attribute which is cached.

```
/* the structure of the cache */
 struct CACHE_STRUCT {
      lock_t lock;  /* different threads may access the same cache slot at */
                    /* the same time, to serialize them, each cache slot uses */
                    /* its own mutex lock */
      void *addr;   /* memory address of the cached attribute */
      unsigned long Ic;       /* import counter */
      char value[SLOT_SIZE]; /* value cached */
 };

/* pseudo code to access the cache looks as follows (C syntax): */
/* (assumes that addr is the address of the attribute being read) */
   slot=CACHE_HASH(addr);
   mutex_lock(cache[slot].lock);
   if(cache[slot].addr!=addr || cache[slot].ic<thread_it) {
      cache[slot].Ic=import_counter;[38]
      far_read(cache[slot].value,addr);
   }
   ret=cache[slot].value;
   mutex_unlock(cache[slot].lock);
   return ret;
```

Figure 6-4: Cache structure and access

	Speed	Data Bandwidth	Delay
standard read	34’200 Integers / sec	137 KB / sec[39]	29.2 μsec / Integer
cached read	510’000 Integers / sec	2 MB / sec	1.9 μsec / Integer

Table 6-2: Reading remote integers on a Meiko CS-2

The time won by using a cache depends obviously on the network used, as the win for a cache hit is much higher for slow networks. Table 6-3 shows the win for using a cache on a Meiko CS-2 for three different distributed programs.

[38] Note that we could unlock the cache slot while executing the far read, however, then it would be possible for several threads to start a far read of the same attribute at the same time, which would not break the system, but would waste system resources.

[39] The bandwidth here seems to be very small. However, every read involves two messages, and transfers just 4 Bytes. As the best round-trip time on a Meiko CS-2 is about 20 μsec, the highest possible bandwidth is about 200 KB / sec.

Program	Clusters	Caching			Caching including all optimizations		
		Hits	%[40]	Speedup	Hits	%	Speedup
distributed Queen	2	4982	50%	1.01	1819	18%	1.01
(boardsize: 14)	4	10397	54%	1.01	9237	54%	1.01
	8	25454	55%	1.04	23535	54%	1.02
	16	103196	56%	1.12	63759	55%	1.08
Distributed	2	39312	69%	1.00	39372	69%	1.00
Differential Evolution	4	54823	64%	1.00	54412	63%	1.03
	8	61059	60%	1.03	60311	59%	1.02
	16	62577	55%	1.19	62690	55%	1.31
Distributed Photo	2	30678	83%	1.00	20463	83%	1.01
retouching	4	61331	83%	1.05	40891	83%	1.04
(2048x2048 pixels	8	122724	83%	1.12	81811	83%	1.13
3x3 filter)	16	183685	82%	1.20	122481	82%	1.36

Table 6-3: Speedup for -O_cache (uses 1024 cache slots)

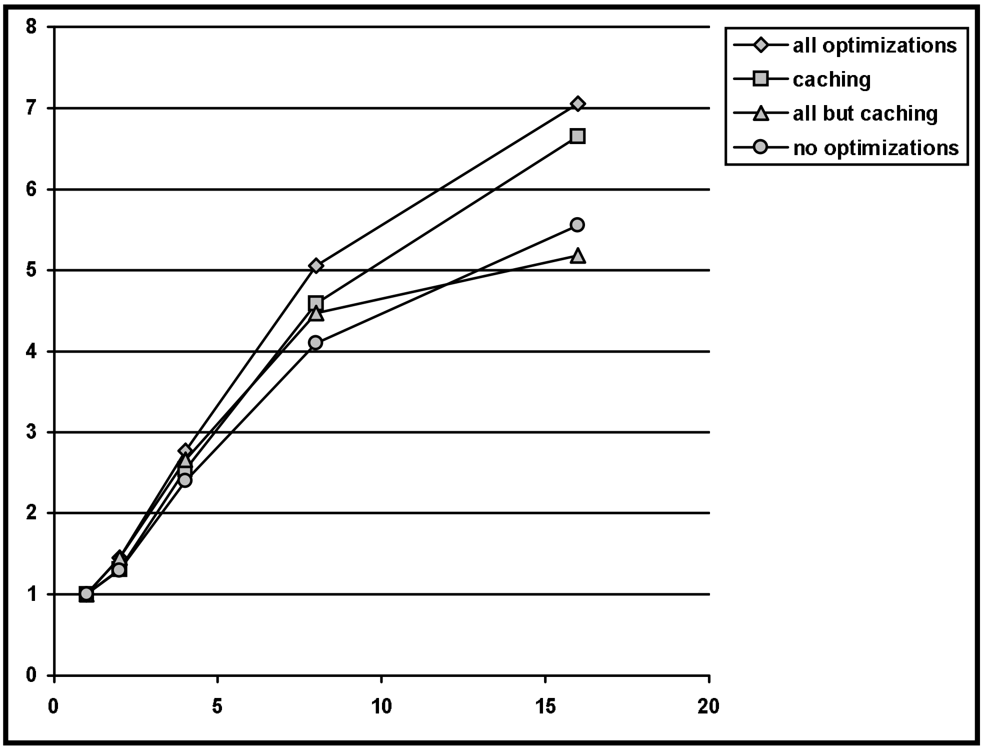

Figure 6-5: Speedup of the photo retouching program when using the cache

---

[40] Percentage of all far reads that were resolved locally.

6.3 Prefetching

Whenever a thread needs some information from another cluster, it will send out the request and wait for the response doing nothing. If, however, the thread could foresee that it needs some information, it could send out the request before it needs the answer, and hopefully the answer is back in time so that unnecessary waiting time can be avoided as shown in figure 6-6, which uses prefetching to get the size of the array before it is actually used in the if statement. Note that not only it is possible to avoid waiting, but by combining prefetch requests to the same cluster, it is also possible to combine messages and save some additional time[41].

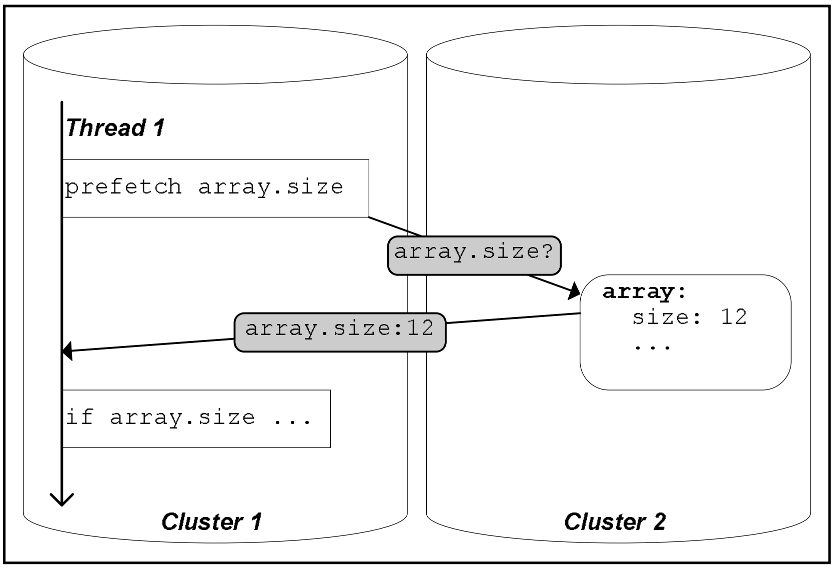

Figure 6-6: Prefetching

Several important points have to be considered when using prefetching:

- **speculative prefetching**: a thread can try to prefetch data it may eventually need. This technique can be useful if the bandwidth of the network is not already used up with more important messages and the remote cluster has either no other work to do anyway or the time to answer a prefetch message is negligible (this can happen for example if the network interface is able to read the main memory without interrupting the main processors). The current pSather compiler has two different options for speculative prefetching:
  `-O_loop_prefetch`: prefetches value before starting a loop, even if they are not used because the loop ends during the first iteration.
  `-O_specul_prefetch`: prefetches any value that could be possibly used.
- **overhead**: prefetching adds at least some small overhead to the program, as for each prefetched value some information has to be stored so that the program

[41] As the size of an active message is limited, this is not an option for the current pSather compiler.

knows if the answer already arrived. Without prefetching the same memory positions can be used for each remote memory read resulting in better cache usage. Therefore the compiler has to make sure that the prefetch request is sent out early enough to compensate for the overhead. As shown in table 6-2, the time until the prefetch request comes back is a little less than 30μsec, and therefore the prefetch has to be sent out ideally 30μsec before the value is used.

Program	Cl.	prefetching only				prefetching including all parallel optimizations			
		prefetch	%[42]	no waits[43]	speedup	prefetch	%	no waits	speedup
distributed Queen	2	496	5%	86%	0.98	481	10%	93%	0.93
(boardsize: 14)	4	739	4%	85%	0.99	719	8%	95%	0.93
GCC optimized	8	3358	3%	88%	0.95	1547	7%	98%	0.90
	16	4042	2%	92%	0.97	4944	6%	0%	0.85
distributed Queen	2	529	5%	87%	1.01	522	10%	91%	1.01
(boardsize: 14)	4	767	4%	88%	1.01	763	8%	95%	1.00
non GCC optimized	8	1612	4%	89%	1.01	1635	7%	99%	1.03
	16	4968	3%	93%	1.02	6325	6%	100%	1.04

Table 6-4: Speedup for -O_prefetch

The first results were quite disappointing, as prefetching did not seem to have any effect, even worse, it slowed the program down as shown in the first four rows in table 6-2. After some investigations it turned out that the GCC optimizer was not able to do a decent job with the prefetch macros. The last four rows in table 6-2 show that without the GCC optimizer prefetching does speed up the code slightly.

However, there are several more questions raised by those numbers. The slowdown measured in the first case is as bad as 17% for a program that runs for about 60 seconds, which seems to be much more than the effect of a bad optimizer. The speedup of 3.7% is also higher than expected, as a far read costs about 30μsec and there were 6325 prefetches, so we can expect to win at most 1.9 seconds. Those 3.7% however represents 6 seconds. Unfortunately, parallel programs are non- deterministic, and a likely problem here is network contention. The generated C code has the property in this case that many prefetches are before far reads, and the AM interface on the Meiko cannot send more than one message at a time. Without the GCC optimizer, it is likely that the prefetch message is sent before the far read message has to get out, with the effect that the prefetch is actually a win. In the optimized version however, it may be that the prefetch message is still in the message buffer when the far read message should be sent, and the thread has to wait.

A similar problem can arise on the receiver side of an active message. If, in the optimized case, it receives several messages at the same time, they will be served in order. However, if a far read arrives after some prefetch messages, it will be delayed far more than if there were no prefetch messages. To solve this problem it should be possible to add a priority to messages. Prefetch messages would then get a lower priority and only served after far read messages. This is not possible with the current im-

[42] Ratio between the number of ordinary reads and prefetches.

[43] Ratio of the prefetch statement that returned their value before it was needed.

plementations of the Active Message library, and to do it efficiently, it would have to be supported by the message interface directly.

Hence, prefetching memory does not seem to be a good idea on the Meiko and with the benchmark programs we used (the photo filter program and differential evolution do not benefit at all from prefetches). If more and larger pSather applications are available, prefetching has to be reevaluated.

6.4 Post Writing

Whenever a thread changes the value of an attribute of an object stored on a remote cluster, it has to notify the remote cluster about this at latest at the next export. The simplest way to do this is to immediately send out the write request and wait for the acknowledge of the remote server. This ensures that at the next export all values are already exported and, as the remote clusters got them in the correct order, they know the correct value of each attribute.

However, sending out the writer request and only waiting for it at the next export reduces the necessary wait times as shown in figure 6-7. The runtime has to make sure however, that either the writer messages are processed by the remote cluster in the same order they are sent, or that for each attribute changes between two exports, only the last update is sent to the remote cluster.

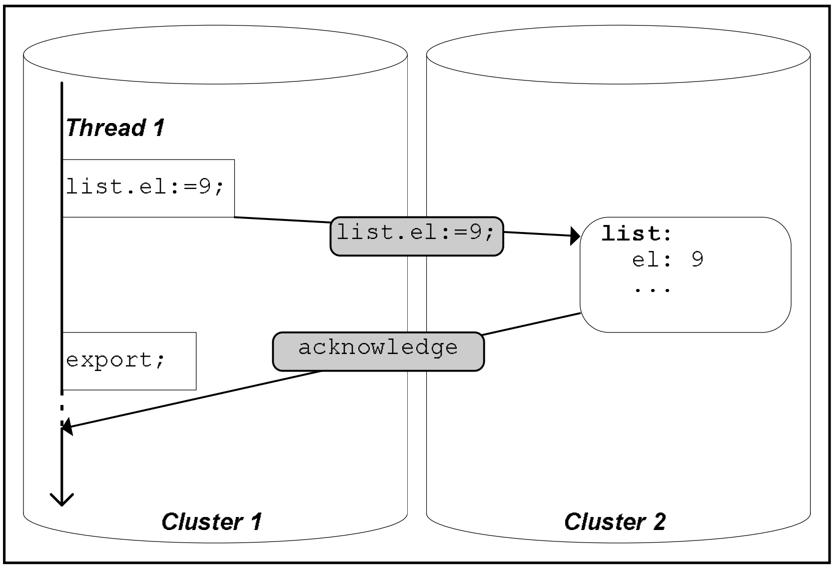

Figure 6-7: Post writing

If the compiler is able to see that several attributes stored on the same cluster are modified together, it can combine the write messages and send the updated information in one message instead of in several ones[44].

[44] Due to the limited size of an active message this is not implemented in the current compiler and runtime.

Table 6-5 shows the result for two of the distributed programs. (The distributed photo retouching program is not included, as it does not write any information to another cluster and therefore does not benefit at all from this optimization).

Program	Clusters	Post Writing			Post Writing including all optimizations		
		Writes	Waits	Speedup	Writes	Waits	Speedup
distributed Queen	2	131	126	1.01	127	124	1.01
(boardsize: 14)	4	175	160	1.00	169	163	1.00
	8	206	171	1.04	216	211	1.00
	16	287	212	1.01	290	285	1.02
Distributed	2	44	26	1.01	44	25	1.00
Differential Evolution	4	132	103	1.01	132	106	1.00
	8	308	260	1.02	308	269	1.01
	16	660	582	1.07	660	594	1.00

Table 6-5: Speedup for -O_post_write

6.5 Loop Fusion

One of the easiest ways to use pSather is the parloop construct, as it is often possible to replace standard loops with a parloop and get instant parallelism as shown in figure 6-8 where a function is called for each element of a list.

```
-- execute function blurp on each list element
loop
   list.elt!.blurp;
end;

-- the same function using the parloop construct
parloop
   elt::=list.elt!;
do
   elt.blurp;
end;
```

Figure 6-8: Using the parloop

However, creating a thread for each function call is excessive, and it would be better to create one thread per processor. Each thread will then execute the function for several of the list elements, while one thread will serve as work distributor. Note that if the compiler knew in advance how many elements have to be processed and it also knew that the time the function works on each element is the same it would be possible to distribute the workload at the beginning of the loop.

This optimization works only if the function inside the parloop cannot block, as otherwise the program could deadlock when all threads block, for example because a **sync** is executed inside the parloop as shown in figure 6-9.

```
-- example of a parloop for which it is not possible to have one thread
-- that executes more than one iteration
parloop
   elt::=list.elt!;
do
   elt.blurp;
   sync
   elt.blorp;
end;
```

Figure 6-9: Parloop with sync

The runtime system implements this optimization as follows:

At the beginning of the program each cluster creates as many work threads as it has processors. All work threads are put to sleep by blocking them on a MUTEX. A large circular buffer is used to store the work to be executed in parloops. Each parloop iteration is stored in one slot in this buffer.

Whenever a cluster enters a parloop, it wakes up all but one of the worker threads, and the thread that entered the parloop (called the loop thread) starts to fill the buffer with work (normally, a new thread would have been created to execute the body of a parloop). If the buffer is full the loop thread wakes up the last of the worker threads and waits until the buffer is nearly empty, at which point one worker thread will go to sleep again while the rest of the work is stored in the buffer by the loop thread. If all work has been stored in the buffer, the loop thread will wakeup the last worker thread again and wait until all work is done[45]. The resulting speedups for an empty parloop and the matrix multiplication program are shown in table 6-6.

	non optimized	optimized
empty parloop, time per iteration	1670 μsec	30 μsec
Matrix Multiplication, 300x300	3.8 sec	3.6 sec

Table 6-6: Speedup for the parloop optimization

[45] The actual system is a little bit more complex, as it allows another parloop to start to create work before the first one is finished to maximize throughput.

6.6 Specialization for Locality

Each time a program reads some attributes of an object, the runtime has to check if this object resides on the current cluster or on some remote cluster. As object oriented languages tend to use a lot of attribute accesses, the code executed by the system is cluttered with many of those checks. Although such a check needs to control just one bit of the pointer in question in the current pSather implementation on the Sparc architecture, these checks add up to be a considerable factor.

However, after the system checks the object stored in a local variable or an attribute it should not check it again unless this variable or attribute may suddenly point to another object. Therefore it is possible for the compiler to eliminate all subsequent checks of a local variable or attribute by generating for the rest of the functions two branches, one when the object is stored remotely and one when it is stored locally. pSather even offers a convenient construct for the programmer to solve this problem by hand, namely the `with near` statement.

One problem remains though, as the information about which attributes are local will not get through a function call, which means that each function will start to repeat those checks anyway. This problem can be solved by generating for one pSather function two or more C functions, which assume that `self` and/or some of the arguments passed are local as shown in figure 6-10.

The current compiler supports three different levels of local optimizations:

- **`-O_local`**: the compiler analyses all locals to see if they can contain only pointers to local objects and, if this is the case, uses the faster array and attribute access routines.
- **`-O_local_call`**: for each function, the compiler decides if it is worthwhile to create several versions, depending on whether some of the arguments are local or not. However, the decision to call an optimized version of the function is made statically at compile time.
- **`-O_local_call_dynamic`**: this option has the same effect as `-O_local_call`, but the compiler inserts code to test at runtime if an optimized version of the function can be called. Figure 6-11 shows how such code looks like.

Two additional options allow the user to fine tune these optimizations:

- **-O_local_call_access**: sets the number of attribute access that have to be performed in a call before an optimized version of the function is created. The default value is 1.
- **-O_local_call_dynamic_accss**: sets the number of attribute access that have to be performed in a call so that the compiler inserts a check before calling the function to see if it an optimized version of the function can be called. The default value is 3.

Program	Clusters	local	local call	local call dynamic
distributed Queen (boardsize: 14)	2	1.00	1.48	1.56
	4	0.98	1.42	1.51
	8	1.04	1.45	1.47
	16	1.04	1.35	1.40
Distributed Differential Evolution	2	1.01	1.27	1.29
	4	1.00	1.27	1.29
	8	1.01	1.24	1.25
	16	1.17	1.13	1.18
Distributed Photo retouching (2048x2048 pixels, 3x3 filter)	2	0.99	1.08	1.10
	4	0.99	1.14	1.11
	8	1.03	1.12	1.05
	16	0.98	0.99	1.04

Table 6-7: Speedup for the three different local optimizations

```
class A_STACK{T} is
   private include AREF{T};
   top:INT; -- index of the top element
   -- checks if a stack implemented with an array is full
   is_full:BOOL is
      return asize=top;
   end;
end;

/* the generated C code, once when we know that self is local,  */
BOOL A_STACKT_is_full_local(A_STACKT self) {
   return self->asize==self->top;
}

/* and once for the case when self is far  */
BOOL A_STACKT_is_full(A_STACKT self) {
   int tmp1,tmp2;
   if(NEAR(self)) tmp1=self->asize;
   else REMOTE_READ(tmp1,self,asize);[46]
   if(NEAR(self)) tmp2=self->top;
   else REMOTE_READ(tmp2,self,top);
   return tmp1==tmp2;
}
```

Figure 6-10: Creating local and remote functions

[46] REMOTE_READ is a C macro that sends a message to the remote cluster and stores the result at the address given.

```
/* we call either A_STACK_is_full_local or          */
/* A_STACK_is_full by checking self before the call  */

   if(NEAR(stack)) res=A_STACK_is_full_local(stack);
   else res=A_STACK_is_full(stack);
```

Figure 6-11: Calling optimized functions at runtime

6.7 Remote Execution

Each function executed by a thread normally runs on the same cluster as the thread that called it, unless the programmer specifically asked to execute it on a remote cluster with the `@` command. The local cluster may however not be the best place to execute a function. Take for example the list insertion function defined in figure 6-12. If this function is executed remotely the system has to fetch the values of the three attributes `head`, `tail` and `size` and rewrite `size` and `head` or `tail`. If, on the other hand, the code is executed on the cluster where the list resides, the only remote access could be the write access to `tail.next`. Additionally, we can use the function generated by the “Local Execution” optimization described above and use the list insertion function that knows that `self` is local.

```
class LIST{T} is
   insert(t:T) is
      l::=#LIST_ELEMENT{T}(t);
      -- make sure that only one thread inserts an element at a time
      lock mutex then
         if size=0 then
            head:=l;
         else
            tail.next:=l;
         end;
         tail:=l;
         size:=size+1;
      end;
   end;
end;
```

Figure 6-12: Inserting a list element

There is however the same problem regarding compiler-inserted remote execution as with the remote execution used by the programmer. In both cases we change values on a remote cluster which may well be used on the local cluster later on, yet we have no import to get those values. Additionally, the remote cluster may use values changed by the local cluster without that the local cluster made an export. An extreme version of the problem is shown in figure 6-13 where the value if the attribute `i` is changed on the local cluster, changed again on a remote cluster and reused on the local one. Without additional support by the compiler, the runtime will not guarantee that this program works as expected.

```
class INCREMENT is
   attr i:INT;
   create:SAME is return new; end;
   incr is i:=i+1; end;
end;

class MAIN is
   main is
      x::=#INCREMENT@(here+1) -- create a remote object
      x.i:=1;
      x.incr@1;
      #OUT+x.i+"\n"; -- x.i is supposed to have the value 2
   end;
end;
```

Figure 6-13: Remote execution

There are several options to deal with this problem:

- **no remote execution**: not a very satisfactory solution.
- **insert `import/export`**: by inserting an `export` before the remote execution and an `import` after, and by enclosing the remote execution call with an `import` at the beginning and an `export` at the end we get the desired functionality. Note that those exports will not hinder other optimizations at all, as they are only relevant for the current thread. Other optimizations, with the exception of caching and post writing, can treat a remote function call the same way as a local function call.
- **insert adapted import/export**: By using the information available for all functions as described in table 5-1 on page 63 the compiler can check which variables have to be exported and imported and only work with those variables. The current pSather runtime would then only wait for the exports of these specific variables and invalidate the cache slots for all variables that have to be imported.

An additional probem arises with functions that create new objects. As objects are usually created on the same cluster on which the create function is executed it would be possible to see the effect of the remote execution optimization by checking the lcoation of newly created objects. Such problems are avoided by not executing any function remotly that creates objects, unless the user specifically allows it by using the option `-O_remote_call_create`.

The current runtime uses the standard import/export functions, not the adapted ones. None of the standard benchmark programs benefit from this optimization, for two reasons:

- Neither the Photo Retouching nor the Differential Evolution program makes use of remote execution at all.
- The Queen program does use it, but it also tries to maximize the use of all processors, so the thread created on a remote cluster to execute the function will compete with the other threads for the available processors and the result is no speedup.

Table 6-8 shows the speed difference between executing `to_identity` and `str` on a 30x30 matrix remotely when using the remote execution option or not. As the `str` function creates a string, it is only executed remotely if the option `-O_remote_call_create` is used.

	normal execution	with remote call optimization
`to_identity`	28 msec.	5 msec.
`str`	190 msec.	166 msec.

Table 6-8: Speedup for the remote execution optimization

7. Synchronization Optimizations

7.1 Fast Lock Interface

In many cases the lock statement is used to ensure that only one thread executes some statements at a time as shown in figure 7-1.

```
lock mutex then
    -- critical section
end;
```

Figure 7-1: Critical section

Unfortunately, the centralized lock manager requires three messages to implement this, namely one sent by the thread entering the lock statement, the answer sent by the lock manager and the unlock message at the end of the lock. If we could implement a protocol that allows a thread to acquire such a lock without bothering the lock manager, we would win a considerable amount of time. However, we have to make sure that the fairness requirements of the pSather specifications are still met.

We solved this problem by adding three new functions to the $LOCK interface (see page 32). Those functions are only used when exactly one lock has to be acquired. Each of these function can either solve the problem (i.e. acquire the lock) or declare itself as not being competent to do so. In the later case the lock manager will be called[47].

function	return values[48]	description
`fast_lock(THREAD_ID):BOOL`	**false**: call the lock manager true: the lock has been acquired	called if no else part exists
`fast_try(THREAD_ID):INT`	-1: execute the else part **0**: call the lock manager 1: the lock has been acquired	called if the else part exists
`fast_unlock(THREAD_ID):BOOL`	false: continue true: notify the lock manager about the change	only called if `fast_lock` or `fast_try` have succeeded

Table 7-1: The fast lock interface

[47] This is the default return value for those functions. Synchronization objects that chose to not use this interface should return those values.

[48] The bold values are the ones that should be returned by all lock objects that do not want to use the additional complexity of the fast lock interface.

Note that `fast_lock` and `fast_try` have to be thread safe, as both functions can be called by different threads at the same time for the same lock object. `fast_unlock` however does not need to be thread safe, unless the lock object can be acquired by more than one thread at a time, as it is the case for the reader part of a reader/writer object.

The current pSather library has only one lock object that uses this interface, namely a `MUTEX` class which works as follows to preserve the pSather requirements: at all times, the `MUTEX` is either managed by itself or by the lock manager. The lock manager will take over whenever

- the `MUTEX` is used in a lock statement with more than one lock,
- or the `MUTEX` is currently locked and another thread waits for it. In this case the lock manager waits until the first thread releases the lock before taking over.

The lock manager will withdraw its responsibility as soon as no thread waits for this lock anymore. As long as the lock manager is responsible for the MUTEX the fast lock functions will never try to lock the lock on their own, but will delegate the work to the lock manager too. This ensures that the semantics defined for the lock, namely fairness and starvation freeness can still be guaranteed.

The pseudocode for the three functions in the class MUTEX is shown in figure 7-2. Not shown there is the necessity to use low level system locks to synchronize the different threads that could call those functions simultaneously. The complete code is available in the pSather library available at `http://www.icsi.berkeley.edu/~sather`.

Cluster on which the lock object resides			*MUTEX*	*SMUTEX*	*FMUTEX*	*FMUTEX*
Cluster where the thread executes						*_SPIN*
Locking unlocked MUTEX	**0**	**0**	40	2210	30	10
	0	**1**	3440	8560	3020	2440
	1	**0**	2960	3480	3550	4390
	1	**1**	90	17240	20	20
Locking locked MUTEX	**0**	**0**	180	2480	20	20
(and failing)	**0**	**1**	2930	8940	2000	2020
	1	**0**	2490	4990	2780	2010
	1	**1**	90	18940	20	10

Table 7-2: Speed differences for different `MUTEX` like classes (in μsec.)

This new interface allowed us to implement an even faster version of the standard `MUTEX`, but this new lock does not guarantee fairness nor can it be used together with other locks in the same lock statement. Deadlock detection does not work either. This class, called `FMUTEX` (for `F`ast `MUTEX`) will never call the lock manager, but instead uses low level system locks. Another similar class called `FMUTEX_SPIN` uses low level spin locks and can be used for very short critical sections. Obviously it should not be used on single processor systems, as it essentially implements a busy wait (it will, at least in the current Solaris implementation, relinquish the processor after some time). Figure 7-3 shows the implementation of the `FMUTEX` class and table 7-2 the speeds of acquiring a standard `MUTEX` that does use the fast lock interface versus the `FMUTEX`, the `FMUTEX_SPIN` and a `SMUTEX` that does not make use of the fast lock interface at all. This table shows that all locks that do not use the fast lock interface should reside on the same cluster as the lock manager and that a distributed lock manager would be a win for all locks used only on one cluster.

```
-- if the lock is available, we take it, otherwise we have to call the lock
-- manager, as more than one thread competes for this lock.
fast_lock(tid:THREAD_ID):BOOL is
   return fast_try(tid)=1;
end;

-- Tries to lock the MUTEX, return
-- -1 if it failed and the else part should be executed
--  0 if the lock manager should be called
--  1 if it was successful
fast_try(tid:THREAD_ID):INT is
   -- if the lock is already locked by tid, increment the lock counter
   -- and return 1
   if locked_by=tid then
      locked:=locked+1;
      return 1;
   end;
   -- if the lock is locked by another thread, we have to execute the else part.
   if locked>0 then return -1; end;
   if lock_manager is in charge then return 0; end;
   locked_by:=tid;
   locked:=locked+1;
   return 1;
end;

-- only one thread can call unlock of a MUTEX at any time (anything else
-- would be a fatal pSather error). fast_unlock returns true
-- if the lock manager has to be notified about the change
fast_unlock(tid:THREAD_ID):BOOL is
   locked:=locked-1;
   if locked=0 and lock_manager has to take over then
      return true;
   end;
   return false;
end;
```

Figure 7-2: Fast lock interface for the class MUTEX

```
class FMUTEX < $LOCK is
   private attr locked_by:THREAD_ID;
   private attr locked:INT;
   private attr ilck:LL_LOCK; -- low level system lock

   fast_lock(tid:THREAD_ID):BOOL is
      -- if the thread already owns the lock, this function returns immediately
      -- after counting the number of times this lock has been locked by the
      -- same thread.
      if locked_by=tid then
         locked:=locked+1;
         return true;
      end;
      ilck.lck; -- wait until we get the low level lock
      locked_by:=tid;
      locked:=1;
      return true;
   end;

   -- Trys to lock the FMUTEX, return -1 if it failed and the else part should be
   -- executed, 1 if it was successful
   fast_try(tid:THREAD_ID):INT is
      if locked_by=tid then
         locked:=locked+1;
         return 1;
      end;
      if ilck.try then
         locked_by:=tid;
         locked:=locked+1;
         return 1;
      end;
      return -1;
   end;

   -- only one thread can call unlock of a FMUTEX at any time (anything else
   -- would be a fatal pSather error).
   fast_unlock(tid:THREAD_ID):BOOL pre locked_by=tid is
      locked:=locked-1;
      if locked=0 then
         locked_by:=THREAD_ID::nil;
         ilck.unlck;
      end;
      return false;
   end;
```

Figure 7-3: Class FMUTEX

Program	Clusters	Fast Lock Interface used	in %
distributed Queen (boardsize: 14)	2	432	88.5%
	4	440	88.7%
	8	446	87.1%
	16	437	80.3%
Distributed Differential Evolution	2	21	100%
	4	41	100%
	8	81	100%
	16	161	100%
Distributed Photo retouching (2048x2048 pixels, 3x3 filter)	2	8	100%
	4	8	80%
	8	14	100%
	16	22	100%

Table 7-3: Usage of the Fast Lock Interface

7.2 Exception Stack and Locks

As shown in chapter **3.6** the exception stack in pSather is used in each lock statement and for each loop to ensure that locks acquired are correctly released if an exception is thrown, a loop quits or a function returns. Unfortunately it is quite expensive to push an element onto the exception stack. In our current implementation for Solaris it takes about 5.5 microseconds on a Sparc 10, 66 MHz.

However, it is not necessary to push each lock and each loop statement onto the exception stack, as the stack is only used when an exception is thrown[49]. A lock statement has to be put onto the exception stack if

- an exception may be called while the lock is held,
- a yield is called inside the lock,
- it is inside a loop and an iterator is called while the lock is held,
- there is a syntactically nested lock or protect statement (this is a requirement of the current runtime and not strictly necessary. As such a nesting does not occur very frequently, we did not change the runtime yet).

In all other cases we know for sure that the thread will pass through the end statement of the lock and therefore we can unlock all locks at this point.

A loop statement has to add an element to the exception stack if

- an iterator is called that has to add an element to the exception stack during a yield (this happens for yields inside locks and yields inside loops that have an element on the exception stack).

[49] Actually, the current compiler uses the exception stack also when a loop quits or when a function returns to unlock all locks acquired inside the function or loop.

The dataflow mechanism (see chapter 5 on page 61) of the ICSI pSather compiler is able to detect most of the cases where the exception stack is unnecessary. Table 7-4 shows the time difference of a lock that needs an entry on the exception stack versus one that does not, and between an iterator call that needs an entry and one that does not on a Sparc 10, 66 MHz.

	non optimized	optimized
`locking`	4.7 μsec	2.2 μsec
`iterator call`	13.0 μsec	9.9 μsec

Table 7-4: Speedup for the exception stack optimizations

8. Conclusion and Future work

8.1 Results

This thesis has shown that parallel, object oriented programs can, in fact, speed up code running on a distributed system without asking the programmer to annotate functions and/or variables, for example by defining which variables may or may not hold references to objects on another cluster. This greatly improves code reusability and readability.

It also presents a very flexible synchronization scheme with an interface that allows the user to define new synchronization objects. Those synchronization objects allow the creation and implementation of the standard synchronization mechanisms and of new ones, like locks, that can only be locked if some condition hold. Synchronization objects can be freely mixed inside the disjunctive lock as shown in several examples.

There are a number of important design decisions however that are needed in order to implement the necessary optimizations:

- **memory model**: an overly restrictive memory model prevents some of the optimizations and makes the system unnecessary slow. By choosing a more relaxed memory model and adding support from the compiler the programmer will nearly never need to worry about it, but it allows the implementation of several important optimizations. Tables 8-1 and 8-2 and figures 8-1 and 8-2 show how much can be gained by using those optimizations.
- **target architecture**: even though the compiler tries to reduce the number of messages sent around, a fast and reliable network is mandatory, as otherwise remote memory accesses are simply too slow to be useable. This is for example the case with TCP/IP implementations of pSather.
- **algorithms**: even though the compiler and runtime system try to hide the latency of remote memory accesses, the user still has to make sure that he uses local values as much as possible. An implementation of the naïve matrix multiplication algorithm for example will not work for a distributed dense matrix, as each node would need a copy of the complete matrix. However, in many cases the optimizations presented in this thesis are sufficient so that the naïve parallelisation yields the expected speedup. This is, for example, the case for the heat program, the n-queen program and the photo filter program.

Program[50]	Speedup for best combinations of all optimizations	only optimizations that work in all memory models
Heat	1.48	
Parallel Heat	1.31	1.23
Matrix Multiplication	7.14	
Parallel Matrix Multiplication	4.61	1.49
N-Queen	1.27	
Parallel N-Queen	1.28	1.12
Differential Evolution	1.29	
Parallel Differential Evolution	1.19	1.00
Photo Filter	1.28	
Photo Filter in Parallel	1.20	1.16
Compiler	1.27	
Compiler with Optimizations	1.27	

Table 8-1: Speedup for the serial and parallel programs

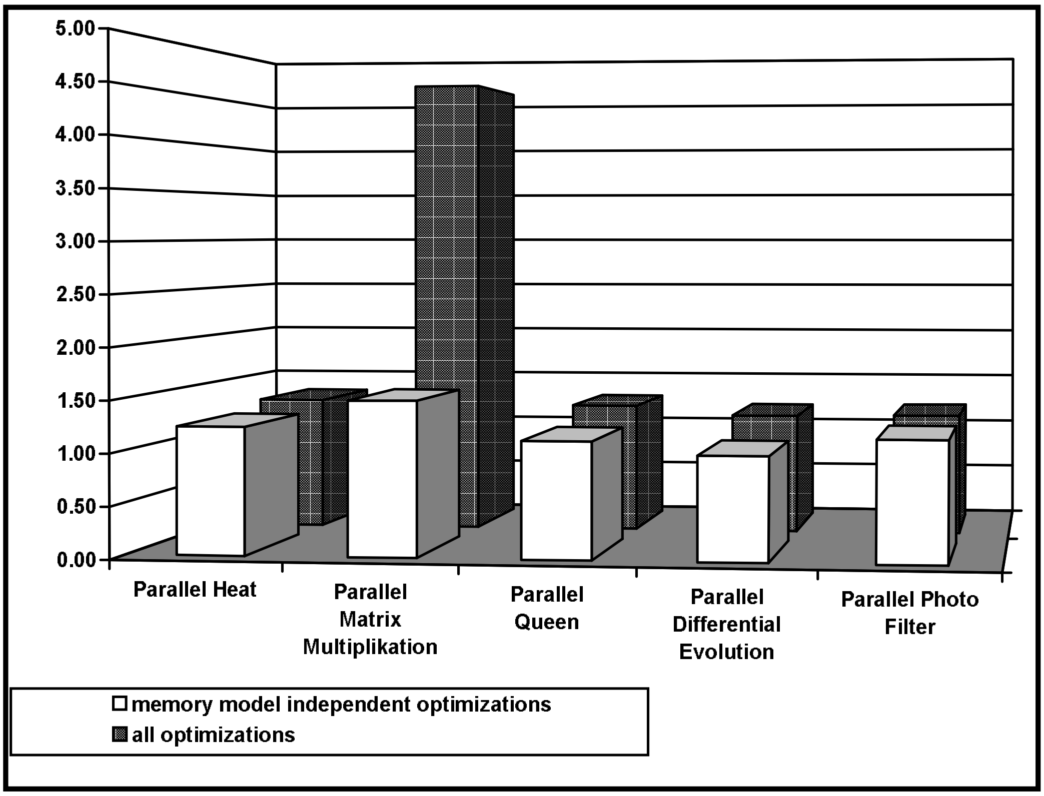

Figure 8-1: Speedup of Parallel Programs

[50] The programs used as benchmarks are described in Appendix C

Program	Clusters	Speed for best combinations of all optimizations (in seconds)	only optimizations that work in all memory models (in seconds)
distributed Queen (boardsize: 14)	1	143	
	2	148	151
	4	81	82
	8	47	48
	16	35	37
distributed Queen (boardsize: 15)	1	1256	
	2	936	939
	4	478	480
	8	243	246
	16	137	137
Distributed Differential Evolution	1	72	
	2	57	59
	4	32	58
	8	19	18
	16	15	18
Distributed Photo retouching (2048x2048 pixels, 3x3 filter)	1	106	
	2	81	128
	4	42	133
	8	22	115
	16	14	110

Table 8-2: Speedup for the distributed programs

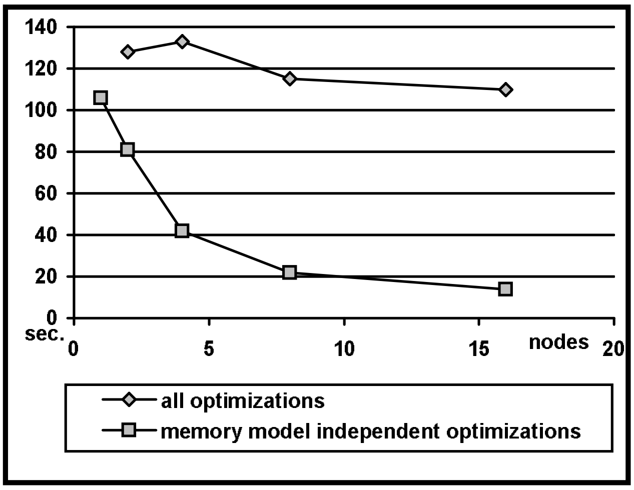

Figure 8-2: Speedup of the Distributed Photo Retouching Program

Even though the results are already very useful, there are several places where the compiler, the runtime and the pSather library could be improved.

8.2 Library Design

The design of parallel libraries was not within the scope of this thesis. Without efficient parallel libraries however pSather is not very useful, especially as we claim that many pSather users will not have to deal with the parallel constructs, as they use parallel libraries. Additionally, none of the container classes with the exception of the stack are thread safe yet. All those Sather classes will have to be changed such that they use the visitor/mutator protocol outlined in chapter 3.2 although the language specification may be changed so that the routines will be marked as visitors and mutators, rather than adding lock statements (see figure 3-8 on page 28).

Besides the container classes new parallel classes are also needed, like parallel and distributed trees, hash-tables and lists, that do allow, for example, to execute a routine on each node in parallel. Quad- and Oct-trees are another example of useful classes for n-body simulation programs, parallel graphs are useful for fluid and gas simulations, distributed vectors and matrices are yet another example of missing pieces in the pSather library.

It is not yet clear if additional synchronization classes are needed or if the currently available ones are sufficient. None of the currently implemented programs with the exception of the n-Queen (see chapter C.3.2) program were in need of other and more sophisticated synchronization objects. However, without creating new synchronization objects it is not possible to create data structures that block until some condition hold[51].

The design and implementation of the pSather library is ongoing research at the International Computer Science Institute and future releases of the Sather/pSather compiler are likely to include more and better libraries.

8.3 Synchronization

The current synchronization system with the centralized lock manager is already very useful. Three points may need some work though:

8.3.1 Dynamic Lock Statement

Currently, the number of synchronization objects inside a lock statement is fixed at runtime. It is not possible to create a list of synchronization objects and suspend a thread until all, some or one of them has been locked[52]. Several possible

[51] Actually, this claim is not quite correct. By using the available synchronization objects it is often possible to create those objects. However, as they use locks internally and cannot usually export them, it is not possible to combine those objects in lock statements as easily as it would be the case if they used their own synchronization objects.

[52] It is actually possible to create a list that can be locked as soon as all objects inside the list can be locked. However, with the current implementation and interface for synchronization objects the system

implementations have been discussed in the Sather development team, but none has been implemented yet, mainly because we did not find a suitable syntax yet.

An example where such a lock would be useful is the distributed queue used in the n-Queen program. Each cluster has its own list of work to do, and ideally we would like to lock any of those lists, preferably the local one to dequeue some work. Figure 8-3 shows how the code could look like.

```
loop
    -- lists is an array such that lists[n] returns a list local to cluster n.
    -- locking lists.any_not_empty will lock one of those lists that is not
    -- not empty, if several lists are not empty, the local one (lists[here])
    -- will be locked if possible.
    lock
    when lists.any_not_empty then
        work::=lists.locked_list.dequeue;
    when lists.all_empty, producers.no_threads then
        return; -- no more work to do
    end;
    -- execute work
end;
```

Figure 8-3: Dynamic lock statement

To implement such container locks and still guarantee fairness, an additional routine that returns all the primary locks used inside the container lock must be added to the `$LOCK` interface. For most locks it will just return `primary`, but container locks will need the function shown in figure 8-4.

```
class LIST_LOCK is
    contains:ARRAY{$LOCK} is
        -- size returns the number of locks used inside the list lock
        c::=#ARRAY{$LOCK}(size);
        -- elt! returns each lock in turn.
        loop c.set!(elt!.primary)); end;
        return c;
    end;
end;
```

Figure 8-4: Contains function for a list lock

will no longer guarantee fairness if one thread tries to lock the list, while others try to lock individual elements of the list. In this case the thread that locks the list may starve.

8.3.2 Distributed Lock Manager

The current implementation uses a centralized lock manager. This works fine unless the number of clusters is too large. A distributed lock manager, despite the fact that its implementation is more complex, would help to ensure the scalability of pSather.

```
-- try to lock any of the branches of the lock statement stored in lck
loop
   branch::=lck.branch!;
   branch_id::=0.up!;

   -- lock all hidden locks to ensure that no other thread believes one of
   -- the locks is locked just because this thread locked it and later found
   -- out that it was not needed.
   loop
      branch.elt!.lock_hidden_lock;
   end;

   -- try to lock all locks of a when branch.
   locked_all::=true;
   loop
      if ~branch.elt!.fast_lock(THREAD_ID::me) then
         locked_all:=false;
         break!;
      end;
   end;

   -- release the hidden lock.
   loop
      branch.elt!.unlock_hidden_lock;
   end;

   -- if successful, return the branch that has been locked.
   if locked_all then return branch_id; end;
end;

-- we could not lock a branch, return failure
return -1;
```

Figure 8-5: Extended fast lock interface

8.3.3 Better Fast Lock Interface

The current fast lock interface works only if there is just one lock in a lock statement, otherwise the fast lock interface is ignored and the lock manager takes over. By adding a hidden lock to any synchronization object, it would be possible to extend the fast lock interface to lock statements where several locks have to be locked. The algorithm to do this is outlined in figure 8-5. Note that now before any function gets called of the standard interface, the hidden lock has to be locked by the lock manager before the function `reservable()` can be executed and it has to be unlocked after either `reserve()` has been called or `reservable()` returned **`false`**.

8.4 Optimizations

The optimizations implemented in the current pSather compiler can already optimize a lot of code, there are two areas where it could still be improved. The dependency algorithm does not yet collect enough information to find all possible optimizations as shown in chapter 5.1, as it only collects information about which attributes of which class is read or written, but not for which argument it happens. Ideally, we should be able to get the same information as if the function call is inlined. This would help the compiler to avoid conservative decisions because of lack of information.

Additionally, several more parallel optimizations should be implemented (see chapter 2.4 for a list of other papers about optimizations), for example:

- **Message Piplining** (combining several messages): As active messages are by definition very small, this optimization is only possible if the underlying communication library is changed.
- **Array Optimizations**: parloops working on arrays can often be optimized to distribute the array data (or broadcast scalars) before creating the threads on each cluster. This avoids a bottleneck on the cluster that started the parloop when all threads try to read the same memory positions at the same time.
- **Combine Optimizations**: the currently implemented optimizations are unaware of each other, and hence act in isolation. Remote execution for example executes a function remotely even if the data needed is already in the cache such that the local execution would be faster in this case.
- **Adaptive import/export**: any import/export statement imports and exports the complete memory, even though it would often suffice to import or export part of the memory. The compiler should be able to tell which parts have to be imported/exported.
- **Semiconstant Attributes**: it is often the case that some attributes of an object are set during creation of the object, but do not change later. If such an attribute is in the cache, it should never be flushed, as it cannot change. Two examples are the size of an array, and the tag of an object (the tag is a number that defines the type of the object and is used for example in `typecase` statements and calls on abstract types).

8.5 Communication Library

pSather uses the active message library for communications between the different clusters. Those libraries are already quite fast on most architectures, some optimizations are nevertheless possible. As discussed in chapter 2.2.3 it is for example possible to use the communication co-processor available on some supercomputers to read and write memory without interrupting the main processor. pSather currently does not use such a library, but it would be quite interesting to see if such an optimized active message library would be an advantage for pSather.

8.6 Porting

The current pSather compiler has been ported to shared memory Solaris workstations, Solaris workstations connected through either Ethernet or Myrinet and the Meiko CS-2. Former pSather targets were also the Thinking Machines CM-5.

As the pSather runtime uses a standard message passing library and normal thread calls it should be fairly easy to port the language to other supercomputers. Whoever is interested in porting the compiler should contact the Sather team at the International Computer Science Institute at `http://www.icsi.berkeley.edu/~sather`.

Appendix A: Tour of the Language

This appendix is a short introduction to Sather and pSather Version 1.1. It is not meant to be a complete language reference manual or a tutorial, as the language specification and several tutorials written by different people are already available online on the World Wide Web at `http://www.icsi.berkeley.edu/~sather`.

The purpose of this chapter is to introduce some of the key features of the language such that the reader is able to understand the different examples shown in the text.

The language is described as follows in the WWW home page:

> *"Sather is an object oriented language designed to be simple, efficient, safe, flexible and non-proprietary. One way of placing it in the "space of languages" is to say that it aims to be as efficient as C, C++, or Fortran, as elegant as and safer than Eiffel, and support higher-order functions and iteration abstraction as well as Common Lisp, CLU or Scheme."*

Sather in a nutshell: Sather is an object oriented language that has parameterized classes, object-oriented dispatch, statically-checked strong (contravariant) typing, separate implementation and type inheritance, multiple inheritance, garbage collection, iteration abstraction, higher-order routines and iterators, exception handling, assertions, preconditions, post-conditions, and class invariants.

pSather in a nutshell: pSather is a parallel extension that addresses non-uniform-memory-access multiprocessor architectures but presents a shared memory model to the programmer. It extends serial Sather with threads, synchronization and data distribution. Unlike actor languages, multiple threads can execute in one object. It offers several synchronization mechanisms like futures, gates, mutex, reader/writer locks, barrier synchronization, rendezvous and a disjunctive lock statement.

The remaining 2 sections in this chapter discuss some of the special features of Sather and pSather that are necessary to understand the examples used in this text.

A.1 Sather

This section will introduce some of the Sather features regarding classes and iterators in Sather. For more information about other features check out [Stoutamire 96].

A.1.1 Classes

The Sather language distinguishes between different classes:

- **immutable classes**[53]: those classes are similar to C structs and C++ classes. Most of the basic types like integers (`INT`), characters (`CHAR`), floating point numbers (`FLT`) are immutable classes. Whenever an immutable object is assigned to another variable, a new copy of that object is created. They are usually passed by value when calling functions[54].
- **reference classes**: the main difference between an immutable class and a reference class is the fact that reference classes are always passed by reference, and if an object is assigned to a new variable, the variable will point to the same object. In C or C++ those classes are similar to classes that are always passed as pointers.
- **partial classes**: a partial class cannot be instantiated, but is meant to be included in other classes trough code inclusion.
- **abstract classes**[55]: abstract classes define new types. They have no implementation, but only an interface. concrete classes (immutable classes and reference classes) can only subtype from one or more abstract classes.

Figures A-1 and A-2 show a possible implementation of a fixed size list. Note how type relation and code inclusion are completely separated, and how features of the abstract class can be defined as routines (as for `clear`), but also as attributes (as for `size`). Attributes actually define two routines, a reader routine with the signature `attr:attr_type` and a writer routine `attr(i:attr_type)`. This allows one to replace an attribute in a class by two routines, a reader and a writer routine. See the discussion of syntactic sugar in the Sather reference manual for more details about this feature and the creation symbol '#'.

Note that Sather uses contravariant subtyping

```
-- define an abstract class for classes that implement a list with elements of type T
abstract class $LIST{T} is
   enqueue(t:T);
   dequeue:T;
   size:INT;
   clear;
end;
```

Figure A-1: An abstract class defining an interface for a list

---

[53] Those classes were called value classes in Sather 1.0 .

[54] Sather supports out and inout arguments.

[55] Abstract classes were called types in Sather 1.0.

```
-- define a concrete LIST implementation that uses a fixed size array
-- and subtypes from $LIST.
class A_LIST{T} < $LIST{T} is
   -- include the implementation of an array, but make all members private
   -- A_LIST does not subtype from array though.
   private include ARRAY{T};

   size:INT; -- attribute that defines the current size of the list

   -- create a new list of size s. new returns a new, uninitialized object of the class
   -- it which it is used. As this class includes an array, new takes one argument,
   -- the size of the array.
   create(s:INT):SAME is
      r::=new(s);
      r.size:=0;
      return r;
   end;

   -- enqueue a new value. It is a fatal error to call this routine if the list is full.
   -- The pre-condition check can be turned of by compiler options.
   enqueue(t:T) pre ~is_full is
      -- [size] is a special case of the standard array access sugar
      -- usually used in array[index]. In this case the expression is equivalent
      -- to self[size]
      [size]:=t;
      size:=size+1;
   end;

   dequeue:T pre ~is_empty is
      size:=size-1;
      return [size+1];
   end;

   is_full:INT is return size=asize; end;
   is_empty:INT is return size=0; end;

   -- empty the list by setting its size to 0
   clear is size:=0; end;
end;
```

Figure A-2: An implementation of a list class

A.1.2 Iterators

Iterators [Murer 93] are a key feature of the language and allow a class to define ways on how elements can be accessed. Several container classes in the standard Sather library, for example lists and arrays, define an `elt!` iterator that yields the elements in order, trees provide recursive iterators that yield nodes in in-, pre- and post-order.

An iterator can only be called from within a loop, although several iterators can be used in one loop. The loop shown in figure A-3 prints the elements of an array as a numbered list.

```
-- sets the i-th element of an array to i^2
loop
   array[0.up!]:=0.up!*0.up!;
end;

-- print all elements of an array, and number them. #OUT is used to create a new
-- OUT object. Everything added to it will be sent to the standard output of the
-- program.
loop
   #OUT+0.up!+". "+array.elt!+"\n";
end;
```

This program prints the following table (for an array of size 4):

```
0. 0
1. 1
2. 4
3. 9
```

Figure A-3: A small program that uses iterators

An iterator has the same format as a function, with the exception that it does not return, but rather **`yield`** a value and can **`quit`**. When an iterator yields a value, the enclosed loop continues, and on the next call of the iterator, the iterator will continue just after the **`yield`**, not at the beginning of the loop. If any of the iterators in a loop quits, the loop terminates immediately.

An iterator knows about two kinds of arguments, once arguments and the default (formerly called the hot) arguments. Hot arguments are evaluated each time the iterator is called, while once arguments are only evaluated the first time it is called and for each subsequent call the same value as the first time is passed into the iterator.

Three iterators are predefined in Sather, but, as shown in the figure A-4, could well be implemented by the user. Figure A-6 shows several simple iterators found in the INT class[56] and figure A-5 some of the array class (among others the iterators used in figure A-3).

```
    -- end the loop immediately
    break! is quit; end;

    -- continue the loop as long as the condition is true as in
    -- while!(~file.eof)
    while!(cond:BOOL) is
        loop
            if ~cond then quit; else yield; end;
        end;
    end;

    until!(cond:BOOL) is
        loop
            if cond then quit; else yield; end;
        end;
    end;
```

Figure A-4: The three predefined iterators

```
class ARRAY{T} is
    -- yield all array elements in order
    elt!:T is
        loop yield [asize.times!]; end;
    end;

    -- set successive elements of an array, as in
    -- loop array.set!(1); end;
    set!(t:T) is
        loop [asize.times!]:=t; yield; end;
    end;
end;
```

Figure A-5: Several useful iterators defined in the ARRAY class

[56] Note that in the current Sather library most of those iterators are defined as builtin iterators and not the way they are defined here.

```
class INT is
    ...
    -- yield all integers in order starting with self
    up!:INT is
        i::=self;
        loop
            yield i;
            i:=i+1;
        end;
    end;

    -- yield all integers from self up to and including n. the first value passed
    -- as an argument will define how far the iterator will go.
    upto!(once n:INT):INT is
        i::=self;
        loop while!(i<=n);
            yield i;
            i:=i+1;
        end;
    end;

    -- the following iterator is equivalent to 0.upto!(self-1)
    times!:INT is
        i::=0;
        loop while!(i<self);
            yield i;
            i:=i+1;
        end;
    end;
end;
```

Figure A-6: Several useful iterators defined in the INT class

For more information about iterators check out the online tutorial about iterators available at the Sather home page.

A.2 pSather

pSather is an extension to serial Sather and adds several features to address parallel execution. A legal Sather program is always a legal pSather program, and yields exactly the same result when executed as a pSather program[57]. The disjunctive lock statement and memory consistency have already been described earlier in chapters 3 and 4 respectively.

---

[57] This is actually not quite true, as a Sather program that uses some of the keywords used in pSather as normal identifiers will break. However, the compiler used at ICSI will not accept pSather keywords as legal Sather identifiers.

A.2.1 Threads

pSather has two different ways to start threads. The first one, the **par/fork** and **parloop** statement allow a thread to create several subthreads and to wait until those threads end their work as shown in figure A-7. The parloop statement is just syntactic sugar, however, it is highly useful to start several threads that have to execute the same code as shown in figure A-8.

```
par
    fork
        -- new thread
    end;
    fork
        -- another thread running in parallel to the first one
    end;
end; -- wait until both threads created end
```

Figure A-7: The par/fork statement

```
parloop                      is equivalent to      par
                                                       loop
    loop body                                              loop body
do                                                         fork
    thread body                                                thread body
end;                                                       end;
                                                       end;
                                                   end;
```

example:

```
-- add two arrays a1 and a2 and store the result in a third one called res.
parloop
    i::=res.ind!;
do
    res[i]:=a1[i]+a2[i];
end;
```

Figure A-8: The parloop statement

There is a second way to create threads in pSather, such that the new thread starts and runs independently of the other ones. Note that a pSather program ends only when all threads started end, not when the main thread exits.

To create an independent thread one has to attach it to an ATTACH object. Any object that subtypes from the abstract class `$ATTACH` or the class `$ATTACH{T}` can be used as one. As `$ATTACH` and `$ATTACH{T}` are a subtype of `$MUTEX`, which itself subtypes from `$LOCK`, such an object also doubles as a synchronization object. Figure A-9 shows the interface of the `$ATTACH` and the `$ATTACH{T}` class.

```
abstract class  $ATTACH < $MUTEX is
   birth;
   death;
   synchronize;
end;

abstract class  $ATTACH{T} < $MUTEX is
   birth;
   death(val:T);
   synchronize;
end;
```

Figure A-9: The $ATTACH class

Figure A-10 shows how a thread can be created and attached to an attach object by using the `:-` operator. If the lhs is an attach object without parameters, the function on the rhs may not return an object, otherwise it must return an object of the same type as the parameter of the attach object. The example uses a GATE and GATE{T} as an attach object. Gates were already available in earlier versions of pSather as builtin special objects, in the current version they are just ordinary library classes. For more information about GATEs check out [Feldman 96].

```
   -- create several threads, do some work and wait until the threads end
   g:GATE:=#; -- create a non parameterized gate
   g :- some_function_1;
   g :- some_function_2;
   -- do some work
   lock g.no_threads then end; -- wait until the threads end;

   g2:GATE{INT}:=#; -- create a parameterized gate
   g2 :- search_function_1;
   g2 :- search_function_2;
   -- do some work

   -- dequeue will block until one of the two threads above returns and the
   -- value it returned is enqueued in the GATE{T}.
   #OUT+"first result found: "+g2.dequeue+"\n";
```

Figure A-10: Attaching threads

Whenever a thread encounters a `:-` it executes the following steps (see also figure A-11):

- Lock the attach object
- Create the new thread and wait until it executed the `birth` function of the attach object.
- Unlock the attach object.

The new thread has to follow these steps:

- Execute the `birth` function of the attach object (as the original thread locked the attach object, the new thread can be sure that no other thread executes the `birth` function at the same time).
- Notify the original thread that it can unlock the attach object.
- Execute the function on the right hand side of the `:-` statement and store an eventual return value.
- If the attach object subtypes from $ATTACH{T}, execute `death` and pass in the return value of the function, otherwise execute `death` without an argument.

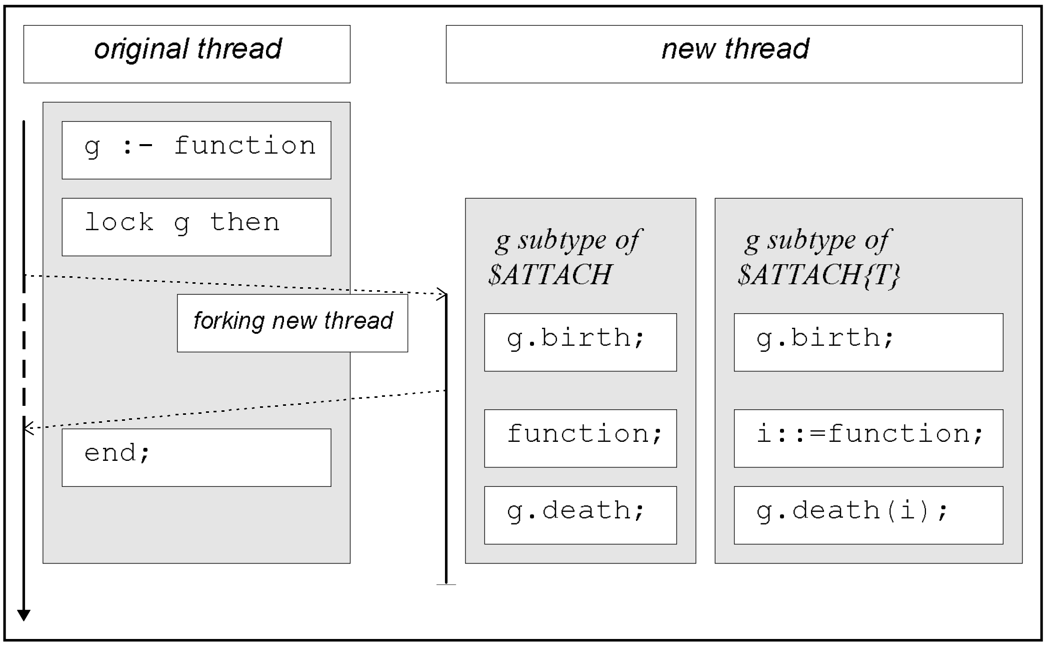

Figure A-11: Attaching a thread

The current pSather runtime offers the following attach objects:

ATTACH < $ATTACH
This class is the most simple attach class possible. It also offers functions to lock the object if threads are attached or not, however, it is not possible to find out how many threads have already terminated.

FUTURE{T} < $ATTACH{T}
Implements the futures found in other languages. Only one thread can be attached to a future object, and the value it returns can be retrieved via the `get` function. It also offers tests to lock the future if it got a value or not and if a thread is attached or not.

GATE < $ATTACH
The GATE class allows many threads to be attached at the same time. Each time a thread ends it increments a counter that can be checked and also manipulated directly.

GATE{T} < $ATTACH
This class is the most complex ATTACH class. The return value of attached threads will be enqueued in an internal list. It is also possible to dequeue val-

ues from this list, however, if the list is empty, the thread that tries to dequeue a value will block. It offers several functions to lock the gate if the internal list is empty or not, and if threads are attached or not.

The synchronization function found in the `$ATTACH` and `$ATTACH{T}` type is called whenever a thread executes a `sync`. Usually, this will block the thread until all other threads attached to the same object either die or enter a `sync` statement too. This is used for example in par and parloops to synchronize threads[58], but threads attached to other objects can make use of it too.

A.2.2 Synchronization Classes

The current pSather libraries offers some of the basic locks found in other languages, plus one more elaborate lock example, the rendezvous lock.

MUTEX:
This lock can only be acquired by one thread at a time, however, unlike in other systems, the same thread can acquire the same lock multiple times.

READ/WRITER:
pSather offers three different reader/writer lock. They all can be locked as either a reader or a writer, and a thread that acquired a writer lock can also acquire a reader lock. However, the reverse results in an immediate deadlock. The three classes differ in that the first one (`RW_LOCK`) gives priority to readers, the second one (`WR_LOCK`) to writers and the last one (`FRW_LOCK`) tries to give both a fair chance.

DOOR:
A door lock can be either open or closed. As long as the lock is open, any thread can lock it, but as soon as it closes, no thread will be able to lock it anymore. This lock is useful to terminate threads that wait in some lock as shown in figure A-12.

```
loop
   lock
   when gate.not_empty then
      w:=gate.dequeue;
   when door.open then
      -- we got the signal to end the thread
      return;
   end;
   -- work on w
end;
```

Figure A-12: Thread termination with a door lock.

[58] Even threads created in **par** and **parloops** are attached to an object (of type `PAR_ATTACH`), however, this class cannot be used outside this construct, and there is no way to access such an object inside the pSather language.

RENDEZVOUS:

Rendezvous lock allow two threads to set up a rendezvous and to exchange objects at the rendezvous point, similar to the rendezvous in ADA. A rendezvous lock has two locks, namely `r1` and `r2`. Each thread that waits for lock `r1` will wait until a thread waits for lock `r2`, in which case both will continue, and an eventual argument will be exchanged (see figure A-13 for an example).

```
r:RENDEZVOUS_LOCK{INT};
par
    fork
        -- some work ...
        get:INT;
        -- wait for a thread that waits on r2, and send it a 12
        lock r.r1(12) then res:=r.get; end;
        -- some more work ...
    end;

    fork
        -- some work ...
        get:INT;
        -- wait for a thread that waits on r1, and send it a 5
        lock r.r2(5) then res:=r.get; end;
        -- some more work ...
    end;
end;
```

Figure A-13: Rendezvous lock

A.2.3 Distributed pSather

pSather offers a shared memory environment, even on a distributed memory machine. Unfortunately accessing an object on another cluster is still much more expensive than accessing a local one, however, executing all threads on one cluster is not a solution either, as most processors would stay idle.

As the compiler is not able to solve all locality problems as good as the programmer can, pSather offers several statements that allow distribution of threads and to check where an object resides. This is necessary to create high performance libraries, but those constructs should show up as little as possible in standard used code.

The number of pSather clusters available to a program is determined at runtime, but it will not change during the run of a program. If the connection to a cluster or a cluster breaks, the whole program will stop, as the current pSather implementation is not fault tolerant and was not designed to be. Each cluster gets a number when the program starts, starting with 0. The first thread, created by the system, will run on cluster 0, and it is the responsibility of the user to start threads on other clusters.

clusters, cluster_size, here

Returns the total number of clusters in the system, the number of processor on the local cluster and the number of the local cluster respectively.

@

The @ sign allows the execution of a thread on another cluster, and to temporarily move a thread to another cluster during the execution of a function (see figure A-14 for some examples).

where()

Returns the cluster that owns the reference object given as argument.

far(), near()

far(x) is equivalent to **where**(x)/=**here**, and **near**(x) is equivalent to **where**(x)=**here**.

with near

The **with near** statement allows the programmer to tell the compiler that some local variables contain all references to locally available objects. This way the compiler can emit more efficient code, as it does not have to insert tests to check for the nearness of variables.

```
-- execute a function of an object such that the object is local
obj.blurp@where(obj);

-- execute all threads of a parloop on another cluster
parloop do@clusters!
    -- thread body
end;

-- make a local copy of a variable if necessary
if far(obj) then
    obj:=obj.copy; -- works only if the class defines a copy function.
end;
-- and tell the compiler that obj now points to a local object.
with obj near
    -- use object
end;

-- create an object on cluster 1
obj::=#@1;
```

Figure A-14: Distributed code fragments

Appendix B: The pSather Runtime

B.1 General Overview

The current pSather runtime (which does not yet support garbage collection) relies on an active message library enhanced with thread calls and calls to access locks and semaphores as shown in figure B-1. Note that the active message library usually has direct access to the network, as this is the only way to get the high efficiency network needed by pSather. The exact implementation of this connection is highly hardware dependent.

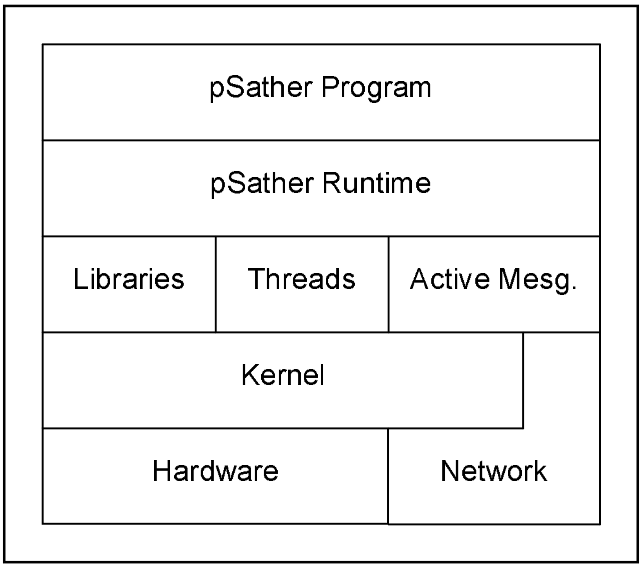

Figure B-1: The general structure of the pSather program

B.2 Low Level Functions

The low level modules provide the following routines (only the ones added to the Sather runtime for pSather are shown here).

Message Passing:

The functions provided by the active message interface [Culler 94] are used in the pSather runtime (`am_reply_n()`, `am_request_n()`, `am_poll()`, `am_put()` and `am_get()`). However, an important change was made to make the interface work with threads.

- Either a thread is started at initialization time that calls `am_poll()` in an infinite loop to drain the network. If this works fine, the other threads will not be able to poll the network (`am_poll()` is just a no-op for those threads).
- Or, if this is not possible because of technical or performance problems, one thread will be created that polls the network at a low priority, and all other threads are expected to poll the network regularly. This is the solution used on the Meiko CS-2, where each thread polls the network at the beginning of each sather loop and whenever a sather function is called.

Thread Creation:

To simplify the thread creation on remote clusters, the same interface as the one used by `am_request_n()` calls is used. Therefore, a call to `thr_create_n()` will create a new thread on the cluster selected, and passes the arguments to the function specified in the call (see figure B-2). Every thread gets a unique id.

message passing:
```
am_request_0(int cluster, func *function);
am_request_1(int cluster, func *function, int arg1);
    ...
am_request_4(int cluster, func *function, int arg1,
                int arg2, int arg3, int arg4);
```
thread creation:
```
thr_create_0(int cluster, func *function);
thr_create_1(int cluster, func *function, int arg1);
    ...
thr_create_4(int cluster, func *function, int arg1,
                int arg2, int arg3, int arg4);
```

Figure B-2: Message passing and thread creation calls

Locks and Semaphores:

A standard interface has been created to interface with the locks and semaphores provided by the system. On system that support spinlocks, those are also supported for short critical sections.

Timing and Signals:

Several calls to allow timers and one signal to be sent to a thread.

Thread Local Memory:

Each thread has access to one local pointer, which is usually initialized to some memory region which can then be used as thread local memory.

B.3 Far Pointers

A program running on a distributed system needs pointers that point to memory on another node. There are basically two solutions, either a far pointer consists now of a pointer and a number that points to the node on which the pointer is valid, or we use some bits of the pointer to designate the node on which it is valid. The second solution has the advantage that the size of a pointer does not change, but it limits the total amount of memory available to the maximum amount available on one system (2^32 Bytes on 32 bit systems).

pSather uses the second solution, and it uses relative node numbers in the pointer. This has the big advantage that a pointer pointing to some local memory has the same representation even in the far pointer form. The disadvantage is of course that each pointer has to change is representation whenever it sent from one node to another. Figure B-3 shows that in addition to the bits used to denote the node, the lowest bit is used to distinguish far pointers from pointers pointing to the high end of the memory on the local system (this usually happens for pointers that point to objects on the stack).

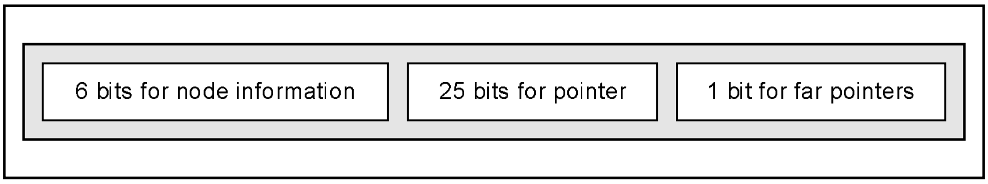

Figure B-3: Bit splitting for far pointers

B.4 Atomic Assignments

pSather guarantees that every assignment is atomic. As most systems that support preemptive threads only guarantee atomic assignments up to a certain size of the value object, such assignments have to be made inside a critical region. To avoid too many serializations, and to avoid to put a lock in every object that is too large for normal assignments, we create a large array of spinlocks[59]. Every variable that needs a lock will lock one of the spinlocks prior to an assignment and release it again after the assignment. The spinlock is selected by masking of some of the bits of the address of the variable.

To simplify the usage of those locks, the runtime offers four macros named `VASS_nn(lhs,type,rhs)`. The first n has to replaced with either a `P` or an `L`, depending if the `lhs` variable is local to the thread or a public visible object. The same has to be done for the second `n` and `rhs` respectively.

[59] The current runtime uses an array of 1024 spinlocks.

B.5 Far Reads and Writes

The pSather runtime distinguishes between four different read and write operations:

- reading and writing of pointers,
- reading and writing of short values that can be assigned atomically (depending on the system this applies usually to values of the size 1, 2, 4 and 8),
- reading and writing of large values that can be assigned atomically,
- reading and writing of values that cannot be assigned atomically.

Additionally, the runtime needs information if the local or far object is visible to other threads or not. Values that are not visible can sometimes be copied more efficiently, as the system does not have to guarantee atomicity.

Similar macros to the ones that execute a far read are available to prefetch a value, while the write macros exist in two versions, one for standard writing and one for post-writing. The complete list of all macros available to read and write far values can be found in the pSather interface file `pSather.h` in the compiler distribution.

Apart from the macros that read and write attribute of objects, there are also a bunch of macros used to read and write complete objects or parts of arrays.

B.6 Remote Thread Creation

Two different remote thread creation exist, one to create a new thread that runs independently from the other one, and one to create a thread that runs synchronously.

- The asynchronous call is used to create a new pSather thread and to attach it to an ATTACH object. Such a thread gets its own exception stack, local memory and pSather THREAD_ID. Several versions exists to accommodate the special requirements of the `par` and `parloop` statements.
- The synchronous call is used to execute a function on a remote node. Such a thread uses the same exception stack as the old one and has the same THREAD_ID. The system makes sure that the old thread will only continue when the new one finished, and makes sure that an eventual return value or exception is correctly forwarded.

B.7 Lock Manager

The lock manager is implemented as a separate thread running on cluster 0. It communicates with the threads that need some locks with several active messages. However, if the fast lock interface is used by the lock, the lock manager will not be involved at all. This is exactly what is needed, except that the deadlock detection mechanism of the lock manager will ignore all threads that wait for a lock without calling the lock manager.

The major problem in implementing the lock manager was to be able to dispatch on a sather object from C (for example, calling the correct `reservable()` function for an object). Unfortunately, the current Sather - C interface does not allow such dispatch, and the only solution was to introduce some special Sather functions with inlined C in the library.

Appendix C: The Benchmark Programs

This appendix describes the programs used to evaluate the optimizations described in this thesis. The programs have been implemented in Sather, pSather, C++, C and parallel C. The parallel C version use the Solaris Thread library directly.

The examples have been selected because they are simple, yet implement widely used methods for solving large problems in parallel.

This appendix shows only code fragments, but the complete code is available over the World Wide Web from `http://www.icsi.berkeley.edu/~fleiner/thesis`.

C.1 Serial Programs

C.1.1 Heat Distribution

The heat distribution program simulates the dissipation of heat in a two dimensional grid. It assumes that each cell in the grid has a specific temperature. During each time step the heat of each cell is recalculated by taking into account the heat of the surrounding cells. It uses the following simple formula, which does not represent the correct heat formula of course, but is fine enough for our demo program:

$$h_{x,y} = \frac{1}{9}\sum_{i=-1}^{1}\sum_{j=-1}^{1} h_{x+i,y+i}$$

We cannot simply calculate the new value of each element and store it in the same matrix again, but by using two matrices, one for the results and one for the values used to calculate the new heat the program works fine.

Additionally, to avoid losing heat at the boundary, we will only use cells inside the matrix to calculate the new heat, and will change the division number from 9 to the number cells used to calculate the sum.

The main loop of the program and the function that calculates the heat are shown in figure C-1.

```
-- calculate the new heat of the cell x, y by using the temperatures stored in t.
heat_of(t:T,x,y:INT):FLT is
   h::=t[x,y];
   d::=1;
   if x>0 then
      d:=d+1;h:=h+t[x-1,y];
      if y>0 then d:=d+1;h:=h+t[x-1,y-1]; end;
      if y<t.size2-1 then d:=d+1;h:=h+t[x-1,y+1]; end;
   end;
   if x<t.size1-1 then
      d:=d+1;h:=h+t[x+1,y];
      if y>0 then d:=d+1;h:=h+t[x+1,y-1]; end;
      if y<t.size2-1 then d:=d+1;h:=h+t[x+1,y+1]; end;
   end;
   if y>0 then d:=d+1;h:=h+t[x,y-1]; end;
   if y<t.size2-1 then d:=d+1;h:=h+t[x,y+1]; end;
   return h/d.flt;
end;

-- calculate the new heat of each cell stored in t1 and store the result again in t1
heat_step is
   -- swap the two matrices t1 and t2
   t::=t1;
   t1:=t2;
   t2:=t;
   loop x::=cols!;
      loop y::=rows!;
         t1[x,y]:=heat_of(t2,x,y);
      end;
   end;
end;
```

Figure C-1: Serial heat distribution

C.1.2 Photo Retouching

Manipulating pictures and photos scanned or taken from satellites is a standard procedure available in most photo manipulating programs. Many of the simple filtering routines simply apply the following function to any pixel. M is a square matrix with an odd sized width M_w and height M_h, *bias* and d are a property of the filter.

$$p_{x,y} = bias + \frac{1}{d}\sum_{i=0}^{M_w}\sum_{j=0}^{M_h} M_{i,j}\, p_{x+i-\frac{M_w-1}{2},\, y+j-\frac{M_h-1}{2}}$$

If the filter is applied to a color image, the filter is applied to each color (usually red, green and blue) separately. Some filter examples are shown in figure C-2. Note that the heat program described above is just a special case of this program, but as the filter is programmed explicitly there are more opportunity for optimizations. Figure C-3 shows the main loop and the function that applies the filter to an individual pixel.

sharpen:	$bias = 0,\; d = 16,\quad M = \begin{pmatrix} -2 & -2 & -2 \\ -2 & 32 & -2 \\ -2 & -2 & -2 \end{pmatrix}$
soften:	$bias = 0,\; d = 13,\quad M = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 5 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
blur:	$bias = 0,\; d = 16,\quad M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$
heat:	$bias = 0,\; d = 9,\quad M = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$

Figure C-2: Several image filters

```
-- Apply filter a to the pixel xp,yp.
-- A pixel is a long integer value, and the red, green and blue part are stored
-- as bits inside it. The filter is applied to all three colors at the same time.
pixel_for(t:T,xp,yp:INT,a:FILTER):INT is
   r::=0;
   g::=0;
   b::=0;
   d::=0;
   loop xa::=a.size1.times!;
      loop ya::=a.size2.times!;
         x::=xp+xa-a.size1/2;
         y::=yp+xa-a.size2/2;
         if x<0 or y<0 or y>=t.size2 or x>=t.size1 then
            d:=d+a[xa,ya];
         else
            c::=t[x,y];
            b:=b+a[xa,ya]*c.band(0xff);
            g:=g+a[xa,ya]*(c.band(0xff00)).rshift(8);
            r:=r+a[xa,ya]*(c.band(0xff0000)).rshift(16);
         end;
      end;
   end;
   b:=b/(a.divisor-d)+a.bias;
   g:=g/(a.divisor-d)+a.bias;
   r:=r/(a.divisor-d)+a.bias;
   return b.band(0xff)+g.band(0xff).lshift(8)
         +r.band(0xff).lshift(16);
end;

apply(a:FILTER) is
   t::=t1;
   t1:=t2;
   t2:=t;
   loop x::=cols!;
      loop y::=rows!;
         t1[x,y]:=pixel_for(t2,x,y,a);
      end;
   end;
end;
```

Figure C-3: The filter program

C.1.3 Matrix Multiplication

The matrix multiplication program used in the benchmarks uses just the simple three loop matrix multiplication program shown in figure C-4. It is not meant to be extremely fast and make use of the cache.

```
-- multiply self with m and return the result as a new matrix
times(m:SAME):SAME pre cols=m.rows is
   r::=#SAME(rows,rows);
   loop x::=rows.times!;
      loop y::=rows.times!;
         loop z::=cols.times!;
            r[y,x]:=r[y,x]+[z,x]*m[y,z];
         end;
      end;
   end;
   return r;
end;
```

Figure C-4: Matrix multiplication in sather

The C and C++ matrix multiplication program use exactly the same loops and store the matrix including the size in a structure as shown in figures C-5 and C-6.

```
class MATRIX {
      inline T &operator()(int x,int y) {
            return t[x+y*cols];
      }

      MATRIX operator*(MATRIX<T> &a) {
            MATRIX r(rows,rows);
            for(int x=0;x<rows;x++)
                  for(int y=0;y<rows;y++) {
                        r(y,x)=0;
                        for(int z=0;z<cols;z++)
                           r(y,x)+=(*this)(z,x)*a(y,z);
                  }
            return r;
      }
}
```

Figure C-5: Matrix multiplication in C++

```
struct MATRIX {
	int rows;
	int cols;
	float *t;
};

#define M(m,x,y) m.t[x+y*m.cols];

struct MATRIX matrix_mult(struct MATRIX a,struct MATRIX b)
{
	int x,y,z;
	struct MATRIX r;
	r=new_matrix(a.rows,a.rows);

	for(x=0;x<a.rows;x++)
		for(y=0;y<b.cols;y++) {
			M(r,y,x)=0.0;
			for(z=0;z<a.cols;z++) {
				M(r,y,x)+=M(a,z,x)*M(b,y,z);
			}
		}
	return r;
}
```

Figure C-6: Matrix multiplication in C

C.1.4 n - Queen Problem

This is the well known problem of putting n queens on an n×n chess board, such that there is just at most one queen on each row, column and diagonal. One of the 92 solutions for the 8 - queen problem is shown in figure C-8.

The problem is solved with the recursive function shown in figure C-7. Exactly one queen will stay in each row and in each column, and therefore we can store a solution in an array of size n. The position of the *i-th* queen will be column *n[i]*, row *i*. The recursive function will try to position the *i-th* queen, and if this worked, it will call itself again to position the *(i+1)-th* queen. A solution is found if all queens have been successfully placed on the board.

```
s_pos(x:INT):INT is
   count::=0;
   -- did we reach the end of the recursion? This means that we found a solution!
   if x=asize then
      if print_result then print; end;
      return 1;
   else
      loop i::=asize.times!;
         -- check if the i-th column is still free
         if [i]=0 then
            ok::=true;
            -- check if any queen stands on the same diagonal
            loop while!(ok);
               j::=asize.times!;
               ok:=[j].int=0 or
                  (x+1-[j]/=i-j and x+1-[j]/=j-i);
            end;
            if ok then
               -- the x-th queen was successfully placed on row x, column i
               -- now try to position the (x+1)th queen
               [i]:=(x+1);
               count:=count+s_pos(x+1);
               [i]:=0;
            end;
         end;
      end;
   end;
   -- return the number of solutions found.
   return count;
end;
```

Figure C-7: N-queen recursive function in Sather

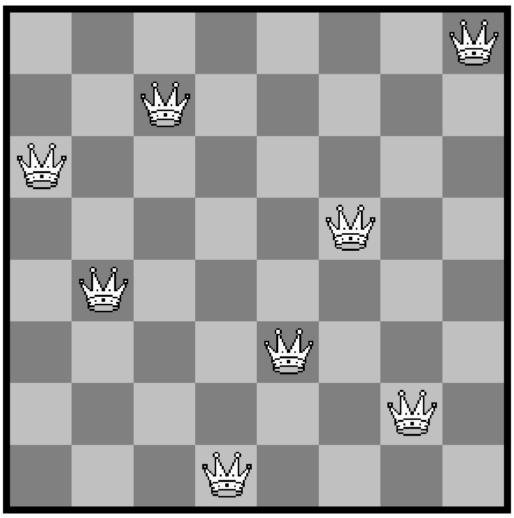

Figure C-8: One solution for the 8-Queen problem

C.1.5 Differential Evolution

The differential evolution program is an iterative program to find solution for a global optimization problem. The algorithm is described in [Storn 95]. The system tries to minimize a function with n arguments. Each possible solution is stored in one of k vectors. During each round of the optimization, each vector v_i is compared against a new vector created by the following function

$$v = v_{r_1} + F(v_{r_2} - v_{r_3})$$

F is a constant, and i, r_1, r_2 and r_3 are three different integers. v and v_i are then merged by taking some values from v and some from v_i. If the new vector yields a better result, v_i is replaced with it, otherwise it is discarded. For more details about this algorithm, its implementation and convergence speed please consult the Technical Report cited above.

C.1.6 The Compiler

As far as we know the Sather compiler with about 50000 lines is the largest Sather application currently running and an excellent test program for the various optimizations. The optimizations are always tested on the compiler compiling itself without any special options, and compiling itself with all serial optimizations turned on. To get reasonable times, the compiler has been patched for those tests such that it does everything except creating the C files and calling the C compiler. This option is now part of the standard compiler and called `-benchmark`. It has also been changed to print out the timing information and how much memory it consumed. As this change is specific to the Solaris environment it is not part of the standard compiler (it has been commented out in the file `cs.sa`).

Unfortunately, when I tried to add the compiler sources, more than 600 pages, to this document, my word processor crashed. Therefore, you have to check them out yourself. They are available from the Sather home page at `http://www.icsi.berkeley.edu/~sather`.

C.2 Parallel Programs

The parallel programs described here are the same ones as above, but make use of a shared memory multiprocessor like the multiprocessor Sparc10. They do not, however, run on multiple clusters and therefore all threads share the same address space. This makes programming easier than programming with distributed pSather, as we do not have to worry about locality of objects.

C.2.1 Heat Distribution

The Heat program is trivial to parallelize, as it consists of two loops, so the outer one can easily be replaced by a parloop as shown in figure C-9. This works especially well together with the parloop optimization described in chapter 6.5. If this optimization would not be available, the parloop should be written such that one thread is created per processor as shown in figure C-10.

```
-- calculate the new heat of each cell stored in t1 and store the result again in t1
heat_step is
    -- swap the two matrices t1 and t2
    t::=t1;
    t1:=t2;
    t2:=t;
    parloop x::=cols!; do
        loop y::=rows!;
            t1[x,y]:=heat_of(t2,x,y);
        end;
    end;
end;
```

Figure C-9: Parallel heat distribution

```
-- calculate the new heat of each cell stored in t1 and store the result again in t1
heat_step is
    -- swap the two matrices t1 and t2
    t::=t1;
    t1:=t2;
    t2:=t;
    parloop p::=cluster_size.times!; do
        loop x::= ((cols*p)/cluster_size).upto!(
                          cols*(p+1)/cluster_size-1);
            loop y::=rows!;
                t1[x,y]:=heat_of(t2,x,y);
            end;
        end;
    end;
end;
```

Figure C-10: Parallel heat distribution, second try

C.2.2 Photo Retouching

As the serial photo filtering program is very similar to the heat program above, the parallel version is too, and therefore the program is not shown here.

C.2.3 Matrix Multiplication

Paralleling the simple matrix multiplication program is trivial, at least if we do not need the highest performance possible as shown in figure C-11[60].

```
mult(b:SAME):SAME is
    a::=#MATRIX{FLT}(rows,b.rows);
    parloop x::=rows.times!;
    do
        loop y::=b.rows.times!;
            loop z::=b.cols.times!;
                a[y,x]:=a[y,x]+self[z,x]*b[y,z];
            end;
        end;
    end;
    return a;
end;
```

Figure C-11: Parallel matrix multiplication

C.2.4 n - Queen Problem

The queen problem can be parallelized by replacing the loop in the recursive function with a parloop (see figure C-7). However, this complicates the counting of the solutions a bit, as the access to the counter variable has to be serialized. To avoid the lock, we created an array so that parloop thread can write its result in a distinguished place in the array. At the end the results are collected and returned. To avoid the creation of too many threads, only the first few recursions use the parloop, the others use then the serial version of the function shown in figure C-12.

[60] The program shown here is a little bit different that the one used in the benchmarks, as the compiler had at that time a problem with parsing the parloop correctly, as it is inside a parameterized class. This bug should be fixed by now.

```
pos(x:INT):INT is
    -- delegate to serial version
    if x>=2 or x=asize then return s_pos(x);
    else
        res:AREF{INT}:=#(asize);
        parloop i::=asize.times!;
        do
            if [i]=0.char then
                ok::=true;
                loop while!(ok);
                    j::=asize.times!;
                    ok:=[j].int=0 or
                            (x+1-[j].int/=i-j and
                             x+1-[j].int/=j-i);
                end;
                if ok then
                    n:QUEEN:=#(asize);
                    n.acopy(self);
                    n[i]:=(x+1).char;
                    res[i]:=n.pos(x+1);
                end;
            end;
        end;
        -- collect the results
        count::=0;
        loop count:=count+res.aelt!; end;
        return count;
    end;
end;
```

Figure C-12: Parallel n-queen function

C.2.5 Differential Evolution

The differential evolution program was parallelized by giving each processor the responsibility over one part of the vectors. Each processor will then try to generate new vectors by using any of the vectors available, but it will only replace the vectors assigned to it. After each processor worked on its vector, the processors synchronize and check if they found the answer. If necessary, they start to replace vectors again.

C.3 Distributed Programs

This section describes real distributed programs that run on distributed systems. The benchmarks that used those programs were run on a Meiko CS-2 with 2 66 MHz processors per node and up to 40 nodes.

C.3.1 Photo Retouching

This program assumes that the photo is distributed over all nodes, and each node will then apply the filter to the part of the photo stored locally. One important problem that has to be solved is the boundary problem, namely the fact that each node needs some of the pixels that are not stored locally to its calculation. One solution is copy those boundary pixels over in one big chunk and another is to store overlapping parts of the picture on each node. However, both solutions make the implementation more complicated than necessary, and they do not use the power of pSather that provides true shared memory even across nodes. The program presented here does not make use of such optimizations, but simply accesses the pixel it needs and relies on the compiler and runtime system to do the correct thing.

It turned out that the main problem to get speedup were not remote memory accesses, but the fact that each thread had to know exactly which coordinates it had to work on. In the first program each thread worked with global coordinates and for each access to the local array it had to make some coordination conversions. Those conversions were much more time consuming than the remote accesses.

After replacing the distributed array with one that allowed each thread to use local coordinates the program ran much faster. Figure C-13 shows the implementation of the distributed array called BIN_CHUNKS2. It works only if the size of the array and the number of processors available is a power of 2 and it uses a special class called SPREAD{T} which allows one to create one object on each cluster at the same time. Each cluster can then work on its own copy, or use remote copies of the object.

```
immutable class BIN_CHUNKS2{T} is
   include SPREAD{LOCAL_BIN_CHUNKS2{T}};

   -- create a distributed array of size s1,s2
   create(s1,s2:INT):SAME;

   -- returns the row, respectively the columns of the array
   size1:INT;
   size2:INT;

   -- returns/sets an element of the global array. x and y are global indexes
   aget(x,y:INT):T is
   aset(x,y:INT,t:T);

   -- returns/sets an element of the GLOBAL array part, but interprets x and y
   -- as coordinates relative to the lowest local x/y coordinate. Therefore x and y
   -- can even be negative.
   local_aget(x,y:INT):T;
   local_aset(x,y:INT,t:T);
end;
```

Figure C-13: Interface of the class BIN_CHUNKS2

The loop that works on each pixel (see figure C-14) looks a little bit more complicated than necessary. However, to avoid that all threads need access to remote values at the same time and therefore creating a jam on the network, the loop tries to spread the calculation of the boundary pixels over the complete calculation.

```
apply(aa:FILTER) is
    t::=t1;
    t1:=t2;
    t2:=t;
    tmp::=t1;

    -- start a thread on each clusters
    parloop do@clusters!;
        a:FILTER;
        if far(aa) then
            a:=aa.copy;
        else
            a:=aa;
        end;

        -- start a thread per processor
        parloop p::=cluster_size.times!; do
            i:INT;
            with a near
                lt2::=t2.local;
                lt1::=t1.local;
                loop x::=(lt2.cs1*p/cluster_size).upto!(
                                   lt2.cs1*(p+1)/cluster_size-1);
                    if x=0 or x=lt2.cs1-1 then
                        loop y::=lt2.cs1.upto!(lt2.cs2-1);
                            lt1.local_aset(x,y,pixel_for(lt2,x,y,a));
                        end;
                        lt1.local_aset(x,0,pixel_for(lt2,x,0,a));
                        lt1.local_aset(x,lt2.cs2-1,
                                       pixel_for(lt2,x,lt2.cs2-1,a));
                    else
                        if x<lt2.cs2 then
                            lt1.local_aset(0,x,pixel_for(lt2,0,x,a));
                            lt1.local_aset(
                                lt2.cs1-1,x,pixel_for(lt2,lt2.cs1-1,x,a));
                        end;
                        lt1.local_aset(x,0,pixel_for(lt2,x,0,a));
                        loop y::=1.upto!(lt2.cs2-2);
                            lt1.local_aset
                                (x,y,local_pixel_for(lt2,x,y,a));
                        end;
                        t1.local_aset(x,lt2.cs2-1,
                                       pixel_for(lt2,x,lt2.cs2-1,a));
                    end;
                end;
            end;
        end;
    end;
end;
```

Figure C-14: Distributed photo retouching program

C.3.2 n - Queen, Using a Distributed Work Queue

The distributed n-queen program uses a special class called DIST_WORK{T} which starts on each cluster as many threads as there are processors, and then distributes some work as evenly as possible. DIST_WORK{T} is a partial class that has to be included in a work class that knows how to work on T. Figure C-15 shows the interface of DIST_WORK. The class tries to maximize locality so that work created locally is executed locally. Only if the local work queue is empty it will try to get work from some other clusters.

```
partial class DIST_WORK{T} is

    -- enqueue some work in the global work bag. If possible it will be executed
    -- locally
    enqueue_work(T);

    -- After enqueuing some work via enqueue_work the threads on all clusters
    -- should be started with this function. After this function has been called, the
    -- threads will run until either all work has been executed, or some thread
    -- called stop_work. After the threads have been started, only those threads
    -- may call enqueue_work anymore.
    start_threads;

    -- call this function if all work should be abandoned.
    stop_work;

    -- the following function is not defined in this partial class, but must be
    -- redefined in any class that includes this one. It should execute the work
    -- represented by t, and it may enqueue new work by calling
    -- enqueue_work.
    work_on(t:T);
end;
```

Figure C-15: Interface of the class DIST_WORK{T}

Note that to implement DIST_WORK{T} correctly, one would assume that a disjunctive lock is exactly what is needed. Unfortunately, this is not the case, as we really need a dynamic disjunctive lock. Figure C-16 shows how the code that selects the non empty queue should look. As this works only if the number of clusters is a compile time constant it is not currently possible to use it. The DIST_WORK{T} class relies on some special synchronization objects that are responsible to select the correct queue and to stop the threads if no more work is available.

```
local::=queue[here];
lock
when local.not_empty then
   w:=local_queue.dequeue;

guard 0/=here when local.empty, queue[0].not_empty then
   w:=queue[0].not_empty

guard 1/=here when local.empty, queue[1].not_empty then
   w:=queue[1].not_empty
...
--  make sure that we stop if no more work is available.
-- all_threads_waiting is a barrier lock that can only be locked if all
-- threads lock it at the same time.
when queue[0].empty, queue[1].empty, ... ,
      all_threads_waiting then
   return;


-- stop when requested to do so
when stop_work then
   return;
end;
```

Figure C-16: Simple, but illegal way to implement a distributed queue

Figure C-17 shows the `work_on()` function for the distributed n-queen problem.

```
work_on(t:QUEEN_PROBLEM) is
   o:QUEEN_PROBLEM;
   -- make local copy
   if far(t) then
      o:=#QUEEN_PROBLEM(size);
      o.acopy(t);o.start:=t.start;
   else
      o:=t;
   end;
   with o near
      if o.start>1 then
         r::=o.s_pos(o.start);
         lock res_mutex then res:=res+r; end;
      else
         x::=o.start;
         loop i::=o.asize.times!;
            if o[i]=0.char then
               ok::=true;
               loop while!(ok);
                  j::=o.asize.times!;
                  ok:=o[j].int=0 or
                        (x+1-o[j].int/=i-j and
                         x+1-o[j].int/=j-i);
               end;
               if ok then
                  no:QUEEN_PROBLEM:=#(o.asize);
                  no.acopy(o);
                  no.start:=o.start+1;
                  no[i]:=(x+1).char;
                  if i=o.asize-1 then
                     work_on(no);
                  else
                     enqueue_work(no);
                  end;
               end;
            end;
         end;
      end;
   end;
end;
```

Figure C-17: `work_on`() for the n-queen problem

C.3.3 Differential Evolution

The distributed differential evolution program uses the same algorithm as the parallel one, it creates one thread per available processor, and each thread takes care of a part of the vectors.

Appendix D: The Optimization Options

This appendix describes all the options necessary to use the optimizations described in this thesis. Some of the options are followed by an integer denoted by a '#'

-O_inline
: Enables inlining for routines and iters, with the default size of 16 for both of them (chapter 5.2, page 64).

-O_inline_routines #
: Enables inlining for routines of a size of up to # (routine calls have a size of 2, iterator calls one of 4, most other statements and expressions a size of 1, chapter 5.2, page 64).

-O_inline_iters #
: Enables inlining for iterators of a size of up to # (chapter 5.2, page 64).

-O_hoist_const
: Moves loop invariants to the beginning of the loop (chapter 5.3, page 67).

-O_hoist_iter_init
: Moves iterator initialization code to the beginning of the loop (chapter 5.4, page 71).

-O_move_while
: Enables the limited loop unrolling described in chapter 5.5 on page 75.

-O_cse
: Turns common subexpression elimination on (chapter 5.6, page 77).

-O_cache
: Enables caching in distributed programs, uses the default cache size of 1024 cache slots a 8 bytes (chapter 6.2, page 83).

-O_cache_size #
: Defines the cache size, must be a power of 2 (chapter 6.2, page 83).

-O_cache_slot_size #
: Defines the cache slot size, should be a power of 2 (chapter 6.2, page 83).

-O_prefetch
: Turns prefetching on in cases were the compiler can prove that the value is actually used later (chapter 6.3, page 88).

-O_loop_prefetch
: Equivalent to `-O_prefetch`, but also enables prefetching in cases were the value may not be used because a loop terminates earlier (chapter 6.3, page 88).

-O_specul_prefetch

Equivalent to `-O_loop_prefetch`, but also enables prefetching in cases were the value may not be used because the function terminates earlier (chapter 6.3, page 88).

-O_prefetch_weight #

defines how many statements and expression have to be between the prefetch call and the use of the value. Default is 15 (chapter 6.3, page 88).

-O_post_write

Enables post writing as defined in chapter 6.4 on page 90.

-O_parloops

Turns parloop optimizations on as defined in chapter 6.5 on page 91.

-O_local

The compiler will now check if any variables or expressions return values that are local and will emit special code for attribute accesses if possible (chapter 6.6, page 93).

-O_local_call

Equivalent to `-O_local`, but the compiler will now emit different versions of a function if it can show that a function will execute faster if it knows that some of the arguments are local. If the compiler can decide at compile time that a faster version of the function can be called, it will do so (chapter 6.6, page 93).

-O_local_call_access #

Defines how many attribute access of a particular argument have to exist inside a function in order to make a special version for it, default is 1 (chapter 6.6, page 93).

-O_local_call_dynamic

Equivalent to `-O_local_call`, but now the compiler will insert code that checks if some arguments are near before calling a function, and, depending on the result of the test, it will call the appropriate function (chapter 6.6, page 93).

-O_local_call_dynamic_access #

Defines how many attribute access have to be saved in order to make a check before calling a function, default is 3, function (chapter 6.6, page 93).

-O_remote_call

If the compiler sees that a function call should be made such that a special argument is local, it will insert the appropriate code (chapter 6.7, page 95).

-O_remote_call_create

The compiler will also make remote calls, even if this means that some of the objects will be created on another cluster (chapter 6.7, page 95).

-O_remote_call_access #

Defines how many attribute access have to be made with a particular function argument to enable a remote call (chapter 6.7, page 95).

-O_yields_in_locks

Turns on the optimization that will make sure that only those loops that actually have iterators that yield inside locks create an entry on the exception stack. Note that in many cases this test is trivial and done even if this option is not used (chapter 7.2, page 103).

-O_locks_on_stack

Checks whether a lock has to create an entry on the exception stack (chapter 7.2, page 103).

Bibliography

[ADA 95] Ada 95 Reference Manual, version 6.0, ISO/IEC/ANSI 8652:1995.

[Adve 95] Sarita V. Adve and Kourosh Gharachorloo, “Shared Memory Consistency Models: A Tutorial”, WRL Research Report 95/7, Digital Equipment Corporation, September 1995.

[Aho 86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques and Tools”, Addison-Wesley Publishing Company.

[Amarasinghe 93] Saman P. Amarasinghe and Monica S. Lam, “Communication Optimization and Code Generation for Distributed Memory Machines”, *Proceedings of the SIGPLAN ‘93 Conference on Programming Language Design and Implementation*, pp. 126-138, Albuquerque, NM, USA, June 1993.

[Anderson 93] Jennifer M. Anderson and Monica S. Lam, “Global optimizations for Parallelism and Locality on Scalable Parallel Machines”, *Proceedings of the SIGPLAN ‘93 Conference on Programming Language Design and Implementation*, pp. 112-125, Albuquerque, NM, June 1993.

[Anderson 95] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang , “Serverless Network File Systems”, *15th Symposium on Operating Systems Principles, ACM Transactions on Computer Systems*, 1995.

[Anderson 95b] Thomas E. Anderson, David E. Culler, David A. Patterson and the NOW team, "A case for NOW (Network Of Workstations)", *IEEE Micro*, 15(1):54-64, February 1995.

[Bacon 93] David F. Bacon, Susan L. Graham and Oliver J. Sharp, “Compiler Transformations for High-Performance Computing”, Technical Report UCB/CSD-93-781, University of California at Berkeley, 1993.

[Barret 92] Geoff Barret, “OCCAM 3 reference manual”, Inmos, 1992.

[Berners-Lee 96] T. Berners-Lee, R. Fielding and H. Frystyk, “Hypertext Transfer Protocol – HTTP/1.0”, Network Working Group, RFC 1945, `http://ds.internic.net/rfc/rfc1945.txt`, May 1996.

[Bernstein 84] P. A. Bernstein and N. Goodman, “An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases”, *ACM Transactions on Database Systems*, 9(4), September 1987.

[Bhatt 95] Sandeep Bhatt, Arjen Lenstra, Marina Chen, James Cowie, Geoffrey Fox and Wojtek Furmanski, “Factoring on the World-Wide Computer (WWC)”, *Supercomputing ‘95*, San Diego.

[Boden 95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic and Wen-King Su, “Myrinet - A Gigabit-per-Second Local Area Network”, in *IEEE-Micro*, Vol. 15, No. 1, pp. 29-36, February 1995.

[Carvalho 83] O. Carvalho and G. Roucairol, “On mutual exclusion in computer networks”, *Communications of the ACM*, 26, pp. 146-147, 1983.

[Chatterjee 95] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber and Shang-Hua Teng, “Optimal Evaluation of Array Expressions on Massively Parallel Machines”, *ACM Transactions on Programming Languages and Systems*, 17(1), January 1995, pp 123-156.

[Chen 92] Tien-Fu Chen and Jean-Loup Baer, “Reducing Memory Latency via Non-blocking and Prefetching Caches”, *ASPLOS-V Proceedings - Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 51-61, 1992.

[Cooperating 96] Cooperating Systems Corporation, “FAFNER: Web Server Support for Factoring RSA130”, Technical Report TR049601, Cooperating Systems, 12 Hollywood Drive, Chestnut Hill MA 02167-2711, USA.

[Culler 93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick, “Parallel Programming in Split-C”, *Proceedings of Supercomputing*, November 1993.

[Culler 94] David E. Culler, Kim Keeton, Cedric Krumbein, Lok Tin Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, Chad Yoshikawa. "The Generic Active Message Interface Specification", UC Berkeley Network of Workstations (NOW) white paper, 1994.

[Cytron 90] Ron Cytron, Jim Lipkis and Edith Schonberg, “A Compiler-Assisted Approach to SPMD Execution”, *Proceedings of Supercomputing ‘90*, pp. 398-406, New York, NY, November 1990.

[Diderich 95] Claude G. Diderich, and Marc Gengler, “Some Strategies for Load Balancing Parallel”, *Algorithms for Irregular Problems: State of the Art, (IRREGULAR '94)*, Chapter 16, (A. Ferreira and J. D. P. Rolim, eds.), pp. 323-338, Kluwer Academic Publishers, 1995.

[Dongara 93] Jack Dongara, Rolf Hempel, Anthony J. G. Hey and David W. Walker, “A Proposal for a User-Level, Message Passing Interface in a Distributed Memory Environment”, ORNL/TM-12231, Oak Ridge National Laboratory, Tennessee, June 1993.

[Dubois 90] Michel Dubois and Christoph Scheurich, “Memory Access Dependencies in Shared-Memory Multiprocessors”, *IEEE Transactions on Software Engineering*, 16(6), pp. 660-673, June 1990.

[Feldman 91] Jerome A. Feldman, Chu-Cheow Lim and F Mazzanti, "pSather monitors: Design, tutorial, rationale and implementation", Technical Report, TR-91-031, International Computer Science Institute, Berkeley, September 1991.

[Feldman 94] Jerome A. Feldman, "Universal High Performance Computing – we have just begun", in Uzi Vishkin, editor, *Developing a Computer Science Agenda for High-Performance Computing*, pp. 26-29, ACM Press, 1994.

[Feldman 96] Jerome Feldman, "A Language Manual For pSather 1.1", available at `http://www.icsi.berkeley.edu`, August 1996.

[Ferrari 94] Adam Ferrari, "TPVM, A Thread-Based Interface and Subsystem for PVM", Technical Report, CSTR-940802, Emory University, Atlanta, GA, USA, 1994.

[Ferrari 95] Adam Ferrari, "Mutliparadigm Distributed Computing with TPVM", Technical Report, CSTR-951201, Emory University, Atlanta, GA, USA, 1995.

[Fleiner 95] Claudio Fleiner, "Active Message Implementation for Shared Memory Machines", `ftp://ftp.icsi.berkeley.edu/pub/sather/am-0.07.tar.gz`

[Fleiner 96] Claudio Fleiner, "Killing Threads Considered Dangerous", *Proceedings of the 1996 Parallel Object-Oriented Methods and Applications Conference*, Santa Fe, NM, USA, February 1996.

[Fleiner 96b] Claudio Fleiner, " Impact of the Different Serial Optimizations of the Sather 1.1 Compiler", `http://www.icsi.berkeley.edu/~fleiner/benchmarks`.

[Fox 87] M. Fox and J. Ywoskus, "Local Area VAXcluster Systems", *Digital Technical Journal*, September 1987.

[Geist 93] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3.0 user's guide and reference manual", Technical Report ORNL TN 37831-8083, Oak Ridge National Laboratory, 1993.

[Geist 94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek and Vaidy Sunderam, "PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing", MIT Press, 1994.

[Gharachorloo 90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons and Anoop Gupta, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proceedings of the 17th Annual Symposium on Computer Architecture*, pp. 15-26, June 1990.

[Gharachorloo 91] Kourosh Garachorloo, Anoop Gupta and John Hennessy, “Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors”, *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 245-257, ACM, New York, 1991.

[Gifford 79] D. K. Gifford, “Weighted Voting for Replicated Data”, *Proceedings of the 7th Symposium on Operating System Principles*, pp. 150-162, December 1979.

[Göhring 93] Hans-Georg Göhring, Franz-Joachim Kauffels, “Token Ring, Principles, Perspectives and Strategies”, Addison-Wesley, 1993.

[Gomes 96] Benedict Gomes, David Stoutamire, Boris Weissman and Holger Klawitter, “A Language Manual For Sather 1.1”, available at `http://www.icsi.berkeley.edu`, August 1996.

[Goodman 89] James R. Doodman, “Cache consistency and sequential consistency”, Technical Report no. 61, SCI Committee, March 1989.

[Goos 95] Gerhard Goos, “Sather-K, The Language”, Report 8/95, University of Karlsruhe, 1995.

[Goscinski 90] A. Goscinski, “Two algorithms for mutual exclusion in real-time distributed computer systems”, *The Journal of Parallel and Distributed Computing*, 9, pp. 77-82, 1990.

[Gunzinger 92] A. Gunzinger, U. A. Müller, W. Scott, B. Bäumle, P. Kohler, W. Guggenbühl, “Architecture and Realization of a Multi Signalprocessor System”, *International Conference On Application Specific Array Processors*, IEEE Computer Society Press, 1992.

[Gupta 92] Manish Gupta, “Automatic Data Partitioning on Distributed Memory Multicomputers”, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

[Hall 95] Mary W. Hall, Brian R. Murphy, Saman P. Amarasinghe, Shih-Wei Liao and Monica S. Lam, “Interprocedural Analysis for Parallelization”, Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, pp. 650-655, San Francisco, CA, USA, February 1995.

[Hirandani 92] Seema Hiranandani, Ken Kennedy and Chau-Wen Tseng, “Compiling Fortran D for MIMD Distributed Memory Machines”, *Communications of the ACM*, 35, (8): pp. 66-80, 1992.

[Hirandani 94] Seema Hirandani, Ken Kennedy and Chau-Wen Tseng, “Evaluating Compiler Optimizations For Fortran D”, *Journal of Parallel and Distributed Computing*, 21(1):27-45, April 1994.

[Johnson 94] Theodore Johnson and Richard Newman-Wolfe, “A Fast and Low Overhead Distributed Priority Lock”, Technical Report TR 10-94, Dept. of CIS, University of Florida, Gainesville.

[Keckler 92] Stephen W. Kecjler and William J. Dally, “Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism”, 19th Annual International Symposium in Computer Architecture, May 1992.

[Kranz 93] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz and Beng-Hong Lim, “Integrating Message-Passing and Shared-Memory: Early Experience”, *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 54-63, San Diego, CA, USA, 1993.

[Krishnamurthy 94] Arvind Krishnamurthy and Katherine Yelick, “Optimizing Parallel SPMD Programs”, *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[Krishnamurthy 96] Arvind Krishnamurthy, Klaus E. Schauser, Chris J. Scheiman, Randolph Y. Wang, David E. Culler and Kathrine Yelick, “Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines”, University of California at Berkeley, CA, USA, 1996.

[Krone 95] Oliver Krone, Christian Renevey and Béat Hirsbrunner, “PT-PVM: A Communication Platform for the Coordination Language CoLa”, unpublished, University of Fribourg, Switzerland, 1995.

[Lamport 78] Leslie Lamport, “Time, clocks, and the ordering of events in a distributed system”, *Communications of the ACM*, 21, pp. 558-564, 1978.

[Lamport 79] Leslie Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”, *IEEE Transaction on Computers,* Vol. C-28, No. 9, pp. 241-248, September 1979.

[Larus 93] James R. Larus, “Compiling for shared-memory and message-passing computers”, *ACM Letters in Programming Languages and Systems”*, 2(1-4), pp. 165-180, 1993.

[Lawson 79] C. L. Lawson, R. J. Hanson, D. Kincaid and F. T. Krogh, “Basic Linear Algebra Subprograms for FORTRAN usage”, *ACM Trans. Math. Soft.*, 5, pp. 308-323, 1979.

[Lim 93] Chu-Cheow Lim, “A Parallel, Object-Oriented System for Realizing Reusable and Efficient Data Abstractions”, Ph.D. Thesis, University of California at Berkeley, TR ICSI TR-93-063.

[Liu 94] Lok Tin Liu, Alan Mainwaring and Chad Yoshikawa, “Whitepaper on Building TCP/IP Active Messages”, University of California at Berkeley, USA, available at `http://now.cs.berkeley.edu/Papers/papers.html`, 1994.

[Mainwaring 95] Active Message Application Programming Interface and Communication Subsystem Organization, Draft Technical Report, Computer Science Division, University of California at Berkeley, `http://now.cs.berkeley.edu/AM/am-spec-2.0.ps`.

[Martin 95] Rich Martin, Lok T. Liu, Vikram Makhija and David E. Culler, “Lanai Active Messages, Release 0.99 PL 14”, `http://now.cs.berkeley.edu/AM/lam_release.html`

[Meiko 93] “Communications Network Overview”, Meiko Limited, 650 Aztec West, Bristol, BS12 4SD, UK.

[Meiko 95] “CS-2 Hardware Overview”, Meiko Limited, 650 Aztec West, Bristol, BS12 4SD, UK.

[Mowry 92] Todd C. Mowry, Monica S. Lam and Anoop Gupta, “Design and Evaluation of a Compiler Algorithm for Prefetching”, *ASPLOS-V Proceedings - Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, 1992.

[Mowry 94] Todd C. Mowry, “Tolerating Latency Through Software-Controlled Data Prefetching”, Ph.D. Thesis, University of Stanford, 1994.

[MPI-FAQ] MPI - Frequently Asked Questions, available at `http://www.erc.msstate.edu/mpi/mpi-faq.html`.

[MPI-FORUM] Message Passing Interface Forum, “MPI-2: Extensions to the Message Passing Interface”, `http://www.cs.wisc.edu/~lederman/mpi2/mpi2-report.ps.Z`.

[Murer 93] Stephan Murer, Stephen Omohundro and Clemens Szyperski, “Sather Iters: Object-Oriented Iteration Abstraction”, Technical Report TR-93-045, International Computer Science Institute, Berkeley, CA.

[Philippsen 93] Michael Philippsen, “Optimierungstechniken zur Übersetzung paralleler Programmiersprachen”, Ph.D. Thesis, University of Karlsruhe, Informatics, 1993.

[Philippsen 95] Michael Philippsen, “Imperative Concurrent Object-Oriented Languages: An Annotated Bibliography”, Technical Report TR-95-049, International Computer Science Institute, Berkeley.

[Philippsen 95b] Michael Philippsen and Ernst A. Heinz, “Automatic Synchronization Elimination in Synchronous FORALLs”, *The Fifth Symposium on the Frontiers of Massively Parallel Computation,* McLean, Virgina, pp. 350-357, February 1995.

[Raymond 91] Eric Raymond (ed.), `The New Hacker's Dictionary', MIT Press, 1991.

[van Renesse 88] Robert van Renesse and Andrew S. Tanenbaum, “Voting with Ghosts”, *Proceedings. Eighth International Conference on Distributed Computer Systems*, IEEE, pp. 456-461, 1988.

[Ricart 81] G. Ricart and A. Agrawala, “An optimal algorithm for mutual exclusion in computer networks”, *Communications of the ACM*, 24, pp. 9-17, 1981.

[Richardson 96] Harvey Richardson, “High Performance Fortran: history, overview and current developments”, TMC-261, Version 1.4, Thinking Machines Corporation, Bedford, MA, USA, April 1996.

[Sather] The Sather/pSather Compiler, available at `ftp://ftp.icsi.berkeley.edu/pub/sather`

[Schauser 95] Klaus E. Schauser and Chris J. Scheiman, “Experience with Active Messages on the Meiko CS-2”. *Int. Parallel Processing Symposium*, Santa Barbara, April 1995.

[Schauser 96] Klaus E. Schauser, Chris J. Scheiman, J. Mitchell Ferguson and Paul Z. Kolano, “Exploiting the Capabilities of Communications Co-Processors”, 10th International Parallel Processing Symposium, Hawaii, April 1996.

[Silberschatz 88] Abraham Silberschatz and James L. Peterson, “Operating System Concepts”, Alternate Edition, Addison-Wesley Publishing Company, 1988.

[Snaman 87] William E. Snaman, Jr. and David W. Thiel, “The VAX/VMS Distributed Lock Manager”, *Digital Technical Journal*, September 1987.

[Stoffel 94] Kilian Stoffel, “Ein neuro-fuzzy gesteuertes Allokationssystem: Verwendung von Methoden der künstlichen Intelligenz zur Realisierung eines dynamischen Allokationssystems vom MIMD-Computersystemen”, Ph.D. Thesis, University of Fribourg, Switzerland, 1994.

[Storn 95] Rainer Storn and Kenneth Price, “Differential Evolution - A simple and efficient adaptive scheme for global optimization over continuous spaces”, Technical Report TR-95-012, International Computer Science Institute, Berkeley, CA, USA, March 1995.

[Stoutamire 96] David Stoutamire and Stephen Omohundro, “Sather 1.1”, available online at http://www.icsi.berkeley.edu/~sather/Specification/Sather-1.1/index.html.

[Tseng 95] Chau-Wen Tseng, “Compiler Optimizations for Eliminating Barrier Synchronization”, *Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming,* Santa Barbara, CA, July 1995.

[von Eicken 92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein and Klaus Erik Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", Technical report UCB/CSD-92-675, University of California at Berkeley, Department of Computer Science, March 92.

[Weaver 93] David L. Weaver and Tom Germond, “The SPARC Architecture Manual, Version 9”, Prentice Hall.

[Weissman 95] Boris Weissman, “Effects of Inlining on Sather Programs”, `http://http.cs.berkeley.edu/~borisv/Sather/inline/inline.html`, 1995.

[Wolfe 89] M. J. Wolfe, “Optimizing Supercompilers fir Supercomputers”, The MIT Press, Cambridge, MA, 1989.

[Yoshikawa 95] Chad Yoshikawa, “Active Message Library for the Meiko CS-2”, available as part of the Sather/pSather Compiler [Sather], 1995.

[Zhou 95] Hombo Zhou and Al Geist, “LPVM: A Step Towards Multithread PVM”, *Journal of Parallel and Distributed Computing*, July 1995.