Design, Implementation, and Analysis of a Split-C Profiler

Björn Haake

University of California at Santa Barbara, USA

Advisor: *Prof. Klaus Erik Schauser, Ph. D.*
Chris Scheiman, Ph. D.

Date: May 2nd, 1996

Contents

1 Introduction **7**

2 Profiling Techniques **11**
2.1 Reasons for a Profiler 11
2.2 How Does a Profiler Work? 12
2.2.1 Data Collection and Data Transformation 12
2.2.2 Displaying Data 12
2.3 The Features of a Profiler 13
2.3.1 Sequential Profiling 13
2.3.2 Parallel Profiling 13
2.4 Profiling Methodologies 14
2.4.1 Online Monitoring 14
2.4.2 Offline Monitoring 14
2.4.3 Reasons for Favoring the Offline Approach 15
2.4.4 The Split-C Profiler - an Overview 15
2.5 Existing Profiling Tools 16
2.5.1 Gprof 16
2.5.2 Upshot 18
2.5.3 VT 19
2.5.4 Paradyn 21
2.5.5 PICL 23
2.5.6 ParaGraph - The Graphical Interface 23
2.5.7 Comparison of Profiling Tools 26

3 The Parallel Hardware and Software Environment **29**
3.1 The Split-C Programming Language 29
3.1.1 Motivation 29
3.1.2 The Programming Model 30
3.1.3 Global Objects 30
3.1.4 Spread Arrays 30
3.1.5 Synchronization 31
3.1.6 A Split-C Example 31
3.2 The Split-C Library 32
3.2.1 Active Messages 32
3.2.2 The Architecture Dependent Routines 33

3.2.3 The Architecture Independent Routines 33
3.3 The Meiko CS-2 34
3.3.1 The Nodes 35
3.3.2 The Network 35

4 Design and Implementation of the Split-C Profiler 37
4.1 Basic Performance Analysis 37
4.1.1 Running a Sequential Profiler in Parallel 37
4.1.2 A Sequential Profiler's Output in Parallel 37
4.2 Data Collection Functionality 38
4.2.1 PICL Modifications 38
4.2.2 Generating Timestamps 38
4.2.3 Problems With the Hardware Clock 41
4.2.4 Changing the Instrumentation Server 42
4.3 Tracing Split-C Events 43
4.3.1 Preparing the Tracing 43
4.3.2 Split-C Keys 44
4.3.3 Starting and Ending the Timer 45
4.4 Graphical Display of Split-C Traces 47
4.4.1 Skipping PICL Events 47
4.4.2 Obtaining Positive Timestamps 48
4.4.3 Mapping Split-C Keys to PICL Keys 48
4.4.4 A Different Graphical Output 48

5 Experimental Analysis 51
5.1 The Overhead of the Profiler 51
5.2 Microbenchmarks of Fundamental Split-C Routines 53
5.2.1 Pingpong Benchmarks 53
5.2.2 Staircase Benchmarks 54
5.2.3 Measuring Split-C Routines Using Statistical Timing 57
5.3 The Data Collection Strategy 62
5.4 Measurements on Larger Applications 64
5.4.1 Overhead from the Profiler 64
5.4.2 Using the Profiler for Peformance Improvements 66

6 Conclusions and Future Work 73

A Split-C Key Mapping Table 75

B Sample Tracefile Output from Split-C 81

Abstract

Because parallel programs are very complex, their execution is hard to understand. A tool that supports the analysis of a parallel program is called a profiler. In this thesis, we compared different profiling techniques and examined existing profiling tools. We used these informations to design and implement a profiler for Split-C. Split-C is a parallel extension to C. Most of the existing implementations are based on active messages, a fast communication mechanism. Split-C provides a global address space, where every processor has access to. Split-phase assignments help to overlap computation with communication. We use a version of Split-C that is implemented on the Meiko CS-2 parallel machine. The Meiko is built from SuperSparc processors that are interconnected via a fat-tree network. This resembles a network of Sparc 10 workstations. The Split-C profiler measures the different communication routines and creates entries for a tracefile in the PICL format. The overhead introduced by the PICL calls varies, in most of the routines it ranges from 1–10 %, only some of the routines imposed a larger overhead between 20 and 40 percent. Roundtrip communication adds larger overhead. The overhead is greater in Split-C compared to MPI (Message Passing Interface, the original target for PICL) because Split-C communication routines are approximately 10 times faster than MPI's. The tracefile can be postprocessed via a graphical interface, such as ParaGraph. We used ParaGraph because it has a very convenient display and uses colors for a helpful and understandable output. We achieved to use the Split-C profiler for improving some large applications.

Chapter 1

Introduction

A sequential program can only be as fast as the processor doing the computation. Parallel programming increases performance by adding more processors to distribute the work. Parallel programming is very powerful because it is much faster than a sequential approach and it suits scientific computation with the large data sets that need to be processed.

Most parallel architectures can be classified as either *SIMD* or *MIMD* [K$^+$94]. SIMD or 'single instruction stream, multiple data stream' performs well for large regular applications that contain a considerable amount of data parallelism. It is not efficient on problems with an irregular data layout. Another model developed is `MIMD` or 'multiple instruction stream, multiple data stream'. Each processor executes different instructions, so that a wider range of problems can be parallelized efficiently.

Parallel architectures can also be classified as *shared memory* or *message passing* [K$^+$94], depending on how they access data. In a shared memory model the processors write and read data from a common piece of memory. In order to avoid data inconsistencies (*read/write* and *write/write* problems) some locking mechanism like a semaphore has to be provided. In a message passing system the data resides on the processors and to access it, a processor sends a request across the communication network to the site where the data is stored. A shared memory model is easier to program than a message passing model because the programmer is not responsible for the data layout or the communication between processors. On the other hand, it is less scalable making it suitable for only a relatively small number of processors.

A parallel program is more difficult to write than a sequential one because distributing the work equally among the processors for efficient use is very hard. We call this problem *load imbalance* [K$^+$94]. Each processor should have more or less the same percentage of utilization. It is also important that the data is distributed evenly across the processors. A big problem in parallel computing on message passing machines is the need to access remote data involves a transfer between different processors over the communication network. This is very expensive since communication over a network is very slow compared to computation. The relationship between these two parameters is known as *communication/computation* ratio. Ideally it is very small indicating a low overhead communication.

If a parallel program does not perform very well it is possible that one of the previously mentioned reasons, like a bad load balance or a high communication/computation ratio, are responsible. The user needs to investigate this further, and he wants to get an 'inside view' of how the program executes including which processors are busy, idle or communicating at a certain time. A tool used for analyzing where a program spends its execution time is called a *profiler* [Fos95]. It shows which functions use most of the time so that these can be examined further. These limiting parts of a program are called *bottlenecks*. The goal in both, sequential and parallel programs, is to minimize bottlenecks in order to increase performance. However, a parallel profiler needs to give more information, like determining at which time each of the processors is idle, busy or handling messages. It should also show the *load imbalance*, the communication/computation ratio and how much overhead is spent on sending and receiving messages.

There are two possible approaches for a profiler, online and offline monitoring. In *online monitoring* the system collects, processes and displays data during run time just like in a debugger. It is possible to set breakpoints, single step through the program etc. Because the data is shown immediately, there is some run time overhead. And to get multiple profiles the program has to be run multiple times. But the output can be different on every invocation, because a parallel program does not run in the same way on each individual run. *Offline monitoring* avoids this problem by working in two parts: In the first phase the entire program runs, collects profile data and flushes it to a file (called a tracefile). In the second phase, a graphical interface postprocesses this tracefile and displays the data graphically. The disadvantage of this solution is that all the collected data needs to be stored. For huge scientific programs this requires huge secondary storage. The good thing about it is that once a tracefile is produced it can be examined over and over again, giving always the same output.

We study the offline profiling in the setting of a global abstract programming language on a distributed memory machine. This machine is the Meiko CS-2, and the language we write the profiler for is Split-C.

The Meiko CS-2 is an example of a message passing machine. It consists of 64 Sparc 10's which are connected via a fast fat-tree network. Each processor has its own Elan co-processor to handle communication.

Split-C [C$^+$95] is a new parallel language, available on the Meiko CS-2 as well as many other platforms. It is an extension to `C` that supports communication between processors. It is based on *active messages* [vECGS92] which provide fast low-latency communication. A global address space makes objects accessible by any processor. A global object is located on a local processor's memory, but can be accessed via a global pointer from any node. A global object is read or written via a *split-phase* assignment, which allows the overlap of communication and computation. After encountering a split-phase assignment, the processor continues execution without waiting for the transfer to complete. At a later point, the processor can check if the transfer has completed. Another helpful Split-C construct are *spread arrays*. These are logically spread over the processors. Together with bulk transfers (transferring a whole group of data with one single command), they allow highly efficient matrix operations.

The goal of this thesis is to implement a profiler for Split-C and to verify that it is working well under the given platform. We thoroughly analyzed the performance of the profiler by doing extensive measurements on Split-C routines as well as on larger applications.

The structure of this thesis is as follows. Chapter 2 gives a general introduction to different profiling techniques along with some examples of existing tools. Chapter 3 introduces the Split-C language, as an example of a parallel programming language and its library. A short overview of the underlying hardware, the Meiko CS-2 and the Elan communication library, is also presented. Chapter 4 describes the changes made to the libraries necessary to obtain a profiler for Split-C programs. It also shows how the tracecalls are inserted.

Chapter 5 examines the experimental results. Microbenchmarks of small Split-C routines measure the overhead imposed by the profiler. Additionally, the profiler is used to examine larger applications. Finally, Chapter 6 gives a conclusion of the work, as well as an outlook on potential future work.

Chapter 2

Profiling Techniques

2.1 Reasons for a Profiler

Somebody has written a program. It runs great and the results are exactly the ones that it should compute. Why care for anything more? The potential problem, especially for large programs, is that they can perform poorly if the implementation is inefficient. It is difficult just by looking at the code to see why the performance is bad, and this task is almost impossible with a parallel program.

We can measure the performance of a parallel program by describing its *speedup*. The *speed-up-theorem* [K$^+$94] says:

$$S = \frac{T_s}{T_p}$$

where S is the speedup, T_s the total sequential execution time with the known best performing algorithm, T_p describes the running time on a parallel machine using p processors. In other words that means that, if an optimal sequential algorithm is taken, and we divide its execution time by the parallel time on each processor, then we get the speed-up. Ideally, the speed-up would be equal to the number of processors, because that would mean that they were all working on the problem all the time. But in practice, some of the time is spent in communication and synchronization, so the speed-up is usually smaller than the number of processors. A challenging goal in parallel programming is achieving a high speed-up. The problem with parallel computing is that there are many interactions between the different nodes. Sometimes they wait in barriers, or they must send variables to other processors, or they must wait until they receive a certain message. There is an arbitrarily complex set of interactions between different nodes, and a badly designed algorithm can result in poor performance in the sense that one processor could do a lot of work while another stays idle or is in a busy-wait loop. This is almost impossible to detect without some tool that provides detailed insights into the utilization and communication of the different processors. Such a tool is called a *profiler*. Its goal is to detect *bottlenecks*, which are defined as inefficient parts of code that slow down execution time. Once detected, the user can eliminate these bottlenecks to increase performance.

2.2 How Does a Profiler Work?

Basically, a profiler is divided into two parts. It has to collect data during the execution and it has to display it so the user knows what is going on in his program. For collecting data, the user's program is modified to recognize certain events and to record them for display. Note that collecting and displaying data will perturb the user's program [SM93].

2.2.1 Data Collection and Data Transformation

To collect the data we describe an *instrumentation system* or IS [WR95]. An IS deals with the collection and management of performance data.

A *monitor* is a tool used to observe the activities of a system. In general, monitors observe the performance of systems, collect performance statistics, analyze data, and display results. Monitors are classified with respect to the implementation level (software, hardware, hybrid). There are three components a software monitor is always composed of:

- Local Instrumentation Server (LIS)

  The Local Instrumentation Server, or LIS, is responsible for inserting the tracecalls into the user's program, which can be done statically before compiling like in PICL [Wor92], or dynamically at run time like in Paradyn [M$^+$95]. The tracecalls, along with the modified communication library, generate entries that go into the tracefile. A usual trace entry consists of the event type, the participating processors, and the time of the event.

- Instrumentation System Manager (ISM)

  The collected data needs to be stored somewhere, and this is done by the Instrumentation System Manager (ISM). It buffers the data and writes it to secondary storage once the buffer is full. The data can either be stored at each node or in a central tracefile. In the latter case the data is either funneled through a central processor during run time, or sent from each node to the central file after the execution of the program.

- Transfer Protocol

  To get the data from the LIS to the ISM an intercommunication network protocol is used. The usual solution is based on the Internet Protocol (IP), but some IS's have sockets (Pablo) [Uni94] or pipes (Paradyn) as a faster means of communication.

2.2.2 Displaying Data

Collecting all the trace data is useless unless we can conveniently display it. Raw data does not tell what is going on in a user's program, so a good graphical output is crucial to the effectiveness of a profiler. The graphs should be straightforward and easy to read without requiring hours going through help pages. Each picture should provide enough information to contribute substantially to the detection of bottlenecks, but extreme details in one image should be avoided because that would decrease the usefulness. It should be possible to see at one glance what a certain window in the graphical interface is all about. Different graphical approaches for a profiler include `OpenGL`, `Tcl/Tk` or `X/Motif`.

2.3 The Features of a Profiler

2.3.1 Sequential Profiling

A fundamental question to address is something that is already important to sequential programs. How much time is spent in each function? This information is necessary to see which functions use most of the time, because those are most likely the ones to speed-up. To get the function execution time we could measure the entry and exit time to and from a function. The problem is that a UNIX clock takes so much time itself that it is impossible to trace small routines correctly. A small function can execute faster than the system clock resolution, thus this function would not contribute to the overall time, even if it is called many times. Unfortunately it is the fundamental concept of C to have these small, fast subroutines, so using the internal hardware clock would not give adequate results.

A solution, that solves the previously mentioned problems, uses a statistical method [GKM83], [GKM82]. At a certain frequency the program counter's location is checked. At the end the total time gets distributed according to the distribution of the program counter's locations. This timing is not real-time, but a statistical measurement. Its advantage lies in the fact that even small routines can be measured adequately and assuming that there are many samples used, its results can be considered a correct approximation. Due to the fact that the method is based on a statistical method, gprof has some limitations on tracing events [Var93].

The second important feature for sequential profiling, which is done rather easily, is counting how often the function was called. A simple counter added to the subroutine entry will do. The only thing to worry about is if it was invoked recursively. This counter does not add a lot of overhead, so it is a straightforward solution.

2.3.2 Parallel Profiling

In a parallel environment there is far more going on than just the sequential processing of functions on each processor. The processors communicate with each other, for synchronizing and exchanging data. To gain a detailed insight we need some tool that observes the parallel execution of a program. In the remainder of this section we discuss the different features that a parallel profiler should implement.

- Compute Time

  For each processor it shows the ratio of CPU-time versus time spent doing things other than waiting or doing communication. This refers to the time when the processors are busy with computations.

- Idle Time

  The time spent in synchronization (barriers) versus the amount of CPU-time gives an idea of how much time the processor is idle.

- Communication Time

  For each node show the amount of time used for communication by the communication co-processor. This gives a general idea of how much time a node spends communicating. It does not show, however, how much of this time was used for computation.

- Overhead Time

  This graph shows how much time the program spent with sending and receiving messages. This is called overhead because it does not belong to the computation time. It is the time necessary from the hardware point of view to deal with these messages, like the latency.

- Message Flow

  For each timestep, it is necessary to show where messages are sent from, where they were sent to, and who has received messages already. This gives an idea of what messages are in the network at a certain time.

- Programming Language and Application Times

  This shows the time spent in the routines specific to the programming language and in the user's application. This gives an idea of the internal language's overhead.

2.4 Profiling Methodologies

We have defined what a profiler is and what it is needed for. The question that comes to mind is how such a tool can be implemented efficiently. Basically, there are two different possibilities: *Online monitoring* directly displays the important features during run time; *offline monitoring* collects data during the execution and runs the actual profiling tool afterwards.

2.4.1 Online Monitoring

Online monitoring works pretty much like a debugger. It displays all the variables of interest in the user's program, and updates the values immediately when they are changed during program execution. Thus when one processor sends a message to another one, the corresponding graph displays this when the program executes the actual send statement. There is a possibility to run the whole program at once or to single step through it. The single step option allows for a very detailed analysis of the program. It is possible to set breakpoints; thus a program can run until a certain part is executed and then it stops automatically. The profiler displays both the program variables and the variables associated with itself.

2.4.2 Offline Monitoring

The idea of offline monitoring is that during program execution the profiler collects data on each of the nodes and at the end writes it to a tracefile. To make this tracefile, we extend the user's source code with special tracecalls, recompile it, and link it against the trace library. Running the instrumented program produces a tracefile, that is postprocessed using a graphical interface. Once collected, it is possible to look at the tracedata over and over again, always having the same result.

The most important problem concerns the decision when to *flush* the buffer from each of the nodes. Flushing refers to the process of writing out data from the main memory to the tracefile. Mainly there are two ways to do so:

- *FOF*: Flushing whenever a buffer on the node is full,
- *FAOF*: Flush all the buffers when one of the node buffers is full.

Detailed studies [WR95] have shown that the second strategy is the more efficient one. Two things were investigated:

- Trace stopping time: the time to fill a buffer for flushing,
- Flushing frequency: ratio of flushes to data collected on one node.

It is shown in the study that when one buffer fills, the chances that the others will soon be filled, too, is very high. In the long run flushing all at once decreases the flush frequency and makes the costly process of writing out more efficient.

2.4.3 Reasons for Favoring the Offline Approach

The great disadvantage in offline monitoring is the large secondary storage it uses for huge tracefiles. It is also a well-known fact that a tracefile becomes huge very easily. However, there are some good reasons to base a `Split-C` profiler on offline monitoring.

The execution run of a debugger-like parallel program can be very unstable. Single-stepping in a wrong order can lead to a deadlock, if the debugger does not prevent this. The prevention on the other hand slows down the execution time. Furthermore, a parallel program does not guarantee the same output for different runs.

Offline monitoring does not involve costly communication between the executable and the profile output server during run time, as in the online approach.

Online monitoring adds run time overhead when it updates the debug variables and the profile graphics. In particular, the communication to the graphics server is very costly. Paradyn is an online approach which shows that the run time instrumentation takes so much time that it makes only sense for long running programs on a large parallel machine.

There are already several good offline monitors available. Online profilers are not that easy to find and though it might be challenging there is still a doubt if it is really the right approach. An online monitor is much harder to write on a parallel machine than on a sequential one. The main difficulty is the communication between the host program and the different nodes. The difficulties that were encountered using the parallel debugger totalview [JD94] on the Meiko CS-2 and trying to port Mantis, a parallel version of the GNU gdb debugger on the same machine, back this last statement.

2.4.4 The Split-C Profiler - an Overview

Figure 2.1 presents the overview of the project. The programmer extends the Split-C code with user-level tracecalls for those parts of the program, which he wants to trace more thoroughly. After the compilation phase, we link the instrumented object code with the modified Split-C library and the trace library. We execute this code on the Meiko and get a tracefile. We can examine the tracefile with ParaGaph, a graphical display, on a uniprocessor machine.

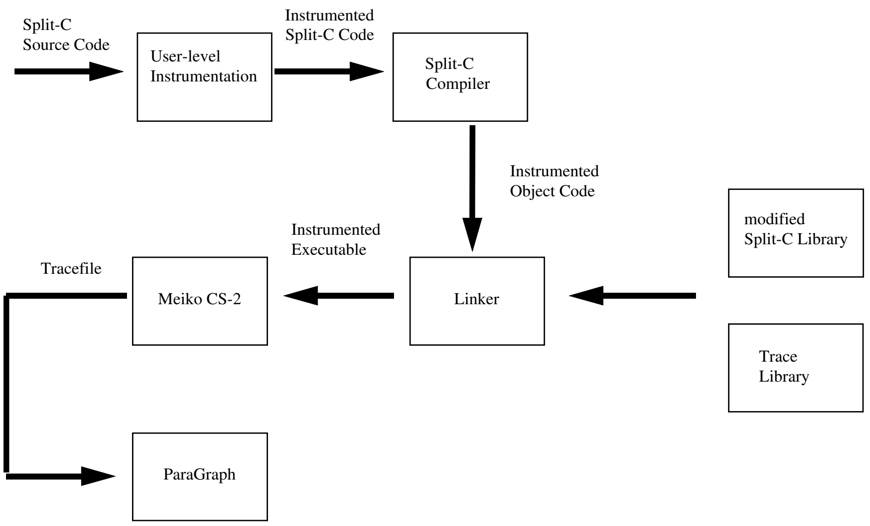

Figure 2.1: Steps Required for a Split-C profiler

2.5 Existing Profiling Tools

There are already some profilers available. We discuss each of them, giving a little bit of information on how they work, what their outputs look like, and describing their advantages and disadvantages.

Some technical reports ([Pik93], [JL93]) already realized the problems for a parallel Split-C profiler, without giving detailed solutions, though.

2.5.1 Gprof

This first program described is *not* a parallel profiler. It is written for UNIX machines and its intention is to determine where a *sequential* `C` program spends its time. This is done by measuring the execution time for each function and counting how often they are called. [GKM83] provides a detailed description. Here, we just give a short overview.

The Instrumentation Server

The instrumentation is done during compile time. The compiler gets a special option, `-pg`, which leads to a modified program after the compilation. The compiler automatically adds a call to a monitoring routine in the prologue of each function call and handles memory allocation for the profiled data. During run time the program counter's (PC) location is checked at a certain rate (the rate depends on the architecture, on a Sun-3 this is 100 times per second) and for each function the PC was found in, a counter is incremented. At the end, the running time of each function is propagated on the distribution of the counters.

<u>The Execution Counts:</u>

The monitoring routine (mcount) can determine the current function by examining its own return address, and it holds a table of counters for each of them. Whenever mcount is called from a certain routine, its counter is incremented, thereby giving the exact number of calls to each routine in the program.

The Execution Time:

Because the timer tick frequency is too slow to get exact timing results, it has to be computed statistically. Periodically (100 times per second), the program is interrupted and the address from the PC is recorded. This does not mean that the PC is in the address range of the same function the whole time of the examined interval, but increasing the number of samples and assuming long execution times should give a good approximation of what routines use the most time. Distribution of samples should be equivalent to the distribution of the execution time. A set of counters records the total address range. Each routine has one or more counters, which are incremented whenever a sample is found in the respective address range. Gprof uses a simple, linear mapping from address ranges to counters. When a routine is called from just one other, all the execution time is accounted to this parent. If more than one function call the same routine, the execution time is accounted to the parents according to the number of calls made from the parents.

The Call Graph:

A call graph is a representation of the relationship between different components in a program. Nodes of the graph refer to the routines and arcs represent the calls between these. An arc is built for every routine that is called by another. Every time an arc is traversed during execution, a counter for this routine is incremented. This counter can be used to determine the number of recursive calls, if the source of the arc is its own routine. The source of an arc can be determined because the compiler generates stack frames and the monitoring routine can get the return address from the called routine.

The Instrumentation System Manager

The whole memory is mapped onto a hash table. Each time a sample is taken, the counter for the according part in memory, where the PC is found, is incremented. Upon returning from the user's program, gprof compacts the table and writes it out to a file (gmon.out by default).

The Graphical Interface

We call the program that provides the gprof output the "graphical interface," to conform with the other programs. Graphical interface is not correct in the sense that gprof just gives statistical output in form of tables. There are no graphics in gprof.

When gprof is run with the user's executable as the argument, and a gmon.out exists, then there are two tables displayed:

- flat profile

  shows the overall distribution of time spent in each subroutine. The time is given as an absolute value as well as a relative compared to the total execution time.

- call graph profile

  gives a detailed statistic for each function on the time it spent running, how many times it was called, how often it was invoked recursively, which function it was called from, and what functions it called.

Postprocessing:

During the program execution the previously mentioned information (execution counts, execution times and call graph) is gathered. After the execution the raw data needs to be transformed so it can be displayed in a convenient and informative way for the user. The time has to be propagated from callee to caller and for each function it must become clear where it is called from and what it calls. The steps that are taken to achieve this are as follows:

1. construct an ordered list of all the routine names in the program.
2. build a dynamic call graph, that shows the relations between different routines, where they were called from and who they called.
3. sum up PC samples within their address ranges and annotate these times to the call graph.
4. sort the call graph in decreasing times, detect cycles and put cycles in one component.
5. propagate the execution time from routine to caller.

2.5.2 Upshot

We now discuss several parallel profilers, starting with upshot. As most of the following examples of parallel profilers, upshot uses an *offline* approach. The steps to produce a profiling image are linking the user's program with a special library, running it to produce a tracefile and postprocessing this tracefile with a graphical interface. Upshot was developed at Argonne National Laboratories. A further introduction is found in [LH91].

The Instrumentation Server

The instrumentation system for upshot is the alog library. The user extends his program by certain tracecalls, then links the library against it. These tracecalls are macros defined in the library, and, during run time, are responsible for collecting the data. Upshot only traces so-called *events*. An event is any action in the user's program, like a call to a function, the entry or exit to or from a function, or a call to a communication routine. It is the user's responsibility to instrument his code in a way that he only traces those parts of a program that take a lot of time or otherwise seem responsible for poor performance (such as long waits in barriers). A buffer exists on each node for gathering the data, and it is written to a file for this node once the buffer is full.

The Instrumentation System Manager

The instrumentation system managing is done apart from the library with a special call to a *merge* program. This takes the different files produced on each node, combines them, sorts them according to their timestamps, and produces one single tracefile.

The Transfer Protocol

To get the data from the distributed buffers to a single tracefiles, upshot uses parallel I/O for communication. Each buffer writes its data to a file on its own node at the same time.

The Graphical Interface

The graphical interface, which is the actual upshot program, deals with the postprocessing of the tracefile. Upshot is based on the X-Window system. It is invoked with the tracefile and optionally a statefile, which is very important to see the most useful features.

- Events

  An event is some sort of action in the user's program, like a call to a function, an entry into or exit from a function or the invocation from some communication routine.

- States

  A state is associated with an ordered pair of events and refers to the part between these events. A number indicates an event, a color indicates a state. It examines the execution time of a routine as well as communication times.

Discussion

Upshot is easy to use and has the advantage that it is already running on the Meiko. It is easy to incorporate the `alog` macro calls into Split-C programs. However, usually this is done by the user to only record the events which are obviously responsible for bad performance. Automation of these steps means to log *every* event. This will slow down the program considerably and produce very large tracefiles. Therefore, the usefulness of the profiling image depends heavily on how the user instruments his code. This poses problems, for example, if somebody wants to profile a large program, that was written by someone else. Additionally, upshot does not provide very much information. It relies heavily on the user-defined events, so setting the log functions effects the tracefile generated.

2.5.3 VT

VT is parallel profiler developed for the IBM SP2. It works only on color workstations. Like the first approach it uses a tracefile generated during runtime and then the graphical interface postprocesses this tracefile.

The Instrumentation Server

The instrumentation server for VT consists of three different parts.

- MPL Communication Library

  The most important component traces the MPL library and accounts for all the communication calls. The library is a modified version of the normal MPL library, so whenever the program calls a communication function, it also generates a trace entry with some statistics like the type of the function or the execution time.

- System Events

  VT also records system events, like page faults, disk traffic or TCP/IP traffic.

- User Events

  The user can specify his own events in the program and trace them separately.

After the buffer fills up, the instrumentation server can either overwrite it, or flush the data to disk.

The Instrumentation System Manager

VT uses a three-tiered approach to write data onto disk. This approach is based on three levels:

- Buffer
- Temporary File
- Permanent File

Once the buffer is filled up, data gets written to the temporary file. If this fills up, too, data finally goes to the permanent file.

The Transfer Protocol

A parallel I/O mechanism collects the data on each node separately. They are later merged into one large, sorted tracefile suitable for postprocessing.

The Graphical Interface

The root window is divided in five parts: communication, computation, system events, network and disk.

<u>Computation</u>

The user and kernel utilization can be viewed as a graph or as a block diagram. There is no indication of which processor is referenced nor if the axis is an absolute time or a percentage. It shows what amount of time was spent computing for the user and how much kernel time is used. The processor idle times show when a processor is not doing any computation. The user load balance says which processors are performing most of the work at any given time. A 3-D graphic shows the processor utilization. Though there are 16 processors for the examined sample file, the 3-D graphic works only for five.

Communication

The interprocessor communication clarifies which processors are communicating with each other. The message status matrix indicates which processor is sending and which one is receiving. And the connectivity graph displays the connection between communicating nodes. A very helpful feature loads the source code of the program. This is done for each node, so each processor can check where the program is at a certain time.

System, Network, and Disk

For the system the page faults, context switches and system calls can be examined for each processor. The number of packets sent and received in the network are shown. A third feature is to display the disk reads and writes.

Discussion

VT provides a lot of different graphs. It not only shows computation versus communication and other things that are characteristic to parallel computing, but also hardware-specific things like I/O traffic, page faults and system calls.

However the graphs are hard to read and it is not easy to understand them without reading a lot of help pages first. The graphs are not scaled and it is difficult to understand what units are used. Another disadvantage is, that VT works only on color workstations.

2.5.4 Paradyn

Paradyn [M+95] is one of the few examples of online monitoring. Its goal is to significantly cut the overhead of collecting trace data by dynamically inserting tracecalls during run time and, once the interesting part of the program is passed, to remove it again. Because the run time insertion and removal of tracecalls is very expensive, Paradyn performs well only on long running programs (e.g. hours or days) because the relative overhead gets smaller. Paradyn works well on machines with many nodes (e.g. thousands), where the offline approach is not possible anymore, because the tracefiles get too large.

The Instrumentation Server

Instrumenting in Paradyn is done by using a local daemon on each node which acts as a server to collect data. A queue on each node along with a buffer models the IS. Every application process running on a node writes its data to this local buffer. The collected data is transferred to the ISM (see below) via a Unix *socket*, which serves as the transfer protocol.

The Instrumentation System Manager

The ISM is also modeled as a queue. It receives and stores data coming from the IS. Paradyn consists of three parts:

- Main Process

  This is the main part of the profiler, it is divided into four segments. The *data manager* handles requests for instrumenting the code and collecting performance data. A *user interface manager* provides visual access to the system's main control while the *performance consultant* controls the automated search for performance problems. At the back end, a *visualization manager* is supervising the communication between the graphical interface and the Paradyn processes.

- Paradyn Daemons

  These are actually inserting the tracecalls into the program, so it is the system dependent part of the code.

- Visualization Process

  Paradyn is build around two data abstractions, the metric-focus grid and the time histogram. These two display data in exact digits and in a graphical representation of these.

The Graphical Interface

The graphical interface in Paradyn constructs a tree, where the leaves try to answer certain questions. This can help the user to find out about bottlenecks.

<u>Locating Performance Problems</u>

The goal of finding a parallel programs bottleneck in Paradyn is achieved using the W^3 search model. This model tries to abstract from parallel programming aspects by answering three questions:

- *Why* is the performance bad?
- *Where* is the performance problem?
- *When* does the problem occur?

<u>Performance Consultant</u>

The performance consultant searches the space of the W^3 model, and displays the results graphically. The output looks like a tree, each node maps one of the three axes in the search model.

Discussion

Paradyn is usable on massively parallel computers with many nodes (e.g. 1000 nodes or more) and with programs that run a long time (e.g. days or hours). The cost of inserting and removing tracecalls during runtime is very expensive. For a small program this would add too much intrusiveness. If the program runs long enough than the relative cost of the insertion compared to the execution time is reduced.

2.5.5 PICL

PICL is a library that allows for collecting trace data to produce a file which is accepted as an input to the graphical interface ParaGraph. ParaGraph is described in detail in the next subsection.

The Instrumentation Server

The instrumentation for the computational part is done explicitly by the user. On the other hand, the library already traces the communication routines, like a send or receive. Each node collects its data into a buffer and, once a buffer fills up, the node writes its data out to a tracefile. This strategy is called FOF (see Section 2.4). This strategy will take longer to perform the flushing, but it is implemented in PICL because it is easier to do. The FAOF strategy requires communication between all nodes, because once a buffer fills up, the corresponding node has to tell all the other nodes to do a barrier, and then the flushing is done at the same time. This approach has the advantage, that during flushing no tracing is done, so even if the flushing takes more time, it does not interfere with the tracing. The problem with the send-receive model is to tell the other nodes, that they should execute a barrier. The only way to do this is to send a message from the one node, where the buffer filled up, to all the others. A send requires a matching receive, but it is not known beforehand, when to execute this receive. Therefore, implementing the FOF strategy, which PICL uses, is much easier.

The Instrumentation System Manager

The instrumentation system manager can either collect the data during runtime on each node and save it into a single tracefile, or it can generate a different tracefile for each node. In the latter case, they have to be merged together into one file. The ISM is responsible for generating the tracefile entries from the raw data. A traceline contains fields with the type of the action (e.g. entry or exit), the type of event (send, receive or idle), the timestamp, the processors that send and receive the message, the message length and some PICL internal fields. A detailed description can be found in [Wor92].

The Transfer Protocol

The transfer protocol used is parallel I/O. The buffers get transferred to the tracefiles concurrently. We will see later that this approach has its limits on the Meiko, so the case to generate a central tracefile is not possible, because it will loose data.

2.5.6 ParaGraph - The Graphical Interface

ParaGraph is another parallel profiler which is based on a tracefile. The Portable Instrumentable Communication Library (PICL), described earlier, generates the tracefile. It works the same way as upshot, generating a tracefile for each node. These tracefiles are later merged into one single file which is used for ParaGraph. [HE91] gives a good overview of ParaGraph. In [HM95], [RJ93] and [KS93] discuss in detail the importance of the convenient graphical output of a profiler. The main aspects ParaGraph keeps track of are communication, computation and tasks.

Typical ParaGraph output is shown in Figure 2.2. The panel in the upper left corner is the control panel. In the upper right we see the output from the tracefile, described in Appendix B. The Animation window shows, which processors are communicating, idle, or busy. The Kiviat diagram provides a different view for the communication. The utilization count displays the busy, idle, and overhead times from the processors over time. There are far too many windows to describe them all.

Computation

For the computation part, ParaGraph shows are the busy and idle times of the processors. They are displayed in a number of graphs; the main difference among them is that in some graphs they give an absolute time and in others they have the percentage of the busy, idle and overhead time. A Gantt diagram shows for each processor at what time it was idle, busy or had overhead.

Communication

There are a lot of different features here. It is possible to see the traffic for each node or on all nodes in bytes at a certain time. The communication matrix shows, which node sends and which receives messages. One of the most interesting features is the animation. It displays the states of the nodes, which can be busy, idle, sending or receiving. The Hypercube gives a network overview for different possible architectures. An interesting one (for example cube) can be chosen and the simulation of the program on it can be seen.

With the node data feature it is possible to see the communication data for single nodes. There are different possibilities, such as the node distance or to which node the communication goes. A type function shows different types of communication in different colors.

Tasks

A task is, just like in upshot, a user-defined event which can be anything (function call, loop, etc.) that the user wants to trace. The user must generate a task with a PICL command. Graphical analysis of tasks includes the percentage of time each task spent running, at which time a task was running on a certain processor and which tasks have been started or are finished.

Statistics

The last part of ParaGraph consists of a summary that shows some statistics. The clock display found here is also very useful to use with the other functions (communication, computation, tasks) to see how far in the program it has gotten. Every action which is performed by a certain processor is shown in the trace event window. And in the statistics window a numerical output is given for a lot of interesting features like the processor's busy and idle times, the time spent sending and receiving messages etc. This is done for each individual node as well as for an aggregate of all nodes.

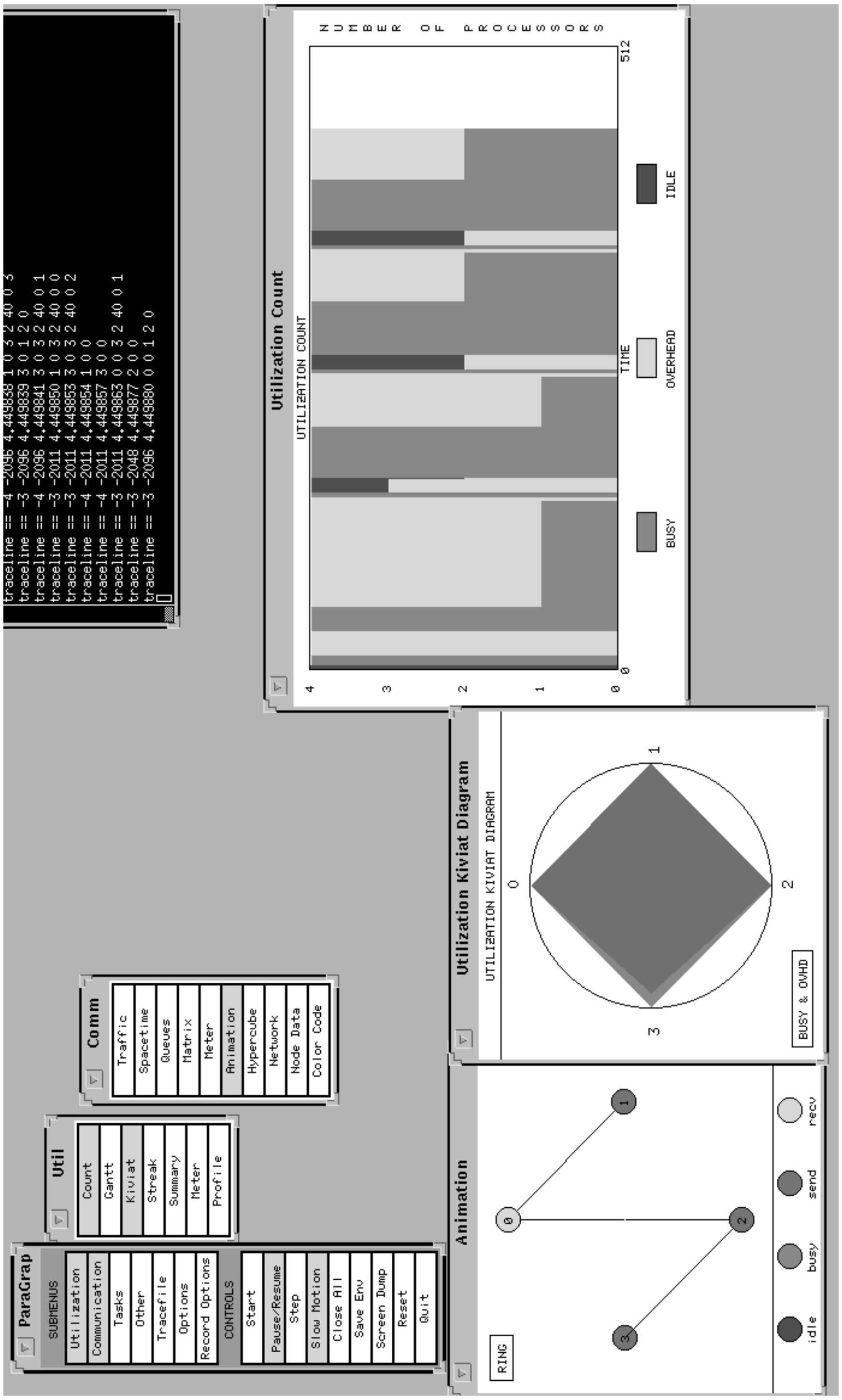

Figure 2.2: ParaGraph output for the Split-C program wator, on 16 processors of the Meiko CS-2.

Discussion

ParaGraph is a very strong parallel profiler which gives very user-friendly and easy-to-read graphical output on color-monitor workstations. It will still work on grayscale and monochrome monitors but with a less convenient output. Some results which depend on different colors (type of message for example) will not be easy to read on these monitors. ParaGraph runs on a lot of different architectures. It can be used with any tracefile that has the same syntax and semantics as the one generated by PICL.

Unfortunately, manual pages are missing to describe the different functions. This would be useful as some of the graph's scales are difficult to understand. This refers to the fact that the axes do not show the unit for the timescales.

Windows that pop up are always placed on the upper left corner (in OpenWindows), which is not very convenient. It is difficult to change the timescale to skip uninteresting events, because some of the windows will show false results after changing the scale during the examination of an application.

2.5.7 Comparison of Profiling Tools

Gprof is a sequential profiler which is fundamental even for a parallel monitor. It provides information on how much time was spent in each function and how often they were invoked. While this can provide information on the busy and idle time, it does not show anything about the communication.

Upshot is the weakest programs of the examined profilers as tracefile generation is dependent on calls to a logging function. This is slow and produces large tracefiles. Also there is not too much functionality in upshot. It only gives an overview of the amount of time spent in each processor for each task and an idle/busy statistic for the different processors over time. The advantage is that the source code is relatively small, it is architecture independent and is compatible with the `C` library.

Paradyn avoids the problem of a huge tracefile by adding and removing tracecalls during program execution. Unfortunately this takes so much time that it is useful only for very large programs (running hours or even days). Considering the given hardware for this thesis and the range of programs it should run, this approach is not considered adequate.

VT is developed by IBM. It offers a lot of functionality, ranging from the usual computation/communication information to disk, network and system specific performance evaluation. It has a very convenient way to start, stop and step the program. The sourcefile can be displayed upon request. A large online help is available. The major drawback is that the graphs are not labeled at all, the processors are not numbered and it is not clear which are the sending and receiving nodes. It does not work on monochrome or grayscale monitors. It is developed for the IBM SP-2 and might be difficult to port.

ParaGraph is a strong profiler which works very good on color workstations, but can be used on monochrome or grayscale monitors as well. The information delivered is good and it covers a lot of interesting aspects. The graphs are well labeled most of the time, and only some of them are not well documented. For instance the timescale is sometimes given in absolute numbers but without referring to the unit of time. ParaGraph runs on various architectures, so the most difficult part for the Meiko should be to get the tracefiles. Most of the features interesting for Split-C are already built-in in ParaGraph. A strong feature is the animation, which shows what various processors are doing at a certain time (busy/idle/sending/receiving).

Unfortunately ParaGraph can not load and display source code, which would be very helpful. Also a help function to describe each possible window would be a great improvement.

Overall VT has the most functionality, but displays the results inconveniently for the user. Upshot lacks too much functionality to be a very helpful tool. ParaGraph has a much more user-friendly output. Although it offers less functionality than VT it seems to be the most promising approach to base our Split-C profiler on.

Chapter 3

The Parallel Hardware and Software Environment

This chapter describes the parallel programming language Split-C, an extension to the widely used sequential language `C`. Split-C was developed as a platform providing efficient software engineering on a parallel machine.

The fundamental concepts in Split-C are a global address space for variables shared among all processors, split-phase assignments to access these global variables, and barriers to synchronize the different processors.

Most implementations of Split-C are based on active messages a method for fast communication [SS95], [vECGS92].

The parallel architecture used for our experimental results is the Meiko CS-2, which consists of 64 Sparc 10s connected on a high speed network. The different processors, also called nodes, are connected via a fat-tree network.

3.1 The Split-C Programming Language

3.1.1 Motivation

Having a parallel machine requires a programming language that effectively supports it. `C` is a fast, general purpose language and therefore widely used for sequential programs in scientific computing. Since it was written for uniprocessors, it lacks the ability of interprocessor communication. Therefore, at the University of California, Berkeley the parallel programming language Split-C [CDG+93] was developed. The underlying language is `C` extended with constructs satisfying two fundamental concepts of parallel programming:

- one active thread of control on each processor, and
- a new level of storage hierarchy involving access to remote memory modules via an interconnection network.

The developers decided to combine the most valuable aspects of message passing, shared memory and data parallel languages to overcome the disadvantages of these three models when used separately. *Message passing* lacks a global view on the data because only two processors communicate with each other at a time and no one else is involved in this dialog. Split-C has a global address space, where global variables are stored, to solve this problem.

3.1.2 The Programming Model

There are different programming models that exist for parallel machines. One that is suited for problems with data subject to identical processing is called data parallelism [K+94]. The language supporting this model is called a data-parallel programming language, where every processor is exactly performing the same steps of execution. This can lead to inefficient code because of the global synchronization involved. The *SPMD* (**S**ingle **P**rogram **M**ultiple **D**ata) model addresses this problem by having each processor running the same program asynchronously. The synchronization is limited to the points where processors need to exchange values. In this model, it is possible to run parts of the code on just a subset of processors. This concept is called *multiple persistent thread* model [CDG+93]. Each processor can be abstracted as a thread of control.

3.1.3 Global Objects

A *global address space* holds data accessible from each processor. Each processor has its own local address range where it can store local variables. The totality of the global address range is made out of all the local ones. The difference is that global objects, accessible from any processor, are dereferenced via global pointers. A global pointer has two parts: one contains the processor number it resides on, the other contains the local address. Thus, a global pointer can always be cast to a local one and vice versa. The dereferencing of a global pointer is far more expensive than that of a local one, but it has the advantage that it can be accessed from any processor. To help overcome the disadvantage of the costly access to a global object, *split-phase assignments* are introduced. During dereferencing a global pointer, which involves expensive communication, the processor can continue with local computation until the thread reaches a synchronization point, waiting for all of the split-phase assignments to be finished. This allows the overlap of communication and computation.

3.1.4 Spread Arrays

A very useful concept in Split-C is the possibility to store different variables on different processors using a *spread array*. This is like an array in `C`, but objects are stored on subsequent processors. Together with bulk assignments, which transfer entire objects, matrix operations can be made very efficient.

Spread arrays are declared by naming the object and giving it at least two dimensions. The dimensions are divided by a '::', called the *spreader*. The part left of the spreader tells on how many processors the object is defined, whereas the part right of the spreader specifies how many objects should reside on each processor. Therefore the statement

```
spread_var[PROCS]::[per_PROCS]
```

is a declaration of a spread array, where each processor, denoted by PROCS, gets per_PROCS elements. Not specifying the part on the right of the spreader associates one item per processor.

Bulk assignments support copies of arbitrary memory regions. Array operations can be made very efficient by using a simple bulk command for any of the memory accessing operations instead of using an expensive loop.

3.1.5 Synchronization

As Split-C follows the SPMD model, it is possible that different processors are running at different speeds (because using `if` statements can mean that some processors can execute different statements than others). This can be a problem when one processor needs to get a value from another, and it cannot be assured which of the two is faster. Then we need some kind of synchronization which assures that the processors follow a specific program sequence. In Split-C the command which ensures that every processor waits for the others to reach the same location in a program is called barrier. While this is a nice way to prevent a program from unpredictable results, the use of a barrier should be done very cautiously, because it adds a lot of overhead. Each processor has to wait until the last one finishes. This can take a long time and leaves the others being idle.

3.1.6 A Split-C Example

In this section, we give a small sample program written in Split-C and explain some of the language characteristics. The program is a straightforward approach which performs poorly. The recursive formula of `fibonacci` is:

$$N = N\text{-}1 + N\text{-}2$$

The given program starts on one processor to compute N by distributing N-1 and N-2 to different processors. This lays out `ibonacci` like a tree structure. After each leaf (or processor) of the tree holds its partial result, the sum of each of the leaves gives the value of fib(N). While this approach changes the semantics of the original idea of `fibonacci` (because of the global summation), it computes the correct result.

```
#include <stdio.h>
#include <stdlib.h>
#include <split-c/split-c.h>
#include <split-c/control.h>
#include <split-c/com.h>

int *global countptr;
int task_queue [PROCS]::, int proz[PROCS]::;

int fib (int n)
{
 if (n <= 1) return 1;
 return (fib (n-1) + fib (n-2) );
}

int distribut ()
{
        int n, p;

        while ((task_queue[MYPROC] ==0)  || (proz[MYPROC] ==0)) am_poll();
```

```
        p = proz[MYPROC];
        n = task_queue[MYPROC];
        while (p>1) {
                p=p/2;
                proz[MYPROC+p] :- p;
                task_queue[MYPROC+p] :- n-2;
                n--;
        }
  return (fib(n));
}

splitc_main(int argc, char **argv)
{
 int n, i, count =0, sum =0, res=0;
 double begin, end, time;

        if (argc !=2) {
            printf("error\n");
            exit (-1);
        }
n = atoi (*++argv);
task_queue[0] = n;        /*init*/
proz[0] = PROCS;
sum = distribut ();
res = all_reduce_to_one_add (sum);
on_one {printf ("result %d\n", res);}
}
```

3.2 The Split-C Library

3.2.1 Active Messages

As most of the implementations of Split-C are based on active messages, we describe these first. Active messages [vECGS92] are a means of very fast communication in large-scale multiprocessors. They were developed because existing message passing libraries have a very high communication overhead. Active messages are based on a lean communication protocol to speed-up the run time of a program.

The idea of how active messages work is easy: Each message has an associated handler which is executed once the message arrives at the receiving side. The purpose is to get the active message out of the network as fast as possible. An active message should be very small and buffering should not be used except for network transportation. This helps in traversing the message quickly through the network and keeping the communication overhead low. In this framework, the network is treated as a pipeline where messages go in and out all the time without delay. The sender has to block until the message can be delivered to the network, and the receiver is responsible for accepting active messages removing them from the network as soon as possible.

The big advantage of active messages over traditional programming models is the elimination of buffering on the receiving side. This can be done either through pre-allocating space in the user's program or the message can hold a simple request to which the handler can reply immediately.

One problem in this model are deadlocks. As we consider modern networks to be deadlock-free, the only condition we want to ensure is, that the handler's reply does not block, e.g. does not busy-wait.

Active messages were implemented to overcome the problem of the communication overhead in message passing architectures. Split-C takes advantage of this fact and its high-level communication routines are build on these low-level messages. The Split-C library has an architecture-dependent and an architecture-independent part.

3.2.2 The Architecture Dependent Routines

This is the portion of the Split-C library where all the functions are located that deal with Meiko specific routines.

The main thing here is of course the implementation of active messages, which were discussed in detail in the previous subsection. The initialization part, the send and receive functions for active messages (including their handlers), and the polling routine, which is used by the receiver to extract messages from the network, are all done here.

The main program is located here because it has to set certain machine-dependent parameters, like the number of processors, before the actual Split-C program is called.

Other machine dependent operations include some global communication routines that take advantage of Meiko's own global routines. These include broadcast, barrier, and reduce. Reduce operations can combine certain variables using arithmetical functions and store the result on one processor. All_reduce_to_one_add (var) for example takes the value of var on each processor, adds them together, and stores this result on one processor (which in usual implementations is processor zero).

3.2.3 The Architecture Independent Routines

In this part of the library contains only Split-C macros which are independent from the architecture. Mainly, these are the operations that deal with global memory access.

Split-phase Assignments

Accessing a global object is up to 10 times more expensive than accessing a local one. Split-C allows the overlap of communication with computation to increase efficiency. This can be achieved by continuing computation while the global variable is transferred from another node. The computation can go on as long as it does not need the value of the global variables. If any node needs the value of the global object, a sync statement ensures that the communication of the global object is complete, and only then can it execute the next statement.

The two basic operations for split-phase assignments are get and put. A get usually has a local variable on the left hand side and a global variable on the right hand side. Get fetches a global value and stores it into a local one. Put is the opposite. A local variable is assigned to a global, with the local being on the right hand side while the global is to the left. The assignment operations for both put and get are indicated by a `:=`. They both work in a similar manner. Once any of the two operations is encountered, they send an active message request to the remote node. The message includes a handler which executes immediately at the receiving side, and sends back an acknowledgment. Figure 3.1 gives an overview of how messages are sent over the network when a global object is accessed.

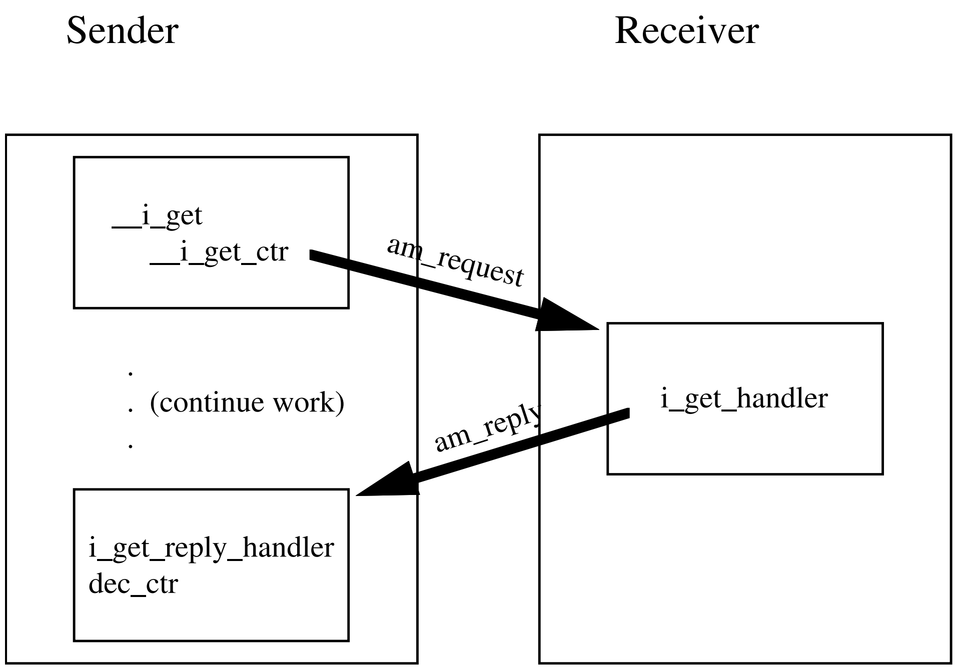

Figure 3.1: The Messages Generated by Split-C for a Get Command

This two-way-send ensures that the sender knows exactly when the get or put has taken place. While a synchronization using get and put has to be made by the programmer explicitly, a read and write is the same operation with an implicit sync already included.

A store is a one-way form of a split-phase assignment. It reduces the overhead by suppressing the acknowledgment from the receiver to the sender, but it is now impossible to assume that the store has been completed. An `all_store_sync` can be used for synchronization among all processors.

Atomic Operations

Atomic operations are necessary in parallel computing to ensure that update operations on any one processor occur sequentially (with mutually exclusive access).

3.3 The Meiko CS-2

The underlying architecture where the Split-C implementation runs is the Meiko CS-2 parallel supercomputer. Each individual node (processing element) is a Sparc 10. The Meiko at UC Santa Barbara consists of 64 of these Sparc processors which are connected via a fat-tree network. The Meiko has a low latency (12 μs one-way writes) and high bandwidth (42 MB/s) for efficient data transfer. As the processing elements are trying to access parts of the memory concurrently, it is very important, that the network is deadlock-free. Fig. 3.2 shows how the machine looks like.

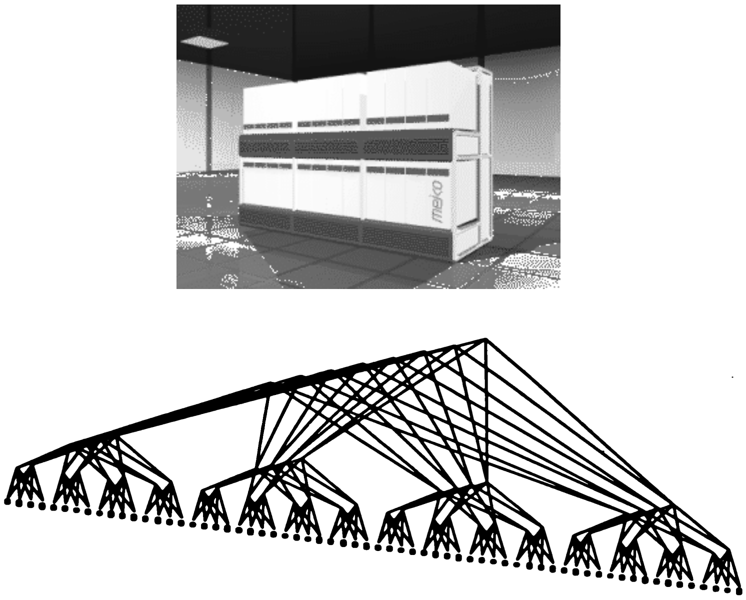

Figure 3.2: The Meiko CS-2

3.3.1 The Nodes

The Meiko is built on Sparc 10 workstations, which are connected like a cluster of workstations. Each of them has 32MB main memory and 40MHz SuperSparcs. They run the Solaris 2.3 Unix operating system. Thus, a single node can operate as an individual Unix machine.

The concept of the Direct Memory Access (DMA) allows direct user-level communication. Using DMA leads to a very high bandwidth (up to 42 MB/s). This makes bulk transfers very efficient. Bulk transfers send a huge amount of data, like an array, with a single command.

Each of the Sparc stations has a co-processor [Mei93] which is responsible for communication. This co-processor performs DMA operations, receives messages, and performs other commands issued by the Sparc 10 processors.

3.3.2 The Network

The individual Sparc stations are connected via a fast, fat-tree communication network. When there is a lot of communication going on in regular tree-structured network, it becomes a bottleneck if processors are using the same subtree. The fat-tree network solves this problem by adding more connections on each level of the tree. The network provides a full connectivity between all processors.

A communications library deals with the software part of the network. The library is used for the initialization phase, where the number of processors are determined. The library also handles things like opening and closing a connection and polling for certain events to occur.

Even if TCP/IP is still a possibility for exchanging messages between the different processors, the Meiko uses its own network, because it is much faster. This is the case with file I/O: processors send and receive files from special server nodes connected on the fat-tree network (8 of the 64 nodes at UC Santa Barbara are server nodes, which contain hard disks shared by the remaining nodes).

Deadlock avoidance in the network is ensured through fixed preallocation of resources.

Chapter 4

Design and Implementation of the Split-C Profiler

The approach chosen for our Split-C profiler is based on an offline monitoring system. The run of a Split-C program, which has been linked with a special profiling library, produces a tracefile. We decided to use the PICL library for the data collection while the visualization is based ParaGraph. This chapter describes the changes to the libraries and to the graphical interface as well as the problems encountered during the development of the profiler for Split-C.

4.1 Basic Performance Analysis

4.1.1 Running a Sequential Profiler in Parallel

The basic performance analysis provides an idea about the time spent in each function, the number of calls to it and the number of recursive invocations. Chapter 2 describes the sequential Unix tool gprof for profiling C programs in detail. In dealing with parallel programs, gprof runs on every node to give the previously mentioned information for each processor separately. Therefore, instead of creating only one filename where gprof writes its output to, we need a unique filename for each node. This is easily done by using the sequential name (gmon.out by default) and adding the processor number (giving gmon.out0, gmon.out1 and so forth). All the file operations, like opening, writing out data, and closing, now perform on all nodes.

4.1.2 A Sequential Profiler's Output in Parallel

Gprof does not provide a graphical output. It only displays tables showing the time spent in different functions. But even without the help of a visualization tool, these tables can provide useful information. In the same way that gprof offers some hints for analyzing sequential programs, it can do this for parallel ones as well. The two ways to show the gprof output from a parallel program are on:

- single nodes

  The analysis on each node is independent from others. This gives useful information on a particular node of the distribution of time and the number of calls to it.

- all nodes

  Gprof provides the ability to specify more profile files. If this is done, it shows the sum of the profile information of all of the input files. This feature is used to give the profiles from all the nodes as input. As the names of the functions are the same on each node, the times for a function are added up and the percentage spent in each function is computed from the sum.

The combination of the two can provide some information about potential bottlenecks in the program. If there is a Function A, that on the output of the sum of all nodes is reported as not very busy, but takes a long time on one particular node, this could be a hint for inefficient code.

To demonstrate gprof, we consider the example Fibonacci program and show the output on a single node and on all nodes in Fig 4.1 and Fig 4.2.

With these two samples, it is already possible to see the differences between a single node (which, in the example happens to be node 0) and the combination of all nodes together. For example, if we look at node 0 separately, `fib` takes more time than `splitc_main`. But on the sum for all nodes, this is vice versa. This shows, that the function fib spends more time in node 0 than on other nodes. In fact, we see that the recursive invocations of fib on node 0 is 11405772, and on all nodes together it is 29860696. So node 0 is responsible for nearly 40% of the total time spent in `fib`. This is a hint for poor load balance.

4.2 Data Collection Functionality

4.2.1 PICL Modifications

The following subsections describe the changes made to PICL. We had to adjust the clock to the given hardware, allow Split-C to generate tracefiles, change the generation of a trace entry, and map the Split-C events to keys readable by ParaGraph.

They include dealing with the hardware clock in order to produce timestamps for the tracefile and recording of Split-C events.

4.2.2 Generating Timestamps

Each hardware platform has its own clock. They vary in the way they are implemented and, especially, in their resolution. The clock resolution is very important to a profiler as timestamps for the trace entries are computed using calls to the hardware clock. The Meiko has a special communications processor. The communications processor is not only used for dealing with sending and receiving messages, but it also provides a system's timer. The clock is started only once, when the system is booted, so consecutive calls to it just generate consecutive numbers. There is no possibility to get the time of the day or other exact timing events. But it is possible to generate two timestamps and subtract them from each other. Thus, the duration of an event can be measured.

```
granularity: each sample hit covers 4 byte(s) for 0.03% of 34.49 seconds

                                    called/total       parents
index  %time    self descendents  called+self    name     index
                                    called/total       children

                                                   <spontaneous>
[1]     45.2   15.58        0.00                 internal_mcount [1]
-----------------------------------------------
                                                   <spontaneous>
[2]     22.3    7.68        0.00                 MAM_poll [2]
-----------------------------------------------
                                                   <spontaneous>
[3]     17.5    6.04        0.00                 MAM_ew_bcast [3]
-----------------------------------------------
                0.00        5.19       1/1           splitc_main [6]
[4]     15.0    0.00        5.19       1         distribut [4]
                5.19        0.00       1/1           fib [5]
-----------------------------------------------
                                 11405772             fib [5]
                5.19        0.00       1/1           distribut [4]
[5]     15.0    5.19        0.00       1+11405772 fib [5]
                                 11405772             fib [5]
-----------------------------------------------
                0.00        5.19       1/1           main [7]
[6]     15.0    0.00        5.19       1         splitc_main [6]
                0.00        5.19       1/1           distribut [4]
-----------------------------------------------
                                                   <spontaneous>
[7]     15.0    0.00        5.19                 main [7]
                0.00        5.19       1/1           splitc_main [6]
-----------------------------------------------

granularity: each sample hit covers 4 byte(s) for 0.03% of 34.80 seconds

  %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 44.8      15.58    15.58                            internal_mcount [1]
 22.1      23.26     7.68                            MAM_poll [2]
 17.4      29.30     6.04                            MAM_ew_bcast [3]
 14.9      34.49     5.19        1  5190.00  5190.00  fib [5]
  0.9      34.80     0.31                            mcount (461)
  0.0      34.80     0.00        1     0.00  5190.00  distribut [4]
  0.0      34.80     0.00        1     0.00  5190.00  splitc_main [6]
```

Figure 4.1: Gprof Output for a Single Node

```
granularity: each sample hit covers 4 byte(s) for 0.01% of 138.50 seconds

                                  called/total       parents
index  %time    self descendents  called+self    name     index
                                  called/total       children

                                                     <spontaneous>
[1]     34.2   47.40        0.00                 MAM_poll [1]
-----------------------------------------------
                                                     <spontaneous>
[2]     29.4   40.69        0.00                 internal_mcount [2]
-----------------------------------------------
               18.71       13.76       4/4           main [4]
[3]     23.4   18.71       13.76       4         splitc_main [3]
                0.00       13.76       4/4           distribut [6]
-----------------------------------------------
                                                     <spontaneous>
[4]     23.4    0.00       32.47                 main [4]
               18.71       13.76       4/4           splitc_main [3]
-----------------------------------------------
                                                     <spontaneous>
[5]     13.0   17.94        0.00                 MAM_ew_bcast [5]
-----------------------------------------------
                0.00       13.76       4/4           splitc_main [3]
[6]      9.9    0.00       13.76       4         distribut [6]
               13.76        0.00       4/4           fib [7]
-----------------------------------------------
                                 29860696             fib [7]
               13.76        0.00       4/4           distribut [6]
[7]      9.9   13.76        0.00       4+29860696 fib [7]
                                 29860696             fib [7]
-----------------------------------------------

granularity: each sample hit covers 4 byte(s) for 0.01% of 139.33 seconds

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 34.0      47.40    47.40                            MAM_poll [1]
 29.2      88.09    40.69                            internal_mcount [2]
 13.4     106.80    18.71        4  4677.50  8117.50  splitc_main [3]
 12.9     124.74    17.94                            MAM_ew_bcast [5]
  9.9     138.50    13.76        4  3440.00  3440.00  fib [7]
  0.6     139.33     0.83                            mcount (461)
  0.0     139.33     0.00        4     0.00  3440.00  distribut [6]
```

Figure 4.2: Gprof Output for All Nodes

In Split-C, there is an explicit call to a clock function. This function is called `get_seconds`, and it uses the elan communication library to compute a time given in seconds. The following example demonstrates, how the clock can be used to measure the time of an event. At the end of the line, example values of get_seconds are given, and the computed time of the duration of *event* is shown.

```
             .
             .
             .
begin_time = get_seconds();       /* begin_time = 857867.003372 */

event();

end_time = get_seconds();         /* end_time = 857907.235691   */

time = end_time - begin_time;     /* time = 40.232319           */
```

4.2.3 Problems With the Hardware Clock

The PICL library is based on the Paragon hardware. The clock that is implemented there is different from the Meiko clock, used by the elan library. The return value from the Paragon clock is stored in two long integers. In the Meiko, a clock value is stored in a double. These two different data structures have to be mapped to each other, so that the PICL library can be used without further changes to incorporate the elan time. This is done by declaring a pointer to a double. The call to the elan clock stores the result into the address, and then it is casted to a struct of two longs. PICL timestamps are based on these structs.

Fig. 4.2.3 shows the code that is described in the previous paragraph.

```
void _hwclock (HWTIME time[2])
{
  double *startd;

startd = (double *) &time[0];
*startd = get_seconds();
}
```

For the profiler it is necessary that the clocks be synchronized, so that they all start with the same time. Every node has its own clock; there is no global clock that could hand down its time to each of the node, thus providing the same time for each processor. Synchronization is necessary to properly trace events in the sequence they occur and that the communication between different processors is traced in the order in they occur. If the clocks were not properly synchronized, it could happen that a message is received on some processor with a timestamp less than the one from the sender. This actually can happen, when messages are very fast, so that the system clock is not fine grained enough to properly record these messages. The occurrence of such a send is called a *tachyon*. The synchronization and adjustment of the first timestamp to 0 in the original PICL library is done by having a loop for each processor until their clocks all have the same time. Using the Paragon machine, this obviously works because the hardware clock of the processors have more or less the same time. On the Meiko, however, this assumption does not hold. The clocks on the processors can be apart from each other a lot, in some cases, differences up to 50 days were found.

While synchronization can still be done without the loop mentioned earlier, most of the timestamps end up being negative. As resetting the nodes on the Meiko to the same time is impossible, a software solution was sought. But even a script from the Meiko Corp. did not work to synchronize the processors. The final solution chosen left out this global clock synchronization completely, thus generating negative timestamps. As ParaGraph cannot handle them properly, they have to be made positive. How this is done exactly is explained in Section 4.4.2.

4.2.4 Changing the Instrumentation Server

The instrumentation server is responsible for transforming the raw, measured data to an entry in a tracefile that can be processed by a graphical interface. The IS is incorporated into the PICL library, and the changes were made within.

According to the encountered event (a send or a receive, for example) the IS generates a tracefile entry. Depending on the event, the number of parameters vary. How a tracefile entry looks like was shown in Section 2.5.5. The generation for Split-C events was made simpler than in the original library. In PICL, there is a case statement for each different command. For Split-C, a single function is doing the same thing. Depending on which event is encountered, the different number of fields in a traceline are generated and the according buffersize has to be allocated.

Regarding the format of a tracefile entry, there are only three different possibilities, depending on the type of event to be traced. Attributes that have always to be added include:

- event entry or exit
- the key value for the type of event
- the timestamp
- the sender
- the process id
- number of attributes
- the type

The field with the number of attributes actually determines, how many of the three remaining entries are going to be included. These entries are as follows:

- message length
- message type
- receiver

After the processing of one line, the pointer in the array has to be incremented to the next field. This has to be done according to how many attributes were added.

4.3 Tracing Split-C Events

For tracing Split-C events, we changed the library. Every call that involves communication is recorded with its start and end time. Thereafter, the amount of communication can be computed as a ratio to the the time spent computing.

4.3.1 Preparing the Tracing

PICL hands down a lot of the responsibility for tracing to the programmer. In Split-C, we decided that the user should no longer have the hassle to call a start and end tracing function. The most time consuming function, detectable by using gprof in parallel, should also be traced automatically. The only job the user has to do himself is if he wants to trace parts of his code as an event which is not caught by the automatic tracecall insertion, and which does not have anything to do with a global communication routine imposed by Split-C. The following program sequence describes the changes to the `main` program of the Split-C library. The initialization of the buffer, the creation of the tracefiles and the final flushing and closing of the tracefiles are all done here. This has the advantage, that the user does not have to do this on his own.

```
main(int argc, char **argv)
{
 int numprocs, me, host;
 char tracefile[20];
 int   n, i;

#ifdef SC_PROFILE
 mpsc_init();
#else
 ew_baseInit ();
#endif
 init_consts();   /* initialize ___PROCS, etc */
#ifdef SC_PROFILE
 tracenode (10000,1,1);  /* bufsize, flush, verbose */
 numprocs = PROCS;
 tracelevel (0,1,0);
 open0 (&numprocs, &me, &host);   /* PICL needs this */
 sprintf (tracefile, "trace%d", MYPROC);
 tracefiles (0, tracefile, 0);
#endif

 splitc_main(argc, argv);  /*  call splitc_main */
#ifdef SC_PROFILE
 traceexit ();      /* flush, close tracefile  */
 close0 ();
#endif

 barrier();
 exit(0);
}
```

The call to *tracenode* determines the size of the buffer, and where the trace data is stored before it is written out to secondary storage. PICL allows the user to decide whether the buffer should be flushed, overwritten, or if tracing should be stopped, if the buffer is full. As the latter two possibilities are not useful, it is implemented into the library that the buffer will be flushed to secondary storage once the buffer is full. The third parameter to tracenode turns synchronization on or off.

The function *tracelevel* enables tracing on different levels (PICL events, user-level, PICL's own trace commands). The only thing we are interested is user-level commands, as the PICL events and its own tracecalls will not appear in the ParaGraph output anyway.

The *tracefile* routine finally is used to create a tracefile, which can be either done on each node separately (where at the end, the different tracefiles have to be merged into one somehow) or it can be created only on processor 0. As we will see, the latter is not possible on the Meiko. When flushing data from all buffers to a file on one node, tracedata gets lost due to problems with the parallel I/O interface on the Meiko.

After returning from the Split-C part, a barrier is performed to ensure that every processor has finished computation of the user's program. The last thing is a call to *traceexit*, which performs the final flush and closes the communication channels and the tracefiles still open.

4.3.2 Split-C Keys

For every Split-C event, like a put or a get, we define a key. This key is used in the graphical interface to generate certain events. A detailed description of ParaGraph will be given in the next section. The key mapping table can be found in the appendix.

There are two different possibilities to generate key values with the Split-C library.

- Directly Mapping Events to PICL Keys

  One approach is to map every single Split-C event to a corresponding PICL key, for example a get to a send. Thus a tracefile is produced that cannot be distinguished from a usual PICL tracefile. The advantage is, that nothing has to be changed for ParaGraph.

- Mapping Events to Split-C Keys

  The second possibility works by mapping each event to a different, Split-C specific key. This key is then mapped to the different states within the graphical interface.

The following reasons led to the second approach. First, with unique keys, the raw tracefile is more useful, because once familiar with it, a programmer can get some insight on what is going on in a program. A big advantage is that it does not exclude the possibility of extending ParaGraph to use more than the three current states (which are send, receive and overhead, all of them distinguished by a different color). If it seems necessary to distinguish Split-C events in more detail, the keys are already defined, and the only thing left to do is to change ParaGraph to use these keys for generating adequate graphics. From the developer's point of view, a key for each event supplies the possibility of easier debugging during the modifications to the library.

4.3.3 Starting and Ending the Timer

In order to get the execution time of a Split-C function, we insert a timer at the beginning and at the end of each routine. Starting the timer is easy. It just reads in the time by a call to the elan hardware clock and writes it into a variable. Ending the timer is a little trickier, because we do not only need to record the stop time but also to which Split-C event it belongs, which node the message came from, where it was sent to and how long the message was. The advantages and disadvantages of calling a timing function at the beginning and at the end of a routine were discussed in detail in Section 2.

The communication routines in Split-C are based on active messages, which work like a pipe, where one puts in messages as soon as they are created. There is no wait for an acknowledgment, the goal is more to get an uninterrupted flow of messages into the network. As ParaGraph is based on the send-receive model, we decided to map the active messages protocol onto this model. This is achieved by describing the initiator of an active message as the sender, and the recipient as the receiver. As in some Split-C routines, an automatic acknowledge is sent back, we included the reply handler as the sender of a message, and the handler on the acknowledge receiving side as the receiver. Thus, we have two messages sent over the network in this case. How the get function call works is graphically shown in Section 3.2. Figure 4.3 illustrates the messages sent over the network and the functions called when tracing is enabled.

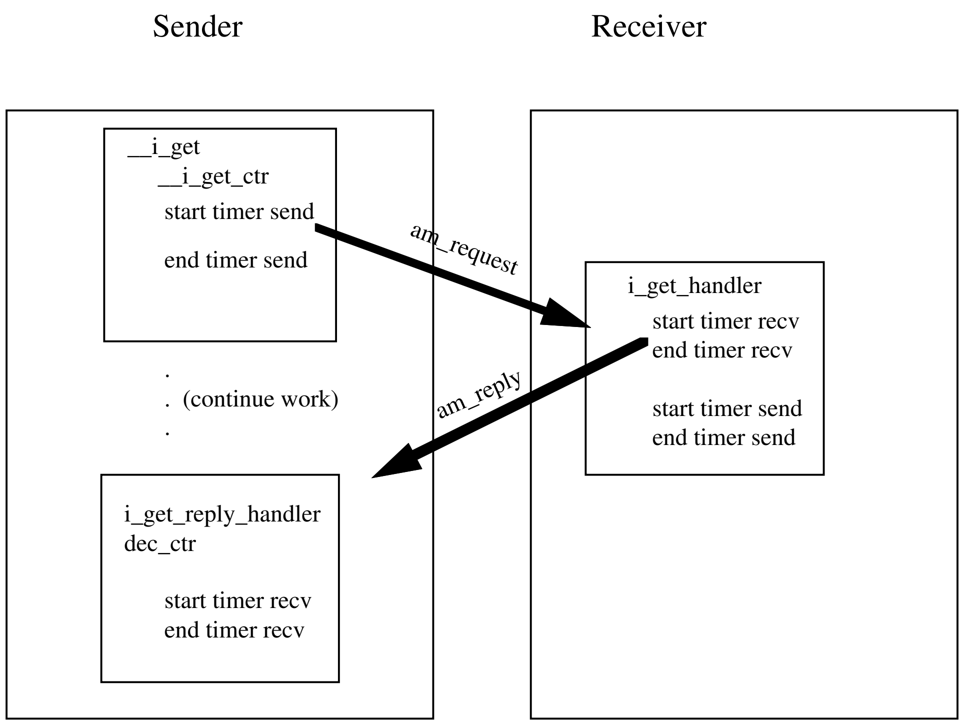

Figure 4.3: The Messages Generated by Split-C for a Get Command

The Different Split-C Levels

The example above exhibits one problem: Split-C functions have more than one level. The highest level can be found in the user's program. The middle part serves as some kind of interface between the high-level user function and the low-level system's function, while in the lowest level, active messages are used for communication. As we are only interested on how much time was spent in a Split-C function, we do not need to trace the active message part. The problem with the remaining two levels is, that they can be either called directly, or the higher-level routine can call the lower level one implicitly. Having a direct call leads to the decision that we need to put the timer in both functions. But then an implicit call to a mid-level function causes the problem that two timers are running at a time. This would slow down the performance, as the timer calls are time consuming. There are two solutions to the problem, that Split-C routines can be called on two different levels:

- Putting the times on a stack

  Putting the times on a stack, so that always matching starting and ending timers are encountered. This ensures that the timestamps for the tracefile are computed correctly.

- Check for tracing

  Once a timer is started on a high-level function, a toggle variable is set to indicate that tracing has been started already. When the lower-level function is called, it has to check whether the toggle variable is set or not. Only when it is called directly the timing in the lower level part will be started.

The latter approach has the advantage that it decreases the overhead, imposed by the first approach due to running the timer twice. Starting and stopping the timer is a costly operation which we try to avoid.

The Parameters to the Timer Function

start_timer is called without any parameters. It just writes the time value in a global variable and returns. The call to end_timer takes the following parameters:

- keyvalue
- sender
- receiver
- message length

The *keyvalue* is essentially the encoding of which Split-C event has been found. Each event has a different keyvalue.

The internal Split-C variable `MYPROC` is the initiator of a message, thus providing the *sender*.

The recipient in a Split-C routine can be computed by decomposing a global pointer. Global pointers were discussed in Section 3.1.3. The processor part can be determined easily, and this is the *receiver*.

The *message length* is given as a parameter to the active message layer in most of the times. In some cases, however, this is not true, and it can be computed on its own by taking the size of the according data structure, using the *sizeof* command.

These parameters are needed for generating the tracefile entries within the PICL library

Changes to the Active Message Layer

It was mentioned earlier what parameters are used for the timing function. Unfortunately, some read routines in the Split-C library that need to be timed do not have the information which processor they are sending to. Thus, the higher-level functions they are called from, have to be extended to transfer a global variable that holds the processor part of the receiver.

4.4 Graphical Display of Split-C Traces

This section deals with how ParaGraph was changed to deal with a Split-C tracefile. The modified ParaGraph first skips uninteresting PICL things, maps keys from Split-C to ParaGraph, then graphically displays the information.

4.4.1 Skipping PICL Events

The PICL tracelibrary was written for the send-receive protocol, where the messages sent over the network are very slow. Split-C communication is much faster. PICL works in a way that its own routines, like open a communication channel and so forth, are also traced. This leads to the following problem. The timescale for the displays in ParaGraph is computed automatically based on the minimum and maximum times in the tracefile. Using PICL functions, which take long times to execute, generates a large time unit, meaning that a long event will be displayed accordingly long, so that it can be seen easily. The problem caused with the Split-C functions is that they are much faster (because they are based on active messages). Thus these fast executions are not shown graphically, because ParaGraph only displays events that take a certain amount of time. If the small timescale is used, so that Split-C events can be shown conveniently, then the problem is vice versa. Now the PICL traces are displayed according to a small timescale, which makes them take a very long time and would present the data in an inconvenient slow way.

One solution could be, that whenever a Split-C event is encountered, the timescale is decreased and increased again for PICL events. But this has the problem that it is hard to see how long each event really executed, because the time units are switched all the time. This can get very confusing.

The final solution chosen was to just skip all the PICL events. This could be achieved by changing the PICL library to not trace its own events, but that would result in a lot of changes to the library. So the responsibility was handed over to ParaGraph, which preprocesses the tracefile and erases all the PICL specific lines. They do not provide any information about the performance of the Split-C program, anyway. So even if the PICL entries are responsible for a large amount of time, they cannot be used to speed up the user's program. Erasing all the PICL specific parts in the tracefile does not effect the performance analysis.

Split-C	PICL
Get, put, read write, store, atomic	send
reduce, broadcast	synchronization
sync, barrier	idle
reply handler	receive

Figure 4.4: Mapping Table

4.4.2 Obtaining Positive Timestamps

Due to the synchronization problems mentioned in Chapter 3, the tracefiles generated from a Split-C program usually have negative timestamps. While this might be alright for some programs, it does not work with ParaGraph in general. The solution to this problem is to get the smallest negative timestamp and to add it to every timestamp. Thus the smallest timestamp is then 0, and the others are positive. A new tracefile with a unique filename has to be created, where the only entries are the Split-C events.

While the addition of the smallest timestamps changes them to positive, there is no loss or modification of the presentation of the times. The relative duration of each timestep is still the same, so the graphical interpretation is correct.

4.4.3 Mapping Split-C Keys to PICL Keys

ParaGraph is based on the PICL library and it is written for the send-receive protocol. In terms of communication there are four states a node can be in: sending, receiving, overhead or idle. Every Split-C event creates a different key. As ParaGraph cannot understand them, they have to be mapped so that they can simulate the send-receive protocol. Table 4.4 shows the mapping of a Split-C key to the corresponding PICL key:

Once the mapping has been done, no other changes are applied. A case statement checks which event has been encountered, and after the mapping is done, it can only be a PICL event.

4.4.4 A Different Graphical Output

We changed the graphs in ParaGraph in order to match them with the Split-C requirements. PICL is based on the send-receive model, so whenever something is sent, a matching receive has to be encountered. With Split-C, where messages can be sent arbitrarily, things get more complicated. Whenever a node is taking any action, the previous state needs to be memorized. A processor can be busy, but then be interrupted to send out a message (an acknowledge, for example, which can come any time). It changes to overhead state, but once the message is sent it should go back to busy. The original ParaGraph is hardcoded, so, depending on the event, it is determined which state the node goes into. Using the original ParaGraph version with Split-C tracefiles produced false graphics because the order of the messages did not match the colors anymore. The following happened: When a node was in a certain state, for example busy, and it was interrupted to do a send, then after the send was finished the node stayed in the state it was in (e.g. overhead). This also distorted the utilization percentage, as now this time was not credited as idle any more. Thus for Split-C purposes, this needed to be changed to record the last event. So when a processor is busy, its state is recorded. Then the send can be issued, changing the state. Once the message is sent, the node has to go back to its original (busy) state.

Chapter 5

Experimental Analysis

This chapter describes the performance analysis of the Split-C profiler and the results of applying our profiling methodology to larger programs. We proceed in two steps. First, we measure overhead imposed by adding the timer calls to the tracing routines, e.g. how much does it slow down the run time of a program. This is done using microbenchmarks of the Split-C routines.

Second, larger programs are traced to show how the profiler can detect bottlenecks.

Before the analysis of the profiler, some definitions will make the following sections more understandable.

A *microbenchmark* is the timing of a small routine such as a system or language function call.

The *testset* refers to the whole environment needed for testing, which includes the hardware, the program to be tested, and the testing program. This includes how the testing was done, e.g. how the run time was measured. To measure the execution time, there are basically two different approaches:

- Record Execution Time only once

  This is the natural approach. A program is run only once, so adding a timer in the beginning and at the end gives the overall time a program or a function spent executing. One problem with running a program or function only once is that the timer can be disturbed by network traffic.

- Average Execution Time over Multiple Runs

  By running a program or a function many (e.g. a thousand or more) times, and then taking the average execution time, a better approximation of the real running time is given. Any intrusiveness by the network is now factored out due to the many measurements.

5.1 The Overhead of the Profiler

To run a parallel program on the Meiko CS-2, instead of just invoking the executable directly, the special startup routine *prun* is called with the executable as an argument. Prun determines on how many processors the program is run, and allocates the partition. It then

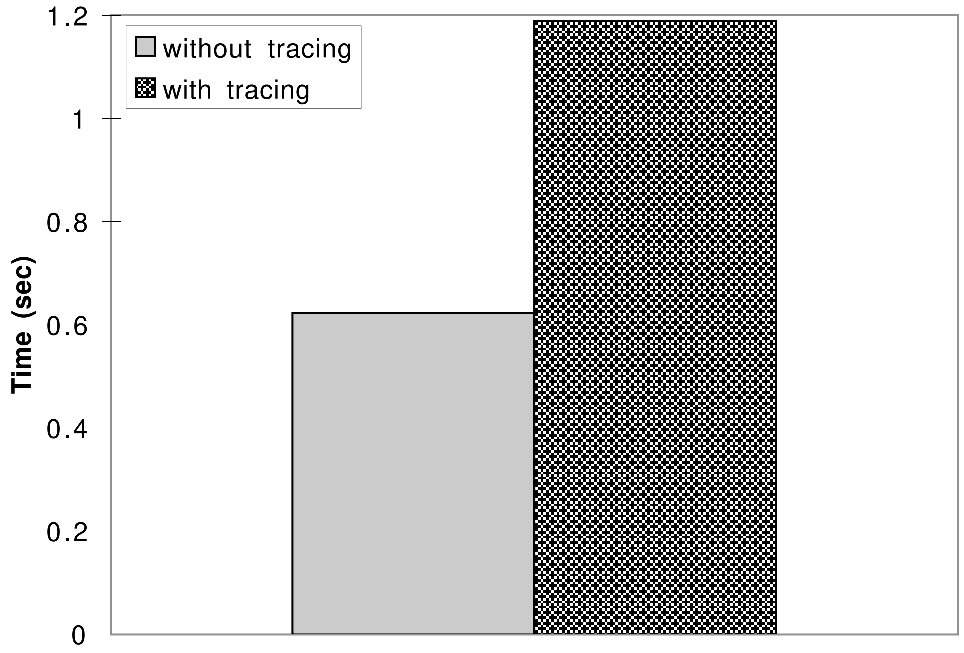

Figure 5.1: Split-C Startup on 32 Procs

loads a local copy of the program from disk to each node. After each processor is set up in this way, the programs can be started. This whole startup can take more than one second, depending on how many processors the program is run on and how busy the network is.

To this usual overhead, the profiler adds more time at the beginning with creating a tracefile, setting up the buffer, setting the flush mode, etc. It also adds overhead at the end by the final flush of the data, closing the connection, and closing the tracefile.

A comparison between the two startup times (with and without tracing) on a different set of processors is shown in Figure 5.1 It is given for completeness, but as the times vary a lot, depending on the network load and on how many user's are trying to run a program, the results can only show that there is a slower startup with tracing. One should be very careful comparing the two in terms of a percentage overhead. The tests are run in such a way that the start and end times occur right at the entry and exit to the main program. The main function of Split-C (called splitc_main) is invoked within the main program, but before this all the tracing functionality as well as the initialization of internal Split-C variables (like PROCS, which holds the number of processors) occurs. Measuring the execution time of the whole main program accounts for the startup overhead and for the trace overhead. The result shows that on 32 processors, executing with tracing takes twice as much time as without tracing. Even though it makes the execution of the program slower, it does not add intrusiveness during the data collection.

The more processors the program is run on, the longer the startup takes. For a small number of processors, the overhead from the trace operations is very small, in the case of 32 processors it can be up to twice the amount of time.

5.2 Microbenchmarks of Fundamental Split-C Routines

This sections deals with the measurements of the Split-C primitives. Times are taken by a ping-pong program, which sends a message to another node and gets a reply back and by the staircase program, where each processor sends a message to its successor. The differences in the various testsets include:

- Buffer Size

  The buffer size determines how often trace flushing needs to be done. If the buffer is small, it is very quick to write it out to secondary storage. But the drawback is that then the I/O needs to be done very often, which in the end makes the execution much slower.

  The increase of the buffer size has the effect that now flushing is done not as often, but it takes more time to write a large buffer.

  Ideally, the problem size would determine the accurate buffer size. If the problem is very large, it does not make sense to use a small buffer, because too many disk I/O operations need to be done, and, vice versa, if the problem size is very small, a very large, but not filled up buffer is flushed, which takes a lot of time.

- Times of Sample Runs

  The more often the samples are run, the more accurate the results are, as discussed in the chapter's introduction. The first of the tests we describe (pingpong programs of Split-C routines) are repeated up to a 1000 times. The execution time is computed in microseconds using the call to the internal Split-C function get_seconds.

5.2.1 Pingpong Benchmarks

A pingpong program is a program that sends a message to one processor, which sends the message back to the initiator. The whole roundtrip time for the message is measured. This gives not only an accurate measurement of how long it takes a message to get sent out, but also how long it takes the handler to process the message and send it back. For all the Split-C operations, like puts and gets, little sample programs, that send out a message and wait for the receive to come back, are measured.

For each of the different routines, Figures 5.2 and 5.3 give a comparison between the routine's execution time of a non-profiled and a profiled version. The programs were run on four processors. The timing is only done for the communication routines only, thus the startup overhead is already factored out.

In the first run, the profiled program has a buffer large enough (e.g. 10 MB) to hold the entire tracedata until the final flush. The overhead for the profiled program is between 40–120%.

An interesting aspect is what happens when the buffer fills up, and the program is not yet finished? There is a need to write out the buffer to secondary storage, so that the data is saved, and the continuing program can refill the buffer. Of course, the I/O operations involved in this take a lot of time, because transfers between disk and main memory are always very slow. Graph 5.3 compares the two outputs. One is the same as above, without any tracing, another with a smaller, 1 MB buffer, that needs to be flushed after a certain

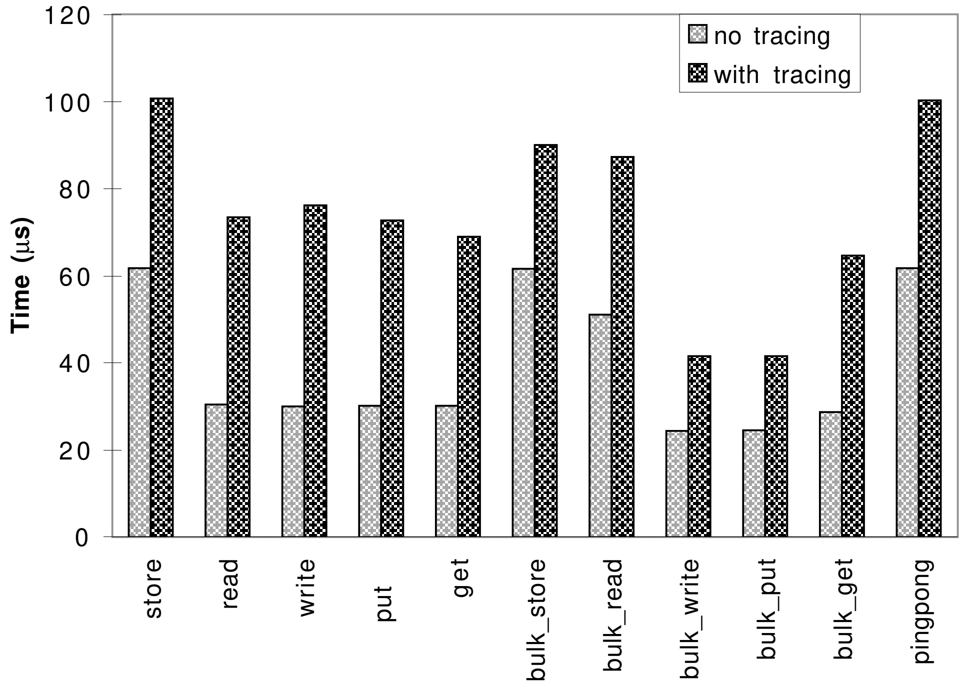

Figure 5.2: Split-C Routines on 4 Processors, Buffer: 10 MB

amount of operations is exceeded. The isolated peaks on this data demonstrate the main disadvantage of an offline trace approach. Once the buffer is full, tracing has to be stopped until the data is saved. Because the PICL library uses the FOF strategy (see Section 2.4), a node flushes its buffer to secondary storage once it fills up. This means that during I/O operations, tracing continues on other nodes, thus influencing the performance of the program. As described in Section 2.4, the preferable method is to flush on all nodes when one buffer fills up. This means, that during flushing no processor is tracing, and there is no intrusiveness. Unfortunately, PICL can not do this.

We see already the tradeoff between a small buffer, which requires a lot of I/O, but has the advantage that the buffer is written very quick, and a large buffer, where flushes only occur rarely, but if they do, they take a lot of time.

Table 5.1 compares a non-traced program versus a traced one that needed flushing for some of the routines (whenever the buffers filled up) and the overhead imposed is printed. The overhead with these programs were between 25 and 150 percent (except for the costly I/O operations, when the buffers are flushed, where the traced routines can run up to a hundred times slower than the original version), which is very high, but can be partly blamed on the inconsistent measuring on the Meiko when I/O is involved.

5.2.2 Staircase Benchmarks

In this section some staircase programs are presented to measure the overhead of the Split-C profiler. In a staircase program every processor sends to its successor in a ring-wise fashion

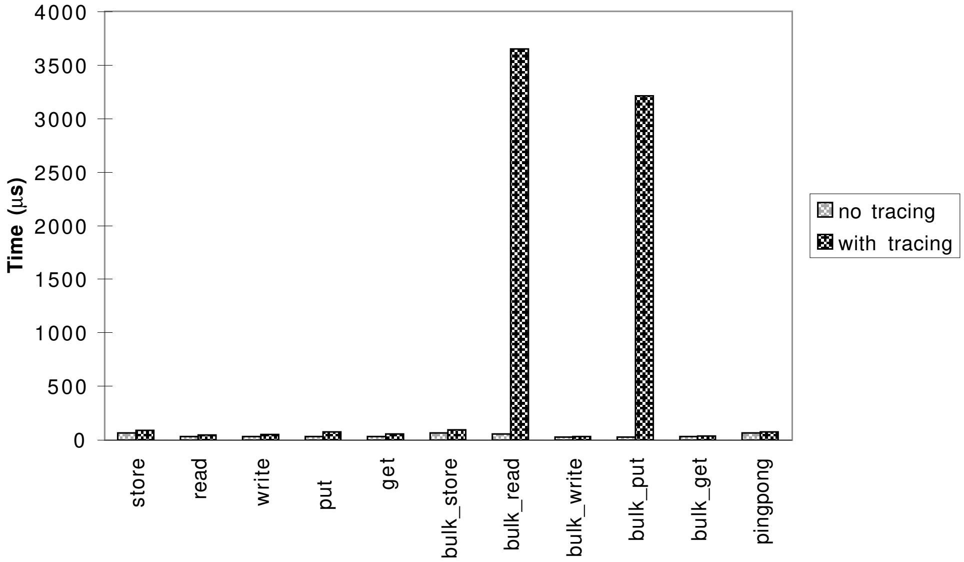

Figure 5.3: Split-C Routines on 4 Processors, Buffer: 1MB

(e.g. zero to one, one to two, and so forth). These programs are implemented for different routines like get, put or store. As in the previous section, various runs are compared against each other. The parameter variations include buffersize, number of runs and, of course, a comparison is made versus a non-traced execution.

The Split-C Implementation of Staircase

The Split-C implementation of staircase works as follows: As long as a local variable on a certain processor does not have a value, it polls. Polling is done in an infinite loop, which is exited if a certain condition becomes true (e.g. a value is assigned to a variable). After it has received a value, it writes this to its successor, and thus the value is passed around like a token on the whole set of processors.

As an example of passing a value in Split-C, we present a staircase benchmark using the command `put`. In Split-C, this is denoted as `:=`, which writes a local value into the global address space of the other processor. This benchmark is presented here:

Split-C Routine	without tracing	with tracing flush on barrier	with tracing flush on full
bulk_store	56.38	90.80	92.90
bulk_store (repeat)	56.05	70.18	94.54
put (multiple dest)	26.48	64.11	63.48
put	24.74	41.35	71.57
get (multiple dest)	26.79	45.22	3565.44
get	24.28	41.80	53.63
store	56.33	70.56	87.08
bulk_read	44.09	61.54	3520.51
bulk_write	18.46	29.18	29.88
bulk_get	35.33	41.01	37.21
bulk_put	19.21	25.00	25.37
bulk_read (multiple dest)	46.11	66.62	3422.59
bulk_write (multiple dest)	18.86	34.87	32.82
bulk_get (multiple dest)	17.30	18.59	20.34
bulk_put (multiple dest)	19.89	29.88	27.62

Table 5.1: Output from Split-C program measuring the roundtrip times of Split-C primitives. Three versions are shown: one without tracing, one with tracing (where data is flushed on barriers), and one with tracing (where data is flushed only when the buffer is full).

```
int *spread iptr, ival;
int idata[PROCS]::[1];

splitc_main(int argc, char **argv)
{
  int i, ii, limit;
  volatile int *flag;
  double beg, end, time;
  flag = tolocal (&( idata[MYPROC][0]) ) ;
  ival = 1;
  if (MYPROC>0){
        while  ( *flag != ival )     /* local var empty ? */
                am_poll();
 }
 iptr = (int *spread)idata[ (MYPROC+1)%PROCS ];
 *iptr := ival;                    /* this is the put  */
}
```

The Analysis of Two Split-C Communication Functions

How the communication functions work was already shown in Chapter 3. Recall that a put involves both a request and reply, so when the message is sent over the network to the other processor, the receiver sends back an acknowledgment to the initiator. This gives a total of two messages for a single put. A get works in a similar way. A store does not send an acknowledge, and is therefore faster.

Diagrams 5.4 and 5.5 compare the output of a non-profiled versus a profiled program on four processors with different buffer sizes. First, a put is presented as an example of a command that creates two messages, followed by a store, which only generates two messages.

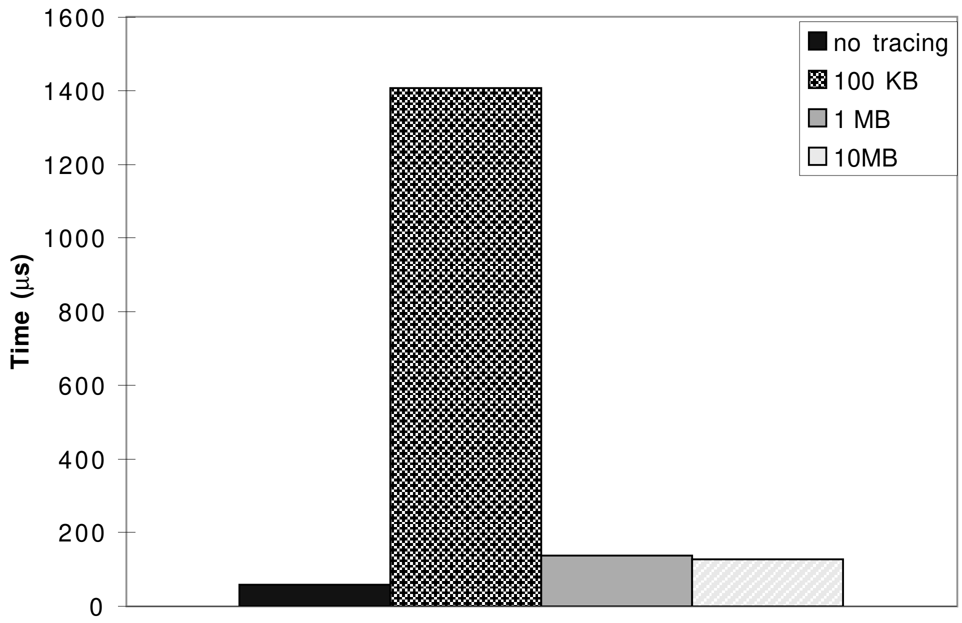

Figure 5.4: Staircase Put on 4 Processors

5.2.3 Measuring Split-C Routines Using Statistical Timing

The third test set is a complete testing of each individual Split-C routine. This uses statistical timing, which ensures a minimum of measure errors. In the previous two test programs, a test was run a fixed number of (for example a thousand) times, and the time is averaged over all the runs. With the approach now described, the timing is done statistically. It works as follows: Initially, 30 iterations are computed, to satisfy the Central Limit Theorem. Then, after each execution of the program, the time is measured, and the average time of all the runs taken. Measurements that differ substantially from the current average are discarded. If finally the times are close enough to the average, running is stopped. This cuts down the errors due to inconsistencies with the clock and the network to a minimum.

The program, called verify, is always used as an extensive test when Split-C is upgraded. Therefore, it is considered adequate for testing the Split-C profiler, too. Every Split-C routine is tested for each of the different data types it uses (char, int, double etc). Diagrams 5.6 to 5.12 compare a traced and a non-traced version on different routines using different number of processors and different buffersizes. The tracefiles are created on each processor and later concatenated to produce one single file. These diagrams cover the measurements of each Split-C routine, for each of the four different data types.

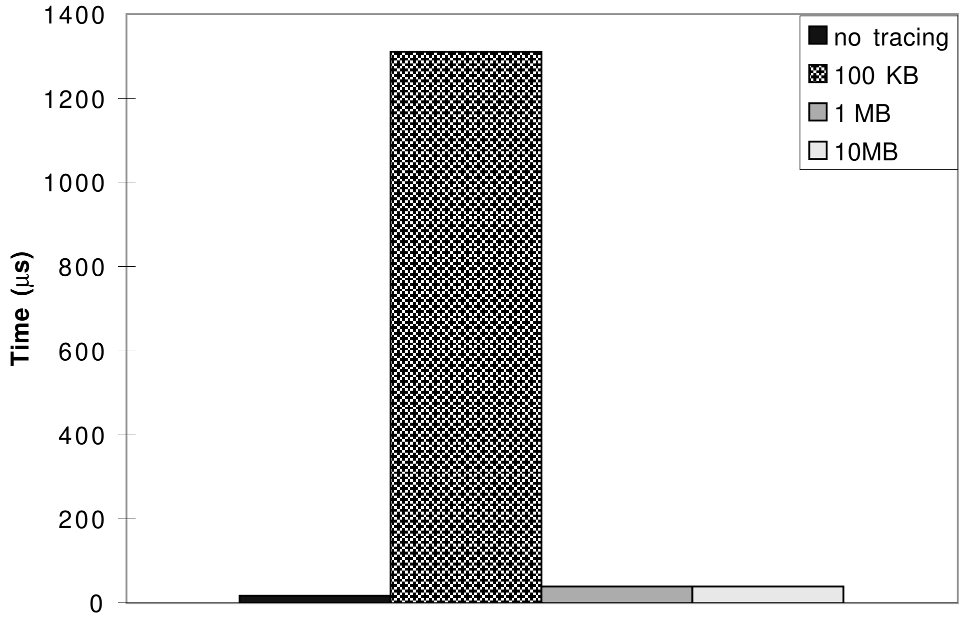

Figure 5.5: Staircase Store on 4 Processors

Fig. 5.6 shows the four different implementations of a store in Split-C. The overhead of the profiler is very small. A store runs for about 10 μs.

The calls to read, write, get, and put are acknowledging, and hence run slower than a store. Fig. 5.7 through 5.10 display the traced versus the nontraced routines.

In Split-C, there are three different types of global communication routines:

- Broadcast

  A broadcast sends a message from one processor to all the others.

- Reduce to One

  A reduce is an arithmetical function, that combines the values of all processors and stores the results on one processor.

- Reduce to All

  This is the combination of a broadcast and a reduce to one. The result is computed by combining all values together, and the result is sent to every other processor.

The reduce functions on a parallel machine are usually implemented for many arithmetical operations such as addition, multiplication and others. In Fig. 5.11, we present just one routine for each of the different types of the global communication.

In Fig. 5.12 we compare the performance of a barrier in a nonprofiled and in a profiled program. We see, that a barrier takes about twices as much time with tracing than without tracing.

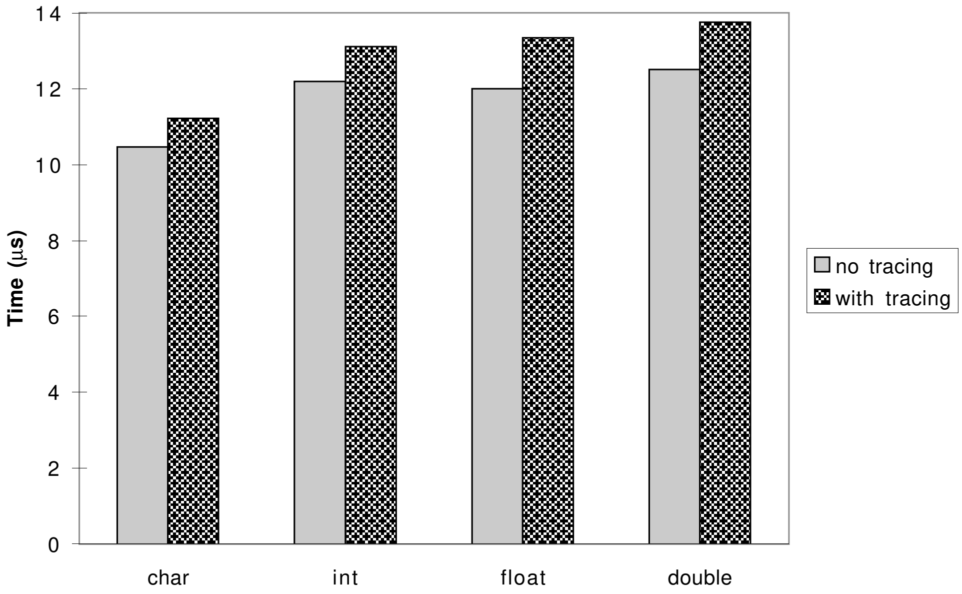

Figure 5.6: Store on 4 Processors

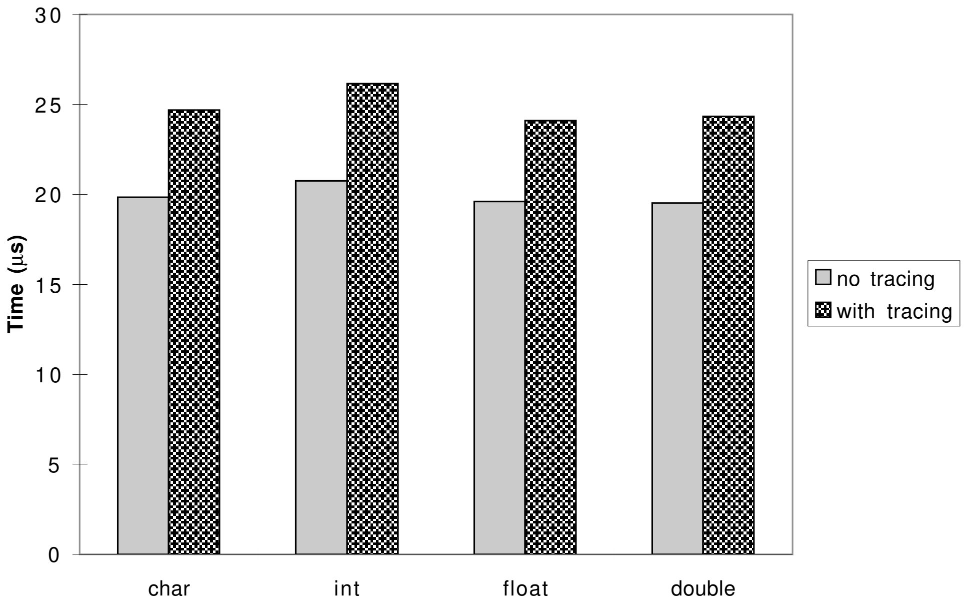

Figure 5.7: Get on 4 Processors

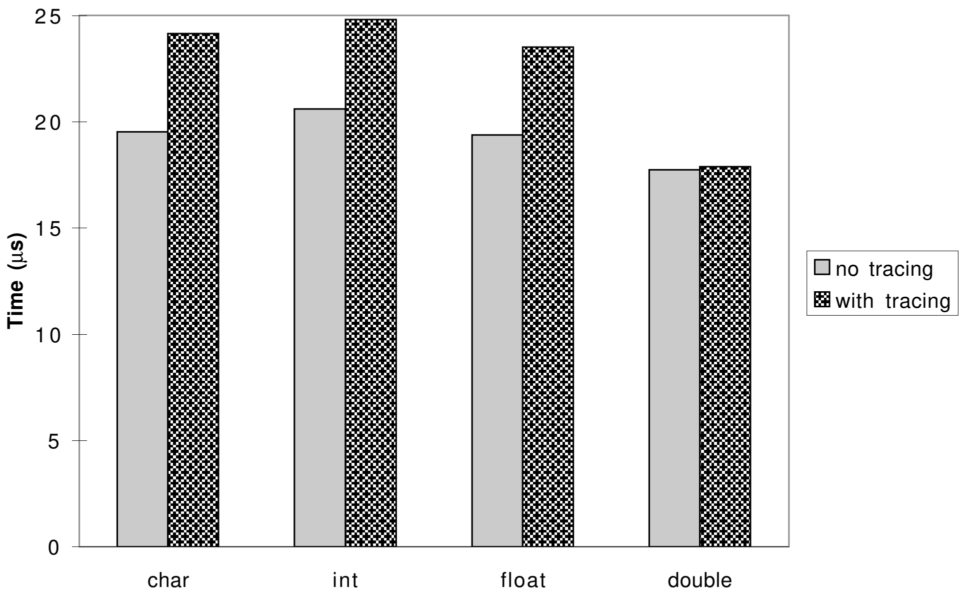

Figure 5.8: Put on 4 Processors

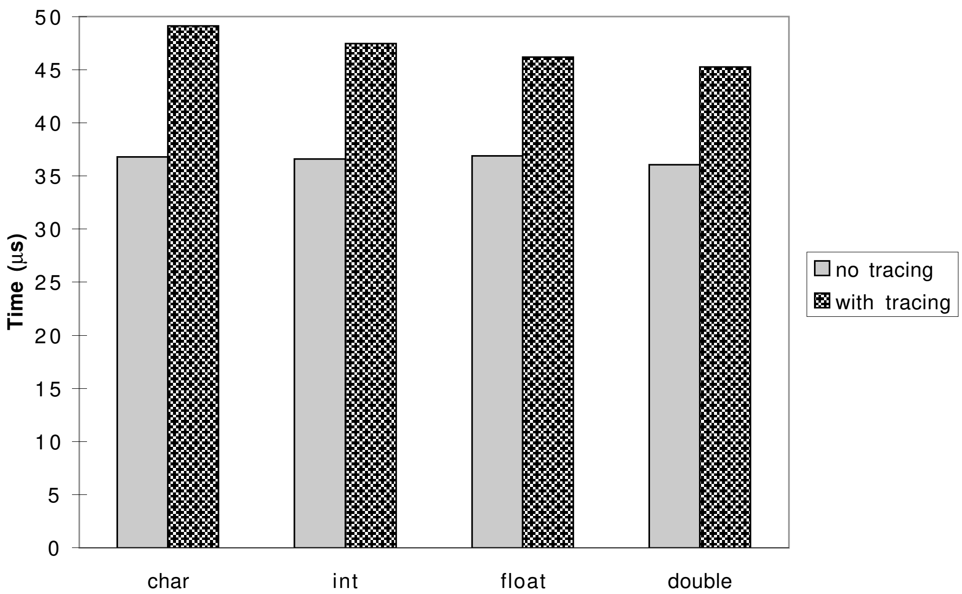

Figure 5.9: Read on 4 Processors

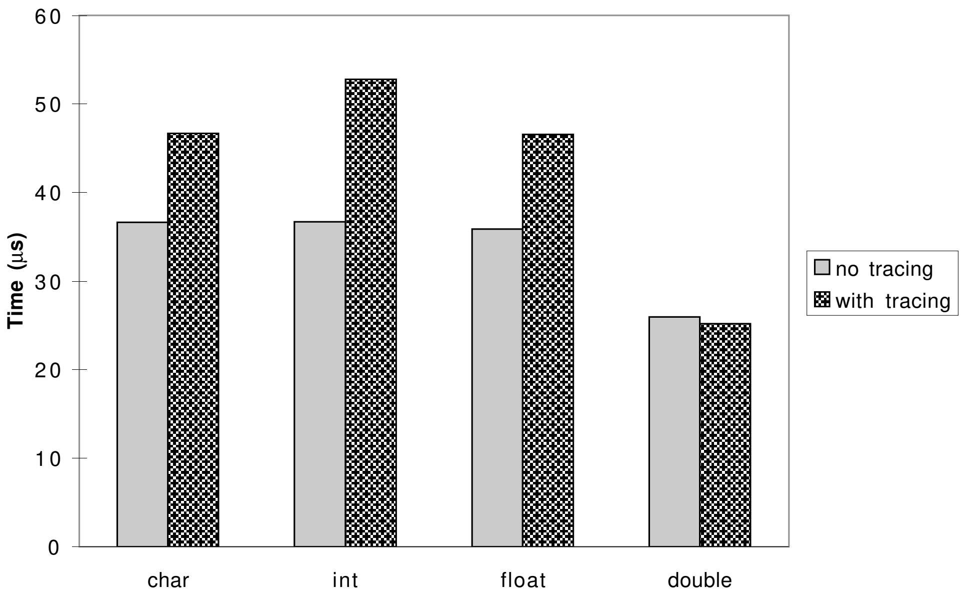

Figure 5.10: Write on 4 Processors

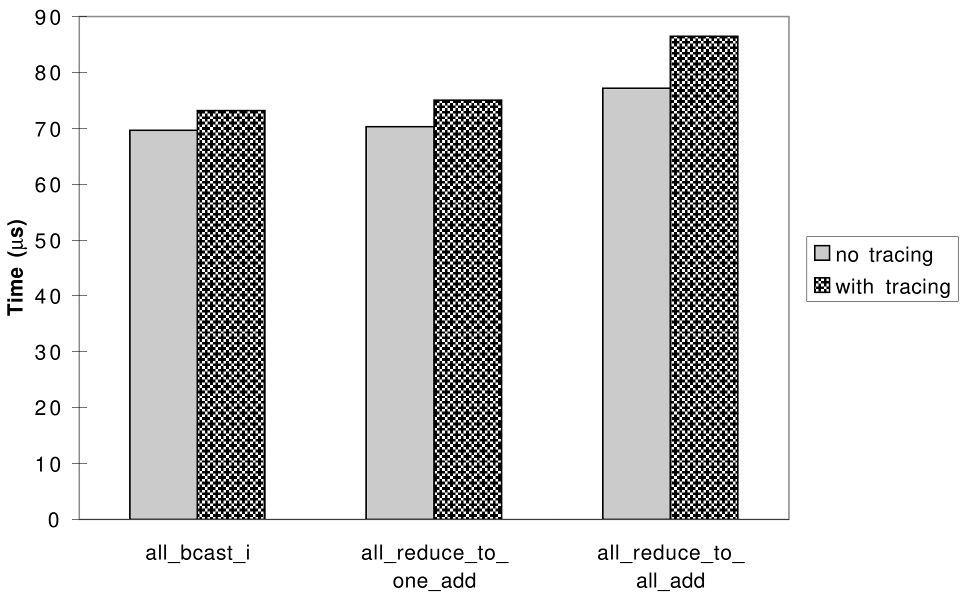

Figure 5.11: Global Communication on 4 Processors

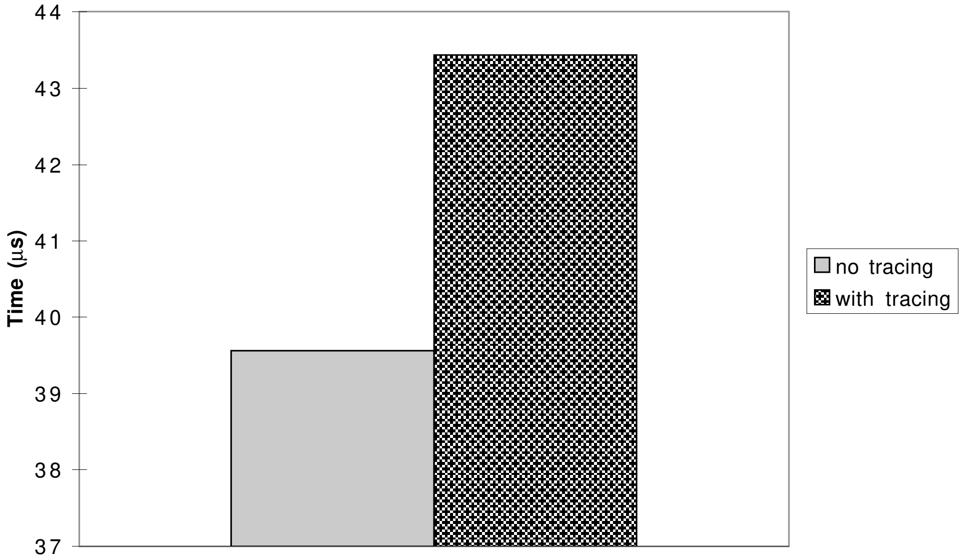

Figure 5.12: Barrier on 4 Procs

5.3 The Data Collection Strategy

There are two ways to produce the tracefiles.

- collect data on each processor separately
- collect data through one central processor

In the first case, each processor collects data on its own and buffers it. There are as many tracefiles produced as there are processors. Only after the program finishes execution are all the tracefiles are merged into one.

In the latter case, each processor collects the data and sends it to a central processor, which stores it into one single tracefile. Unfortunately, we realized a loss of data during the measurements. When the nodes write out tracedata to the central tracefile, tracelines are thrown away. That seems to be due to the fact of parallel I/O problems on the Meiko.

This is one of the reasons why the first approach is chosen. A second reason is that creating as many tracefiles as there are processors is faster than processing the whole data through one central processor.

It is of course necessary to account the time needed for concatenating all tracefiles together to the profiling overhead. For example, concatenating 16 tracefiles of about 20 MB to a total of 345 MB takes 3:10 minutes. The sorting algorithm we use takes linear time.

Figures 5.13 and 5.14 compare runs for different numbers of processors and different buffer sizes for a program performing put operations.

As mentioned earlier, a large buffer avoids writing out data too often, but writing out the buffer takes a long time. Figure 5.15 shows the impact that a large buffer can have, even if it is written only once, namely at the end of the execution. Each of the four programs were run on 32 processors, one time without any tracing, and three times with varying buffersizes.

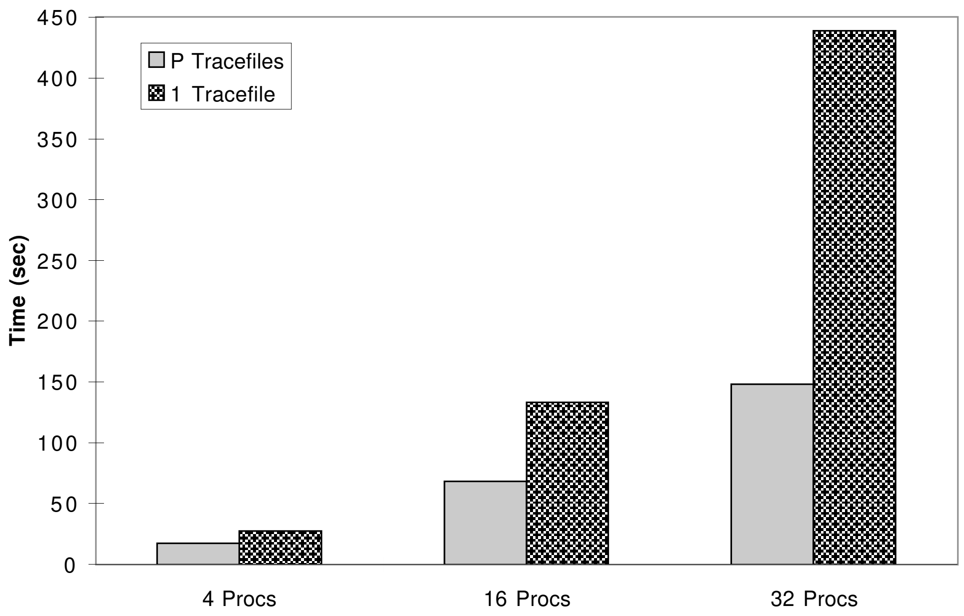

Figure 5.13: Put, Total Execution Time, Buffer: 100 KB

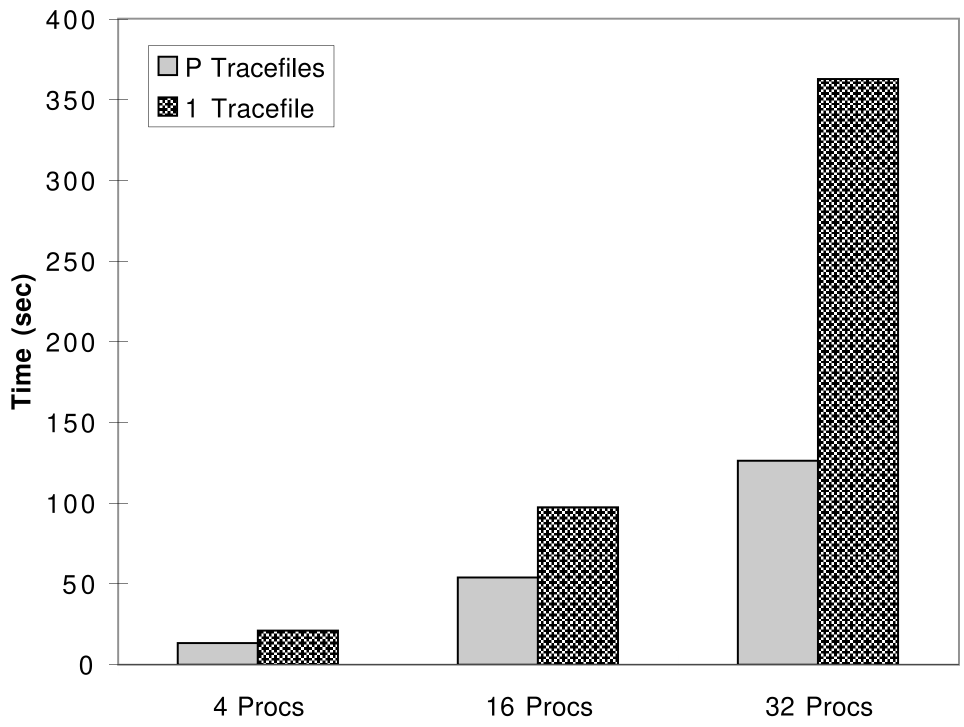

Figure 5.14: Put, Total Execution Time, Buffer: 10 MB

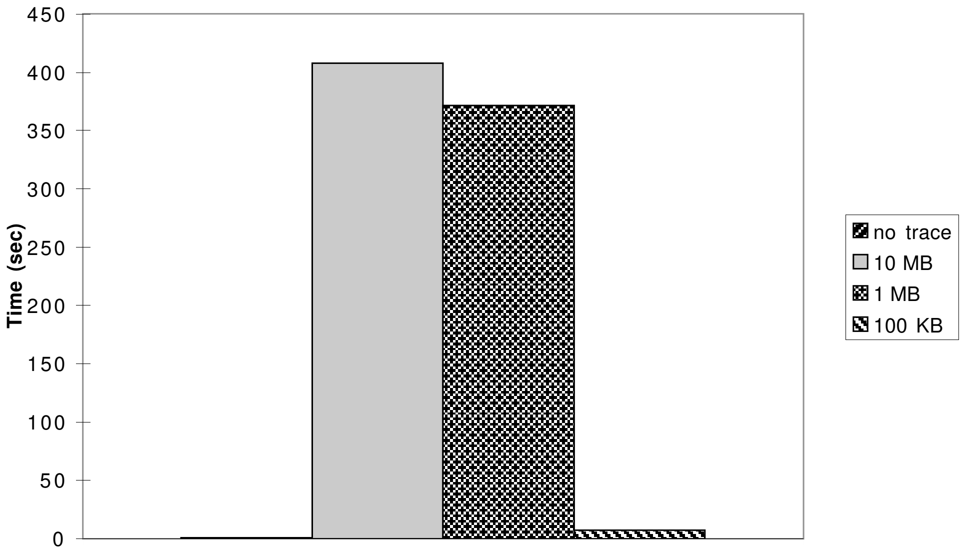

Figure 5.15: Flushing Time on 32 Procs, Varying Buffer Sizes

5.4 Measurements on Larger Applications

Testing 'real life' programs shows if the profiler performs in the expected way and if the profiler is usable in practice. We tested a lot of longer Split-C applications, that are available here at UC Santa Barbara. This section first provides information about the overhead of the profiler and how large the tracefiles are. Later, we will actually see how the Split-C profiler can be used to improve existing applications.

5.4.1 Overhead from the Profiler

Tables 5.2 and 5.16 compare the execution and the filesizes from the executables of a non-traced version against a traced version. Table 5.17 provides more details about the traced version in terms of the size of the produced tracefile and how much time the concatenation of all the tracefiles and the sorting takes. It is easy to see that it is not possible to give a certain factor of how much slower the programs perform when it is profiled. Of course, the execution time depends on how many trace entries the profiler generates. For example, fft only runs 5 seconds without tracing, but more than 2 minutes with tracing. There are a lot of communication routines executed, so the tracefile gets very large. The time consuming part is the flushing. So the larger the tracefile, the longer it takes to flush the file. We see that `fibonacci`, which takes nearly 10 minutes, has hardly any additional overhead. There are not many communication functions executed, so the tracefile stays fairly small.

Tracefile Sizes

A well known problem with offline profilers is the amount of disk space they need to store the profiling data. Table 5.3 shows the size of the generated trace files for some Split-C applications with various input parameters and number of processors. We see that programs

Program	Description	Without Tracing	With Tracing
mm	matrix multiply	3.208 s	6.985 s
cannon	Cannon matrix multiply	2.519 s	5.192 s
em3d	Electromagnetic wave simulation	0.570 s	7.187 s
fft	FFT using small transfers	4.959 s	129.138 s
fft_bulk	FFT using bulk transfers	1.503 s	3.723 s
pi	Approximating pi	0.527 s	4.583 s
wator	N-body simulation	109.078 s	421.397 s
shellsort	Shell sort	0.719 s	15.638 s

Table 5.2: Execution times for traced and untraced Split-C applications.

Program	Problem Size	4 Processors	16 Processors
FFT	1K points	105 KB	161 KB
	10K points	960 KB	1.25 MB
	100K points	9.4 MB	11.6 MB
Radix	1K numbers/proc	2.6 MB	12 MB
	2K numbers/proc	3.3 MB	15.8 MB
	4K number/proc	4.8 MB	23.5 MB
Bulk Radix	1K numbers/proc	233 KB	1.4 MB
	2K numbers/proc	233 KB	1.4 MB
	4K number/proc	233 KB	1.4 MB
Wator	16 particle, 64 iterations	800 KB	4.2 MB
	32 particles, 82 iterations	2 MB	10 MB

Table 5.3: Tracefile sizes for Split-C applications for various problem sizes.

such as FFT, radix, and wator, which use fine-grained communication, generate huge tracefiles even for small input sizes. This is particularly true for the simulation program wator, which produces a 10 MB trace file when modeling only 32 particles for a relatively short number of iterations. Note that the bulk radix program generates trace files that are functions of the number of processors, not the input size. It uses bulk transfers, and although the amount of transferred data increases the number of messages does not. As in all of the programs, however, increasing the number of processors increases the size of the trace files.

Executable Sizes

A comparison of the size of the executable files show is given in Table 5.16 We see, that most of the executables of the applications have a size of 700 to 800 KB. The profiled versions of these applications add an overhead of about 30 to 40 KB.

Interesting aspects in terms of the tracefiles are: How large is the tracefile, how long does it take to concatenate them together, and how long does it take to sort. Table 5.17 provides these details. Note that 'fast' refers to the fact that the command finished within a few

Name of the Program	Without Tracing	With Tracing
mm	769 KB	804 KB
cannon	767 KB	803 KB
em3d	788 KB	824 KB
fft	731 KB	767 KB
fft_bulk	746 KB	783 KB
pi	702 KB	738 KB
wator	770 KB	813 KB
shellsort	767 KB	803 KB
fibonacci	715 KB	751 KB

Figure 5.16: Comparing Filesizes from Traced and Nontraced Split-C Applications

Name of the Program	Tracefile Size	cat (in min)	sort (in min)
mm	39 KB	fast	fast
cannon	39 KB	fast	fast
em3d	340 MB	fast	2:38
fft	120 MB	2:05	??
fft_bulk	125 KB	fast	fast
pi	8 KB	fast	fast
wator	130 MB	2:02	??
shellsort	293 MB	fast	1:53
fibonacci	4 KB	fast	fast

Figure 5.17: Tracefilesizes and Sorting Times for Traced Split-C Applications

seconds and a '??' means that due to the size it was not possible to sort the tracefile with the UNIX `sort` command.

5.4.2 Using the Profiler for Peformance Improvements

We conclude with example trace outputs for some of the Split-C applications. We are interested in how the Split-C profiler can be used to increase performance on larger programs. The first application that we tested is a bulk fft program. The profiler shows, that this program works very good already. The second program is wator, a particle simulation program. This program has some bottlenecks which can be detected with the Split-C profiler

FFT

We examine a Split-C program that computes the fast Fourier transform. The program runs in three phases: two compute phases that sandwich an all-to-all communication phase, which performs a cyclic-to-blocked remap. ParaGraph output for the communication phase is shown in Figure 5.18, where the communication consists of fine-grained store operations. We see that that all of the processors have a large communication overhead (the light gray region in the lower window). Since there is no computation in this large overhead region, we know that none of the data is immediately used, and we can optimize the communication by combining short messages into longer bulk transfers. Of course, one does not need a profiler to discover this optimization, but this does show how a profiler can help even without knowledge of the underlying algorithm.

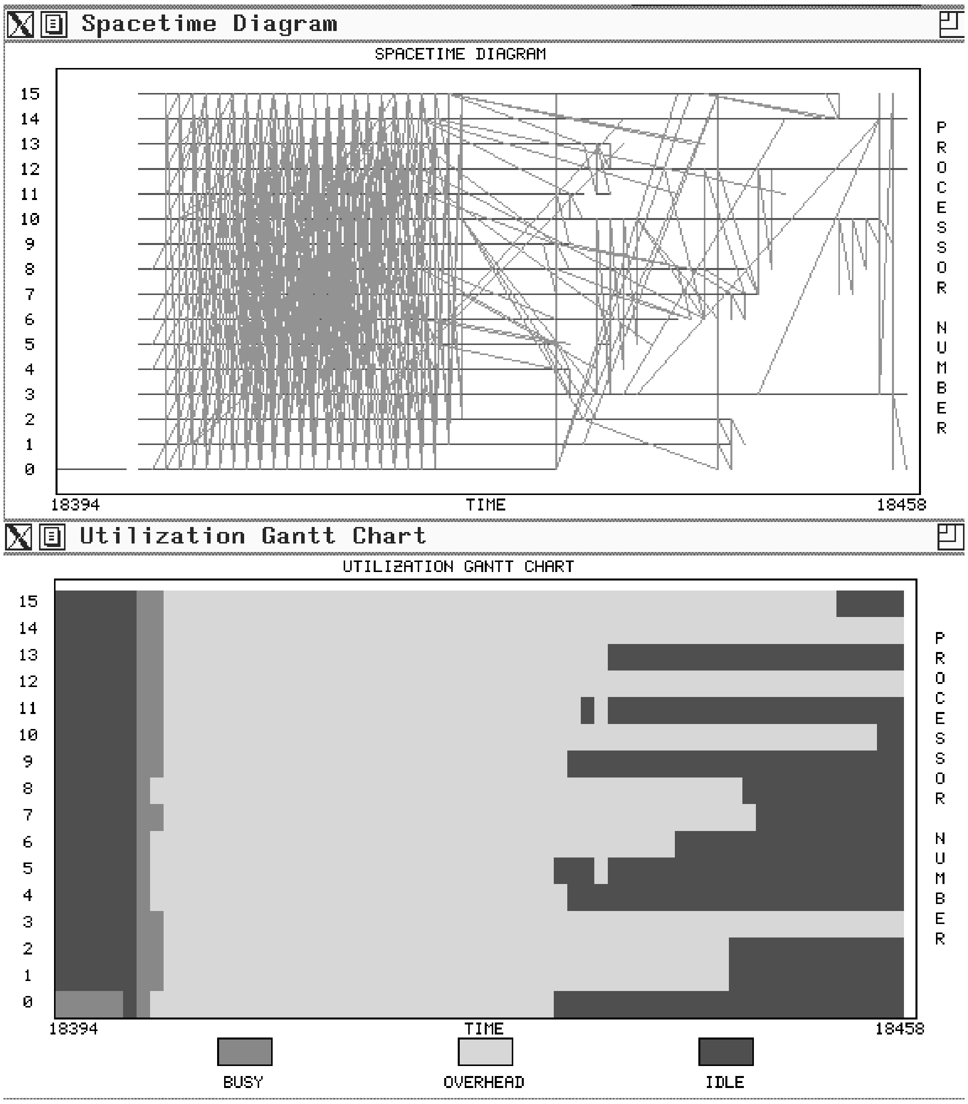

Figure 5.18: ParaGraph output for the Split-C program FFT, using fine-grained store operations, on 16 processors of the Meiko CS-2.

ParaGraph output for the communication phase (and the last computation phase) of the optimized FFT algorithm is shown in Figure 5.19. We see that the communication phase actually consists of a number of steps, each of which communicates (as shown in the lower window), computes (light gray area in the upper window), then synchronizes (dark gray area). The "computations" within these phases are actually just local memory copies: moving data transferred by a bulk_store into its proper place in memory. The second computation phase of the FFT is comparatively small: it is just the right-most light gray region. Figure 5.19 also shows that the profiler accurately depicts the communication pattern found in this program, though in this case one must have an understanding of the underlying code to make sense of it.

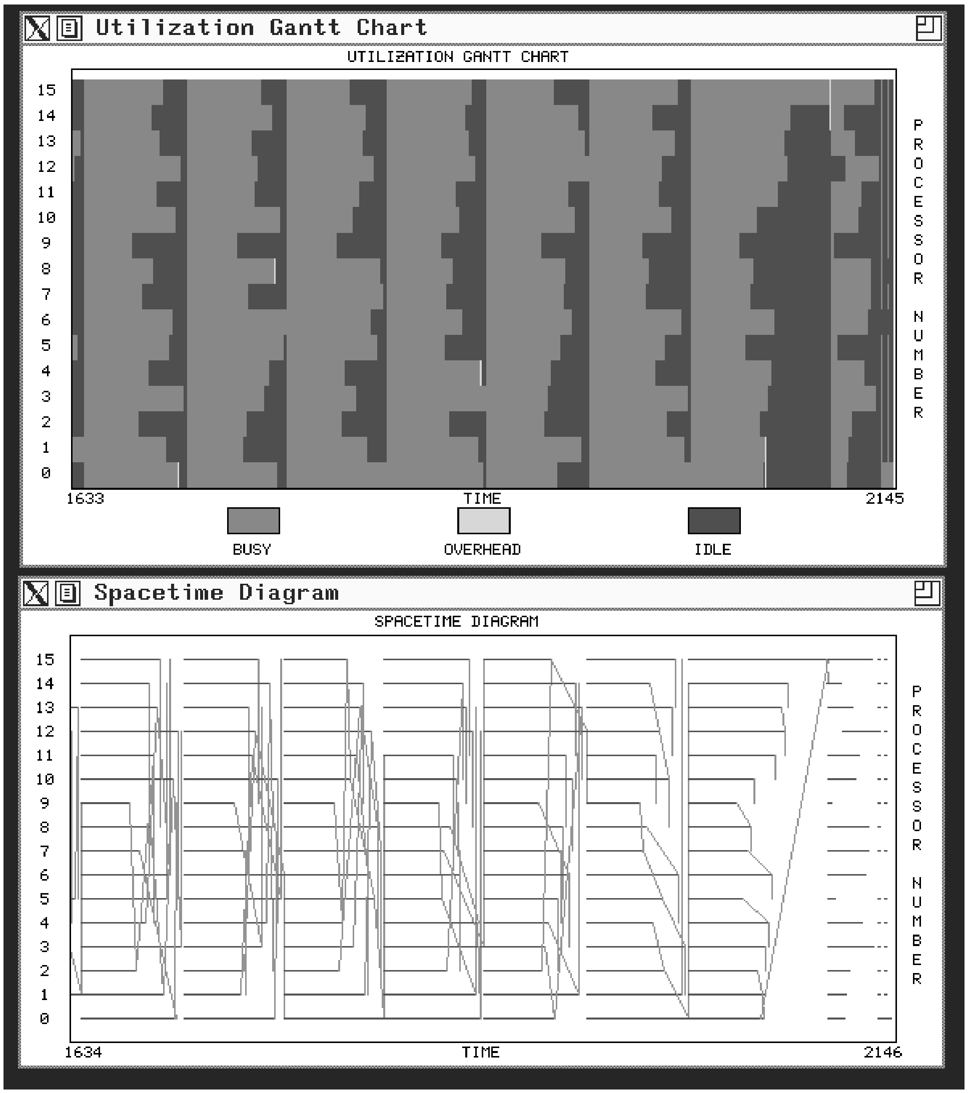

Figure 5.19: ParaGraph output for the optimized Split-C program FFT, based on bulk transfers, on 16 processors of the Meiko CS-2.

Wator

We now examine the Split-C program wator, which is a particle simulation program. In this program, every processor is responsible for a fraction of the particles; it computes the new position of a particle based on the positions and velocities of all of the other particles. So the heart of the program is a read–compute loop which acquires information about other particles, then computes new locations and velocities. Another important component of this program is the display of the particles, which is done every 10th iteration. This is done by processor 0 using X Window library calls. Figure 5.20 shows the ParaGraph output for the original wator program. We can see from the utilization window in this figure that most of the processors have long idle sections while processor 0 is busy computing: These are the points were processor 0 is displaying the data (X library calls involve I/O and are very slow).

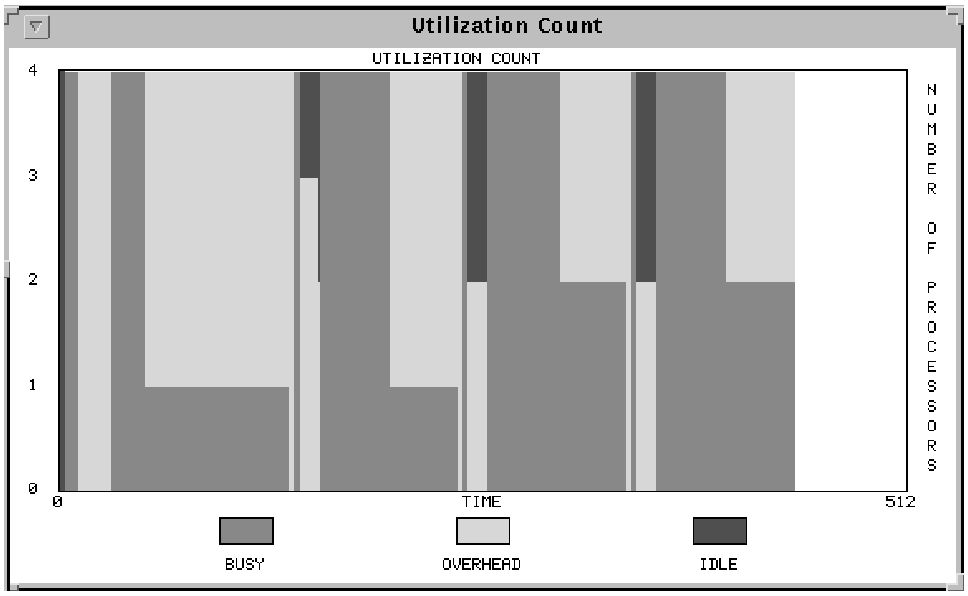

Figure 5.20: ParaGraph output for the Split-C program wator, on 16 processors of the Meiko CS-2.

Figure 5.21 shows another view of this. It displays the communication pattern between the processors. We see that most communication occurs in very short segments. The vertical lines show that the messages are received almost as fast as they are sent. However, there are some other diagonal lines extending from processors 1, 2, and 3 to processor 0. These are requests for data that processor 0 does not respond to until it has finished its display computation. The problem is that processor 0 does not poll for messages during this time.

This is quantified in Table 5.4, which shows the range of times processors must wait while a read request is serviced. As the data shows, the average waiting time for a read request is 150–300 μs on 4 processors, and 600–800 μs on 16 processors. To reduce the times, polls must be added. One way to do this on the Meiko CS-2 is to have the communications co-processor do the polling and service read requests. We have designed another Split-C implementation using this approach [SSFK96], and the optimized numbers are shown in the right most column of the table. They substantially reduce the wait times to 23–33 μs.

By adding some polls on the main processor, we obtain the communication pattern shown in Figure 5.22. In this case, we see that processor 3 is now able to get its reply in half the previous time (the reply is the vertical line from processor 0 to processor 3). In this example, however, processors 1 and 2 are still not able to get their requests answered by processor 0. This is because only a few polls were added to the code. By adding more polls to processor 0's main display compute loop, we are able to speed up the program by 20%. Thus, ParaGraph

	Wait times for Read Request	
	Main	Co-proc
4 processors	144-288 μs	23–24 μs
16 processors	600-800 μs	25–33 μs

Table 5.4: Time (in μs) spent waiting for a read request to complete in the wator program. The range of times represents the wait time over all processors. Results are shown for 4 and 16 processors, under both the co-processor and main processor implementations of Split-C.

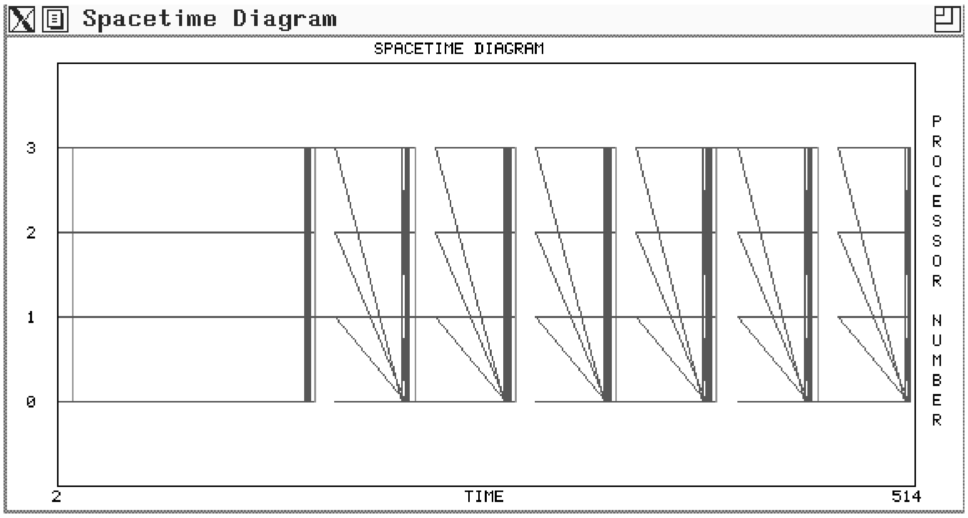

Figure 5.21: Communication pattern for wator particle simulation program. This version has no polls during the computation phase.

provided us with the insight to improve our program. It showed us that some messages were taking a long time in getting a reply, and we were able to insert polls to correct this problem.

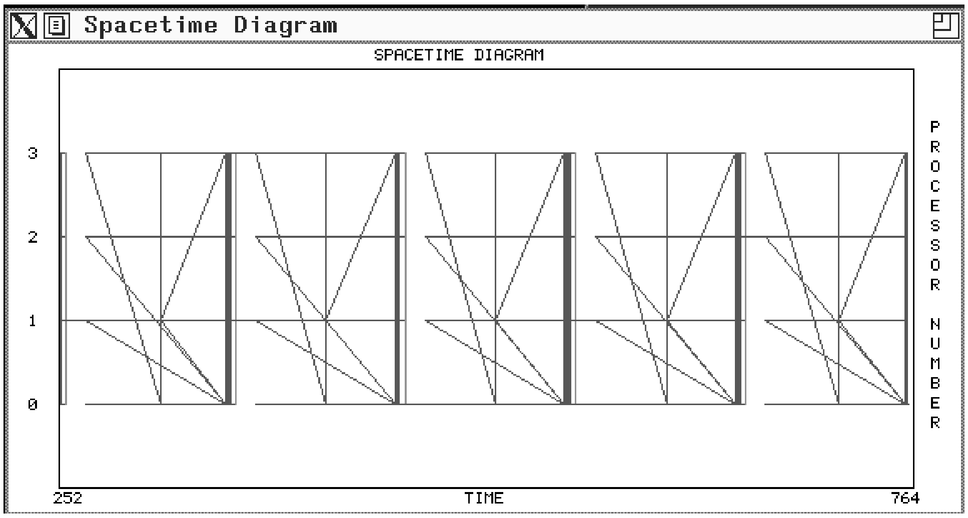

Figure 5.22: Communication pattern for wator particle simulation program. This version has some polls during computation phase.

On the other hand, a 20% improvement is not nearly as good as the improvement found in the co-processor implementation measured in Table 5.4. The reason is that it is difficult to insert polls in the X window I/O routines that are doing most of the computation on processor 0. This would require modifying the X library and system call routines. Another alternative is to change the structure of the wator program. Since the problem is that most of the processors have trouble communicating while processor 0 is busy computing, we will separate the communication and computation phases by adding a barrier. The resulting communication pattern is shown in the upper window of Figure 5.23. As expected, all communication occurs only during non-display phases, and all messages are handled very fast (i.e. the vertical lines). The lower window of Figure 5.23 shows the utilization of the processors for this modified program. Comparing it to Figure 5.20, we see that processors 1,2, and 3 are no longer in overhead time (waiting for an arriving message), but are in idle time (waiting in a barrier).

For a small number of processors and particles, such as those used in these trace runs (4 processors and 16 particles), this modified version of the program runs at the same speed or even slower than the original. The extra barrier just adds overhead. But for more processors and more particles (e.g., 200 particles on 16 processors), there is a large improvement in the run times (up to 400% faster on 16 processors).

Unfortunately, trace files generated for runs with more than a few particles become very large (on the order of megabytes), and ParaGraph is very slow in processing these. A programmer examining only more manageable tracefiles might not consider the barrier improvement discussed above, since it only helps for larger runs, and its benefits are not apparent for small problem sizes.

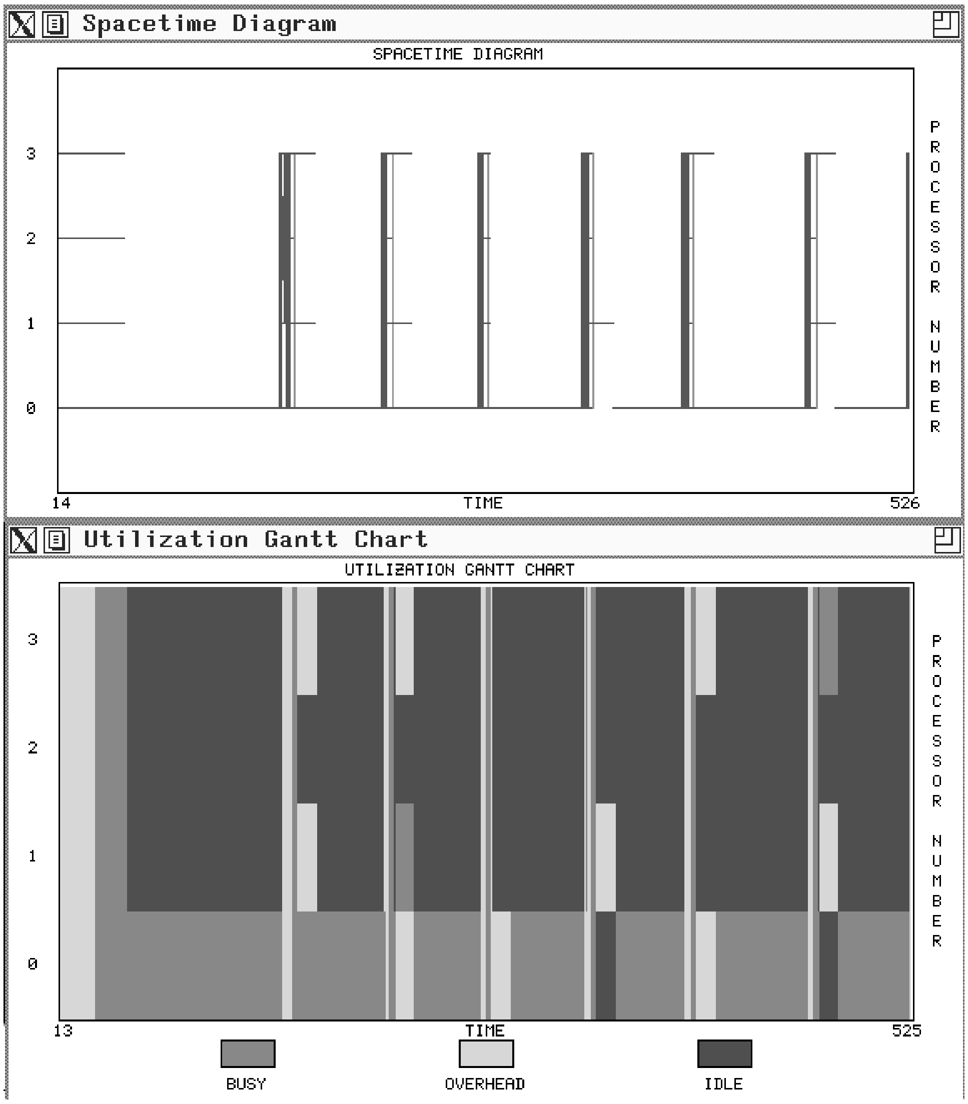

Figure 5.23: Communication and utilization patterns for the wator particle simulation program. This version has a barrier between the communication and computation phases

Chapter 6

Conclusions and Future Work

In this thesis we have presented a profiler for the parallel programming language Split-C. Because parallel algorithms are difficult to implement efficiently, we need some tool that provides the user with details about the performance of the program. A profiler supplies this information by collecting data about the communication and computation time and by displaying it graphically so that the user can watch and analyze the execution of his program. A convenient graphical interface is crucial to the effectiveness of the profiler. The better it displays the collected data, the easier it is for the user to locate bottlenecks, which are responsible for poor performance in the program.

This thesis first studied different ways of profiling and summarized existing tools. The the chosen Then a profiler for Split-C was implemented based on the offline approach. In the end, extensive measurements were performed on Split-C routines, as well as larger applications, and these measurements were used to verify that the profiler works correctly, to see how large the overhead is. In the end, we tried to increase performance on some of the larger applications using the Split-C profiler. We were able to achieve this goal.

There are two ways of implementing a profiler: an online approach, where data is displayed immediately during program execution, and an offline approach, which first collects the data and then postprocesses it using a graphical displaying tool. We have chosen to implement the offline approach. Its disadvantage of producing large amounts of data is balanced by the fact that it does not have any overhead for the communication between the data collector and the data display server during program execution. It does not have to deal with the very expensive insertion and removal of tracecalls during run time. Furthermore, the trace data can be examined repeatedly without the need to rerun the user's program.

At Oak Ridge National Laboratory, the PICL library was developed as an architecture independent way for profiling Message Passing Interface (MPI) programs. We modified the PICL library to match the Split-C requirements. When adding this Split-C specific code we preserved the architecture independence, thus the profiler can be ported to any platform. Our version of the profiler is implemented on a parallel machine, the Meiko CS-2 with 64 nodes.

After the implementation, extensive measurements on the Split-C routines were made using both, microbenchmarks and real applications. By just looking at the raw times, it seems that the overhead is very high. But most programs overlap computation with communication. This decreases the overhead for collecting the tracedata. The only very expensive part is flushing the buffer to secondary storage. The Split-C routine measurements were also used to examine the correctness of the produced traces. This is done by analyzing the traces with ParaGraph. Since we know what the different functions should do, we can check that the profiler is working correctly.

The last and most important measurements include the analysis of larger applications. This is necessary to see how the profiler performs with real programs, to see what is happening, and when the tracefiles get really large.

Large tracefiles are a general problem to the offline monitoring approach. Our Split-C profiler is no exception to this rule. Depending on how many communication routines are executed, tracefiles can easily reach sizes of several hundred megabytes. However, for smaller tracefiles this profiler is a useful tool for analyzing Split-C programs.

As this is the first version of the implementation, there are still optimizations possible to increase performance and decrease the overhead. One of them is to eliminate successive calls to the timer. These have to be done to satisfy the mapping to the send-receive model. In some parts of the library, they follow directly on each other, thus one call to the start timer function could be saved.

If the user wants to get a more specific trace of certain functions, he can add trace event calls. Trace event calls will collect more detailed data for the part of the code they surround (there is always an event entry and an event exit). A possible way to make the trace event calls a little more convenient is to have them inserted automatically for the most time consuming functions. Gprof could be used to determine these functions.

The problem with a Split-C profiler based on PICL is due to the fact that PICL is written for send-receive protocols. These protocols are very slow, and the messages generated run some hundred microseconds. As Split-C is based on active messages, it is much faster. The elapsed times between different actions are much shorter (e.g. microseconds). Furthermore, active messages are asynchronous. A processor can send a message, then continue its work. The matching receive from the handler may arrive later, with other actions taking place in the mean time. This is harder to read with ParaGraph, the graphical interface used to analyze the tracefiles. Thus, it is much harder to profile Split-C than a language based on a simpler send-receive protocol.

The implementation for the Split-C profiler associates different events with different keys. ParaGraph basically only knows three different keys, for send, receive, and idle events. The goal for the Split-C profiler should be to integrate the integrate more keys, so that different events can be distinguished. For example it would be nice to see if a communication function is due to a get or put. Incorporating new keys should be done carefully, though, because introducing too many keys will make the output confusing.

The experimental analysis of the Split-C profiler showed us that it is a useful tool for helping us to increase performance on larger applications. While there are still possible improvements for the profiler, especially to the graphical interface, this thesis has provided a valuable tool for analyzing and improving parallel programs. We showed, that the profiler can be used to speed up programs. One of the examined applications executed four times faster after the profiling.

Appendix A

Split-C Key Mapping Table

Tables A.1 to A.5 describe the mapping from the Split-C keys to the numbers that ParaGraph can understand. The Split-C keys are included in the Split-C library, the mapping is done within the PICL library.

BULK_STORE_KEY	-2000
STORE_KEY	-2001
BULK_READ_KEY	-2002
BULK_GET_CTR_KEY	-2003
GET_KEY	-2004
BULK_PUT_CTR_KEY	-2005
BULK_PUT_KEY	-2006
BULK_WRITE_KEY	-2007
PUT_KEY	-2008
WRITE_KEY	-2009
BULK_GET_KEY	-2010
GET_CTR_KEY	-2011
PUT_CTR_KEY	-2012

Figure A.1: Split-C Keys for send related commands

KEY_all_reduce_to_one_add	-2013
KEY_all_reduce_to_one_uadd	-2014
KEY_all_reduce_to_one_fadd	-2015
KEY_all_reduce_to_one_dadd	-2016
KEY_all_reduce_to_one_mult	-2017
KEY_all_reduce_to_one_umult	-2018
KEY_all_reduce_to_one_fmult	-2019
KEY_all_reduce_to_one_dmult	-2020
KEY_all_reduce_to_one_max	-2021
KEY_all_reduce_to_one_umax	-2022
KEY_all_reduce_to_one_fmax	-2023
KEY_all_reduce_to_one_dmax	-2024
KEY_all_reduce_to_one_min	-2025
KEY_all_reduce_to_one_umin	-2026
KEY_all_reduce_to_one_fmin	-2027
KEY_all_reduce_to_one_dmin	-2028
KEY_all_reduce_to_one_or	-2029
KEY_all_reduce_to_one_uor	-2030
KEY_all_reduce_to_one_xor	-2031
KEY_all_reduce_to_one_uxor	-2032
KEY_all_reduce_to_one_and	-2033
KEY_all_reduce_to_one_uand	-2034
KEY_all_bcast	-2035
KEY_all_bcast_i	-2036
KEY_all_bcast_u	-2037
KEY_all_bcast_f	-2038

Figure A.2: Split-C Keys for global reduction

KEY_all_bcast_d	-2039
KEY_all_reduce_to_all_add	-2040
KEY_all_reduce_to_all_uadd	-2041
KEY_all_reduce_to_all_fadd	-2042
KEY_all_reduce_to_all_dadd	-2043
KEY_all_reduce_to_all_mult	-2044
KEY_all_reduce_to_all_umult	-2045
KEY_all_reduce_to_all_fmult	-2046
KEY_all_reduce_to_all_dmult	-2047
KEY_all_reduce_to_all_max	-2048
KEY_all_reduce_to_all_umax	-2049
KEY_all_reduce_to_all_fmax	-2050
KEY_all_reduce_to_all_dmax	-2051
KEY_all_reduce_to_all_min	-2052
KEY_all_reduce_to_all_umin	-2053
KEY_all_reduce_to_all_fmin	-2054
KEY_all_reduce_to_all_dmin	-2055
KEY_all_reduce_to_all_or	-2056
KEY_all_reduce_to_all_uor	-2057
KEY_all_reduce_to_all_xor	-2058
KEY_all_reduce_to_all_uxor	-2059
KEY_all_reduce_to_all_and	-2060
KEY_all_reduce_to_all_uand	-2061
KEY_all_bcast_buffer	-2062
KEY_com_init	-2063

Figure A.3: Split-C Keys for global reduction

ALL_SCAN_ADD_KEY	-2064
ALL_SCAN_UADD_KEY	-2065
ALL_SCAN_FADD_KEY	-2066
ALL_SCAN_DADD_KEY	-2067
ALL_SCAN_MULT_KEY	-2068
ALL_SCAN_UMULT_KEY	-2069
ALL_SCAN_FMULT_KEY	-2070
ALL_SCAN_DMULT_KEY	-2071
ALL_SCAN_MAX_KEY	-2072
ALL_SCAN_UMAX_KEY	-2073
ALL_SCAN_FMAX_KEY	-2074
ALL_SCAN_DMAX_KEY	-2075
ALL_SCAN_MIN_KEY	-2076
ALL_SCAN_UMIN_KEY	-2077
ALL_SCAN_FMIN_KEY	-2078
ALL_SCAN_DMIN_KEY	-2079
ALL_SCAN_OR_KEY	-2080
ALL_SCAN_UOR_KEY	-2081
ALL_SCAN_XOR_KEY	-2082
ALL_SCAN_UXOR_KEY	-2083
ALL_SCAN_AND_KEY	-2084
ALL_SCAN_UAND_KEY	-2085

Figure A.4: Split-C Keys for global reduction

SYNC_KEY	-2086
IS_SYNC_KEY	-2087
SYNC_CTR_KEY	-2088
ALL_STORE_SYNC_KEY	-2089
ALL_STORE_SYNC_CTR_KEY	-2090
STORE_SYNC_KEY	-2091
STORE_SYNC_CTR_KEY	-2092
IS_STORE_SYNC_KEY	-2093
IS_STORE_SYNC_CTR_KEY	-2094
BARRIER_KEY	-2095
RECV_KEY	-2096
ATOMIC_KEY	-2097
ATOMIC_I_KEY	-2098
ATOMIC_D_KEY	-2099
ALL_ATOMIC_SYNC_KEY	-2100
ATOMIC_LONG_KEY	-2101

Figure A.5: Split-C Keys for receive, sync, barrier, atomic

Appendix B

Sample Tracefile Output from Split-C

Here, we present a sample output from a Split-C tracefile. We start by describing the structure of an entry line in a tracefile. The first colomn specifies if the event is a function entry (-3) or exit (-4). The second value provides the type of the event as described in Appendix A. The third colomn displays the timestamp, followed by the processornumber the command was issued from. The number of the remaining fields varies, providing the information of how long a message is and to which processor the message is sent to (last colomn).

The original program is doing a ringwise sending of messages, e.g. each processor sends a message to its successor and this send is only possible when a message has been received. The tracefile shows the barriers (key 2095) and the command `put` (key 2012) performed on each processor. The receive is denoted by the key 2096 and key 2008 is for acknowledging the `put`.

```
-3 -2095 -0.951663 2 0 0
-3 -2095 -0.940972 0 0 0
-3 -2095 -0.930234 1 0 0
-3 -2095 -0.920531 3 0 0
-4 -2095 -0.918831 0 0 0
-4 -2095 -0.918831 3 0 0
-4 -2095 -0.918830 2 0 0
-4 -2095 -0.918829 1 0 0
-3 -2012 -0.918782 0 0 3 2 4 0 1
-4 -2012 -0.918778 0 0 0
-3 -2095 -0.918762 0 0 0
-3 -2096 -0.918749 1 0 1 2 0
-4 -2096 -0.918746 1 0 3 2 4 0 0
-3 -2008 -0.918739 1 0 3 2 4 0 0
-4 -2008 -0.918736 1 0 0
-3 -2012 -0.918705 1 0 3 2 4 0 2
-3 -2096 -0.918705 0 0 1 2 0
-4 -2096 -0.918702 0 0 3 2 4 0 1
-4 -2012 -0.918701 1 0 0
-3 -2095 -0.918692 1 0 0
-3 -2096 -0.918679 2 0 1 2 0
-4 -2096 -0.918676 2 0 3 2 4 0 1
```

```
-3 -2008 -0.918669 2 0 3 2 4 0 1
-4 -2008 -0.918666 2 0 0
-3 -2012 -0.918638 2 0 3 2 4 0 3
-4 -2012 -0.918635 2 0 0
-3 -2096 -0.918630 1 0 1 2 0
-4 -2096 -0.918627 1 0 3 2 4 0 2
-3 -2095 -0.918626 2 0 0
-3 -2096 -0.918614 3 0 1 2 0
-4 -2096 -0.918611 3 0 3 2 4 0 2
-3 -2008 -0.918604 3 0 3 2 4 0 2
-4 -2008 -0.918602 3 0 0
-3 -2012 -0.918570 3 0 3 2 4 0 0
-4 -2012 -0.918566 3 0 0
-3 -2096 -0.918566 2 0 1 2 0
-4 -2096 -0.918563 2 0 3 2 4 0 3
-3 -2095 -0.918556 3 0 0
-3 -2096 -0.918542 0 0 1 2 0
-4 -2096 -0.918539 0 0 3 2 4 0 3
-3 -2008 -0.918534 0 0 3 2 4 0 3
-4 -2008 -0.918532 0 0 0
-4 -2095 -0.918502 2 0 0
-4 -2095 -0.918498 0 0 0
-4 -2095 -0.918498 1 0 0
-3 -2096 -0.918498 3 0 1 2 0
-4 -2096 -0.918494 3 0 3 2 4 0 0
-3 -2012 -0.918494 2 0 3 2 4 0 3
-4 -2012 -0.918491 2 0 0
-3 -2012 -0.918490 1 0 3 2 4 0 2
-3 -2012 -0.918489 0 0 3 2 4 0 1
-4 -2012 -0.918488 1 0 0
-4 -2012 -0.918487 0 0 0
-4 -2095 -0.918487 3 0 0
-3 -2095 -0.918487 2 0 0
-3 -2095 -0.918483 1 0 0
-3 -2095 -0.918482 0 0 0
```

Bibliography

[AG89] G. Almasi and A. Gottlieb. *Highly Parallel Programming.* Benjamin/Cummings, Redwood City, CA and others, 1989.

[Bra89] S. Brawer. *Introduction to Parallel Programming.* Academic Press, San Diego, CA, 1989.

[BSM95] R. Block, S. Sarukkai, and P. Mehra. Automated Performance Prediction of Message-Passing Parallel Programs. In *Proceedings SUPERCOMPUTING '95*, San Diego, CA, November 1995.

[C^+95] D. Culler et al. Introduction to Split-C. Technical report, UC Berkeley, 1995.

[CDG^+93] D. Culler, A. Dusseau, S. Goldstein, et al. Parallel Programming in Split-C. In *Proceedings SUPERCOMPUTING '93*, Portland, OR, November 1993.

[Fos95] I. Foster. *Designing and Building Parallel Programs.* Addison Wesley, Redwood City, CA, 1995.

[GKM82] S. Graham, P. Kessler, and M. McKusick. A Call Graph Execution Profiler. *Software-Practice and Experience*, 17(6):120–126, 1982.

[GKM83] S. Graham, P. Kessler, and M. McKusick. An Execution Profiler for Modular Programs. *Software-Practice and Experience*, 13(8):671–685, 1983.

[HE91] M. Heath and J. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, 1991.

[HM95] S. Hackstadt and A. Malony. Visualizing Parallel Programs and Performance. *IEEE Computer Graphics and Applications*, pages 12–14, 1995.

[JD94] Jr. J. DelSignore. Meiko CS-2 TotalView. Technical report, BBN, 1994.

[JL93] J. Jones and X. Li. A Performance Analysis Tool for Split-C Programs. Technical report, 1993.

[K^+94] V. Kumar et al. *Introduction to Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1994.

[KR88] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[KS93] E. Kraemer and J. Stasko. The Visualization of Parallel Systems: An Overview. *Journal of Parallel and Distributed Computing*, 18:105–117, 1993.

[LH91] E. Lusk and V. Herrarte. Studying Parallel Program Behaviour with Upshot. Technical report, Argonne National Laboratory, 1991.

[M+95] B. Miller et al. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[Mei93] Meiko, Waltham, Ma. *The Elan Communications Processor*, 1993.

[Pik93] G. Pike. The Design and Implementation of a Split-C Profiler. Term project, 1993.

[RJ93] D. Rover and C. Wright Jr. Visualizing the Performance of SPMD and Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:129–146, 1993.

[SM93] S. Sarukkai and A. Malony. Perturbation Analysis of High Level Instrumentation for SPMD Programs. *SIGPLAN Notices*, 28(7):44–53, 1993.

[SS95] K. Schauser and C. Scheiman. Active Messages Implementations for the Meiko cs-2. In *9th International Parallel Processing Symposium (IPPS'95)*, Santa Barbara, CA, April 1995.

[SSFK96] K. E. Schauser, C. J. Scheiman, J. M. Ferguson, and P. Z. Kolano. Exploiting the Capabilities of Communications Co-processors. In *10th International Parallel Processing Symposium*, April 1996.

[Uni94] University of Illinois, Urbana, Ill. *Pablo Instrumentation Environment User's Guide*, 1994.

[Var93] D.A. Varley. Practical experience of the limitations of gprof. *Software-Practice and Experience*, 23(4):461–463, 1993.

[vECGS92] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[Wor92] P. Worley. A New PICL Trace File Format. Technical report, Oak Ridge National Laboratory, 1992.

[WR95] A. Waheed and D. Rover. A Structured Approach to Instrumentation System Development and Evaluation. In *Proceedings SUPERCOMPUTING '95*, San Diego, CA, November 1995.