AN ABSTRACT OF THE THESIS OF

Anil Kumar Yadav for the degree of Master of Science in Electrical and Computer Engineering presented on October 26, 1989.

Title : Performance Monitoring of Parallel Applications at Large Grain Level

Redacted for privacy

Abstract approved : ______________________________

Prof. T. G. Lewis

This thesis is an attempt to create a methodology to analyze the performance of parallel applications on a wide variety of platforms and programming environments. First we determined the monitoring functions required to collect traces for accurate representation of the parallel application. We used the Extended Large Grain Data Flow (**E L G D F**) representation of an application to determine granularity and which monitoring functions should be inserted for sufficient feedback to application designer. The monitoring routines (real time clock access procedures) with a common interface were developed for the Sequent™ multiprocessor machine and the C-Linda programming environment . We also developed an Execution Profile Analyzer(**E P A**) for post-processing the traces. The **E P A** gives feedback to the mapping and scheduling (**TaskGrapher**) tool by providing actual performance data. These tools are being developed as a part of Parallel Programming Support Environment (**P P S E**) research . Results indicate that when actual grain execution time is made available to scheduling tools, accurate projections of program behavior are obtained.

Performance Monitoring of Parallel Applications at Large Grain Level

by

Anil Kumar Yadav

A THESIS

submitted to

Oregon State University

in partial fulfillment of

the requirements for the

degree of

Master of Science

Completed October 26, 1989

Commencement June 1990

APPROVED :

Redacted for privacy

Prof. T. G. Lewis, Electrical and Computer Engineering
in-charge of major

Redacted for privacy

Head of Department of Electrical and Computer Engineering

Redacted for privacy

Dean of Graduate School

Date thesis is presented October 26, 1989.

ACKNOWLEDGEMENT

I would like to take this opportunity to thank the staff and faculty of the Departments of Electrical and Computer Engineering and Computer Science for extending their constant support and encouragement.

In particular I am very grateful to my major professor Dr. T. G. Lewis for introducing me to the exciting field of parallel computing and Dr. Shreekant Thakkar of Sequent Computer Systems, Inc. Beaverton, Oregon, for his constant guidance and support.

I would also like to express my sincere thanks to Oregon Advanced Computing Institute, Beaverton, Oregon and Sequent Computer Systems Inc, for letting me have access to their valuable equipment for conducting the experiments.

I am also very thankful to the PPSE researchers and guests at the PPSE meeting on 2nd and 3rd October 1989. The presentation of this work and subsequent discussions made me view this thesis from a broader perspective.

And last but not the least I am thankful to my friends and my family members for making things easy for me.

Anil Kumar Yadav

TABLE OF CONTENTS

1	INTRODUCTION	1
2	AN APPROACH TO PERFORMANCE ANALYSIS	5
	2.0 Introduction	5
	2.1 Existing Analysis Tools	5
	2.2 Overview of P P S E	6
	2.3 Steps in Performance Analysis	10
	2.4 Instrumentation Experiments	13
3	TIMING ROUTINES	19
	3.0 Introduction	19
	3.1 Common Interface	19
	3.2 Implementation on Sequent™ machine	24
	3.3 Implementation using Linda-C	30
	3.4 Extensions to Timing Routines	31
4	EXECUTION PROFILE ANALYZER	37
5	ALL TOGETHER	46
6	CONCLUSIONS AND SUMMARY	58
	BIBLIOGRAPHY	62
	APPENDICES	
	Appendix A : Instrumented Code of Travelling Salesman Problem	65
	Appendix B : Source Code of Execution Profile Analyzer	74
	Appendix C : Instrumented Code generated by SuperGlue	95

LIST OF FIGURES

Fig. 2.1	Overview of P P S E	8
Fig. 2.2	Performance Analysis Methodology	11
Fig. 2.3	A generic Simulated Annealing Algorithm	14
Fig. 2.4	ELGDF representation of TSM	16
Fig. 2.5	Instrumentation Levels	17
Fig. 3.1	Grain Model and timing interface	22
Fig. 3.2	Trace file format	23
Fig. 4.1	Top level ELGDF of E P A	38
Fig. 4.2	Data structures of E P A	40
Fig. 4.3	Data structures of E P A	41
Fig. 4.4	Data Structures of E P A	42
Fig. 4.5a	General Performance Report	42
Fig. 4.5b	Actual Task Graph file	43
Fig. 4.5c	Actual Gantt chart file	43
Fig. 5.1	ELGDF representation of pi approximation problem	47
Fig. 5.2	Specifications of an arc	48
Fig. 5.3	Specification of a node	49
Fig. 5.4	Code fragment for node "start"	50
Fig. 5.5	Task graph at 'cold start'	50
Fig. 5.6	Execution Gantt Charts	51
Fig. 5.7	Task graph after first run	53
Fig. 5.8	Speed up projections with 'cold start' data	54
Fig. 5.9	Speedup projections with actual data	55
Fig. 5.10	Actual speedups	56
Fig. 5.11	Effect of Grain Size on actual Speedup	57

LIST OF TABLES

Table 3.1	Timing Routines for Sequent™ Symmetry™ multiprocessor	24
Table 3.2	Parallel Programming Library Microtasking Routines	29
Table 3.3	Timing Routines for Linda-C environment	33
Table 4.1	Procedures in E P A	44

Performance Monitoring of Parallel Applications at Large Grain Level

Chapter 1

Introduction

One of the most serious problems in the full utilization of multiprocessor architectures for parallel applications is the lack of tools and techniques for debugging and doing performance analysis. An approach to performance analysis of parallel applications should provide the program designer with sufficient synchronization and execution information for fine tuning the application. It should present a consistent interface and parameters for evaluation across a wide variety of machines and programming environments. The methodology should be independent of specific architectures and should provide a high level abstraction of application performance.

The performance of parallel programs can be analyzed in basically two ways :- 1) On-line analysis and 2) Postmortem analysis. In on-line analysis execution trace is collected and analyzed during a program execution [1, 2, 3]. The on-line analysis approach requires the creation of trace collection and analysis process on the machine. Online analysis leads to an interactive analysis tool that is dynamically controlled by the programmer. Since the profile gathering computations take place on an additional processor, this method is less intrusive then the Postmortem method. Furthermore, online

analysis is capable of handling large profiling tasks since the traces are summarized prior to storing them in off-line database files. Online analysis has intrinsic machine dependencies since it needs an additional processor and may be ineffective for architectures which have substantial interprocessor communicational delays such as cube machines. Therefore while online analysis yields excellent results for specific architectures (e.g. shared memory multiprocessors) , it can not form the basis of a more general approach.

Postmortem analysis of applications [4, 5, 6, 7] involves collection of execution traces and then processing the traces off-line to obtain performance parameters such as processor utilization, order of execution, task mapping and speedup. This method is more intrusive than on-line analysis since the executing application is diverted to collection of trace information. However this approach has no machine dependencies and can therefore form the basis of a more general approach. The trace data can also be analyzed by more sophisticated programs thus giving different views of the history of execution of a program.

One of the earliest systems for monitoring multiprocessor programs was the **METRIC** system [3] for applications that communicate over a local area network. This system is divided into three parts : *probes*, *accountants* and *analysts*. *Probes* are the procedure calls used by a programmer to generate trace data. A *probe* inserted into the program source code sends data over the network to an *accountant*. *Accountants* record or filter the incoming data. *Analysts*

are the collection of processes which summarize the trace data. This system helped to define the explicit separation of data generation, selection and analysis. The **METRIC** system has been the basis of later analysis tools.

Most of the recent work in performance monitoring [1, 2, 4, 5, 6, 7] has centered around creating exclusive environments for trace collection and analysis. Trace collection has been done either by inserting special routines in the source code or by means of special hardware hooks. The trace analysis approach used most frequently is the creation of a relational database of execution history and a set of queries for obtaining performance information about the application. These methods have shown efficacy for the systems on which these were implemented but the portability of such methods over completely different architectures and the integration of performance analysis tools with programming environments are issues which have been addressed marginally at best [8]. The more portable UNIX™ profilers *prof* and *gprof* are sufficient for static analysis of sequential programs but fail to represent the synchronization behavior of parallel applications.

The work of this thesis is an attempt to create a methodology to analyze the performance of parallel applications on a wide variety of platforms and programming environments. First we determined the monitoring functions required to collect traces for accurate representation of the parallel application. We used the Extended Large Grain Data Flow (**E L G D F**) [10] representation of an application to

determine granularity and which monitoring functions should be inserted for sufficient feedback to application designer. The monitoring routines (real time clock access procedures) with a common interface were developed for the Sequent™ multiprocessor machine and the C-Linda programming environment[1]. We also developed an Execution Profile Analyzer(**E P A**) for post-processing the traces. The **E P A** gives feedback to the mapping and scheduling tool (**TaskGrapher**) by providing actual performance data. These tools are being developed as a part of Parallel Programming Support Environment (**P P S E**) research [11].

The rest of this thesis is organized as follows : Chapter 2 describes the approach and the experiments for determining the instrumentation and correct granularity for performance analysis. The context of this work is further explained with a brief overview of the **P P S E**. Chapter 3 describes the timing routines and their usage with the **SuperGlue** [12] source code generator. The mapping of these routines to real machines is also explained. The analysis program, **E P A** is explained in Chapter 4. Chapter 5 presents a complete example of application design and performance analysis using the **P P S E** tools. And finally chapter 6 summarizes the work done, effectiveness of the approach developed and the work needed to be done in the future.

---

[1] C-Linda supports a common interface for parallel application development on different architectures such as cube, shared memory and connection machines .

Chapter 2

An Approach to Performance Analysis

2.0 Introduction

This chapter describes the proposed performance analysis methodology in concert with the **P P S E** research. We used the solution of the Traveling Salesman Problem (**T S M**) by Simulated Annealing [13] as a test for determining suitable instrumentation, levels of granularity at which to insert instrumentation, and the effects of such instrumentation on program execution time. The experiments and the results are also summarized in this chapter.

2.1 Existing Analysis Tools

In our brief survey of general purpose monitoring tools we found *gprof* and *parasight* to be portable across different UNIX™ platforms. *gprof* is a stand-alone utility used in post-mortem mode[14]. The application to be profiled, is compiled with an option to link the trace collection procedures e.g

```
>>  <  cc, pascal, fortran > -pg  prog.< c, p, f >
```

On successful execution the application dumps a binary trace file *gmon.out*. The *gprof* utility then translates this trace to yield a call graph of the application along with call statistics such as time spent in

a particular function, number of times a function is called and percentage of user time occupied by various functions. While *gprof* presents a coarse overview of a sequential program, it is unsuitable for analyzing parallel applications since it completely hides the order of execution and the synchronization events.

Another interesting approach to monitoring parallel programs is considered in *parasight* [1]. It is a parallel programming environment for shared memory multiprocessors. The main idea behind *parasight* is to use one processor in the machine to harbor monitoring processes called *parasites*. These *parasites* periodically examine the process tables of the running application and build an on-line view of the application. The approach is very non-intrusive, has scope for dynamic monitoring and control of the application by the programmer, and produces a summary of traces rather than raw traces. However we find this approach unsuitable because it requires an additional processor for monitoring and may not be effective if the inter-processor communication delays are more than grain execution times.

2.2 Overview of P P S E

The Parallel Programming Support Environment (**P P S E**) is a set of software tools developed under the aegis of Oregon Advanced Computing Institute (**O A C I S**) to address the problems associated with parallel application development in an integrated fashion [11]. Numerous prototype tools address almost all aspects of parallel programming such as :-

- partitioning of application into parallel parts
- mapping parallel parts onto multiple processors
- optimal scheduling of parallel parts
- reverse engineering existing serial code into parallel programs
- performance measurement and analysis
- how to coordinate design, coding, debugging and performance

Fig. 2.1 illustrates the various **P P S E** tools and their interaction. The application designer uses the Application Design System (**Parallax**) to construct a data flow design of the desired program. The design system produces a data flow graph which describes the interconnections and data dependencies of the proposed program.

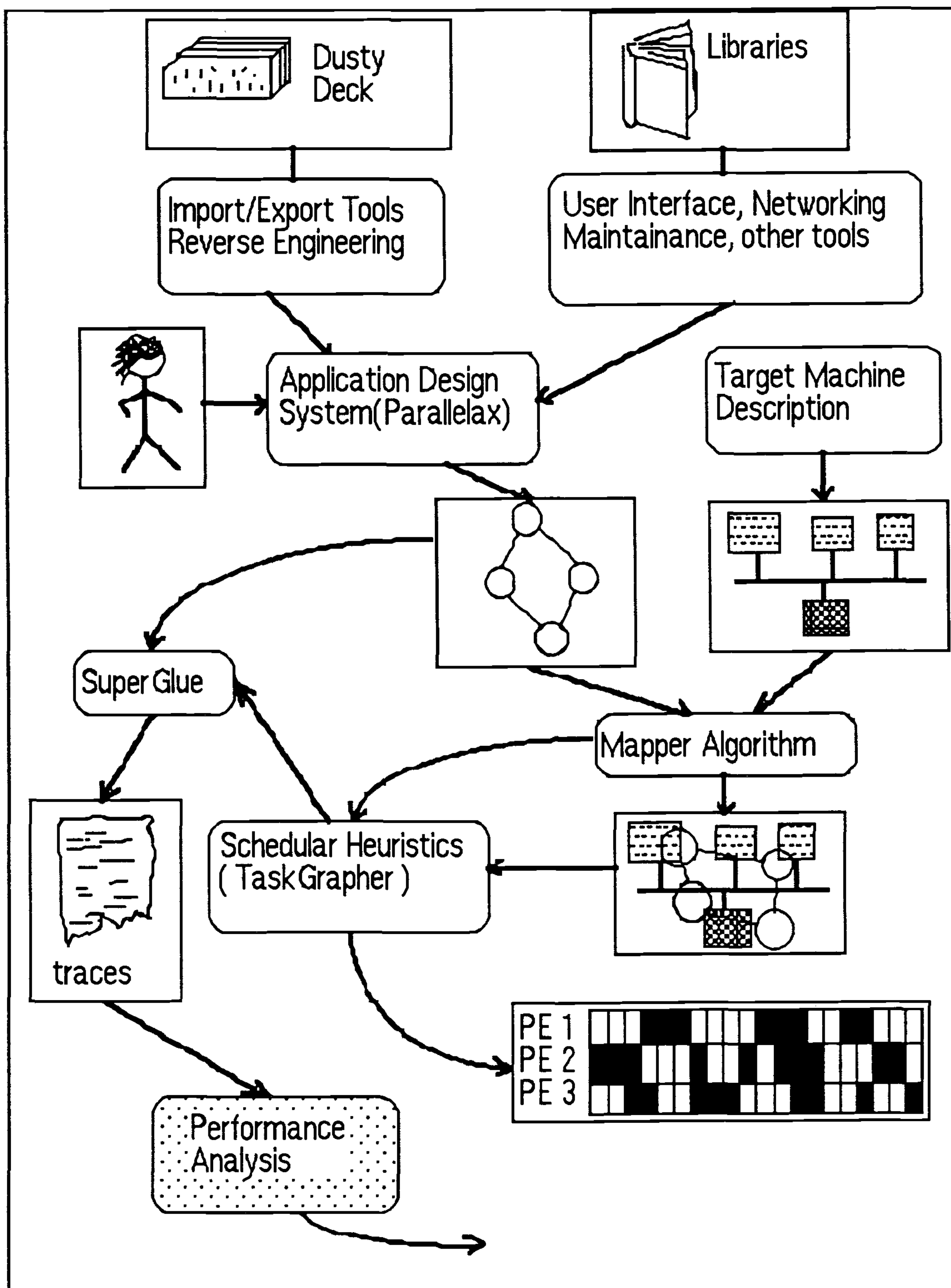

Fig. 2.1 : Overview of P P S E

Parallax can also be used to write the code fragments for implementing the nodes in the data flow diagram in a particular programming language. These code fragments are used by source code generator **SuperGlue** [12] to produce compilable source code for a particular platform. Similarly the target machine(s) for the proposed program is designed and stored as a file of architectural properties such as connection topology, communication delays, cache structures etc. using the Target Machine Editor (**T M E**).

The programmer next uses a mapper and scheduler (**TaskGrapher**) to allocate and schedule the data flow design onto the target machine. This produces an execution schedule as a Gantt chart of processing elements versus time. The output at this stage also includes the plot of the speed up versus number of processors (PEs). This information is used to evaluate the " goodness" of the design (the program and the target machine).

From this preliminary analysis of a number of designs the programmer choses one for actual execution on a real machine. At this stage **SuperGlue** is used to yield the compilable code for the specified machine and programming language. When the resulting program is actually run on the machine, an execution profile is collected and analyzed by Execution Performance Analyzer (**E P A**). This analysis yields feedback to the application design system about the actual execution time, actual scheduling Gantt chart and statistics for speedup, processor utilization and distribution of execution time

among grains. Collecting traces for generating feedback is the work of this thesis.

2.3 Steps in Performance Analysis

The methodology which we found appropriate for analyzing parallel applications in **P P S E** is illustrated in Fig. 2.2. It consists of the following steps :-

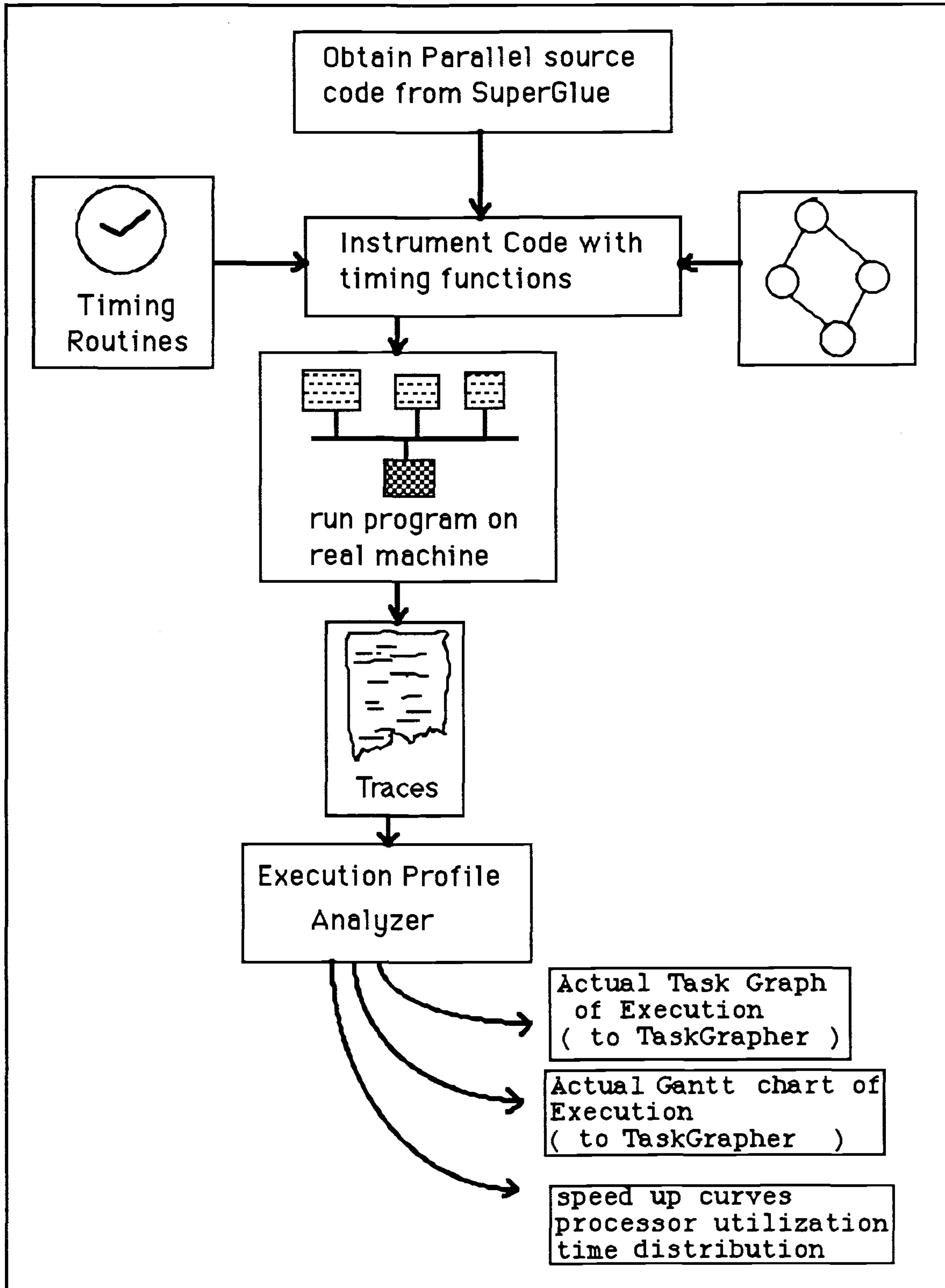

Fig.2.2 : Performance Analysis Methodology

• Identify synchronization points and grain boundaries in the program with the help of **E L G D F** description and programming language constructs such as `m_fork` and `m_multi` in Sequent™ environment and `in()` and `out()` in the C-Linda language.

• Insert real-time measurement routines into the source code of the application at the synchronization points and large grain boundaries[1] (the time measurement routines are described in chapter 3) .

• Run the instrumented program on the target machine and collect the execution traces.

• Analyze the traces off-line using the analysis program (described in chapter 4) .

In selecting this particular approach we addressed the following issues :-

• What execution data is necessary and sufficient for validating the prediction of current **P P S E** tools ?

• At what level of granularity should the measurement functions be placed ? (" level of granularity" is shown in Fig. 2.5 and explained in section 2.5)

---

[1] **SuperGlue** automatically inserts timing functions at the grain boundaries if "Timing" option is chosen while generating source code.

• What are the measurement functions which make this performance analysis method portable ?

• What is the effect of the measurement functions on the execution time of the application ?

To determine the answers to these issues we conducted a series of instrumentation experiments using the Sequent™ Symmetry™ shared memory multiprocessor. These experiments are summarized in section 2.5. Using this methodology we conducted analysis of applications developed using **P P S E** tools. A complete example is explained in Chapter 5.

2.4 Instrumentation Experiments

We solved the parallel implementation of the Traveling Salesman Problem (**T S M**) using simulated annealing to fine tune the methodology and to address the issues mentioned in section 2.4. The **T S M** problem is an example of the NP-complete class of combinatorial problems. In **T S M** a salesperson visits N cities, each of which is represented by coordinates [Xi, Yi] on a two dimensional plane. Each city is to be visited exactly once and the total path length for such a trip is to be minimized. This rather simple problem is of practical importance in cell placement in VLSI design, and process-processor mapping in parallel machines, for example.

In NP class of combinatorial problems the computational load increases exponentially with the number of variables handled. Therefore approximate solutions along with manual improvements have been found to yield optimal results [15]. Simulated annealing [13] solves this class of problems by iterative improvement of an assumed solution in association with a hill climbing algorithm. Fig. 2.3 shows a generic simulated annealing algorithm.

```
 S = state of solution.
 Sn = new state of solution.
 E = cost function.
 T = control parameter for finding
     low-cost state.
To = initial value of control parameter.
Tf = Final value of control parameter
 c = rate of change in T.

frozen = FALSE
Start with some state, S
T = TO
Repeat {
  While ( not in equilibrium ) {
    Perturb S to get new state Sn
    ΔE = E(Sn) - E(S)
    if ΔE < 0
      Replace S with Sn
    Else with probablity exp(-ΔE/T )
      replace S with Sn.
    }

  T = c*T.  /* c < 1 */
  if ( T <= Tf )
    frozen = TRUE
  } Until( frozen )
```

Fig. 2.3 : A generic Simulated Annealing Algorithm

An initial guess to the solution is made (in our case the path of the salesman), then this solution is modified slightly and the variable to be minimized is recalculated. If there is an improvement in the solution, the new solution is accepted; otherwise the new solution is *still* accepted with a probability which depends on the 'state' of the solution. Therefore this algorithm makes it possible to escape from local minima in search space and approach absolute minima with greater probability of success.

A top level **E L G D F** description of **T S M** problem is shown in Fig.2.4. The program begins with the grain `Read Data` which inputs the cities to be covered by the salesman. Then the grain `Link Points` creates a circular linked list of cities thus forming the initial guess about the salesman path. The program then forks a selected number of parallel processes `Anneal` which perform simulated annealing on the salesman's path. During an iteration each parallel process makes modification to the shared structure `Path`. After the selected number of iterations have been performed , the final path is written to an output file by the grain `Write Data`.

In our experiments we manually wrote the source code for this application in C language on Sequent™ Symmetry™ machine [1].

---

[1] Current prototype of **SuperGlue** provides source code in C-Linda language only.

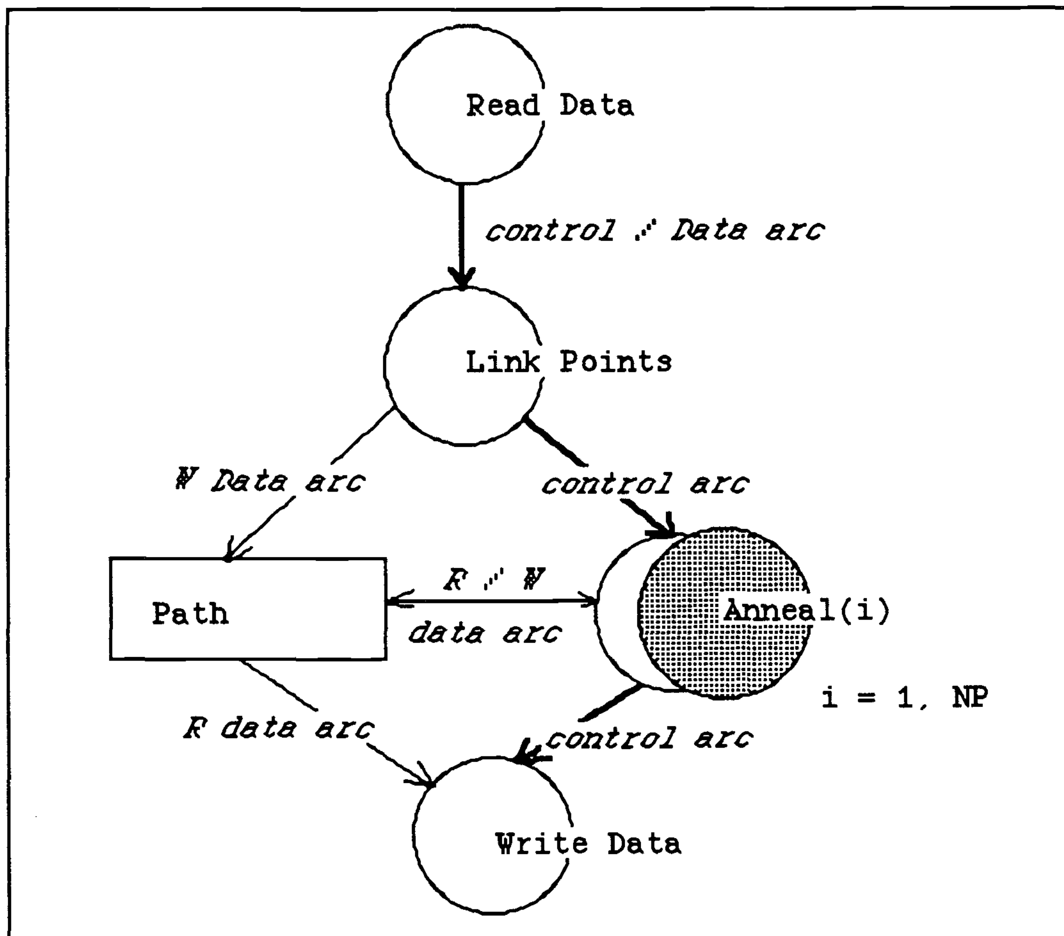

Fig.2.4 : ELGDF representation of TSM

To enforce parallel execution, routines from micro-tasking and parallel library were used [16, 17, 18]. These routines also served to identify the synchronization points and grain boundaries in the source code and thus helped in placing time measurement functions. Appendix A show the implemented source code of one of the versions of **TSM** problem[1].

Execution traces can be collected at three levels of detail as shown in Fig.2.5. Program level provides total execution time, large

[1] The timing routines have been highlighted for easy identification.

grain level provides the execution time for each grain. The small grain level of instrumentation involves tracking each variable and the time spent on their access. We conducted experiments with all the three levels of instrumentation and found that in **P P S E**

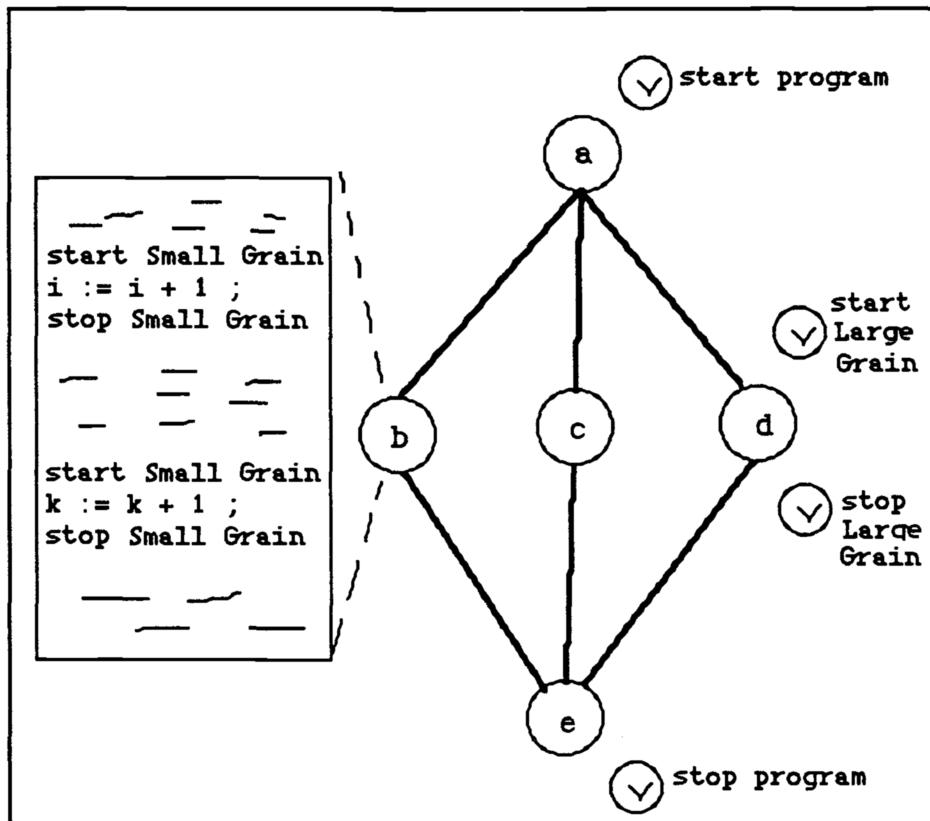

Fig.2.5 : Instrumentation Levels

the large grain level of instrumentation is suitable as it provides sufficient and manageable information on program execution.

The experiments also indicated that real-time measurement at the beginning and ending of each large grain is the data required. By comparing the total execution time of the *clean* and *instrumented*

programs, it was deduced that instrumentation at large grain level produced at most 8% deviation. It was also noted that as grain size became smaller the perturbance due to instrumentation increased. The trace collected could also identify differences between the underlying hardware by means of the differences in the execution times. However, the trace collected is not enough to identify which part of the architecture (CPU, I/O subsystem, Cache, primary memory) is responsible for delay/speedup of an application.

In conclusion, by instrumenting programs at the large grain level with real time measurement we were able to characterize the performance of the parallel application without significant distortion in actual behavior.

Chapter 3

Timing Routines

3.0 Introduction

This chapter describes the timing routines used to generate an execution profile of an application. These routines are the common interface of performance analysis to the **SuperGlue** source code generator. The actual implementation of these routines for a machine or programming environment is dependent on the timing functions available on a particular system. The implementation of these routines and the procedure for incorporating them into **SuperGlue** are described for the Sequent machine and Linda-C programming environment.

3.1 Common Interface

Following are the common interface routines for the **SuperGlue** source code generator :-

`initialize_clock()`
Routine to set up and activate the time measurement facility on the target machine. It involves creation and initialization of variables so that times for the application start with 0.

Routines for Measuring Grain Execution Time :-

`get_grain_start_time( processor_id, grain_id)`

Routine to access the current value of the clock when the grain begins computation. The identifier *start* [1], the variables `processor_id` and `grain_id` and value of the time are printed on the standard output[2].

`get_grain_stop_time(processor_id, grain_id)`

Routine to access the current value of the clock when computations in a grain are finished. The identifier *stop* , the variables `processor_id` and `grain_id` and value of the time are printed on the standard output.

Routines for finding data dependencies between grains and approximating[3] communication delays :-

`get_send_begin_time(grain_id, structure_name)`

Routine to access the current value of the clock prior to placing a variable into tuple[4] space. The identifier *sendBegin*, the variable `structure_name` and `grain_id` and the value of time is printed on the standard output.

---

[1] The identifiers *start*, *stop*, *sendBegin*, *sendEnd*, *recvBegin* and *recvEnd* are printed on the execution trace. These are used by **EPA** to recreate actual Gantt Chart, Task Graph etc.

[2] The redirected standard output file is the execution trace file of the application.

[3] The communication delay is approximate since I/O activity in parallel with the receiving grain can make the variable collect time for receiving grain appear when measured using these routines.

[4] The word 'tuple' is Linda specific. Here it means a message queue for hypercube, shared structure for shared memory machine etc.

`get_send_end_time(grain_id, structure_name)`
Routine to access the current value of the clock after placing a variable into tuple space. The identifier *sendEnd*, the variable `structure_name` and `grain_id` and the value of time is printed on the standard output.

`get_recv_begin_time(grain_id, structure_name)`
Routine to access the current value of the clock prior to obtaining a variable from tuple space. The identifier *recvBegin*, the variable `structure_name` and `grain_id` and the value of time is printed on the standard output.

`get_send_end_time( grain_id, structure_name)`
Routine to access the current value of the clock after obtaining a variable from tuple space. The identifier *recvEnd*, the variable `structure_name` and `grain_id` and the value of time is printed on the standard output.

`processor_id` = integer number identifying a processor on the target machine.
`grain_id` = integer number identifying a grain in the ELGDF description of the problem.
`structure_name` = string identifying a structure passed between two grains.

The communication time for a structure is calculated by the following expression :-

communication time = (sendEndTime - sendBeginTime) +
(recvEndTime - recvBeginTime).

These routines are suitable for obtaining dependencies between grains when message passing paradigm is used for inter-grain communication. In Linda the message passing is implemented by the functions `in()` and `out()` while on a hypercube this is implemented by the system functions `send()` and `receive()`.

The placement of these routines and general model of grain is shown in Fig. 3.1.A section of the execution trace generated using these routines is shown in Fig. 3.2.The execution trace is used by Execution Profile Analyzer (**E P A**) to yield the execution times of the grain and the communication delay or synchronization wait time.

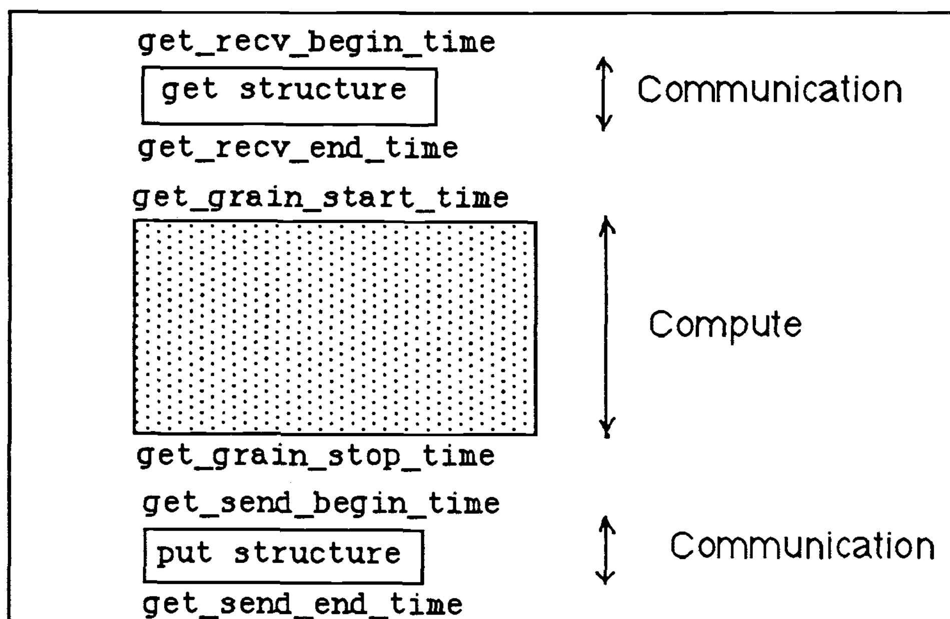

Fig. 3.1 : Grain Model and timing interface.

```
recvBegin     superGlue01     1     520
recvEnd       superGlue01     1     540
start           2             1     540
stop            2             1     2350
sendBegin     superGlue16     1     2350
sendEnd       superGlue16     1     2360
recvBegin     superGlue05     5     2360
recvEnd       superGlue05     5     2360
start           2             5     2360
stop            2             5     4170
```

Fig. 3.2 : Trace file format

3.2 Implementation on Sequent™ Machine

Following steps are required to use the timing routines on the Sequent™ Symmetry™ machine :-

1. Include the interface file in the source code of the application by

```
#include "time_sequent_c.h"
```

The time_sequent_c.h file is as shown in Table 3.1

Table 3.1 : Timing Routines for Sequent™ Symmetry™ multiprocessor

---

```
#include <usclkc.h>             /* microsecond clock header file */
#include <stdio.h>
#define CONVERSION_FACTOR 1000 /* factor to convert to milli-seconds */
usclk_t clock_offset ;          /* value of real time at start */
void initialize_clock()           /* procedure to setup the clock */
{
        usclk_init();
        clock_offset = getusclk();
}
void  get_grain_start_time(  processor_id,  grain_id  )
int processor_id ;
int grain_id ;
```

```
{
        usclk_t temp ;          /* scratch variables */
        usclk_t time_now ;    /* get the time since program start */
        temp = ( getusclk() - clock_offset ) ;
        time_now = temp / CONVERSION_FACTOR ;
        printf(" start \t%d\t%d\t\t%d\n", processor_id, grain_id,
                  time_now ) ;
}
void  get_grain_stop_time(  processor_id,  grain_id  )
int processor_id ;
int grain_id ;
{
        usclk_t temp ;          /* scratch variables */
        usclk_t time_now ;    /* get the time since program start */
        temp = ( getusclk() - clock_offset ) ;
        time_now = temp / CONVERSION_FACTOR ;
        printf(" stop \t%d\t%d\t\t%d\n", processor_id, grain_id,
                  time_now ) ;
}
get_send_begin_time(  grain_id,  structure_name  )
int grain_id ;
char structure_name[100];
{
        usclk_t temp ;          /* scratch variables */
        usclk_t time_now ;    /* get the time since program start */
        temp = ( getusclk() - clock_offset ) ;
        time_now = temp / CONVERSION_FACTOR ;
```

```
        printf(" sendBegin  %s   %d    %d\n", structure_name, grain_id,
                       time_now ) ;
}

get_send_end_time(  grain_id,  structure_name  )
int grain_id ;
char structure_name[100];
{
        usclk_t temp ;          /* scratch variables */
        usclk_t time_now ;    /* get the time since program start */
        temp = ( getusclk() - clock_offset ) ;
        time_now = temp / CONVERSION_FACTOR ;
        printf(" sendEnd  %s   %d    %d\n", structure_name, grain_id,
                       time_now ) ;
}
get_recv_begin_time(  grain_id,  structure_name  )
int grain_id ;
char structure_name[100];
{
        usclk_t temp ;          /* scratch variables */
        usclk_t time_now ;    /* get the time since program start */
        temp = ( getusclk() - clock_offset ) ;
        time_now = temp / CONVERSION_FACTOR ;
        printf(" recvBegin  %s   %d    %d\n", structure_name, grain_id,
                       time_now ) ;
}
get_recv_end_time(  grain_id,  structure_name  )
```

```
int grain_id ;

char structure_name[100];

{

        usclk_t temp ;          /* scratch variables */

        usclk_t time_now ;    /* get the time since program start */

        temp = ( getusclk() - clock_offset ) ;

        time_now = temp / CONVERSION_FACTOR ;

        printf(" recvEnd  %s   %d    %d\n", structure_name, grain_id,

                        time_now ) ;

}
```

The microsecond clock on the Symmetry™ machine is a 32-bit register which is synchronously incremented across all the processors. The clock overflows after 4195 seconds and there is no system function to inform the process of this event. Thus these routines can give incorrect time measurements and are not suitable for measuring programs which last more then 4195 seconds. This drawback could be removed by using system clock to determine such overflows. However using system clock makes the execution time of the clock routines large thus suppressing the 1 microsecond resolution of the clock[1]. For tracing programs with large execution times the clock routines in `time_linda_c.h` file should be used. The timing routines in the file `time_linda_c.h` use the standard UNIX™ system timer and do not make any Linda dependent calls. Thus these routines could be used on any UNIX™ system and C programming environment.

2. Modify compilation option to include the microsecond timing library `/usr/lib/libseq.a` e.g.

```
cc -o myprog myprog.c -lseq
```

3. Placement

`init_clock` : preferably the first executable statement in the source code of the application. This must be run prior to any other timing routine.

---

[1]In course of instrumentation experiments it was found that clock resolution of 1 microsecond is not necessary for monitoring performance at large grain level.

`get_grain_start_time` : the first statement in the grain to be measured.

`get_grain_stop_time` : the last statement in the grain to be measured.

`get_send_begin_time` : before an `m_lock()` statement.

`get_send_end_time` : after an `m_unlock()` statement.

`get_recv_begin_time` : before an `m_lock()` statement.

`get_recv_end_time` : after an `m_unlock()` statement.

The grain boundaries can be detected by the synchronization points in the program. Table 3.2 shows the synchronization functions available in the parallel programming/microtasking library of the Sequent™ Symmetry™.

Table 3.2: Parallel Programming Library Microtasking Routines

Routine	Descriptions
m_fork	execute a subprogram in parallel
m_get_myid	return process identification number
m_get_numprocs	return number of child processes
m_kill_procs	terminate child process
m_lock	lock a lock
m_multi	end of single process code section
m_next	increment global counter
m_park_proc	suspend child process execution

m_rele_procs	resume child process execution
m_set_procs	set number of child process
m_single	begin single process code section
m_sync	check in at barrier
m_unlock	unlock a lock
shared	indicates a shared variable
private	indicates private variable

Source : Guide to Parallel Programming on Sequent Computer Systems, Second Edition, 1987.

Appendix A shows an example source code instrumented for Sequent™ Symmetry™ machine.

4. For detailed information about the implementation of the timing routines refer to the Dynix™ Programmer's Manual, system commands *getusclk(3), usclk_conf(8), usclk(4).*

3.3 Implementation using Linda-C

Following steps are required for using the timing routines on *any* machine supporting Linda-C parallel programming environment :-

1. Include the interface file in the source code of the application by

```
#include "time_linda_c.h"
```

The file `time_linda_c.h` is shown in Table 3.3.

2. There are no compile time modifications.

3. Placement

`initialize_clock` : preferably the first executable function in the source code.

`get_grain_start_time` : the first statement in the grain to be measured.

`get_grain_stop_time` : the last statement in the grain to be measured.

`get_send_begin_time` : before an `out()` statement.

`get_send_end_time` : after an `out()` statement.

`get_recv_begin_time` : before an `in()` statement.

`get_recv_end_time` : after an `in()` statement.

In Linda-C environment the grains are implemented as functions, thus there are no specific grain boundary identifiers.

4. For detailed information about the implementation of the timing routines refer to the UNIX Programmer's Manual, system commands *gettimeofday(2)*.

3.4 Extensions to Timing Routines

The ability of SuperGlue to insert timing functions from a common interface into source code of parallel application makes it easy to extend the performance analysis methodology to animation and

debugging. For example if we replace the current implementation of `init_clock` by having this routine setup an animation process and if we change the other 'timing' routines to send the event messages to the animation process then we can animate the application as it is run on a real machine. Similarly debugging can be viewed as performance monitoring at a small grain level.

Table 3.3 : Timing Routines for Linda-C environment

```
#include <sys/time.h>

struct timeval clock_offset ;


initialize_clock()

{

        gettimeofday( &clock_offset, 0 );

}


get_grain_start_time(  processor_id,  grain_id  )

int processor_id ;

int grain_id ;

{

        struct timeval temp ;

        long time_elapsed ;

        gettimeofday( &temp, 0 );

        time_elapsed = ( temp.tv_sec - clock_offset.tv_sec ) * 1000 ;

        time_elapsed += ( temp.tv_usec - clock_offset.tv_usec )/1000 ;

        printf(" start   %d    %d     %d\n",processor_id, grain_id,

                time_elapsed);

}


get_grain_stop_time(  processor_id,  grain_id  )

int processor_id ;

int grain_id ;
```

```
{
        struct timeval temp ;
        long time_elapsed ;
        gettimeofday( &temp, 0 );
        time_elapsed = ( temp.tv_sec - clock_offset.tv_sec ) * 1000 ;
        time_elapsed += ( temp.tv_usec - clock_offset.tv_usec )/1000 ;
        printf(" stop     %d     %d          %d\n",processor_id,
                 grain_id, time_elapsed);
}

get_send_begin_time(  grain_id,  structure_name  )
int grain_id ;
char structure_name[100];
{
        struct timeval temp ;
        long time_elapsed ;
        gettimeofday( &temp, 0 );
        time_elapsed = ( temp.tv_sec - clock_offset.tv_sec ) * 1000 ;
        time_elapsed += ( temp.tv_usec - clock_offset.tv_usec )/1000 ;
        printf(" sendBegin     %s    %d        %d\n",structure_name,
              grain_id, time_elapsed);
}

get_send_end_time(  grain_id,  structure_name  )
int grain_id ;
char structure_name[100];
{
```

```
        struct timeval temp ;
        long time_elapsed ;
        gettimeofday( &temp, 0 );
        time_elapsed = ( temp.tv_sec - clock_offset.tv_sec ) * 1000 ;
        time_elapsed += ( temp.tv_usec - clock_offset.tv_usec )/1000 ;
        printf(" sendEnd     %s     %d       %d\n",structure_name,
              grain_id, time_elapsed);
}
```

get_recv_begin_time(grain_id, structure_name)

```
int grain_id ;
char structure_name[100];
{
        struct timeval temp ;
        long time_elapsed ;
        gettimeofday( &temp, 0 );
        time_elapsed = ( temp.tv_sec - clock_offset.tv_sec ) * 1000 ;
        time_elapsed += ( temp.tv_usec - clock_offset.tv_usec )/1000 ;
        printf(" recvBegin     %s       %d       %d\n",structure_name,
              grain_id, time_elapsed);
}
```

get_recv_end_time(grain_id, structure_name)

```
int grain_id ;
char structure_name[100];
{
        struct timeval temp ;
```

```
        long time_elapsed ;

        gettimeofday( &temp, 0 );

        time_elapsed = ( temp.tv_sec - clock_offset.tv_sec ) * 1000 ;

        time_elapsed += ( temp.tv_usec - clock_offset.tv_usec )/1000 ;

        printf(" recvEnd     %s         %d          %d\n",structure_name,

              grain_id, time_elapsed);

}
```

Chapter 4

Execution Profile Analyzer

Execution Profile Analyzer (**E P A**) is a program which reads the execution profile produced by the timing routines in a parallel application and generates result files which serve as input to **TaskGrapher**. It also produces a general report giving the overall speedup, overall processor utilization and per processor data on utilization and the time distribution among the grains which were mapped onto it. This chapter explains the operation and structure of **E P A** so that this analyzer can be used for generating different reports.

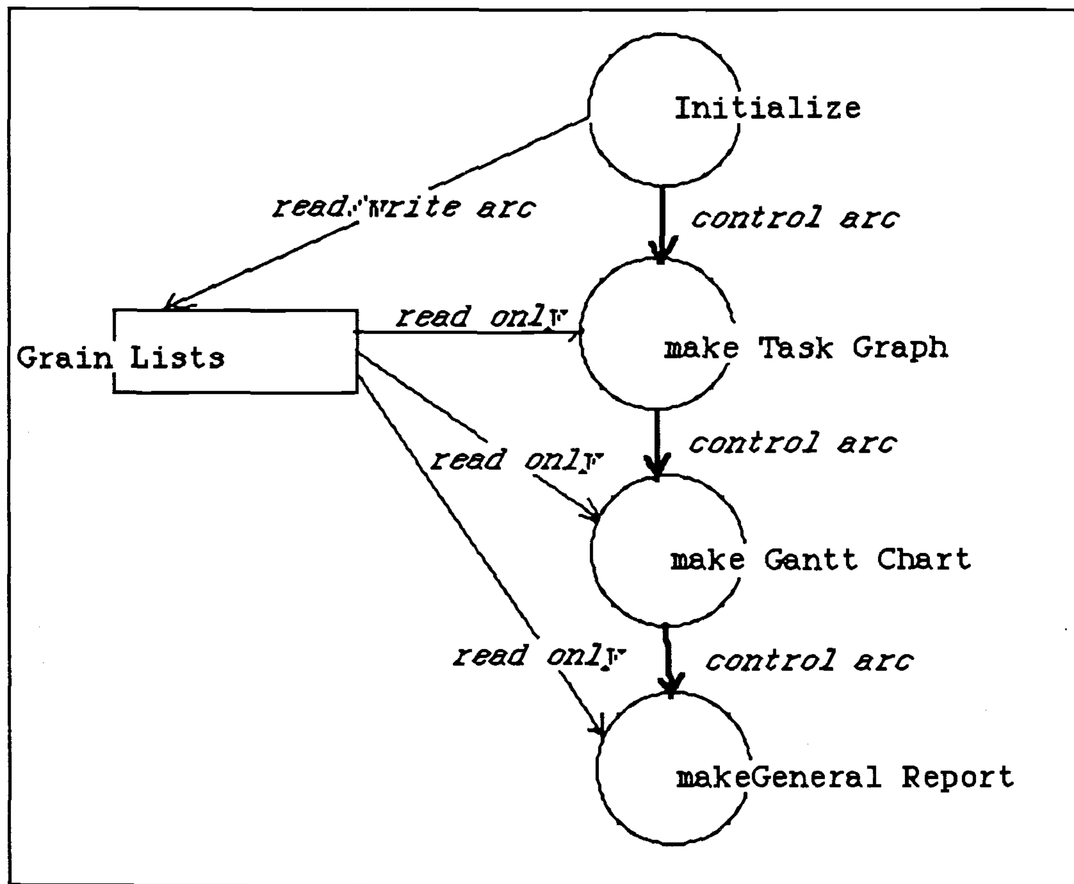

Fig. 4.1 : Top level ELGDF of E P A

The **E P A** is implemented using the Standard Pascal available on most UNIX™ machines (see Appendix B). The traces generated from an instrumented program can be piped directly to the **EPA** thus removing the intermediate trace storage. The **EPA** is also implemented on Apple™ Macintosh™ computer for off-line analysis. The UNIX™ version of **EPA** was developed for easy portability to different systems. The top level **E L G D F** of the program is shown in Fig.4.1. The main data structures manipulated in E P A are a doubly linked list of trace elements for all the processors in the application (see Fig. 4.2), a doubly linked list of all the arc elements (see Fig.4.3), and a doubly linked list of all the processors which

were used in the application(see Fig. 4.4). Each of the processor in the list of processors also has a doubly linked list of trace elements of the grains which were mapped onto that processor during actual execution.

The program first reads the trace file to create the three linked lists. Then report generation functions are executed yielding textual description of actual Gantt chart of execution, task graph of application and a General performance report. The General performance report contains the actual speedup, processor utilization and the percentage of times occupied by the grains mapped on the processors. Fig. 4.5 show an example of the reports generated. Table 4.1 gives the list of the procedures and functions in the current version of **E P A**. A complete listing of the program for UNIX™ based machines is provided in the Appendix B .

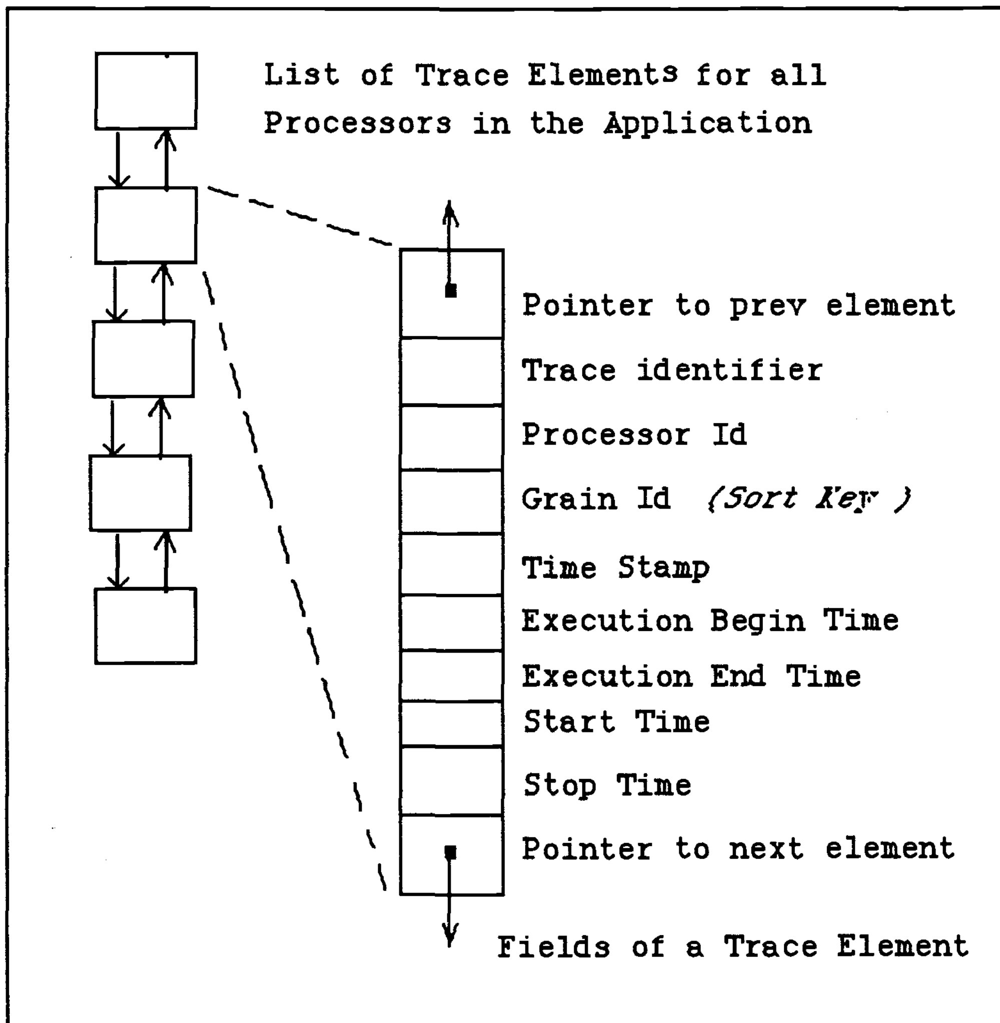

Fig. 4.2 : Data structures of E P A

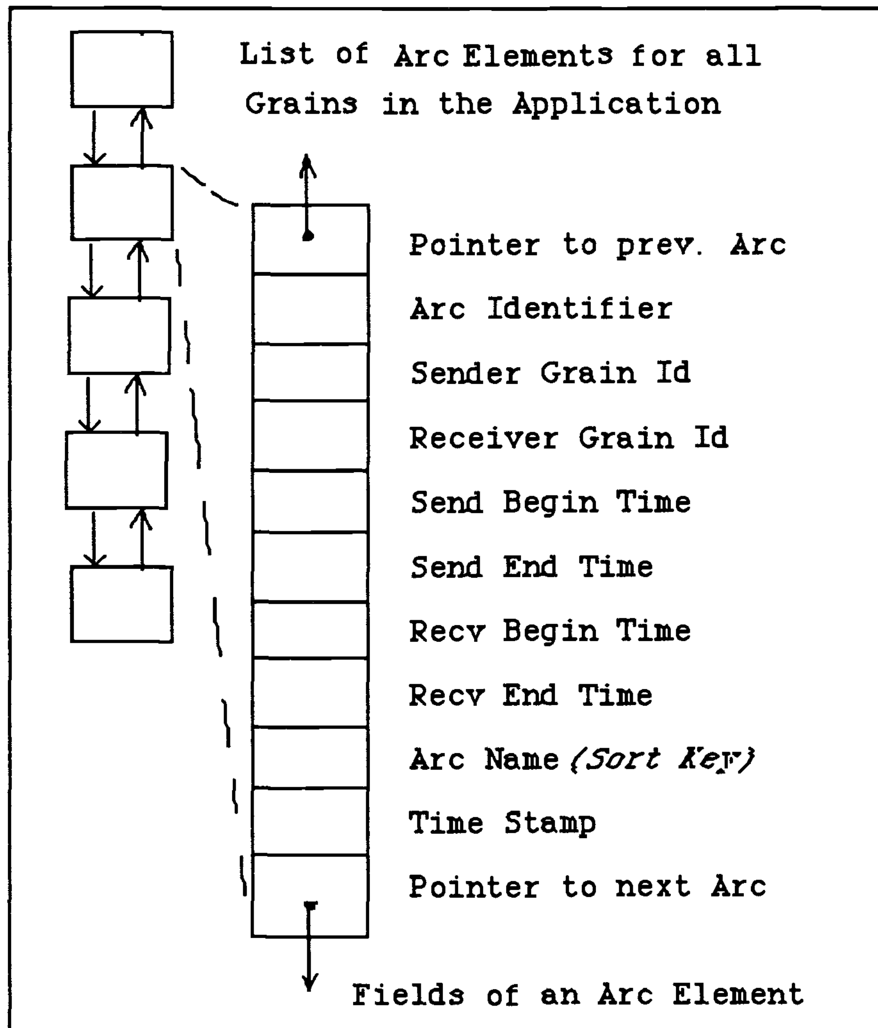

Fig. 4.3 Data structures of E P A

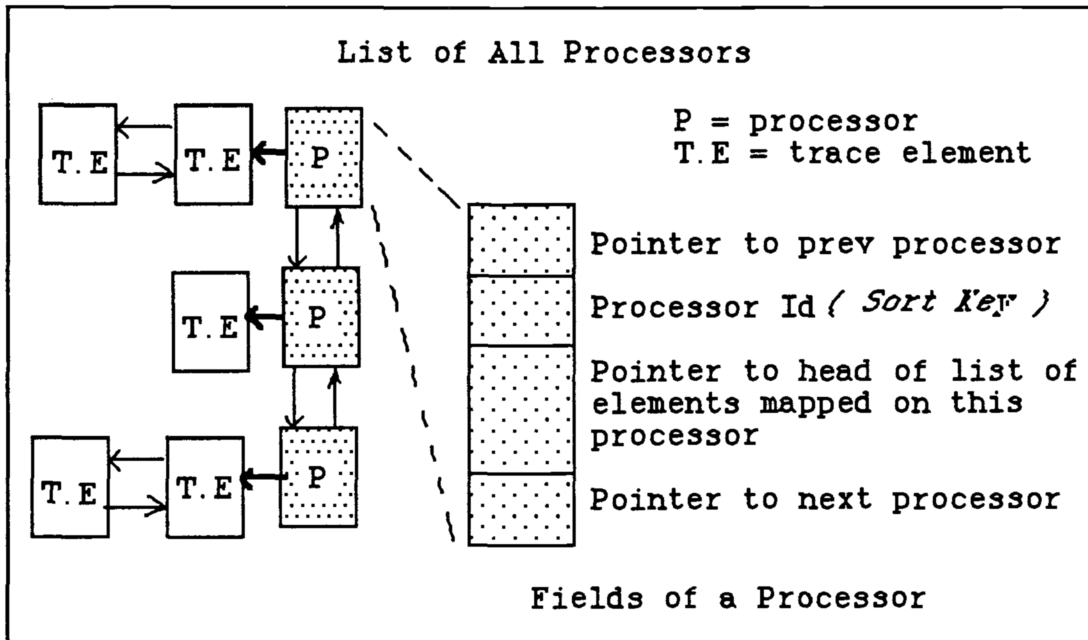

Fig. 4.4 : Data Structures of E P A

```
  Speed up = 1.55
Overall Processor Utilization = 77
Data on Processor 1
Utilisation = 93
Grain id     % time
1      0
3      31
4      30
5      30
7      0

Data on Processor 2
Utilisation = 62
Grain id     % time
2      31
6      31
```

Fig. 4.5a : General Performance Report

```
Task_Graph  <Identifies a Task Graph file >
7    < Total Number of Nodes >
     1     20      40     80      6     0
     2     1810    40     160     1     1
     3     1810    40     240     1     1
     4     1790    40     320     1     1
     5     1800    40     400     1     1
     6     1810    80     80      1     1
     7     10      80     160     0     6
11     < Total Number of Edges >
     1     3     0
     1     4     20
     1     5     0
     1     7     10
     1     6     10
     1     2     20
     2     7     10
     3     7     0
     4     7     10
     5     7     0
     6     7     10
```

Fig. 4.5b : Actual Task Graph file

```
Schedule          < Identifies this as a schedule file >
    7   < Total Number of Tasks on All Processors >
    2     1.0   < Number of Processors, Transfer Rate >
    1    < Processor# >
    5   < Number of Tasks on this processor >
    1     290      310     0     0
    3     350      2160    0     0
    4     2170     3960    0     0
    5     3980     5780    0     0
    7     5810     5820    0     0
    2    < Processor# >
    2   < Number of Tasks on this processor >
    2     590      2400    0     0
    6     2410     4220    0     0
```

Fig. 4.5c : Actual Gantt chart file

Table 4.1 : Procedures in E P A

Procedure	Called By	Calls
readMarker	getTracePoint	none
getTracePoint	makeTraceList	readMarker
swapTracePoints	orderByGrainId	none
swapProcessors	orderByProcessorId	none
swapProcessorTracePoints	orderByStartTime	none
placeTraceElement	makeTraceList	none
makeTraceList	initialize	getTracePoint
		placeTracePoint
orderByGrainId	initialize	swapTracePoints
orderByStartTime	orderByProcessorId	swapProcTracePts.
regroupTraceByProcessor	initialize	none
orderByProcessorId	initialize	swapProcessors
		orderByStartTime
makeELGDFfile	program	none
getTraceCount	makeMacScheduleReport	none
	makeGeneralReport	
getProcessorCount	makeMacScheduleReport	none
getTotalExecutionTime	makeGeneralReport	none
makeMacScheduleReport	program	getTraceCount
		getProcessorCount
		getTotalExecTime

getRealExecutionTime	makeGeneralReport	none
makeGeneralReport	program	getRealExecTime
		getTotalExecTime
initialize	program	makeTraceList
		orderByGrainId
		regroupTraceByPrc
		orderByProcIc

Chapter 5

All Together

This chapter presents an example of the usefulness of **P P S E** tools for parallel programming. We consider the complete design, implementation and performance analysis of a program consisting of large grained, loosely coupled processes.

The example we consider approximates pi using the rectangle rule[19]. The parallel version of this program is a typical broadcast, calculate and aggregate (BCA) problem where the data is sent(broadcast) to a number of worker tasks which after doing the calculations, send the data to a task which collects (aggregate) the results. An **E L G D F** expression[1] of the problem using the **Parallax** editor is shown in Fig.5.1.

[1] A succinct **ELGDF** representation of this problem would replace the workers by a Repetition symbol. However this is not done here since current version of **SuperGlue** requires simple nodes in the **ELGDF** representation of problem.

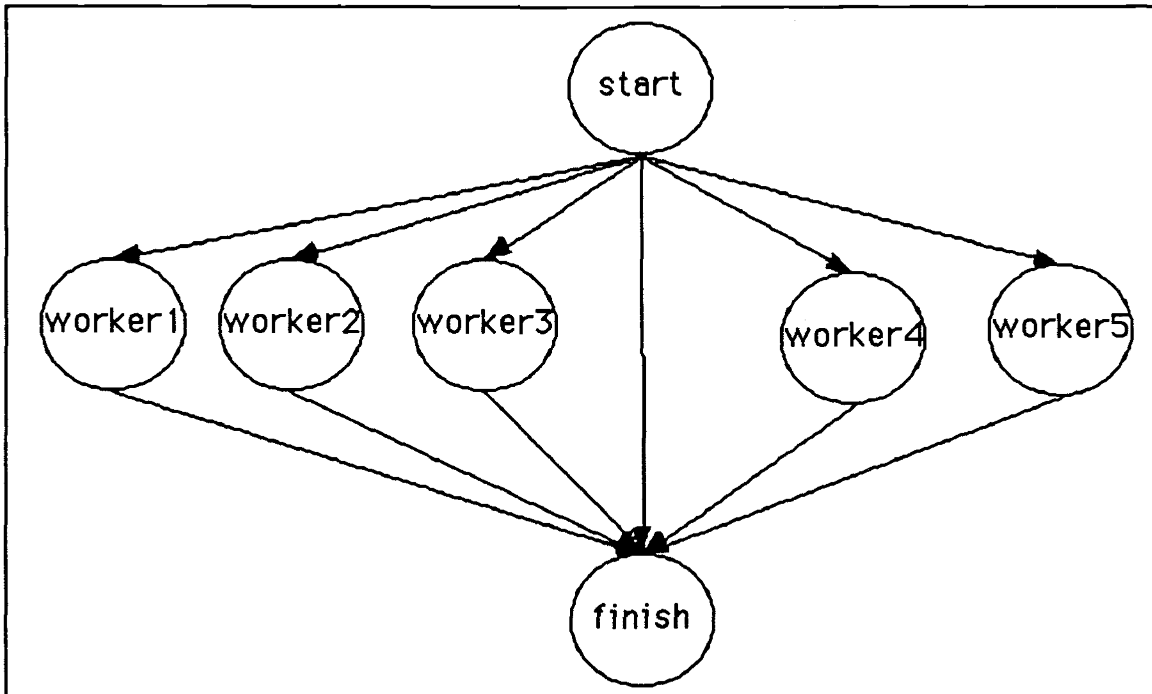

Fig. 5.1 : ELGDF representation of pi approximation problem

While editing the **ELGDF** description we enter the data to be passed along the arcs. Fig. 5.2 show the dialog box for entering data for the arc between the node "start" and "worker 1" .

Arc Inforamtion

Arc Vars interval,start1,stop1

Arc Usage ○ Read ◉ Write ○ R/W

Mutual Exclusion ◉ No ○ Yes

Compound Arc ◉ No ○ Yes

Num. Of Iteration 1 Times

Message Size 1 Bytes

Documentation refer start_worker1

Cancel OK

Fig. 5.2 : Specifications of an arc

The specifications of the nodes are similarly entered. Fig.5.3. shows the data entry dialog box for the node "start". The code fragment implementing the node start could be also entered at this stage[1]. The code fragment for node "start" is shown in Fig.5.4.

After the Data flow design of application is completed, a transformer[2] would convert this ELGDF description into a task graph of the application.Using the task graph of the application we schedule the program using the scheduling tool **TaskGrapher**. From the ELGDF description of the problem it is clear that having five processors would yield the greatest speedup for this problem. However for this example

[1] Current implementation of **SuperGlue** uses a separate code fragment file rather than the code fragments entered through **Parallax**.

[2] This transformer has not been implemented in **PPSE**. Currently this transformation is done manually.

we used *Hu's Highest Level First* heuristic with two processors to illustrate the optimizing features of **TaskGrapher**.

Symbol Information

Compound Symbol ◉ No ○ Yes

Symbol Name start

Execution Time 1 NS

Num. Of Interation 1 Times

Documentation refer start_code

Cancel OK

Fig. 5.3 : Specification of a node

Fig. 5.5 shows the task graph for this example. Each bubble represents the task to be scheduled. The number on the top of each bubble is the `grain_id` while the number at the bottom is the estimated time of execution. Note that at this stage we have no idea as to how long each task will take. Thus we use the default value of 1 at 'cold start'. The numbers on the arcs represent the communication delay which are similarly assumed to be unity.

```
<$$StartCodeFragmentBlock$$>
start
<$$StartFragment$$>
int i,l=1,m,n, workers,processors;
int start1,start2,start3,start4,start5;
int stop1,stop2,stop3,stop4,stop5;
double interval,h;
<$$EndDCL$$>
n = 10000;
interval = 1.0/n;

start1=1;
start2=2000;
start3=4000;
start4=6000;
start5=8000;
stop1=1999;
stop2=3999;
stop3=5999;
stop4=7999;
stop5=10000;
<$$EndCodeFragmentBlock$$>
```

Fig. 5.4 : Code fragment for node "start"

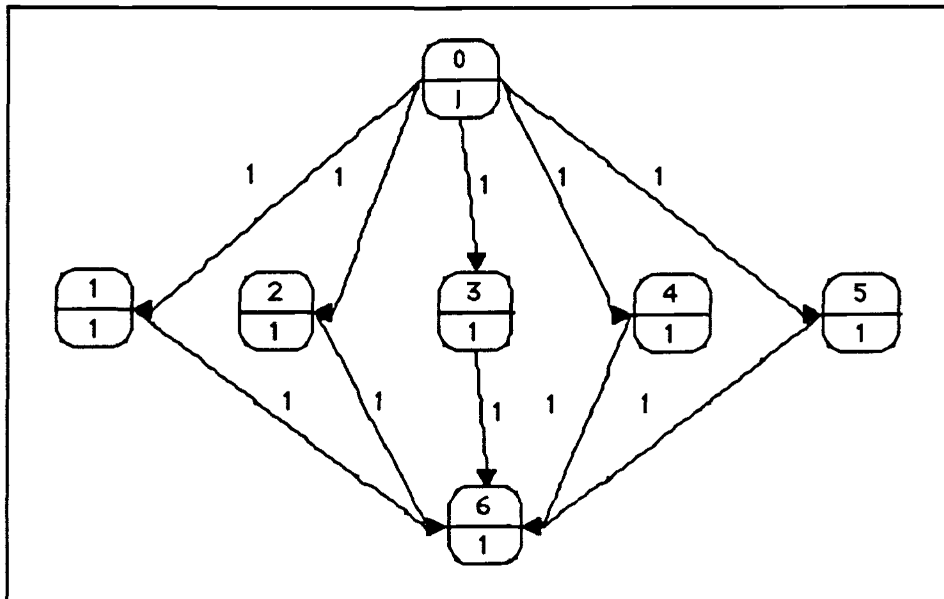

Fig. 5.5 : Task graph at 'cold start'

On the basis of this information the projected execution Gantt chart is obtained from the **TaskGrapher**. This is chart #1 in Fig.5.6.

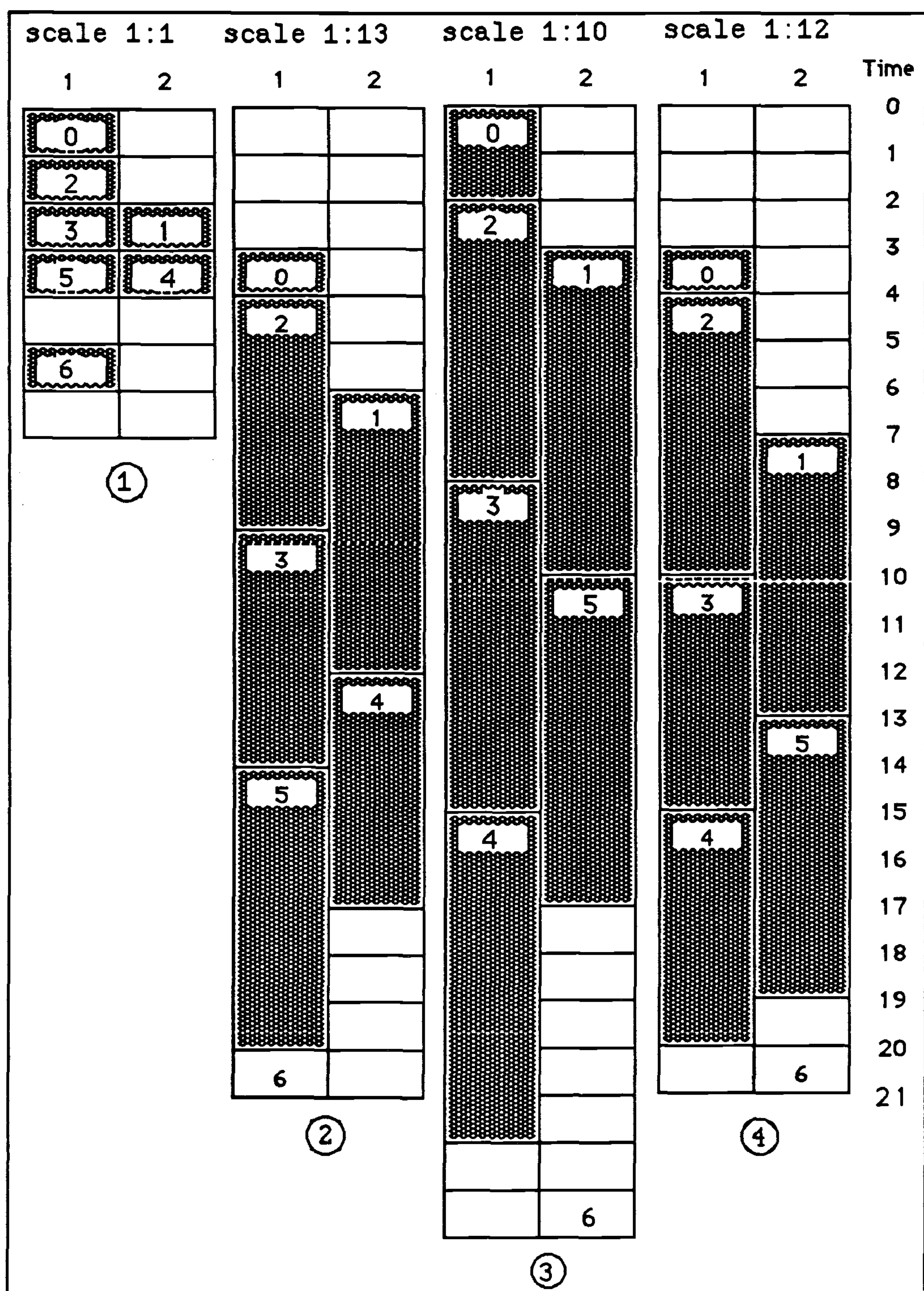

Fig. 5.6 : Execution Gantt Charts

Now we use **SuperGlue** to generate the source code for the Sequent™ machine and Linda-C parallel programming environment. **SuperGlue** takes in the description of the target machine, code fragments and the Gantt chart to yield the compilable source code as shown in Appendix C. The instrumented source code is produced when **SuperGlue Build Application** menu is selected with the timing option enabled. The program was compiled and run on the Sequent™ Symmetry™ in at 3:00 am when there were only two users on the machine. This was done to minimize the effects of operating system on process scheduling.

When the execution trace was analyzed by Execution Profile Analyzer(**E P A**) we obtained the actual execution time of the grains and the actual schedule. The actual Gantt chart of the first iteration is shown as chart # 2 in Fig.5.6. The total execution time was 260 milli-second, therefore the Gantt chart was scaled by a factor of 13 to fit on the screen.

Note that in the actual Gantt chart the execution of the program begins at the time of 40 milliseconds and not at 0 as shown in the projected Gantt chart. This apparent anomaly is due to the fact that the first 40 milliseconds are spent in the initialization of the program and the first grain starts execution after this interval. **TaskGrapher** assumes that the grains start executing when the program is launched. The order in which grains are executed is the same as projected but the execution times are much bigger than expected.

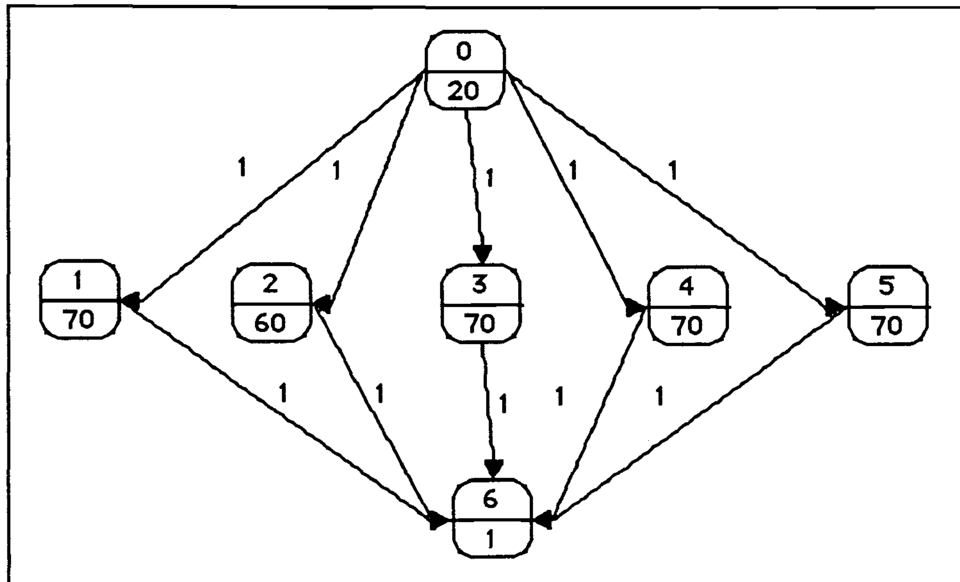

Fig. 5.7 : Task graph after first run

The task graph of the actual application is shown in Fig.5.7. Again using *Hu's Highest Level First* heuristic we get the projected execution Gantt chart. The projected execution time is 222 milliseconds, the Gantt Chart is too long to fit on the paper, so we scale it by 1:10. The scaled execution chart is shown as chart # 3 in Fig.5.6. We also note that on the basis of actual execution times the mapping of the tasks on the processors has been changed. The grains 5 and 6 have been moved to processor 2 while grain 4 has been mapped onto processor 1.

Armed with the new schedule, we go back again to **SuperGlue** and generate the instrumented source code and run it on the target machine. The traces are collected and analyzed. The actual execution

Gantt chart on the basis of this feedback is shown as chart # 4 in Fig.5.6. This actual chart is very close to the projected chart # 3 except for the initialization part of the program.

We also used **TaskGrapher** to get the speedup in this application as the number of processors in the machine is changed. The number of processors was set to five in the **TaskGrapher**. Then the **Speed up Curve** item in the **Analyze** menu is chosen. The **TaskGrapher** then draws the speedup curves for the selected scheduling heuristic(in this example *Hu's highest level first*). Fig. 5.8 shows the speedup curve for the graph in Fig. 5.5. For the graph in Fig.5.7 the projected speedup curve is shown in Fig. 5.9.

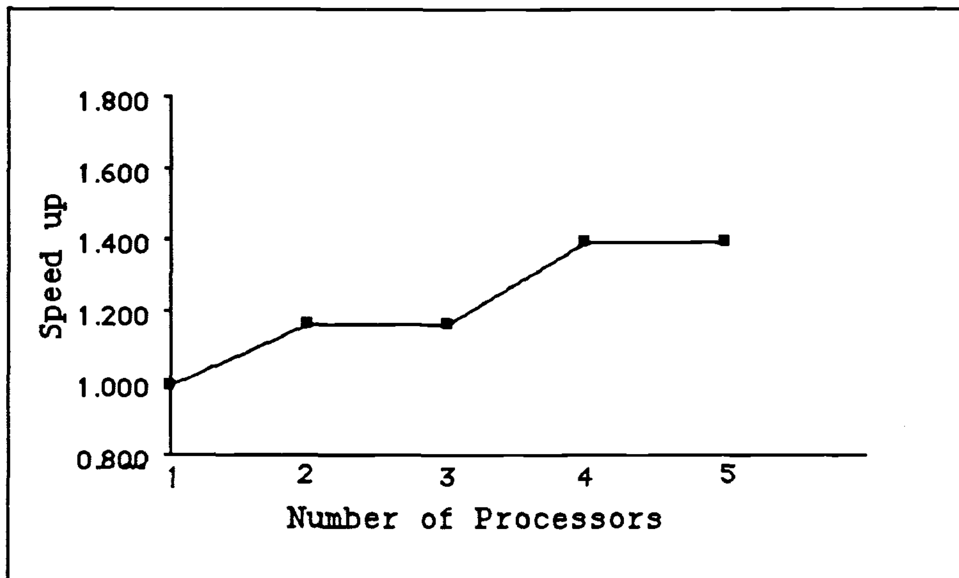

Fig. 5.8 : Speed up projections with 'cold start' data

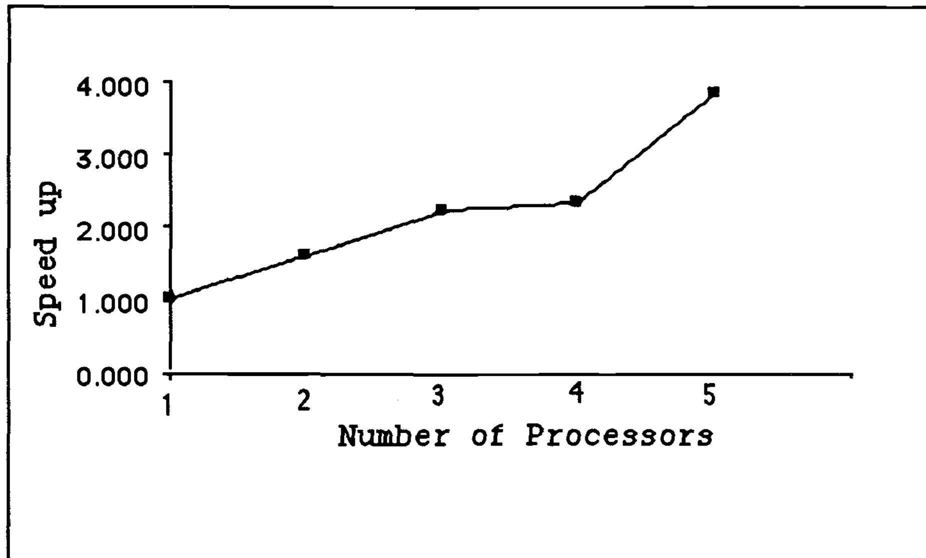

Fig. 5.9: Speedup projections with actual data

By changing the number of processors in the Task Graph of the application we obtained a set of Gantt charts for *Hu's highest level first* heuristic. These were used by SuperGlue to generate source code for the selected number of processors. When these applications were run we obtained the actual speedup. The actual speedups are shown in Fig. 5.10.

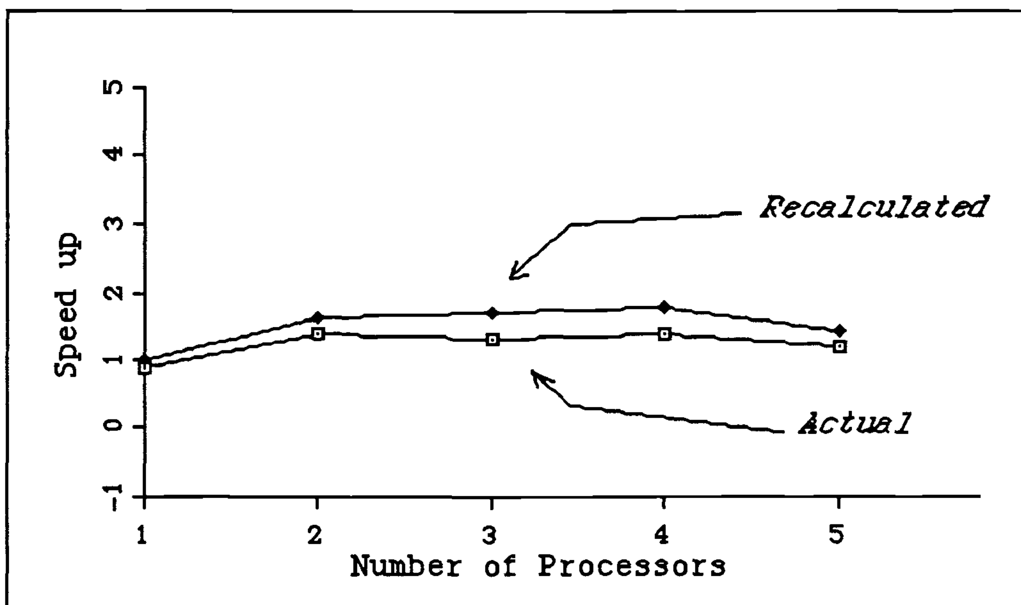

Fig. 5.10: Actual speedups

It is seen that there is considerable difference between the speed-ups especially when the number of processors is increased. A possible cause for this could be the fact that initialization time is not used by **TaskGrapher** for calculating speedups. We recalculated the speedups from the Gantt charts after ignoring the initialization to check if this was the only cause. As seen in Fig. 5.10 ignoring the initialization cost is not the only reason for mismatch. Notice that each grain in this example, is implemented as a set of three phases :- get variables, compute and put variables. By changing the number of iterations in the compute portion of the grain we can change the grain size (execution time). We experimented with changing the grain size to see how it compares with TaskGrapher predictions. The results of these experiments are summarized in Fig. 5.11.

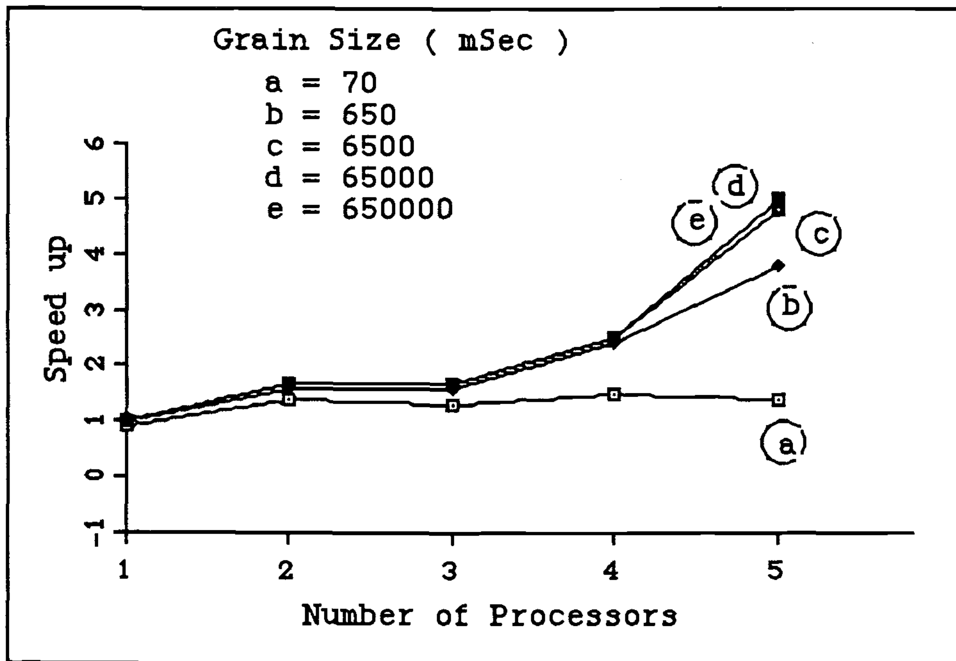

Fig. 5.11 : Effect of Grain Size on actual Speedup

It is seen that the prediction of **TaskGrapher** is close to the actual speedup for large grains (execution time more then 6500 milliseconds) for the C-Linda environment on the Sequent™ Symmetry™ machine. This also indicates that it is okay for **TaskGrapher** to ignore the initialization and process creation time in making the projections if large grains are used.

Chapter 6

Conclusions and Summary

In this thesis a methodology to analyze the performance of parallel applications was devised. The methodology involves inserting real-time measurement routines in the application's source code, collecting execution traces and off-line analysis of traces. It was seen that measurement of grain start and stop time in the **E L G D F** representation of an application provided data which makes it possible for **TaskGrapher** tool to accurately predict the application performance. The methodology presented in this thesis is most useful to the software engineer. However this method is unable to identify hardware bottlenecks within a specific architecture for a given application.

While developing the instrumentation and conducting the experiments, it was noted that Linda-C programming environment provided a straight forward model of parallel application in terms of interprocessor communication via shared tuple space. This high level abstraction frees the programmer from being concerned about synchronization or process scheduling. On the other hand the Sequent's parallel programming/microtasking library provides low level synchronization primitives which require a programmer's constant attention. On the basis of rather limited experience with these two environments, I found the choice of Linda based parallel

programming environment for **SuperGlue** to be appropriate in **P P S E** context.

The **P P S E** tools used most in connection with the work of this thesis were **E L G D F / Parallelax**, **TaskGrapher** and **SuperGlue**. The tools **E L G D F** and **TaskGrapher** are conceptually fully functional, but the human interface needs work. **Parallax** needs the following features :-

- On screen display of basic information about the nodes and arcs of the data flow diagram. This includes full name and type of the symbol[1].
- Printing dataflow diagram or a part of it.

TaskGrapher requires the following improvements :-

- Ability to scale Gantt charts so that a full chart can be seen on the screen.
- Zoom out sections of the selected charts i.e the user selects a section of the Gantt chart and then TaskGrapher redraws the screen so that the selected section covers the complete screen.
- If the number of processors is so large that all Gantt charts can not fit on the screen then, it should be possible to select

[1]This could lead to a cluttered display for large designs. But on the other hand this cluttering would indicate to the user that he must coalesce some of the nodes into compound nodes.

processors and see their Gantt chart clearly.

- Print all or selected Gantt charts or selected part of a Gantt chart.
- An alternative to long Gantt charts could be the use of color to represent length i.e using green to show small grains, red to indicate long grains, other shades for intermediate grain sizes. The use of colors would also quickly identify the important grains.

The current version of **SuperGlue** generates source code for a flat **E L G D F** diagram of an application consisting of simple nodes and arcs only. The source code is generated in Linda-C language for the Sequent™ Balance™ machine. The Parallel Program Data Base is not functional at this stage or **P P S E** research. As a result the interfacing between different tools is via one-to-one compatible files. This made the movement from one tool to another rather clumsy.

In **P P S E** context, a useful extension to **Parallelax** would be to link the actual execution time data with an execution time estimator within the **Parallelax**. Another possibility is to build a database of execution reports. Then the **Parallelax** could direct queries to it for getting the estimate of execution time a particular grain.

We had started with the intuition that high resolution clocks are required for performance monitoring. The results indicate that for **E L G D F** model of parallel programs and for accurate **P P S E** tools predictions, high resolution real-time clocks are not necessary. In fact

the standard UNIX™ system clock with a resolution of 10 milli-seconds is sufficient. The real-time measurement is of tremendous use in performance monitoring of parallel programs since it captures the synchronization of events in an unambiguous manner. Therefore implementing global clocks on the system along with simple routines to access them is an idea which parallel computer manufacturers should seriously consider.

A limitation of this proposed performance analysis methodology is that the inter-grain communication delay can not be measured accurately in the presence of parallel I/O channels in the system. This is because the instrumentation itself is a part of the application and it is subject to manipulation by the system on which the application is running. A valuable extension to this methodology would be to develop a mechanism which can relate the communications occurring at the hardware level (bus traffic, cache hit etc.) to the higher level language inter-grain communication primitives (send, receive etc.).

References

[1] Aral Z. and Gertner I. " High Level Debugging in Parasight " *ACM Workshop on Parallel and Distributed Debugging*, University of Wisconsin - Madison, May 1988.

[2] Brandis C. and Thakkar S. " A Parallel Program Event Monitor " *Proc. of the Twentieth Annual Hawaii International conference on System Sciences*, 1987.

[3] McDaniel G. " METRIC : A kernel instrumentation system for distributed environments " *Proceedings of the 6th Symposium on Operating System Principles*, November 1975.

[4] Snodgrass E. " Monitoring Distributed Systems : A Relational Approach " PhD dissertation, Carnegie Mellon University, December 1982.

[5] Maples C. " Analyzing Software Performance in a Multiprocessor Environment " *IEEE Software* , July 1985.

[6] Miller B. " DPM : A measurement system for distributed programs " *5th International Conferences on Distributed Computing Systems, IEEE Computer Society*, May 1985.

[7] Segall S. and Rudolph R. " PIE : A programming and instrumentation environment for parallel processing " *IEEE Software*, November 1985

[8] Summary of Proceedings, *Workshop on Parallel and Distributed Debugging*, May 1988, University of Wisconsin - Madison, in SIGPLAN Notices v24 n1, January 1989.

[9] *prof, gprof* : UNIX™ utilities for displaying profile data, DYNIX™ Programmer's Manual, revision 1.6 86/05/13

[10] El-Rewini H. and Lewis T. G. " Software Development in Parallax : The ELGDF Language " Technical Report (88 - 60 -17), Dept. of Computer Science, Oregon State University, July 1988.

[11] Lewis T. G. " Parallel Programming Support Environment Research " TR-PPSE-89-1, Oregon Advanced Computing Institute, Beaverton, Oregon, 1989.

[12] Rudd W. G., El-Rewini H., Handley S., Judge D. V. and Kim I. " Status Report : Parallel Programming Support Environment Research at Oregon State University " Corvallis, 1989.

[13] Kirkpatrick S, Gelatt C, Vecchi M " Optimization by Simulated Annealing ", *Science* v220, n459, pp45 - 54 May 1983.

[14] Graham S. L. , Kessler P. B. , McKusick M. K. " gprof : a Call graph Execution Profiler" *Proceedings of the SIGPLAN ' 82 Symposium on Compiler Construction, SIGPLAN Notices*, v17, n6, June 1982.

[15] Durand M. D. " Parallel Simulated Annealing : Accuracy vs Speed in Placement " *IEEE Design and Test of Computers*, pp 8 - 34 , June 1989

[16] *Dynix Programmer's Manual*, revision 1.6 86/05/13.

[17] *Symmetry Technical Summary*, 1987, Sequent Computer Systems, Beaverton, Oregon

[18] *A Guide to Parallel Programming on Sequent Computers*, Book, 2nd Edition, 1987.

[19] Karp A. and Babb R. A. " A comparision of 12 Parallel Fortran Dialects " *IEEE Software*, Sept 1988, pp. 52-67.

Appendix A : Instrumented code of Travelling Salesman Problem

```
/**********************************************************************
 *
 * Traveling Salesman Problem
 *
 * This is a parallel implementation of the Travelling Salesman Problem
 * First the initial path of the salesman is read as x and y coordinates
 * of the cities to be visited. Next the annealing parameters are
 * inputted. The  annealing parameters determine how quickly and
 * accuratly the path of the salesman will be converged.
 * In parallel implemetation, each of the processor is given a copy of
 * the current path. The processor reduce the path for a number of
 * iterations and then they join at a barrier. The best solution is
 * selected and made the latest path. This contines till the annealing
 * parameters are satisfied.
 *
 *      ORIGINAL VERSION BY : Martha, 01/1986
 *      CURRENT VERSION BY  : Anil Kumar Yadav, 09/1989
 *
 *********************************************************************/
#include "time_sequent_c.h"    /* timing routines for execution trace */
#include <stdio.h>
#include <math.h>
#include <parallel/microtask.h>
#include <sys/time.h>
#include <sys/resource.h>

#define TRUE 1
#define FALSE 0

#define MAXCYC 800                /* maximum temperature drops */
#define MAXPTS 1000               /* maximum points in path */

#define MAXPROC 100
#define EPSILON 0.000001          /* for floating point comparison */
#define RANF ((double)((randx = randx * 1103515245 + 12345) &
0x7fffffff)) / 214
7483647.

typedef int boolean;

typedef struct pointtype {
    float x;
    float y;
    struct pointtype *prev;     /* ptr to previous point */
    struct pointtype *next;     /* ptr to next point */
    } POINT;                    /* x,y coords of point on path */

shared float beta;              /* temperature for swapping */
shared float currtemp;          /* current temperature */
shared float drop;              /* drop in temperature */
shared float ftemp;             /* final temperature */
shared float itemp;             /* initial temperature */
shared long  niter;             /* iterations per drop in temp */
shared int   nprocs;            /* number of processes participating */
shared int   npts;              /* number of points in the path */
shared POINT *mypath[MAXPROC];  /* first point for each processor */
```

```
shared float mypathlen[MAXPROC]; /* pathlength for each process */
shared POINT *path;             /* first point in salesman's path */
shared float pathlen;           /* current path length */
shared float plog[MAXCYC];    /* path lengths at various temperatures */
shared static long randx;       /* for randomization of swapping */
shared float tlog[MAXCYC];      /* temperature log */
shared float timelog[MAXCYC];   /* time per iteration */


/*
* function to get the latest path from common path
*/
void get_current_path(pathpointer)
POINT *pathpointer;
{
        POINT *t, *s;
        int i;

        for( i = 0, t=pathpointer, s= path;i < npts; i++, s++, t++)
          {
          t->x = s->x;
          t->y = s->y;
          }

        /* link the points */
        for(i=0, t=pathpointer; i < npts; i++,t++)
          {
          if(i==(npts-1))
            t->next = &pathpointer[0];
          else
            t->next = &pathpointer[i+1];
          if(i==0)
            t->prev = &pathpointer[npts -1];
          else
            t->prev = &pathpointer[i - 1];
          }
}


/*
* function to put the latest path to  common path
*/
void put_current_path(pathpointer)
POINT *pathpointer;
{
        POINT *t, *s;
        int i;

        for( i = 0, t=pathpointer, s= path;i < npts; i++, s++)
          {
          s->x = t->x;
          s->y = t->y;
          t=t->next;
          }

        /* link the points */
        for(i=0, t=path; i < npts; i++,t++)
          {
```

```
          if(i==(npts-1))
            t->next = &path[0];
          else
            t->next = &path[i+1];
          if(i==0)
            t->prev = &path[npts -1];
          else
            t->prev = &path[i - 1];
          }
}


/*
* function to return the index of the process having the smallest path
*/
int get_smallest_path_index()
{
        int currentIndex, i;
        float minPath;

        currentIndex = 0;
        minPath = mypathlen[0];
        for( i=0; i< nprocs; i++)
          {
          if( mypathlen[i] <= minPath)
            {
            currentIndex = i;
            minPath = mypathlen[i];
            }
          }
        return(currentIndex);
}

/*
 * Swap points t and t->next if they make the path length shorter or if
 * they don't with a certain probability.Calculate distances using the
 * point below i and the two points above.
 * This function called by iterate and run by multiple processes.
 *
 * 1/21/86:  I originally had locked this entire routine since the
 * values that the pointers pointed to could change, but there was no
 * speedup at all when I added multiple processors.  This was because
 * the swaps were essentially being done sequentially.  By only locking
 * the section where the swap takes place, I am introducing a little
 * more randomness into the algorithm.  The actual points could be
 * changed by the time they are swapped, but now there is some speedup.
 *
 * 9/1/89 : In the barrier based version each processor is minimizing
 * its own copy of the path, thus there is no contention in accessing
 * this function. Hence there are no locked parts in this functions
 */
cswap(t, swap, eint, erem)
POINT *t;                       /* path pointer */
boolean *swap;                  /* true if points interchanged */
float eint[], erem[];           /* for randomization of swapping */
{
float delta;                    /* distance between points */
int idel;                       /* integer of delta */
```

```
POINT point, *p;                    /* temporary coordinates */

p = &point;

p->x = t->prev->x - t->next->x;
p->y = t->prev->y - t->next->y;
delta = sqrt(p->x * p->x + p->y * p->y);

p->x = t->x - t->next->next->x;
p->y = t->y - t->next->next->y;
delta += sqrt(p->x * p->x + p->y * p->y);

p->x = t->prev->x - t->x;
p->y = t->prev->y - t->y;
delta -= sqrt(p->x * p->x + p->y * p->y);

p->x = t->next->x - t->next->next->x;
p->y = t->next->y - t->next->next->y;
delta -= sqrt(p->x * p->x + p->y * p->y);

*swap = FALSE;
if (delta <= 0.0) *swap = TRUE;
else {
    delta *= beta;
    if (delta > 49.0) delta = 49.0;
    idel = (int)delta;
    if (RANF < eint[idel] * erem[(int)(1000.0 * (delta - idel))])
        *swap = TRUE;
    }

/* Switch t and t->next. */

if (*swap) {
    t->prev->next = t->next;
    t->next->prev = t->prev;
    t->next = t->next->next;
    t->next->prev = t;
    t->prev->next->next = t;
    t->prev = t->prev->next;
    }
}

/*
 * Calculate the distance between two points.
 */
float distance(a, b)
POINT *a, *b;
{
POINT point, *p;

p = &point;
p->x = a->x - b->x;
p->y = a->y - b->y;

return(sqrt(p->x * p->x + p->y * p->y));
}
```

```
/*
 * Each process tries to reduce the size of its path at each
 * temperature drop.  At the end of its iterations it recalculates
 * the length of its path.
 */
void iterate()
{
static float eint[100];          /* for randomization of swapping */
static float erem[1000];         /* "              "              */
int high;                        /* index of top point I work on */
long i;                          /* loop index */
int j, k, l;                     /* loop indices */
int keep;                        /* points I work on */
int low;                         /* index of bottom point I work on */
POINT *mylow;                /* pointer to starting index of my points */
int rem;                     /* remainder when points div. by procs */
static struct rusage start, finish;     /* for timing */
boolean swapped;                 /* true if points interchanged */
POINT *t, *s, *tprime;           /* temporary path pointers */
double x;                        /* for randomization of swapping */

/* Each node sets up randomizing constants for swapping. */
get_grain_start_time( m_myid,  ( 3 + m_myid )  );

randx = (long)(27 * m_myid);
for (i = 0; i < 50; i++)
    eint[i] = exp(-(double)i);
for (i = 0, x = 0.0005; i < 1000; i++, x += 0.001)
    erem[i] = exp(-x);

m_single();

/*
* since the processes are acting on parallel on the same path each will
* will be given a part of the iterations
*/
niter = niter / nprocs;

for (currtemp = itemp, j = 0; currtemp > (ftemp - EPSILON) && currtemp >
0.0;
   currtemp -= drop, j++) {
    m_multi();
    get_current_path(mypath[m_myid] );
    low = 0;
    high = low + npts - 1;
/* Find and save a pointer to the first point in my portion of the path.
*/

    mylow = mypath[m_myid];      /* save the lowest point in my path */
    beta = 1.0 / currtemp;

    /* Move points around */

    for (i = 0L; i < niter; i++) {
        if (keep > 2) {
            /* swap pts within my path? (first point not involved) */
            for (k = low + 1, tprime = mylow->next; k < high - 1; k++) {
                cswap(tprime, &swapped, eint, erem);
```

```
                tprime = tprime->next;
                }
            }
        /* mylow may be swapped in these two calls; want to keep it
           pointing to lowest one in MY path. */
        if (keep > 1) {
            cswap(mylow, &swapped, eint, erem );
         /* swap my first two points? */
            if (swapped) {
                mylow = mylow->prev;     /* back to my lowest */
                }
        }
        cswap(mylow->prev, &swapped, eint, erem );
        /* swap with previous point? */
        if (swapped) {
            mylow = mylow->next;         /* back to my lowest */
            }
        }

    /* Calculate the length of my path and its connection to my next
       neighbor.  Update the shared path length. */

    mypathlen[m_myid] = 0.0;
    for (l = low, t = mylow; l <= high; l++, t = t->next) {
        mypathlen[m_myid] += distance(t, t->next);
        }

/* all processes come together */
    m_sync();
    m_single();
    l = get_smallest_path_index();
    pathlen = mypathlen[l];
    put_current_path(mypath[l]);
/* Calculate iteration time. */
    tlog[j] = currtemp;
    plog[j] = pathlen;
    }
   m_multi() ;
get_grain_stop_time( m_myid,  ( 3 + m_myid  )  );
}

/*
 * Use simulated annealing to approach the solution.
 */
void anneal()
{
int i,j,k;                      /* loop indices */
char yn;                        /* yes/no answer */
char filename[30];              /* log file */
FILE *fp, *fopen();             /* log file pointer */
POINT *s, *t;
get_grain_start_time( 0,  3 )  ;
/* Get input from user. */

printf("\nEnter initial temperature: ");
scanf("%f", &itemp);
printf("Initial temperature is %6.3f\n", itemp);
```

```
printf("\nEnter drop in temperature: ");
scanf("%f", &drop);
printf("Drop in temperature is %6.3f\n", drop);

printf("\nEnter final temperature: ");
scanf("%f", &ftemp);
printf("Final temperature is %6.3f\n", ftemp);

printf("\nEnter the number of iterations per drop in temperature: ");
scanf("%ld", &niter);
printf("Number of iterations per cycle is %ld\n", niter);

/* Set up multiple processes. */

printf("\nEnter number of processors: ");
scanf("%d", &nprocs);
printf("nprocs = %d\n", nprocs);
if (m_set_procs(nprocs) != 0) {
    perror("It didn't work\n");
    printf("I asked for %d processes.\n", nprocs);
    exit(-1);
    }

/* shared memory for each process is generated */
for( i = 0; i < nprocs; i++ )
  mypath[i] = ( POINT * ) shmalloc(sizeof( POINT )* npts);

get_grain_stop_time( 0, 3 ) ;
m_fork(iterate);
}
 /*
 * Write data to file.
 */
void
write_data()
{
int i;                          /* loop index */
FILE *fopen();                  /* for opening file */
char outfile[30];               /* data file names */
FILE *outfp;                    /* file pointer */
POINT *t;                       /* runs along points on path */

do {
    printf("Enter output data file name: ");
    scanf("%s", outfile);

    if  ((outfp = fopen(outfile, "w")) == NULL)
        printf("Error opening output file.\n");
    } while (outfp == NULL);

fprintf(outfp, "%d\n", npts);
t = path;

for (i = 0; i < npts; i++, t = t->next)
    fprintf(outfp, "%6.3f\t%6.3f\n", t->x, t->y);
fclose(outfp);
}
```

```
main()
{
char command;                   /* user-entered */
boolean done;                   /* boolean */
FILE *fopen();                  /* to open file */
int i;                          /* loop index */
static char infile[30];         /* data file names */
static FILE *infp;              /* file pointer */
POINT *t;                       /* runs along points on path */

initialize_clock()  ;
get_grain_start_time(  0  ,  1  );

do {
    printf("Enter name of data file: ");
    scanf("%s", infile);
    printf("\nfile = %s\n", infile);
    if ((infp = fopen(infile, "r")) == NULL)
        printf("Input file not found.\n");
    } while (infp == NULL);

fscanf(infp, "%d", &npts);
if (npts > MAXPTS) {
    printf("Number of points in file (%d) exceeds maximum allowed
(%d)\n",
       npts, MAXPTS);
    exit(1);            /* error exit */
    }

printf("npts = %d\n", npts);
path = (POINT *) shmalloc(sizeof(POINT) * npts);

for (i = 0, t = path; i < npts; i++, t++) {
    fscanf(infp, "%f %f", &(t->x), &(t->y));
#   ifdef DEBUG
    printf("path[%d]:\tx = %6.3f\ty = %6.3f\n", i, t->x, t->y);
#   endif
    }
get_grain_stop_time(  0  ,  1  )  ;

/* Link the points. */
get_grain_start_time(  0  ,  2  )  ;

for (i = 0, t = path; i < npts; i++, t++) {
    if (i == npts - 1)
        t->next = &path[0];
    else
        t->next = &path[i + 1];
    if (i == 0)
        t->prev = &path[npts - 1];
    else
        t->prev = &path[i - 1];
    }
get_grain_stop_time(  0  ,  2  )  ;

fclose(infp);
anneal();
get_grain_start_time(  0  ,  (3  +  nprocs  ))  ;
```

```
write_data();
get_grain_stop_time( 0 , (3 + nprocs) ) ;
}
```

Appendix B : Source Code of Execution Profile Analyzer

```
{                                                              }
{           PROGRAM EXECUTION TRACE ANALYZER                   }
{                                                              }
{              Written By : Anil Kumar Yadav                   }
{                      On : September 01, 1989                 }
{             Last Modified : October 01, 1989                 }
{                                                              }
{--------------------------------------------------------------}
{                                                                  }
{ This program reads the execution trace of a program and          }
{ generated the following reports :-                               }
{     ELGDFReport : a list of grains and their execution           }
{                 and their execution time.                        }
{     MacScheduleReport : A MacSchedular compatible Gnatt          }
{                       chart file to see the Gnatt                }
{                       of actual execution                        }
{     GeneralReport : a file containing overall speedup data       }
{                 and individual processor utilisation             }
{--------------------------------------------------------------}
{--------------------------------------------------------------}
program analyser (ELGDFReport,
               MacScheduleReport,GeneralReport,input,output);

const
      TRUE = 1 ;
      FALSE = 0 ;
      START = 0 ;
      STOP = 1 ;
      SENDBEGIN = 2 ;
      SENDEND = 3 ;
      RECVBEGIN = 4 ;
      RECVEND = 5 ;
      CHARTLENGTH = 25 ;
type
      identifier = ( start, stop, sendBegin, sendEnd, recvBegin,
                  recvEnd ) ;  { trace identifiers }
      text = file of char ;
      tracePointer = ^traceElement ;
      traceElement = record
            traceType : identifier ;
            processorId : integer ;
            grainId : integer ;
            flashTime : integer ;
            startTime : integer ;
            stopTime : integer ;
            prev : tracePointer ;
            next : tracePointer ;
            end ;

      processorPointer = ^processorElement ;
      processorElement = record
            processorId : integer ;
            grainListHead : tracePointer ;
            next : processorPointer ;
            prev : processorPointer ;
            end ;
```

```
      arcPointer = ^arcElement ;
      arcElement = record
            arcType : identifier ;
            sender : integer ;
            receiver : integer ;
            sendBeginTime : integer ;
            sendEndTime : integer ;
            recvBeginTime : integer ;
            recvEndTime : integer ;
            arcName : array[1..100] of char ;
            flashTime : integer ;
            next : arcPointer ;
            prev : arcPointer ;
            end ;

var
      ELGDFReport : text ;
      MacScheduleReport : text ;
      GeneralReport : text ;

      arcHead : arcPointer ;
      processorHead : processorPointer ;
      traceHead : tracePointer ;

{--------------------------------------------------------------------}
function readMarker : integer ;
{                                                          }
{ This function reads in the standard input till it finds the    }
{ keywords 'start' or 'stop' which mark the begining of a        }
{ trace point. It returns 0 for 'start', 1 for 'stop' and 99 for}
{ if the keywords are not found.                          }
{ Called From : getTracePoint                             }
{       Calls : none                                      }
var
      character : char ;
      word :  array[1..100] of char ;
      found : integer ;
      counter : integer ;
      status : integer ;
begin
      status := 99 ;
      found := FALSE ;

      while(( not eof ) and ( found = FALSE )) do
      begin
        counter := 1 ;        { initialize a blank word }
        while( counter <> 101 ) do
        begin
          word[counter] := ' ' ;
          counter := counter + 1 ;
        end ;

        counter := 0 ;        { read word from input }
        read(character) ;

        while((character <> ' ')and( character <> chr(9))
                and ( counter < 100 )) do
        begin
```

```
          counter := counter + 1 ;
          word[counter] := character ;
          read(character) ;
        end ;

      if( word = 'start' ) then     { check if it is an identifier }
        begin
          status := START ;
          found := TRUE ;
        end ;

      if( word = 'stop' ) then      { check if it is an identifier }
        begin
          status :=STOP ;
          found := TRUE ;
        end ;

      if( word = 'sendBegin' ) then { check if it is an identifier }
        begin
          status :=SENDBEGIN ;
          found := TRUE ;
        end ;

      if( word = 'sendEnd' ) then   { check if it is an identifier }
        begin
          status :=SENDEND ;
          found := TRUE ;
        end ;

      if( word = 'recvBegin' ) then { check if it is an identifier }
        begin
          status :=RECVBEGIN ;
          found := TRUE ;
        end ;

      if( word = 'recvEnd' ) then   { check if it is an identifier }
        begin
          status :=RECVEND ;
          found := TRUE ;
        end ;
      end ; { while }

      if( found = TRUE ) then       { return the search status }
       readMarker := status
      else
       readMarker := 99 ;

end ; { readMarker }
{----------------------------------------------------------------}
function getArcPoint ( marker : integer ) : arcPointer ;
var
      temp : arcPointer ;
      counter : integer ;
      character : char ;
      word : array[1..100] of char ;
begin
      new( temp ) ;
```

```
      case marker of
        SENDBEGIN : temp^.arcType := sendBegin ;
        SENDEND   : temp^.arcType := sendEnd ;
        RECVBEGIN : temp^.arcType := recvBegin ;
        RECVEND   : temp^.arcType := recvEnd ;
      end ; { case }

      counter := 1 ;
      while ( counter <> 101 ) do
      begin
        word[counter] := ' ' ;
        counter := counter + 1 ;
      end ;

      counter := 0 ;
      read(character) ;
      while(( character = ' ' ) or ( character = chr(9))) do
        read(character );

      while(( character <> ' ' ) and ( character <> chr(9) )
            and ( counter < 100 ) ) do
      begin
        counter := counter + 1 ;
        word[counter] := character ;
        read(character) ;
      end ;

      temp^.arcName := word ;

      case marker of
        SENDBEGIN, SENDEND : read( temp^.sender ) ;
        RECVBEGIN, RECVEND : read( temp^.receiver) ;
      end ;  { case }

      read(temp^.flashTime ) ;
      readln ;
      getArcPoint := temp ;
end ; { getArcPoint }
{-------------------------------------------------------------}
function getTracePoint( marker : integer ) : tracePointer ;
{                                                  }
{ This function reads in one trace point from the standard      }
{ input and stores it in a newly created record.            }
{ Called From : makeTraceList                           }
{       Calls : readMarker                              }
{                                                  }
var
      temp : tracePointer ;
begin
      new( temp ) ;

      if(( marker = STOP ) or ( marker = START) ) then
       begin
        read( temp^.processorId ) ;
        read( temp^.grainId ) ;
        read( temp^.flashTime ) ;    { read the time stamp }
        readln ;
        temp^.next := nil ;
```

```
        temp^.prev := nil ;
        temp^.startTime := 0 ;
        temp^.stopTime := 0 ;
        if( marker = STOP ) then
          temp^.traceType := stop
        else
          temp^.traceType := start ;
        getTracePoint := temp ;     { return the pointer }
       end
      else
        getTracePoint := nil ;

end ; { getTracePoint }
{------------------------------------------------------------}
procedure swapArcPoints( one, two : arcPointer ) ;
{                                                }
{ This procedure exchanges two trace elements in the linked of  }
{ trace points of all the  processors                         }
{ Called By : orderBySenderId                          }
{     Calls : none                                     }
{                                                }
var
      temp : arcPointer ;
begin
      temp := one^.prev ;
      if ( temp = nil ) then
        begin
        one^.next := two^.next ;
        one^.prev := two ;
        two^.prev := nil ;

        if ( two^.next <> nil ) then
          begin
          two^.next^.prev := one ;
          end;

        two^.next := one ;

        if ( one = arcHead ) then
          arcHead := two ;

        end { if }
      else
        begin
          one^.next := two^.next ;
          one^.prev^.next := two ;
          two^.prev := one^.prev ;
          one^.prev := two ;

          if ( two^.next <> nil ) then
            begin
            two^.next^.prev  := one ;
            end ;

          two^.next := one ;
        end ; { else }
end ; { swapArcPoints }
{------------------------------------------------------------}
```

```
procedure swapTracePoints( one, two : tracePointer ) ;
{                                                     }
{ This procedure exchanges two trace elements in the linked of  }
{ trace points of all the  processors                          }
{ Called By : orderByGrainId                          }
{     Calls : none                                    }
{                                                     }
var
      temp : tracePointer ;
begin
      temp := one^.prev ;
      if ( temp = nil ) then
        begin
        one^.next := two^.next ;
        one^.prev := two ;
        two^.prev := nil ;

        if ( two^.next <> nil ) then
          begin
          two^.next^.prev := one ;
          end;

        two^.next := one ;

        if ( one = traceHead ) then
          traceHead := two ;

        end { if }
      else
        begin
          one^.next := two^.next ;
          one^.prev^.next := two ;
          two^.prev := one^.prev ;
          one^.prev := two ;

          if ( two^.next <> nil ) then
            begin
            two^.next^.prev  := one ;
            end ;

          two^.next := one ;
        end ; { else }
end ; { swapTracePoints }
{---------------------------------------------------------------}
procedure swapProcessors( one, two : processorPointer ) ;
{                                                     }
{ This procedure exchanges two processor records in the global  }
{ linked list of processors.                          }
{ Called By : orderByProcessorId                      }
{    Calls  : none                                    }
{                                                     }
var
      temp : processorPointer ;
begin
      temp := one^.prev ;
      if ( temp = nil ) then
        begin
        one^.next := two^.next ;
```

```
        one^.prev := two ;
        two^.prev := nil ;

        if ( two^.next <> nil ) then
          begin
           two^.next^.prev := one ;
          end;

        two^.next := one ;

        if ( one = processorHead ) then
          processorHead := two ;

        end { if }
      else
        begin
          one^.next := two^.next ;
          one^.prev^.next := two ;
          two^.prev := one^.prev ;
          one^.prev := two ;

          if ( two^.next <> nil ) then
            begin
            two^.next^.prev  := one ;
            end ;

          two^.next := one ;
        end ; { else }
end ; { swapProcessors }
{-----------------------------------------------------------------}
procedure swapProcessorTracePoints( myProcessor : processorPointer ;
                              one, two : tracePointer ) ;
{                                                    }
{ This procedure exchanges the trace points within the linked    }
{ of the tracepoints of a particulat processor.                }
{ Called By : orderByStartTime                                 }
{   Calls   : none                                       }
var
      temp : tracePointer ;
begin
      temp := one^.prev ;
      if ( temp = nil ) then
        begin
        one^.next := two^.next ;
        one^.prev := two ;
        two^.prev := nil ;

        if ( two^.next <> nil ) then
          begin
          two^.next^.prev := one ;
          end;

        two^.next := one ;

        if ( one = myProcessor^.grainListHead ) then
          myProcessor^.grainListHead := two ;

        end { if }
```

```
      else
        begin
          one^.next := two^.next ;
          one^.prev^.next := two ;
          two^.prev := one^.prev ;
          one^.prev := two ;

          if ( two^.next <> nil ) then
            begin
            two^.next^.prev  := one ;
            end ;

          two^.next := one ;
        end ; { else }
end ; { swapProcessorTracePoints }
{----------------------------------------------------------------}
procedure placeArcElement ( newPoint : arcPointer ) ;
var
      temp : arcPointer ;
      found : integer ;
begin
      if ( arcHead = nil ) then
      begin
        case newPoint^.arcType of
          sendBegin : newPoint^.sendBeginTime := newPoint^.flashTime ;
          sendEnd : newPoint^.sendEndTime := newPoint^.flashTime ;
          recvBegin : newPoint^.recvBeginTime := newPoint^.flashTime ;
          recvEnd : newPoint^.recvEndTime := newPoint^.flashTime ;
        end ; { case }

        arcHead := newPoint ;
      end
      else
      begin
        found := FALSE ;
        temp := arcHead ;

        while (( temp <> nil ) and ( found = FALSE )) do
        begin
          if( temp^.arcName = newPoint^.arcName )  then
            found := TRUE
                else
            temp := temp^.next ;
        end; { while }

        if ( found = FALSE ) then
        begin
         case newPoint^.arcType of
          sendBegin : newPoint^.sendBeginTime := newPoint^.flashTime ;
          sendEnd : newPoint^.sendEndTime := newPoint^.flashTime ;
          recvBegin : newPoint^.recvBeginTime := newPoint^.flashTime ;
          recvEnd : newPoint^.recvEndTime := newPoint^.flashTime ;
         end ; { case }

         newPoint^.next := arcHead ;
         arcHead^.prev := newPoint ;
         arcHead := newPoint ;
        end
```

```
         else
         begin
          case newPoint^.arcType of
           sendBegin : begin
                   temp^.sendBeginTime := newPoint^.flashTime ;
                   temp^.sender := newPoint^.sender ;
                   end ;
           sendEnd : begin
                   temp^.sendEndTime := newPoint^.flashTime ;
                   temp^.sender := newPoint^.sender ;
                   end ;
           recvBegin : begin
                   temp^.recvBeginTime := newPoint^.flashTime ;
                   temp^.receiver := newPoint^.receiver ;
                   end ;
           recvEnd : begin
                   temp^.recvEndTime := newPoint^.flashTime ;
                   temp^.receiver := newPoint^.receiver ;
                   end ;
          end ; { case }

          dispose( newPoint ) ;
         end ;
       end ;
end ; { placeArcElemenet }
{-------------------------------------------------------------}
procedure placeTraceElement(  newPoint : tracePointer ) ;
{                                                  }
{ This procedure takes in the raw timestamped trace and then    }
{ recreates the starting and ending times of the grain based on }
{ the marker type. The newly created trace element is put on the}
{ top of the global linked list of such trace elements          }
{ Called By : makeTraceList                             }
{    Calls  : none                                      }
var
      temp : tracePointer ;
      found : integer ;
begin
      { if this is the first element in the list the create the list }
      if ( traceHead = nil ) then
        begin
          if( newPoint^.traceType = start ) then
            newPoint^.startTime := newPoint^.flashTime
          else
            newPoint^.stopTime := newPoint^.flashTime ;

          traceHead := newPoint ;
        end { if }
      else
        begin
          found := FALSE ;
          temp := traceHead ;

          { check if any element related to this grain already exists }
          while (( temp <> nil ) and ( found = FALSE )) do
          begin
            if ( temp^.grainId = newPoint^.grainId ) then
            found := TRUE
```

```
          else
          temp := temp^.next ;
        end ; { while }

        if ( found = FALSE ) then
          begin
          if( newPoint^.traceType = start ) then
            newPoint^.startTime := newPoint^.flashTime
          else
            newPoint^.stopTime := newPoint^.flashTime ;

          newPoint^.next := traceHead ;
          traceHead^.prev := newPoint ;
          traceHead := newPoint ;
          end { if }
        else
          begin
          case newPoint^.traceType of
              stop :
                    temp^.stopTime   := newPoint^.flashTime ;
            start :
                    temp^.startTime   := newPoint^.flashTime ;
          end ; { case }
          end ; { else }

      end ; { else }

end ; { placeTraceElement }
{---------------------------------------------------------------}
procedure makeTraceList ;
{                                                   }
{ This procedure creates the initial global linked list of      }
{ grains, their mapping on processes and their start and stop   }
{ times.                                            }
{ Called By : initialize                                 }
{   Calls    : getTracePoint, placeTracePoint                 }
var
      kuka : tracePointer ;
      temp : arcPointer ;
      marker : integer ;
begin

      while ( not eof ) do
      begin
        marker := readMarker ;
        case marker of
          START, STOP : begin
                      kuka := getTracePoint( marker ) ;
                      placeTraceElement( kuka ) ;
                     end ;
          SENDBEGIN, SENDEND, RECVBEGIN, RECVEND :
                     begin
                       temp := getArcPoint( marker ) ;
                       placeArcElement( temp ) ;
                     end ;
          99 : ;
        end ; { case }
      end ; { while }
```

```
end ; { makeTraceList }
{---------------------------------------------------------------}
procedure orderBySenderId ;
{                                                   }
{ This procedure goes through the global linked list of trace    }
{ elements and arranges the grains by their grain Id in an  }
{ ascending order.                                       }
{ Called By : initialize                                 }
{   Calls   :  swapArcPoints                             }
var
      temp : arcPointer ;
      kuka : arcPointer ;
      changed : integer ;
begin
      changed := TRUE ;

      while ( changed = TRUE ) do
      begin
       temp := arcHead ;
       changed := FALSE ;

       kuka := temp^.next ;
       while ( kuka  <> nil ) do
         begin
           if ( temp^.sender > temp^.next^.sender ) then
             begin
             swapArcPoints( temp, temp^.next ) ;
             changed := TRUE ;
             end ;

           temp := temp^.next ;

           if ( temp = nil ) then
             kuka := temp
           else
             kuka := temp^.next ;
         end ; { while }
      end ; { while }
end ; { orderBySenderId }
{---------------------------------------------------------------}
procedure orderByGrainId ;
{                                                   }
{ This procedure goes through the global linked list of trace    }
{ elements and arranges the grains by their grain Id in an  }
{ ascending order.                                       }
{ Called By : initialize                                 }
{   Calls   :  swapTracePoints                                   }
var
      temp : tracePointer ;
      kuka : tracePointer ;
      changed : integer ;
begin
      changed := TRUE ;

      while ( changed = TRUE ) do
      begin
       temp := traceHead ;
       changed := FALSE ;
```

```
        kuka := temp^.next ;
        while ( kuka  <> nil ) do
          begin
            if ( temp^.grainId > temp^.next^.grainId ) then
              begin
              swapTracePoints( temp, temp^.next ) ;
              changed := TRUE ;
              end ;

            temp := temp^.next ;

            if ( temp = nil ) then
              kuka := temp
            else
              kuka := temp^.next ;
          end ; { while }
      end ; { while }
end ; { orderByGrainId }
{------------------------------------------------------------}
procedure orderByStartTime( tempProcessor : processorPointer ) ;
{                                                          }
{ This procedure arranges the trace elements for a processor     }
{ in the increasing order of start time, which is used for       }
{ creating MacSchedule report file.                        }
{  Called By : orderByProcessorId                          }
{    Calls   : swapProcessorTracePoints                          }
var
      temp : tracePointer ;
      kuka : tracePointer ;
      changed : integer ;
begin
      changed := TRUE ;

      while ( changed = TRUE ) do
      begin
       temp := tempProcessor^.grainListHead ;
       changed := FALSE ;

       kuka := temp^.next ;
       while ( kuka  <> nil ) do
         begin
           if ( temp^.startTime > temp^.next^.startTime ) then
             begin
             swapProcessorTracePoints(tempProcessor, temp, temp^.next )
;
             changed := TRUE ;
             end ;

           temp := temp^.next ;

           if ( temp = nil ) then
             kuka := temp
           else
             kuka := temp^.next ;
         end ; { while }
      end ; { while }
end ; { orderByStartTime }
```

```
{--------------------------------------------------------------}
procedure regroupTraceByProcessor ;
{                                                   }
{ This procedure reads in the global linked list of the trace   }
{ elements and makes a linked list of processors with each      }
{ processor having a linked list of grains which were mapped on }
{ itself.                                           }
{ Called By : initialize                                 }
{   Calls   : none                                       }
var
      found : integer ;
      tempHead : processorPointer ;
      tempTrace : tracePointer ;
      temp : tracePointer ;
begin
      temp := traceHead ;

      while( temp <> nil ) do
      begin
        new(tempTrace);
        tempTrace^.processorId := temp^.processorId ;
        tempTrace^.grainId := temp^.grainId ;
        tempTrace^.startTime := temp^.startTime ;
        tempTrace^.stopTime := temp^.stopTime ;
        tempTrace^.next := nil ;
        tempTrace^.prev := nil ;

        if ( processorHead = nil ) then
          begin
            new( processorHead ) ;
            processorHead^.processorId := tempTrace^.processorId ;
            processorHead^.grainListHead := tempTrace ;
          end      { if }
        else
          begin
            found := FALSE ;
            tempHead := processorHead ;

             while ( ( found = FALSE ) and ( tempHead <> nil ) ) do
             begin
             if ( tempHead^.processorId = tempTrace^.processorId ) then
               found := TRUE
               else
               tempHead := tempHead^.next ;
             end; { while }

             if ( found = TRUE ) then
             begin
               tempTrace^.next := tempHead^.grainListHead ;
               tempHead^.grainListHead^.prev := tempTrace ;
               tempHead^.grainListHead := tempTrace ;
             end { if }
             else
             begin
               new( tempHead ) ;
               tempHead^.prev := nil ;
               tempHead^.grainListHead := tempTrace ;
               tempHead^.processorId := tempTrace^.processorId ;
```

```
               processorHead^.prev := tempHead ;
               tempHead^.next := processorHead ;
               processorHead := tempHead ;
             end ; { else }
          end ; { else }

      temp := temp^.next ;
      end ; { while }
end; { regroupTraceByProcessor }
{-----------------------------------------------------------------}
procedure orderByProcessorId ;
{                                                 }
{ This procedure rearranges the linked list of processors to      }
{ give an ascending order. Also within each process it            }
{ rearranges the list of trace elements in ascnding order by    }
{ by the start time.                                      }
{ Called By : initialize                                  }
{   Calls    : swapProcessors, orderByStartTime                }
var
      temp : processorPointer ;
      kuka : processorPointer ;
      changed : integer ;
begin
      changed := TRUE ;

      while ( changed = TRUE ) do
      begin
       temp := processorHead ;
       changed := FALSE ;

       kuka := temp^.next ;
       while ( kuka  <> nil ) do
         begin
           if ( temp^.processorId > temp^.next^.processorId ) then
             begin
             swapProcessors( temp, temp^.next ) ;
             changed := TRUE ;
             end ;

           temp := temp^.next ;

           if ( temp = nil ) then
             kuka := temp
           else
             kuka := temp^.next ;
         end ; { while }
      end ; { while }

      temp := processorHead ;

      while( temp <> nil ) do
      begin
        orderByStartTime( temp ) ;
        temp := temp^.next ;
      end ; { while }
end ; { orderByProcessorId }
{-----------------------------------------------------------------}
function getTraceCount( head : tracePointer ) : integer ;
```

```
{                                                    }
{ This function goes through a list of trace elements and    }
{ returns the number of elements present in the list.        }
{  Called By : makeMacScheduleReport, makeGeneralReport            }
{    Calls    : none                                       }
var
      tempCount : integer ;
      temp : tracePointer ;
begin
      tempCount := 0 ;
      temp := head ;

      while ( temp <> nil ) do
      begin
        tempCount := tempCount + 1 ;
        temp := temp^.next ;
      end ; { while }

      getTraceCount := tempCount ;
end ; { getTraceCount }
{----------------------------------------------------------------}
function getProcessorCount : integer ;
{                                                    }
{ This function counts the number of elements present in the       }
{ linked list of processors.                               }
{  Called By : makeMacScheduleReport                             }
{    Calls    : none                                       }
var
      tempCount : integer ;
      tempProc : processorPointer ;
begin
      tempCount := 0 ;
      tempProc := processorHead ;

      while ( tempProc <> nil ) do
      begin
        tempCount := tempCount + 1 ;
        tempProc := tempProc^.next ;
      end ;

      getProcessorCount := tempCount ;
end ; { getProcessorCount }
{----------------------------------------------------------------}
function getTotalExecutionTime : integer ;
{                                                    }
{ This function goes through the linked list of all the trace      }
{ elements and yeilds the overall execution time. Overall    }
{ execution time is the higest value of the stop time among all }
{ the grains.                                              }
{  Called By : makeGeneralReport                           }
{    Calls    : none                                       }
var
      temp : tracePointer ;
      maxTime : integer ;
begin
      maxTime := 0 ;
      temp := traceHead ;
```

```
      while ( temp <> nil ) do
      begin
        if ( temp^.stopTime > maxTime ) then
          maxTime := temp^.stopTime ;

        temp := temp^.next ;
      end ;

      getTotalExecutionTime := maxTime ;
end ; { getTotalExecutionTime }
{----------------------------------------------------------------}
function getActualStartTime : integer ;
{                                                        }
{ This function goes through the linked list of all the trace      }
{ elements and yeilds the actual start time . Overall       }
{ execution time is the higest value of the stop time among all }
{ the grains.                                                 }
{  Called By : makeGeneralReport                              }
{    Calls   : none                                           }
var
      temp : tracePointer ;
      minTime : integer ;
begin
      minTime := 9999999 ;
      temp := traceHead ;

      while ( temp <> nil ) do
      begin
        if ( temp^.startTime < minTime ) then
          minTime := temp^.startTime ;

        temp := temp^.next ;
      end ;

      getActualStartTime := minTime ;
end ; { getActualStartTime }
{----------------------------------------------------------------}
procedure makeMacScheduleFile ;
{                                                        }
{ This procedure goes through the linked list of the processors }
{ and prints a file named 'MacScheduleReport' in the format of  }
{ of the MacSchedule 1.0 Gnatt Chart.                         }
{  Called By : program                                        }
{    Calls   : getTraceCount, getProcessorTime,               }
{            getTotalExecutionTime,                           }
var
      tempProc : processorPointer ;
      tempTrace : tracePointer ;
      count : integer ;
      factor : integer ;
begin
      rewrite(MacScheduleReport);
      writeln( MacScheduleReport,'Schedule',
        '   < Identifies this as a schedule file >' ) ;

      count := getTraceCount( traceHead ) ;
      writeln( MacScheduleReport, count,
        '   < Total Number of Tasks on All Processors >' ) ;
```

```
      count := getProcessorCount ;
      writeln( MacScheduleReport, count,  '     1.0',
        '   < Number of Processors, Transfer Rate > ' );

      count := getTotalExecutionTime ;
      writeln('The total execution time for this program was ',count,
             ' msec');
      factor := 1 ; {count div CHARTLENGTH ; }
      writeln( ' The Gantt Chart and Task Graph have been scaled by : ',
              factor,'  for proper display');
      tempProc := processorHead ;

      while( tempProc <> nil ) do
      begin
        writeln( MacScheduleReport, tempProc^.processorId,
          '    < Processor# >') ;

        count := getTraceCount( tempProc^.grainListHead ) ;
        writeln( MacScheduleReport, count,
          '   < Number of Tasks on this processor > ');

        tempTrace := tempProc^.grainListHead ;

        while( tempTrace <> nil ) do
        begin
          write( MacScheduleReport, tempTrace^.grainId );
          write( MacScheduleReport, tempTrace^.startTime div factor );
          write( MacScheduleReport, tempTrace^.stopTime div factor  );
          count := 0 ;
          writeln( MacScheduleReport, count, count );
          tempTrace := tempTrace^.next ;
        end ;
        tempProc := tempProc^.next ;
      end ;
end ; { makeMacScheduleFile }
{-----------------------------------------------------------------}
function getRealExecutionTime( temp : tracePointer ) : integer ;
{ This procedure totals the execution time of all the grains to }
{ indicate the amount of time it will take if the program were  }
{ to be executed in serial fashion.                        }
{  Called By : makeGeneralReport                           }
{    Calls   : none                                        }
var
      time : integer ;
begin
      time := 0 ;

      while( temp <> nil ) do
      begin
        time := time + ( temp^.stopTime - temp^.startTime ) ;
        temp := temp^.next ;
      end ;

      getRealExecutionTime := time ;
end ; { getRealExecutionTime }
{-----------------------------------------------------------------}
procedure makeGeneralReport ;
```

```
{                                                        }
{ This procedure prints the information regarding overall    }
{ performance of the program and also utilzation by each of the    }
{ the processor and grains                                  }
{ Called By : program                                       }
{  Calls    : getTotalExecutionTime, getRealExecutionTime    }
var
      count, totalTime, goodTime : integer ;
      efficiency : real ;
      tempTrace : tracePointer ;
      tempProc : processorPointer ;
begin
      totalTime := getTotalExecutionTime ;
      goodTime := getRealExecutionTime( traceHead );

      rewrite(GeneralReport) ;
      writeln(GeneralReport,'                          Speed up = ',
                   (goodTime / totalTime):10:1 );
      count := getProcessorCount ;
      efficiency := ( (goodTime * 100) /( totalTime * count )) ;
      writeln(GeneralReport,'    Overall Processor Utilization = ',
                   efficiency:10:1,' %' ) ;
      writeln(GeneralReport) ;
      tempProc := processorHead ;

      while( tempProc <> nil ) do
      begin
        writeln(GeneralReport,'     Data on Processor ',
                       tempProc^.processorId:6 ) ;
        goodTime := getRealExecutionTime( tempProc^.grainListHead );
        writeln(GeneralReport,'     Utilisation = ',
                      (goodTime*100/totalTime):10:1,' %' );
        writeln( GeneralReport,'     Grain id  % time ');
        tempTrace := tempProc^.grainListHead ;

        while ( tempTrace <> nil ) do
        begin
          writeln(GeneralReport,tempTrace^.grainId,
                ((tempTrace^.stopTime - tempTrace^.startTime)
                       * 100 / totalTime):10:1 ) ;
          tempTrace := tempTrace^.next ;
        end ; { while }

        tempProc := tempProc^.next ;
        writeln(GeneralReport) ;
      end ; { while }
end ; { makeGeneralReport }
{----------------------------------------------------------------}
function getArcCount : integer ;
var
      temp : arcPointer ;
      count : integer ;
begin
      temp := arcHead ;
      count := 0 ;

      while ( temp <> nil ) do
      begin
```

```
        count := count + 1 ;
        temp := temp^.next ;
      end ; { while }

      getArcCount := count ;
end ; { getArcCount }
{-----------------------------------------------------------------}
function getNumberOfSuccessor( nodeId : integer ) : integer ;
var
      temp : arcPointer ;
      count : integer ;
begin
      count := 0 ;
      temp := arcHead ;

      while( temp <> nil ) do
      begin
        if ( temp^.sender = nodeId ) then
          count := count + 1 ;
        temp := temp^.next ;
      end ; { while }

      getNumberOfSuccessor := count ;
end ; { getNumberOfSuccessor }
{-----------------------------------------------------------------}
function getNumberOfPredecessor( nodeId : integer ) : integer ;
var
      temp : arcPointer ;
      count : integer ;
begin
      count := 0 ;
      temp := arcHead ;

      while( temp <> nil ) do
      begin
        if ( temp^.receiver = nodeId ) then
          count := count + 1 ;
        temp := temp^.next ;
      end ; { while }

      getNumberOfPredecessor := count ;
end ; { getNumberOfPredecessor }
{-----------------------------------------------------------------}
procedure makeELGDFfile ;
{                                                }
{ This procedure reads the global linked list of the grains and }
{ prints out the grain Id and the execution time in the file    }
{ named 'ELGDFReport'                                  }
{ Called By : program                                  }
{    Calls  : none                                     }
var
      factor : integer ;
      top : integer ;
      left : integer ;
      count : integer ;
      mynum : integer ;
      delay : integer ;
      temp : tracePointer ;
```

```
      tempArc : arcPointer ;
      executionTime : integer ;
begin
      rewrite( ELGDFReport ) ;
      count := getTotalExecutionTime ;
      factor := 1 ; { count div CHARTLENGTH ; }
      writeln( ELGDFReport, 'Task_Graph  <Identifies a Task Graph file
>' );
      temp := traceHead ;
      count := getTraceCount( traceHead ) ;
      writeln( ELGDFReport, count,' < Total Number of Nodes >' );
      count := 0 ;
      top := 40 ;
      left := 80 ;

      while( temp <> nil ) do
      begin
        write(ELGDFReport, ( temp^.grainId + 1 ) ) ;
        executionTime := temp^.stopTime - temp^.startTime ;
        write(ELGDFReport, ( executionTime div factor ) ) ;
        write(ELGDFReport, top ) ;
        write(ELGDFReport, left ) ;
        mynum := getNumberOfSuccessor( temp^.grainId ) ;
        write(ELGDFReport, mynum) ;
        mynum := getNumberOfPredecessor( temp^.grainId ) ;
        write(ELGDFReport, mynum) ;
        writeln(ELGDFReport) ;
        count := count + 1 ;
        top :=  40 + 40 * ( count div 5 )  ;
        left := 80 + 80 * ( count mod 5 ) ;
        temp := temp^.next ;
      end ; { while }

      tempArc := arcHead ;
      mynum := getArcCount ;
      writeln(ELGDFReport, mynum,'  < Total Number of Edges >') ;

      while( tempArc <> nil ) do
      begin
        delay := ( tempArc^.sendEndTime - tempArc^.sendBeginTime ) ;
        delay := delay+(tempArc^.recvEndTime - tempArc^.recvBeginTime) ;
        write(ELGDFReport, (tempArc^.sender + 1 )) ;
        write(ELGDFReport, ( tempArc^.receiver + 1 )) ;
        writeln(ELGDFReport, ( delay div factor )) ;
        tempArc := tempArc^.next ;
      end ;
end ; { makeELGDFfile }
{----------------------------------------------------------------}
procedure  getCorrectSpeedup ;
var
      startTime : integer ;
      stopTime : integer ;
      pcount : integer ;
      speed : real ;
      realTime : integer ;
begin
      pcount := getProcessorCount ;
      stopTime := getTotalExecutionTime ;
```

```
      realTime := getRealExecutionTime( traceHead ) ;
      startTime := getActualStartTime ;
      speed := ( realTime ) / ( stopTime - startTime ) ;
      writeln('processors = ',pcount,'   speedup is =',speed:10:2 );
end ;
{----------------------------------------------------------------}
procedure initialize ;
{                                                  }
{ This procedure makes calls to appropriate routines to     }
{ create the linked lists of trace elements, processors, arrange}
{ them in order suitable for proper file generation         }
{  Called By : program                                 }
{    Calls   : see inside                              }
begin
      processorHead := nil ;
      traceHead := nil ;
      arcHead := nil ;
      makeTraceList ;
      orderByGrainId ;
      regroupTraceByProcessor ;
      orderByProcessorId ;
      orderBySenderId ;
end; { initialize }
{----------------------------------------------------------------}
begin { program }

      initialize ;
      makeELGDFfile ;
      makeMacScheduleFile ;
      makeGeneralReport ;
end. { program }
```

Appendix C : Instrumented Code generated by SuperGlue

```
#include <linda.h>
#include "time_linda_c.h"
#include <stdio.h>
    /*It worked *********************!!*/
real_main()
{
    int df0;
    int df1;
    int df2;
    int df3;
    int df4;
    int df5;
    int df6;
    int i;
     initialize_clock();
    scheduler();
}  /* End of MainLine!! */
df0()
{
    int i,l=1,m,n, workers,processors;
    int start1,start2,start3,start4,start5;
    int stop1,stop2,stop3,stop4,stop5;
    double interval,h;

     get_grain_start_time(1,0);
    n = 100000;
    interval = 1.0/n;

    start1=1;
    start2=20000;
    start3=40000;
    start4=60000;
    start5=80000;
    stop1=19999;
    stop2=39999;
    stop3=59999;
    stop4=79999;
    stop5=100000;

     get_grain_stop_time(1,0);

     get_send_begin_time(0,"superGlue06");
    out("superGlue06", interval);
     get_send_end_time(0,"superGlue06");

     get_send_begin_time(0,"superGlue05");
    out("superGlue05", interval, start5, stop5);
     get_send_end_time(0,"superGlue05");

     get_send_begin_time(0,"superGlue04");
    out("superGlue04", interval, start4, stop4);
     get_send_end_time(0,"superGlue04");

     get_send_begin_time(0,"superGlue03");
    out("superGlue03", interval, start3, stop3);
     get_send_end_time(0,"superGlue03");
```

```
    get_send_begin_time(0,"superGlue02");
    out("superGlue02", interval, start2, stop2);
    get_send_end_time(0,"superGlue02");

    get_send_begin_time(0,"superGlue01");
    out("superGlue01", interval, start1, stop1);
    get_send_end_time(0,"superGlue01");
}  /* End of function df0*/
df1()
{
    int start1,stop1;
    int i;
    double interval,x,result1=0.0;

    get_recv_begin_time(1,"superGlue01");
    in("superGlue01", ?interval,?start1,?stop1);
    get_recv_end_time(1,"superGlue01");

    get_grain_start_time(2,1);
    for(i=start1;i<=stop1;++i)
     {
       x=(i-0.5)*interval;
       result1+=4.0/(1.0+x*x);
     }

    get_grain_stop_time(2,1);

    get_send_begin_time(1,"superGlue16");
    out("superGlue16", result1);
    get_send_end_time(1,"superGlue16");
}  /* End of function df1*/
df2()
{
    int start2,stop2;
    int i;
    double interval,x,result2=0.0;

    get_recv_begin_time(2,"superGlue02");
    in("superGlue02", ?interval,?start2,?stop2);
    get_recv_end_time(2,"superGlue02");

    get_grain_start_time(1,2);
    for(i=start2;i<=stop2;++i)
     {
       x=(i-0.5)*interval;
       result2+=4.0/(1.0+x*x);
     }

    get_grain_stop_time(1,2);

    get_send_begin_time(2,"superGlue26");
    out("superGlue26", result2);
    get_send_end_time(2,"superGlue26");

}  /* End of function df2*/
df3()
{
```

```
    int start3,stop3;
    int i;
    double interval,x,result3=0.0;

    get_recv_begin_time(3,"superGlue03");
    in("superGlue03", ?interval,?start3,?stop3);
    get_recv_end_time(3,"superGlue03");

    get_grain_start_time(1,3);
    for(i=start3;i<=stop3;++i)
     {
       x=(i-0.5)*interval;
       result3+=4.0/(1.0+x*x);
     }

    get_grain_stop_time(1,3);

    get_send_begin_time(3,"superGlue36");
    out("superGlue36", result3);
    get_send_end_time(3,"superGlue36");
}  /* End of function df3*/
df4()
{
    int start4,stop4;
    int i;
    double interval,x,result4=0.0;

    get_recv_begin_time(4,"superGlue04");
    in("superGlue04", ?interval,?start4,?stop4);
    get_recv_end_time(4,"superGlue04");

    get_grain_start_time(1,4);
    for(i=start4;i<=stop4;++i)
     {
       x=(i-0.5)*interval;
       result4+=4.0/(1.0+x*x);
     }

    get_grain_stop_time(1,4);

    get_send_begin_time(4,"superGlue46");
    out("superGlue46", result4);
    get_send_end_time(4,"superGlue46");
}  /* End of function df4*/
df5()
{
    int start5,stop5;
    int i;
    double interval,x,result5=0.0;

    get_recv_begin_time(5,"superGlue05");
    in("superGlue05", ?interval,?start5,?stop5);
    get_recv_end_time(5,"superGlue05");

    get_grain_start_time(2,5);
    for(i=start5;i<=stop5;++i)
     {
       x=(i-0.5)*interval;
```

```
        result5+=4.0/(1.0+x*x);
      }

     get_grain_stop_time(2,5);
     get_send_begin_time(5,"superGlue56");
    out("superGlue56", result5);
     get_send_end_time(5,"superGlue56");
}  /* End of function df5*/
df6()
{
    double result1,result2,result3,result4,result5;
    double h,pi_approx=0.0, interval;

     get_recv_begin_time(6,"superGlue56");
    in("superGlue56", ?result5);
     get_recv_end_time(6,"superGlue56");

     get_recv_begin_time(6,"superGlue46");
    in("superGlue46", ?result4);
     get_recv_end_time(6,"superGlue46");

     get_recv_begin_time(6,"superGlue36");
    in("superGlue36", ?result3);
     get_recv_end_time(6,"superGlue36");

     get_recv_begin_time(6,"superGlue26");
    in("superGlue26", ?result2);
     get_recv_end_time(6,"superGlue26");

     get_recv_begin_time(6,"superGlue16");
    in("superGlue16", ?result1);
     get_recv_end_time(6,"superGlue16");

     get_recv_begin_time(6,"superGlue06");
    in("superGlue06", ?interval);
     get_recv_end_time(6,"superGlue06");

     get_grain_start_time(1,6);
    pi_approx=result1+result2+result3+result4+result5;
    pi_approx=pi_approx*interval;
    printf("pi approximation %20.15lf\n",pi_approx);

     get_grain_stop_time(1,6);
}  /* End of function df6*/
gant1() {
     df0();
     df2();
     df3();
     df4();
     df6();
     out("gant_done");
      }  /*end of gantt chart 1 */
gant2() {
     df1();
     df5();
     out("gant_done");
      }  /*end of gantt chart 2 */
scheduler()
```

```
{
    eval(gant1());
    eval(gant2());
    in("gant_done");
    in("gant_done");
}
```