Association for Information Systems
AIS Electronic Library (AISeL)

PACIS 1993 Proceedings

Pacific Asia Conference on Information Systems (PACIS)

December 1993

# A Visualizer for Observing Parallel Program Behavior

Ay-Hwa Liou
*Tamkang University*

R. Loganantharaj
*Center for Advanced Computer Studies*

Follow this and additional works at: http://aisel.aisnet.org/pacis1993

Recommended Citation
Liou, Ay-Hwa and Loganantharaj, R., "A Visualizer for Observing Parallel Program Behavior" (1993). *PACIS 1993 Proceedings*. 10.
http://aisel.aisnet.org/pacis1993/10

This material is brought to you by the Pacific Asia Conference on Information Systems (PACIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in PACIS 1993 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

# A Visualizer for Observing Parallel Program Behavior

Ay-Hwa Andy Liou
Dept. of Information Management
Tamkang University
Tamsui, Taipei, Taiwan, ROC
email:tkut093@twnmoe10.edu.tw

R. Loganantharaj
Center for Advanced Computer Studies
USL P.O. Box 44330
Lafayette, LA 70504, USA
email:logan@cacs.usl.edu

## Abstract

In an environment for developing and running parallel programs, it is not always possible to capture effectively the true behaviors of the concurrently running processes. The degree of complexity resulted from the parallel execution and the unavailability of the suitable facilities contribute considerably to this situation. This paper introduces the experiences of developing a tracing mechanism for visually observing and studying the behaviors of a parallel automated reasoning system. This visualizer is capable of displaying the execution status of the multiprocessors as well as analyzing the speedups and related informations. It has been successfully implemented on the SunView windowing environment. Also discussed in the paper are the improvements upon a parallelism made possible by employing the information obtained from the visualizer.

## 1 Introduction

The availability of the multiprocessors makes it possible to apply parallelism to many interesting algorithms and improve the overall performance. However, it is not straightforward to design and implement an effective algorithm for a given parallel mechanism. Among the difficulties, the high degree of complexity and the large amount of information generated form the parallel executions are the most apparent one. Since the processors are running in parallel, their executions are blended together and their outputs are a mixture of different types of informations. To sort out this composite of output and solve this *behavioral comprehension* problem, one must provide more information, such as the identification of a processor which generated a particular output, or direct the output of different processors to different places which can be files or different areas of the screen. However, the outputs are usually not well-organized and none of these methods serves the purpose of easy understanding of the parallel behaviors or debugging the parallel programs.

During our process of developing a parallel automated reasoning system, the desire of obtaining the behavioral comprehension was even stronger. The overall control over an automated reasoning system is crucial for a successful result to be derived. For the purpose of being able to direct an automated reasoning system to a fruitful direction and therefore improve the speed of the system, the details about the specific operations that each processor is carrying out at a specific time is very helpful. The design and implementation of a tracing tool (a visualizer) for monitoring the executions of each processor and therefore allowing the comprehension of the parallel behaviors is thus initiated.

The SUN workstation is chosen to be the platform for this visualizer. The parallel automated reasoning system is implemented in C on an Encore Multimax using a package for writing parallel program. This package, form the Argonne National Laboratory, contains most of the constructs which is useful for implementing parallel programs. The tracing message generated from the execution of the parallel reasoning system is transported form the Encore Multimax to the SUN and displayed in the SunView windowing environment. This mechanism allows the researchers to visually observe the behaviors of the parallelism by observing what kind of operations a particular process is doing during concurrent execution. The characteristic of parallelization for a particular problem can be studied using this tool. This mechanism will not directly help the execution of the parallel system in an obvious way, but will enhance the knowledge of the researcher about the parallelism and subsequently improve the algorithm.

Described in the next section is the parallel automated reasoning system which is the program the visualizer is tracing for. The parallelism governing the execution of the automated reasoning system is also presented. It also introduces the parallel package which is used for implementing our parallel reasoning system. The tracing tool itself is presented in Section three along with some example. Section four discusses how this tool can help to improve the parallelism. A summary of this paper is given in Section five.

## 2 Parallel Automated Reasoning

Automated reasoning offers an elegant framework for solving several interesting but difficult problems such as program verification, logic circuit design, and solving some open questions in mathematics. The success of these applications make automated reasoning a promising area in AI. However, the process of reasoning is, in general, combinatorially explosive. Effective control strategy will lead the operations of a reasoning system towards a fruitful direction, such as a proof or a desirable derivation. Unfortunately, even with an effective control strategy, the search space of a deductive proof grows exponentially. Applying parallelism is one of the methods which may improve the performance of the automated reasoning systems.

We take advantages of the explicit availability of the growing search space of the connection graph representation to study different kinds of parallelism systematically. A *connection graph* is a schema for representing a set of first-order clauses in a refutational proof [Kow75]. The parallel link resolution [LL90] of connection graphs have been categorized into (1) global parallelism and (2) local parallelism. OR and AND parallelism are examples of local parallelism. Local parallelism is suitable for densely connected graphs while global parallelism, such as DC parallelism [Log87], is appropriate for larger sparsely connected graphs.

The basic operation in a connection graph refutation is *link resolution*, in which a link is selected and the pair of clauses incident to the link is resolved. In parallel link resolution each process is responsible for resolving a link: create a resolvent, update the inherited links, and finally remove the link being resolved. Consider resolving a link connecting to *C1* and *C2*. The information on the inherited links are obtained by reading both *C1* and *C2*. The process then updates a resolvent and modifies the inherited clauses to establish the inherited links between the resolvent and the inherited clause. For more details of connection graph refutation refer to [Kow75, SP82, SP84].

Practicing different strategies of selecting parallel links resulted in different parallelism. For example, in DC parallelism, links connecting to distinct clauses are resolved concurrently, while, in OR parallelism, links connecting to a specific literal, called *sun* literal, are resolved concurrently. The links selected by a particular parallelism to solve concurrently are called *parallel links*. Consider the connection graph of Figure 1. The parallel link sets of the graph that can be solved by DC parallelism include {1,5}, {2,6,4}, {1,6,7}, {3,8}, while the parallel links of the OR parallelism will be link number 4,5,7, and 8 if the clause *C* is selected as the sun literal. For a successful application of any parallelism, it is essential to have an algorithm that obtains the *parallel links* with a minimum overhead. In extended DC parallelism, parallel links are selected by partitioning the graph such that links that are selected one from each partition form parallel links. Most of the examples in this paper, used for explaining the functions of the tracer, are selected from running extended DC parallelism.

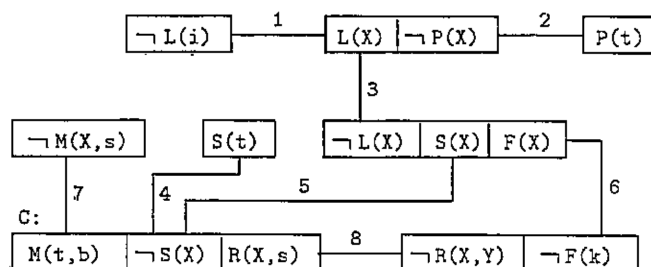


**Figure 1**: A Connection Graph example.

Logical inconsistency has been observed when practicing each different parallelism [Log85, HKK81]. By *logical inconsistency*, we mean the derivation of both the empty clause and the empty graph from the same graph when parallel links are resolved concurrently. The derivation of the empty clause means that the hypothesis is proved while the derivation of the empty graph means that the same hypothesis is not proved. Logical inconsistency occurs in parallel link resolution because parallel processes access and manipulate the same shared resource concurrently.

The necessary and sufficient conditions for a correct parallel link resolution can be easily satisfied by only allowing the access to the shared resources to happen within a critical section. The possible failure of the sufficient condition suggests that it may be possible to have a circular waiting and hence a deadlock situation. The formal details analyzing the possible deadlock situations in different parallelisms were shown in [LL91]. We use a higher level parallel proof procedure to refer to all the different parallelism which will select their parallel links based on the employing parallelism and solve these parallel links concurrently.

## 2.1 Task Pooling

In parallel processing on shared memory system, there are usually tasks to be distributed, or assigned to different processes for solving. The scheduling of the task distribution can be done in dissimilar fashions. In *pre-scheduling,* the assignments of tasks to processes are decided before the processes actually started working, while, in *self-scheduling*, the assignings are performed "on-the-go." A simple example of different scheduling would be the addition of two matrices. Pre-scheduling would require designated indexes to be assigned to identified processes before the addition actually begin. For example, the addition of two 3×3 matrices would assign each row to different process in pre-scheduling, while, in self-scheduling, each process may dynamically pick up one element of the matrix for addition and then come back to pick up next available one, without a pre-determined order.

For the purpose of executing our parallel algorithm, it is necessary to use the self-scheduling approach because the tasks are not pre-determinable. There is no way to know, before hand, which link is going to be resolved and which nodes are the inherited nodes. It is therefore necessary to distribute them once they become available.

It is adequate to use *task pooling* to realize the self-scheduling mechanism. Before describing the task pooling, we first identify the *master* process and the *slave* processes. The 'master process', which is the original process created by the operation system, is responsible for creating and depositing tasks into the pool, while the set of 'slave processes' are responsible for link resolution.

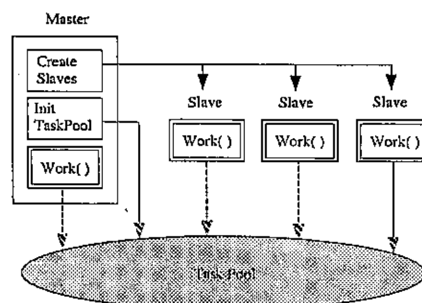


**Figure 2**: The conceptual diagram of the task pooling.

In task pooling, the tasks are created and deposited into a *task pool*. Figure 2 illustrates the relationships between the master process and the slave processes. The master creates the slaves and then places initial tasks into the task pool. After depositing tasks into the task pool, the master picks up a task from the task pool and behaves like a slave in terms of resolving links. The master joins with the slaves to solve the problems by calling the same subroutine, Work( ), called by the slaves. Tasks also can be created, and deposited into the task pool, after all the processes starts working. If a process cannot get a task because the task pool is empty, it should wait, somewhere inside the monitor, until all the processes are waiting.

## 2.2 The Macro Package

Although primitive iperations are always available for most of the parallel machines, they are usually low level and insufficient for parallel operations that requires more complicated control and coordination. A more sophisticated construct is needed for complex operations such as task pooling in our algorithm.

The parallel proof procedure discussed in this paper is implemented using a macro package [BBD87] of Argonne National Laboratory (call it MP from now on) which provides rich constructs for parallel programming. There are several reasons for using MP. First, it is easier to encode the parallel programs than using the primitives of the machine, provided that a proper construct that well matched to the problem is used. Before a programmer can master the skill of programming in parallel, extensive practice must acquired. Using MP can organize and standerize the parallel concepts that enable the user to write logical or structural parallel programs without low level planning.

Another advantage, as claimed by Argonne, is that MP will not impair the performance of parallel program. As evidented in our experiment, this is an important feature any parallel package should possess. Finally, MP is portable. The parallel constructs provided in the macro package can be viewed as an intermediate language for parallel programming. Thus a program developed with the package can run on several parallel machines, provided that one uses the appropriate macro expansion for the selected underlying parallel machine.

In this research, we use the monitor construct of the task pooling mechanism. A monitor encapsulates the data structures defined within the monitor and provides an exclusive access to the procedures defined in it. Two types of monitors are used in this implementation: the ASKFOR monitor and the BARRIER monitor. In order to understand the parallel-proof procedure introduced in the next sub-section, these two monitors, along with their syntax, are shortly described below.

ASKFOR monitor: The task pooling is maintained by an 'ASKFOR' monitor which stands for a monitor to which processes 'ask for' an available task from the pool. To ask for a task from the pool, the operations inside the ASKFOR guarantee exclusive access to the pool of tasks. A call to the ASKFOR monitor, appears as

ASKFOR(task_identifier);

in the pseudo codes, will return the next available task to the caller. It also maintains 'delay' and 'continue' operation for processes to wait inside the monitor.

BARRIER monitor: The synchronization between processes can be achieved by another monitor of the package called the BARRIER monitor. A process that reaches a BARRIER statement, appeared as

BARRIER;

in the pseudo codes, will stay in the associated queue until all the other processes reach a BARRIER. At this time they are released all together and the concurrent operations proceed. This *barrier synchronization* allows the alignment of processes at a specific point during the concurrent execution.

The details concerning the implementation and the utilization of these monitors are going to be discussed elsewhere.

## 2.3 Parallel Proof Procedure

The overall control of the automated reasoning system using connection graphs can be described by a procedure called "parallel-proof." The pseudo codes of this procedure are written in Figure 3. This procedure is general to all the parallelisms, which means it can be applied to a particular parallelism with only minor modifications to its details while keeping the high level structure the same.

The parallel proof procedure is called by a master process which controls the proof procedures and monitors the termination condition. The parallel-proof procedure creates $N$ slave processes (line 3) where $N$ is usually one less than the number of processors in the system. The proof procedure terminates (line 4) successfully when the empty

```
1.  procedure parallel-proof;
    /* This procedure is called by the master process */
2.  {  Initialize the termination conditions;
3.     Create N Slave processes; /* they initially wait on a BARRIER */
4.     while (terminating condition is not true) {
5.         Create parallel tasks;
6.         Deposit them into the task pool;
7.         BARRIER; /* Release the Slave processes*/
8.         Work; /* Join the slaves to work */
9.         Apply simplification such as removal of subsumed clauses;
10.    }
11.    Terminate all the Slave processes;
12. }

13. process Slave;
15.    while (terminating condition is not true) {
17.        BARRIER;
16.        Work;
18.    }
19. }

20. procedure Work;
21. {  while (the task pool is not empty) {
22.        ASKFOR(task); /* Get a task from the task pool */
23.        if (task ≠ null) resolve-update(task);
24.    }
25. }
```

**Figure 3:** The Parallel-Proof Procedure of DC parallelism.

clause is derived, or it terminates unsuccessfully when the necessary conditions for refutation failure are satisfied. The necessary conditions for refutation are: there must be at least one positive and one negative clause in the graph, and the graph cannot collapse.

Once the parallel links are obtained (line 5), they are deposited into a task pool that is used for processes scheduling. Notice that different selection of parallel links will result in different parallelism.

When all the processes reach the BARRIER, they will be released together. These released slave processes will call a procedure 'Work' (line 16) where it repeatedly grabs a task (a link) from the task pool by calling the ASKFOR monitor (line 21) and, if the ASKFOR returns a valid task, resolves the link by calling a process called "resolve-update" (line 22). This resolve-update procedure, appeared in [Log87], stands for the codes that is used to solve the tasks at hand, and it may match to different content for different parallelism.

While the slave processes are resolving the links, the main process also gets a link from the task pool and resolves it by calling the procedure 'Work' (line 8), instead of waiting. Before all the processes complete their works, if a process cannot acquire a link from the task pool by calling ASKFOR, it will be placed on the queue associated with that ASKFOR monitor. The ASKFOR will release all of the processes together when all the processes are waiting in the queue, which will happen when all the links in the task pool have been resolved. This synchronization mechanism provided by the ASKFOR monitor will prevent the master process from starting the simplification (line 9) prematurely before all the slaves finish their work. That is to say, the macro package is capable of blocking all the processes in the monitor before all the tasks in the task pool have been finished, such that no process can starts another round of works before the works in the present round have been completed. After all the processes finished their 'Work', the slaves will go back to the BARRIER (line 15) while the master proceeds through the simplifications (line 9) and goes back to create parallel links for the next round.

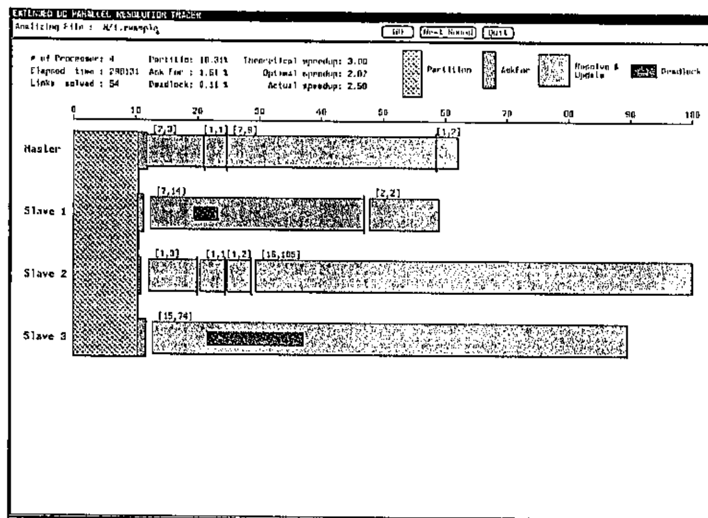

**Figure 4:** A particular example of the output of the trace mechanism.

# 3 The Visualizer

To trace the execution of the parallel processors, it is required that the timing information about the details of each processor's execution be generated. The instructions for acquiring the timing information are machine dependent. To get the timing on Encore Multimax, a

timer_init( )

instruction must be set initially to enable a 32-bit, free running, microsecond timer. It must be called before the initialization of all the concurrent processors in order to allow each of them to access the timer successfully. Subsequently, in the parallel program, the current value of the microsecond timer can be returned by a

timer_get( )

instruction. To acquire the exact execution time of a block of instructions, the difference of the time between the start and the end of the block, let it be 'diff', is calculated as following.

```
s = timer_get( );
...
(block of instructions)
...
e = timer_get( );
diff = e - s
```

The granularity of the block of instructions is assumed to be large enough such that the time of executing the timer_get( ) instruction will not cause any notable interference with the parallel programs' execution. Fortunately, timer_get( ) is an efficient machine instruction and its execution time is negligible.

After the timing informations are available, they must be recorded in a data structure and then saved in a file. The timing information cannot be printed out directly to the output when the parallel programs are executing because the I/O will interferes seriously with the concurrent executions of the parallel programs. In order to record all the timing information, a specially designed data structure is maintained in the shared memory. This cause a large amount of valuable shared memory space to be occupied by the data structure for maintaining the timing information. Unfortunately, this is inevitable.

All the data maintained in the timing structure is dumped to a file after all the parallel executions are terminated. Then, these data are shipped to the SUN workstation for a graphical trace. This visualizer is written in C in the SunView windowing environment. Figure 4 shows an example output created by the tracer in which different tasks of each processor at different time are presented in bar chart form. The examples from running extended DC parallelism is going to be used for discussion after this point. The trace output in extended DC parallelism includes following sub-tasks: partitioning, askfor, resolve and update the resolvent, and waiting to resolve dead-lock. Each sub-task is shown in a different shaded pattern. Each round is separately scaled such that it can fit in the whole screen while maintaining the relative length of each sub-task.

In the trace, the times taken by the partitioning algorithm are represented by an area starting at the beginning of the time axis. This is indicating the fact that the overhead is common to all the processors because, when the master processor is partitioning the connection graph, all the slaves are waiting on the barrier and doing nothing.

Since the macro package is very efficient, in the sense that the ASKFOR operations take a small amount of time compared with the overall execution time, the time taken by ASKFOR is relatively short or negligible. Usually, the first invocation of the ASKFOR monitor requires some set up time and appeared longer than those happened at the later stage of the trace. Those ASKFORs after the first one usually take a negligible amount of time and hence they appear as vertical lines.

Those overhead (partitioning, askfor, and deadlocks) are subtracted form the duration of the link resolutions to obtain the time required to perform the useful resolution operations. The areas representing the time spent on solving deadlocks are drawn inside the areas representing the time taken for resolution and update as in indication that the resolve-deadlock procedure is called within the resolution procedure. This tracer can help us to visually observe and realize the behavior of the extended DC parallelism.

To get an insight of the parallel-proof procedure we have defined the following speedups: Theoretical Speedup (TSP), Optimal Speedup (OSP), and Actual Speedup (ASP). TSP is calculated ignoring all the overheads involved in each iteration of the extended DC parallelism. OSP is calculated by taking all the overheads except

the time taken for creating the partitions. The ASP is the real speedup after considering all the overheads including the partitioning time.

The tracer can be used to graphically explain how the speedups are calculated. During an iteration of an extended DC parallelism, suppose processor $i$ is assigned a total of $r_i$ partitions and $t_{i1},\ldots,t_{i2},\ldots,t_{ir}$ are the useful times taken for resolving all the links in each of these partitions, then the *actual speedup* (ASP) is

$$\Sigma_{n=1}^{p}(t_{n1}+t_{n2}+\ldots+t_{nr_n})/e$$

where $p$ is the number of processes used and $e$ is the elapsed time of overall resolution. Note here that each $t_{ij}$ does not include any overhead. It means that the ASP is the real speedup after taking all the overhead into consideration, including the time for partitioning.

If the time taken for partitioning is zero, although it is impossible, the resulted speedup will be an optimal, respecting to the partitioning time. We call the speedup obtained based on this assumption as *optimal speedup* (OSP). Notice that the word *optimal* is used with respect to the effectiveness of the partitioning algorithm. Suppose the partitioning time is $g$, then the OSP is given by the expression

$$\Sigma_{n=1}^{m}(t_{n1}+t_{n2}+\ldots+t_{nr_n})/(e-g)$$

It is actually representing the speedup with the assumption that partitioning operations take no time at all.

The theoretical speedup, TSP, is calculated without considering any kind of overhead. Although this is impossible in the real world, it helps to understand the performance of the proposed algorithm. It is calculated as the ratio of the summation of all the useful work and the time taken by the longest stream of the useful work. Note here that no more than one processor is allowed to work on a single partition. The time taken, $m$, for the longest stream of the useful work is given by:

$$m = max\{(t_{n1}+t_{n2}+\ldots+t_{nr_n}) \quad for\ n = 1\ldots p\}$$

Thus the theoretical speedup is given by

$$\Sigma_{n=1}^{p}(t_{n1}+t_{n2}+\ldots+t_{nr_n})/m$$

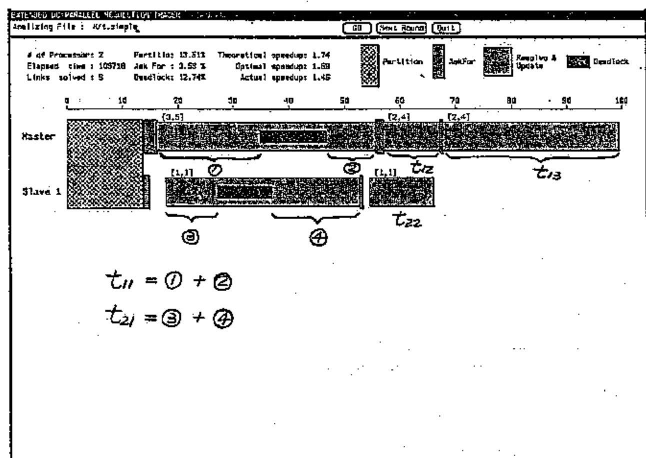


**Figure 5**: A simple case for explaining the calculation of the speedup.

To explain the calculations using the visualizer, consider the execution trace shown in Figure 5. There are two processes, the Master, which is assigned three partitions, and the Slave 1 which is assigned two partitions. Therefore, $p = 2$ with $r_1 = 3$, and $r_2 = 2$. The times used for resolving these partitions by the process 1 and 2 are respectively $t_{11}, t_{12}, t_{13}$ and $t_{21}$, $t_{22}$ as indicated in the Figure 5. The elapsed time $e$ is scaled from point 0 to 100 and the maximum summation of the useful work $m$ is

$$max\{(t_{11}+t_{12}+t_{13}),(t_{21}+t_{22})\} \ = t_{11}+t_{12}+t_{13}$$

The TSP is therefore

$$\Sigma_{n=1}^{2}(t_{n1}+t_{n2}+\ldots+t_{nr_n})/m$$

$$= ((t_{11}+t_{12}+t_{13})+(t_{21}+t_{22}))/m$$

$$= 1.74$$

If $t_{11} + t_{12} + t_{13} = t_{21} + t_{22}$, the theoretical speedup in this example will be equal to 2, but it is very unlikely to happen in practice. After including the system overhead, askfor overhead, and deadlock overhead, without considering the partitioning overhead, the OSP is calculated as

$$\Sigma_{n=1}^{2}(t_{n1}+t_{n2}+\ldots+t_{nr_n})/e-g$$

$$= ((t_{11}+t_{12}+t_{13})+(t_{21}+t_{22}))/e-g$$

$$= 1.69$$

Finally, the ASP is simply

$$\Sigma_{n=1}^{2}(t_{n1}+t_{n2}+\ldots+t_{nr_n})/e$$

$$= ((t_{11}+t_{12}+t_{13})+(t_{21}+t_{22}))/e$$

$$= 1.46$$

# 4 Improving the Parallelism Using the Visualizer

This section shows how the visualizer can help to realize as well as improve the parallelism by using examples in extended DC parallelism.

When the number of processors in use is increased, the speedup usually is expected to increase. However, when the number of tasks in not large enough, this might not be the case. Using the tracer, one of the exception can be depicted in Figure 6 and 7. When four processors are used instead of three, the works are more sparsely distributed therefore lowered the speedup. When the number of processors keep growing (Figure 8), there are simply not enough work available and the processors without being assigned any task are sitting idle all the way. Figure 9 and 10 are showing the situation for another example when number of processors is increased from five to six. It is simply saying that one should use more processors only when there are enough works to do.

How, exactly, the tracer can improve the parallelism? The following example shows how the decision of a series of modifications upon the extended DC parallelism can be made by examining the output created by the tracer. The original parallel proof procedure solves the links of a partition without resolving any new links of the partition. This will result in a small granularity and subsequently a relatively large partitioning time. Figure 11 shows a trace of the program execution where the percentage of partitioning time (8.49%) is relatively large. This is because the granularity of the useful works are not large enough and the partitioning time is relatively large and becomes a noticeable factor. To improve the granularity, we have to put more work into the partition, which can be done by solving not only the original links of a partition but also the new links inserted into the partition. This will cause the partition

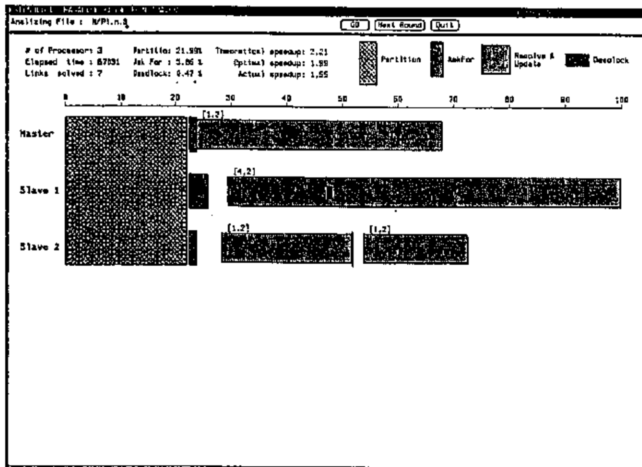


Figure 6: An example with three processes in use.

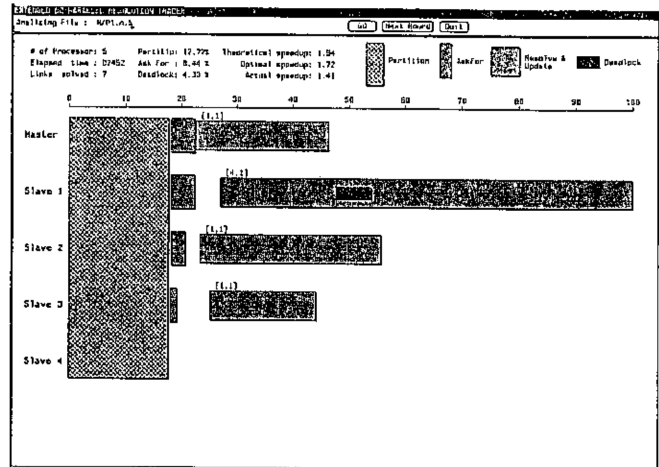


Figure 8: Increasing the number of processor further wil not increase the speedup because they are not doing any work.

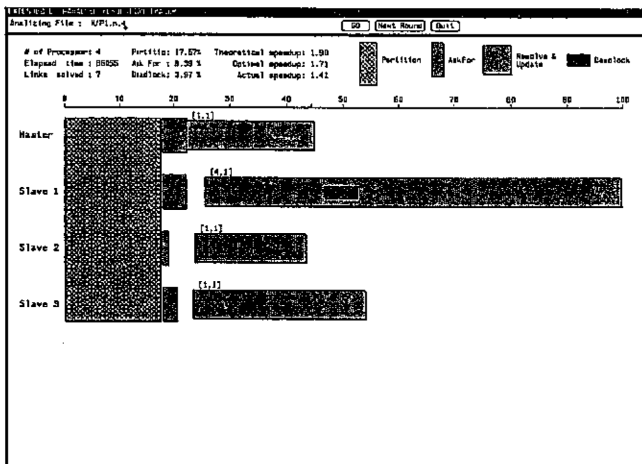


Figure 7: Increasing the number of processor from three to four is actually decreasing the speedup.

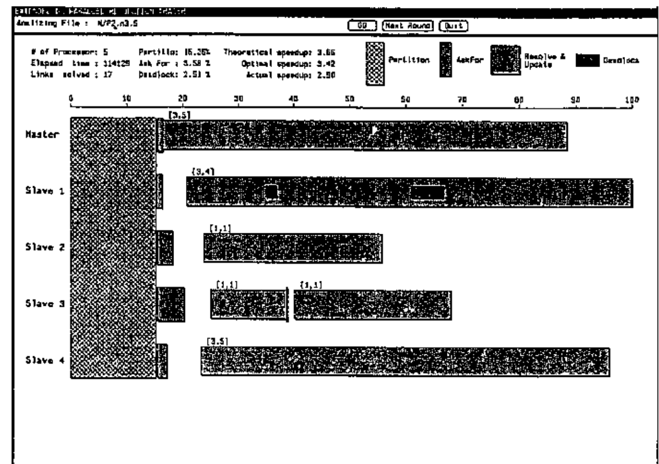


Figure 9: An execution with five processors available.

to expand along with the solving of the links in the partition. Therefore, in modifying the proof procedure, we allow each partition to expand without any restriction and to continue until all the links are solved within a partition.

Under the policy of allowing the expansion of a partition, the elapsed time of resolving each partition is extended a great deal therefore the overhead are become relatively insignificant. When we allow a partition to expand, the partitioning overhead becomes small compared to the useful work. In Figure 12, the Master and the Slave 1 are working on partitions that expanded to large sizes and the overhead percentages are lowered, for example, to 0.05% for 'ask for' and 1.2% for partitioning. Compare Figure 12 with Figure 11 which is executed under the original parallel proof procedure. The percentage time of partitioning operation is relatively lowered significantly.

Now that the problem of significant partitioning time is solved. However, another problem is revealed when we examine the trace show in Figure 13. This trace depicts the situation when the uneven sizes of partitioning happens. An evened amount of works should be distributed among parallel processors such that a best speedup can be reached. Unfortunately, not every partition has the opportunity to grow at the same rate and hence some processors may complete their work earlier than the others. In Figure 13, the Master, the Slave 1, and the Slave 2 are exhausted their tasks and the idling is affecting the speedup considerably. This problem of uneven distribution can be solved by setting up a threshold value of maximum idling processors. When the threshold value is reached, it means a pre-decided number of processors are idling, all the parallel processors at this point will relinquish its partition and finish. In this situation the resolution of over-expanded partitions can be stopped immediately and the possible uneven distribution caused by partition expansion will not be able to happen.

Compare Figure 13 and Figure 14 where they are working on the same connection graph with the same number of processors, but Figure 13 represents the trace without setting up the threshold value while 14 represents the trace with the threshold value. Obviously, the achieved improvement is a result of the fact that the threshold value becomes effective when there are too many idling processors. The break-even point between partition expansion and uneven distribution is the key to a better resource utilization.

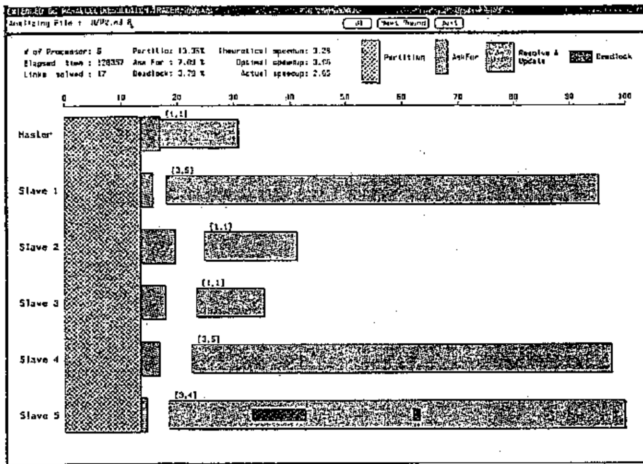

**Figure 10:** Increase the number of processors from five to six will not help.

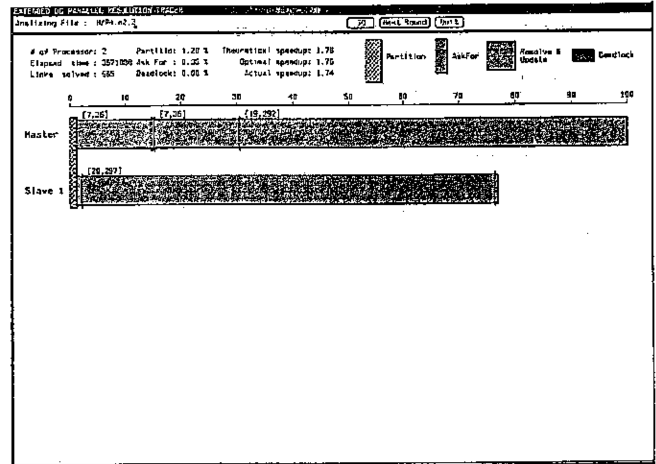

**Figure 12:** The partitioning overhead is relatively lowered when the partitions are allowed to expand.

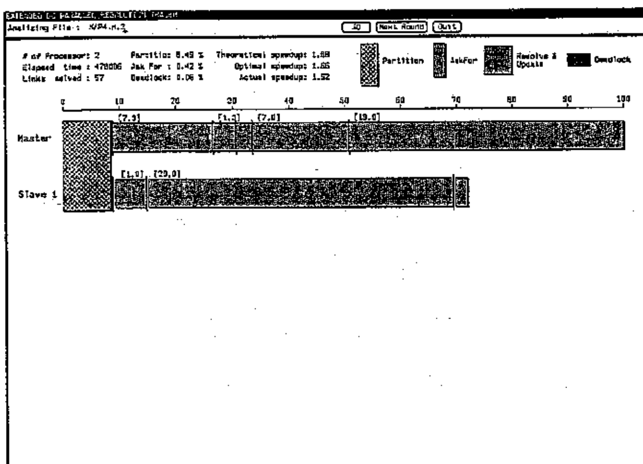

**Figure 11:** The partitioning time is relatively large.

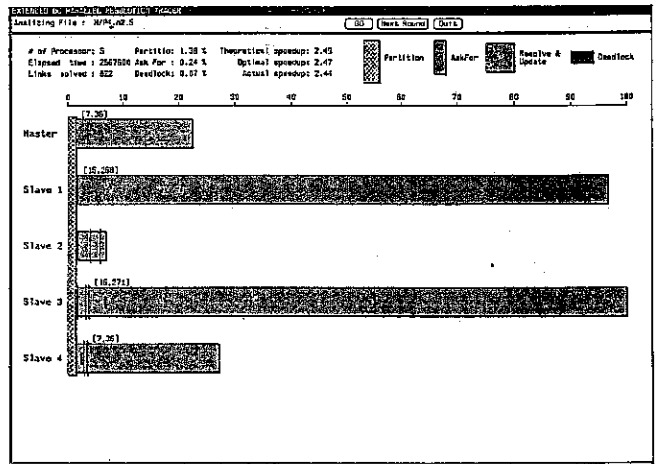

**Figure 13:** Uneven sizes of partitioning may cause processors to be idled.

# 5 Summary

In general, to completely comprehend the true behavior of the parallel processors in a parallel environment is not usually achievable. Simultaneous parallel operations make the task of analyzing the parallelism very difficult. This paper discussed a tracing tool (a visualizer) which is successfully developed as a part of a parallel automated reasoning program. This visualizer can be modified to trace the output of any other parallel systems as long as it can generate the timing information following the proper format. This visualizer, as demonstrated, is capable of helping the researchers to realize what have really happened among those parallel processors, and, most importantly, assisting the designers to improve their algorithms.

# References


[BBD87] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt Rinehart and Winston, Inc., New York, 1987.

[Efe82] K. Efe. Heuristic models of task assignment scheduling in distributed systems. IEEE Computer, 15(6):55-66, June 1982.

[GN72] R. Garfinkel and G. L. Nemhauser. *Integer Programming*. Wiley New York, 1972.

[HKK81] G. Hornung, A. Knapp, and U. Knapp. A parallel connection graph proof procedure. In J. H. Siekmann, editor, *German Workshop on Artificial Intelligence*. Springer-Verlag, Berlin, 1981.

[Kow75] R. Kowalski. a proof procedure using connection graphs. *Journal of the ACM*, 22(4):572-595, October 1975.

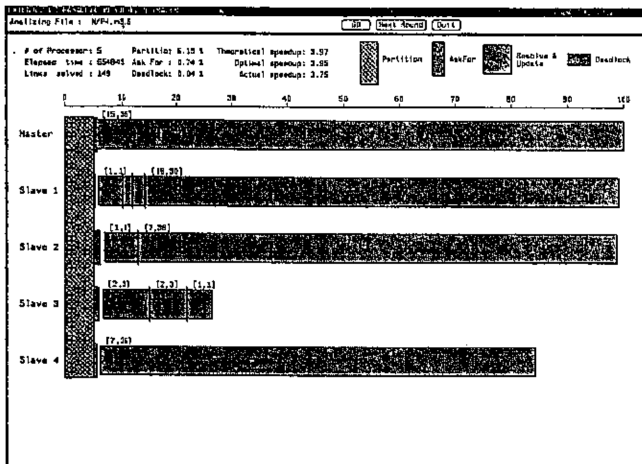

**Figure 14**: Processor un-idling can improve the speedup a great deal.

[LL90] R. Loganantharaj and A. A. Liou. Experience with extended dcparallelism of connection graphs. In *Proceedings of the ISMM Parallel and Distributed Computing, and System*, New York, October 1990.

[LL91] R. Loganantharaj and A. A. Liou, Parallel Deduction of Connection Graphs, in R. W. Wilkerson, editor, *Advances in Logic Programming and Automated Reasoning*. Ablex Publishing Corporation, New Jersey, 1991.

[Log85] R. Loganantharaj. *Theoretical and implementational aspects of parallel link resolution in connection graphs*. PhD thesis, Department of Computer Science, Colorado State University, 1985.

[Log87] R. Loganantharaj. Dc parallel link resolution of connection graph. In *Fall Joint Computer Conference*, October 1987.

[SP82] Seki-Projekt. The markgraf carl refutation procedure: the logic engine. Technical Report Nr. 24/82, Institut fuer Informatik I, Universitat Karlsruhe, West Germany, 1982.

[SP84] Seki-Projekt. The markgraf carl refutation procedure. Technical Report Memo-SEKI-MK-84-01, Institut fuer Informatik I, Universitat Karlsruhe, West Germany, 1984.