Parallel OPS5 User's Manual

Dirk Kalp, Milind Tambe, Anoop Gupta[1], Charles Forgy,
Allen Newell, Anurag Acharya, Brian Milnes, Kathy Swedlow

The Production System Machine Project
Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania
15213-3890

13 November 1988

Abstract

This document is a reference manual for *ParaOPS5*, a parallel version of the OPS5 production system language. ParaOPS5 is implemented in C, and runs on shared memory multiprocessors. While it is intended for use on shared memory multiprocessors, it is also suitable for uniprocessor machines. ParaOPS5 does not change the computational model or the functionality of OPS5. Instead, it exploits parallelism in the match phase of OPS5. Exploiting parallelism at a very fine granularity in the match phase has allowed ParaOPS5 to achieve significant speed-ups.

---

[1]Department of Computer Science, Stanford University, Stanford, CA 94305

1. Introduction

The goal of the production system machine (PSM) project at Carnegie Mellon University is to create software and hardware technology for parallel rule-based systems. This document is a reference manual of our parallel implementation of the OPS5 production system language [1, 2], called *ParaOPS5*. ParaOPS5 is based on the parallel implementation proposed in [5]. ParaOPS5 does not change the computational model or the functionality of OPS5. Therefore, OPS5 programs can be run on this system without any changes.

ParaOPS5 exploits parallelism in the match phase of OPS5. Exploiting parallelism at a very fine granularity in the match phase has allowed ParaOPS5 to achieve significant speed-ups. An optimized C-based implementation provides ParaOPS5 with additional speed-ups over the serial Lisp-based implementation of OPS5. Thus, ParaOPS5 has shown up to 200 fold speedup in certain OPS5 systems: about 10-12 fold from parallelism and 15-20 fold from the optimized C-based implementation.

ParaOPS5 is intended for use on shared memory multiprocessors. However, it is also suitable for use on uniprocessor machines as well since the implementation technology used provides it with considerable speed-ups over the serial Lisp-based versions of OPS5. We are also investigating the implementation of ParaOPS5 on message passing computers (MPCs). Our analysis [6] indicates that MPCs are quite suitable for production systems.

We expect you, the readers of this document, to be familiar with the OPS5. In fact, for reading this document, the *OPS5 User's Manual* [2] is an essential reference. Familiarity with the implementation proposed in [5] may be helpful, but is not necessary. For the interested reader, the ParaOPS5 implementation is described in detail in some other papers related to the PSM project [7, 8].

Section 2 provides some background information. Section 3 presents information about installation and system requirements for ParaOPS5. Section 4 describes how to use the system and presents an overview of ParaOPS5, its command interface and run-time options. Section 5 describes some limitations of ParaOPS5, and its differences with OPS5. Section 6 gives some recommendations on the style and design of ParaOPS5 programs.

2. Background

This section is intended to provide a very brief review of OPS5, the Rete matching algorithm, and our ParaOPS5 implementation. In addition, this section introduces terminology which will be used in the rest of this document.

2.1. OPS5

An OPS5 [1] production system is composed of a set of *if-then* rules, called productions, that make up the *production memory*, and a database of temporary assertions, called the *working memory*. The individual assertions are called working memory elements (WMEs), and are lists of attribute-value pairs. Each production consists of a conjunction of condition elements (CEs) corresponding to the *if* part of the rule (also called the left-hand side or LHS), and a set of actions corresponding to the *then* part of the rule (also called the right-hand side or RHS).

The CEs in a production consist of attribute-value tests, where some attributes may contain variables as values. The attribute-value tests of a CE must all be matched by a WME for the CE to match; the variables in the condition element may match any value, but if the variable occurs in more than one CE of a production, then all occurrences of the variable must match identical values. When all the CEs of a production are matched, the production is satisfied, and an instantiation of the production (a list of WMEs that matched it), is created and entered into the *conflict set*. The production system uses a selection procedure called *conflict-resolution* to choose a production from the conflict set, which is then *fired*. When a production fires, the RHS actions associated with that production are executed. The RHS actions can add, remove or modify WMEs, or perform I/O.

The production system is executed by an interpreter that repeatedly cycles through three steps:

1. Match
2. Conflict-resolution
3. Act

The matching procedure determines the set of satisfied productions, the conflict-resolution procedure selects the highest priority instantiation, and the act procedure executes its RHS. These three steps are collectively called the *recognize-act cycle*.

2.2. Rete

Rete [3] is a highly efficient match algorithm that is also suitable for parallel implementations [5]. Rete gains its efficiency from two optimizations. First, it exploits the fact that only a small fraction of working memory changes each cycle by storing results of match from previous cycles and using them in subsequent cycles. Second, it exploits the commonality between CEs of productions, to reduce the number of tests performed.

Rete uses a special kind of a data-flow network compiled from the LHSs of productions to perform match. The network is generated at compile time, before the production system is actually run. The entities that flow in this network are called *tokens*, which consist of a *tag*, a *list of WME time-tags*, and a *list of variable bindings*. The tag is either a + or a – indicating the addition or deletion of a WME. The list of WME time-tags identifies the data elements matching a subsequence of CEs in the production. The list of variable bindings associated with a token corresponds to the bindings created for variables in those CEs that the system is trying to match or has already matched.

There are primarily three types of *nodes* in the network which use the tokens described above to perform match:

1. *Constant-test nodes*: These are used to test the constant-value attributes of the CEs and always appear in the top part of the network. They take less than 10% of the time spent in Match.

2. *Memory nodes*: These store the results of the match phase from previous cycles as state. This state consists of a *list* of the tokens that match a part of the LHS of the associated production. This way only changes made to the working memory by the most recent production firing have to be processed every cycle.

3. *Two-input nodes*: These test for joint satisfaction of CEs in the LHS of a production. Both inputs of a two-input node come from memory nodes. When a token arrives from the *left memory*, i.e., on the left input of a two-input node, it is compared to each token stored in the *right memory*. All token pairs that have consistent variable bindings are sent to the successors of the two-input node. Similar action occurs when a token arrives from the right memory. We refer to such an action as a *node-activation*.

Figure 2-1 shows the Rete net for a production named P1.

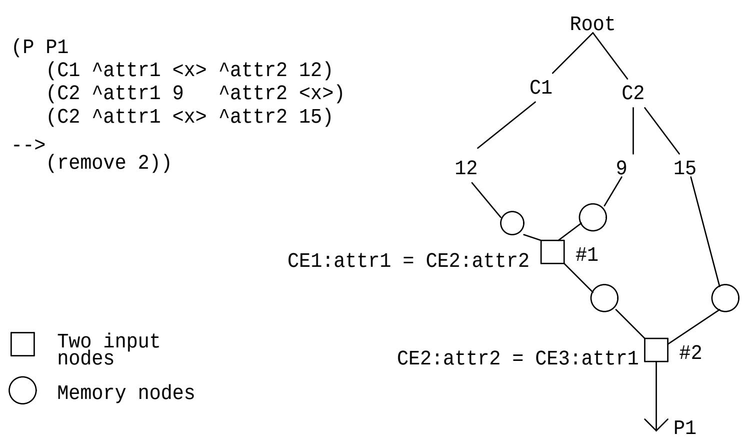

Figure 2-1: The Rete network.

2.3. ParaOPS5: Parallel Implementation of OPS5

ParaOPS5 [8] is a highly optimized C-based parallel implementation of the OPS5 production system. It produces a machine coded version of the Rete data-flow network. Before starting a run, the ParaOPS5 compiler generates a tree structured representation of the Rete network for the current set of productions. This data structure is used to generate OPS83 [4] style assembly code for the network. The system then uses the assembling, linking and loading facilities provided by the operating system to create the executable image. This process is explained in detail in later sections.

ParaOPS5's run-time environment consists of one *control process* that selects and then fires an instantiation and one or more *match processes* that actually perform the Rete match. ParaOPS5 exploits parallelism at the granularity of *node activations*. Previous work has demonstrated that to achieve significant speed-ups via parallelism in production systems, it is necessary to exploit parallelism at a very fine granularity [5]. A node activation consists of the address of the code for a node in the Rete network and an input token for that node. These node activations are called *tasks* and are held in one or more shared *task queues*. Each individual match process performs match by picking up a task from one of these queues, processing the task and, if any new tasks are generated, pushing them onto one of the queues. When the task queues become empty, one production system cycle ends; the control process applies conflict resolution to select and fire an instantiation from the CS. Figure 2-2 shows the speed-ups achieved with our current implementation for three different systems: Rubik, Weaver and Tourney. The speed-ups are for an implementation on the Encore Multimax and are reproduced from [8]. Though Rubik and Weaver are seen to achieve good speed-ups, the speed-ups in Tourney are quite low. The speed-ups are a function of the characteristics of the productions in the production system. We will comment on this in detail in Section 6.

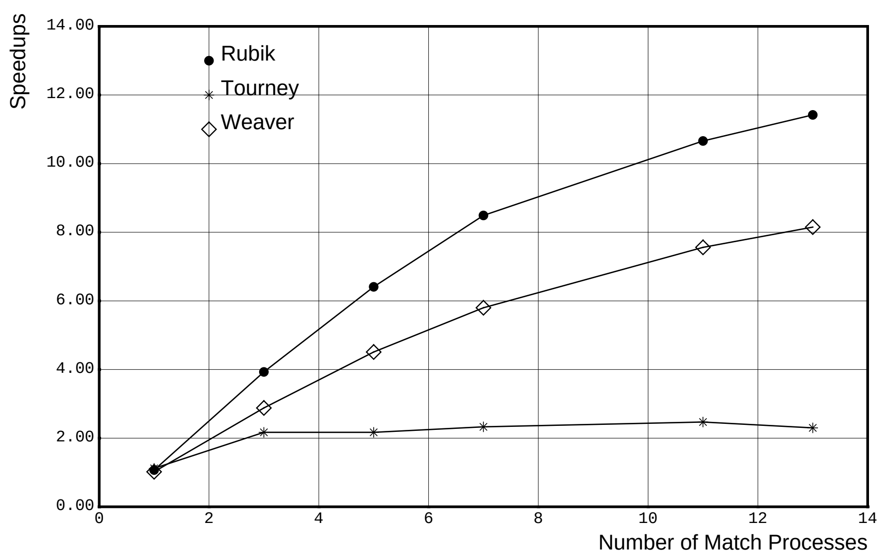

Figure 2-2: Speedups for OPS5 on the Encore Multimax [8].

Since ParaOPS5 exploits parallelism in the match at the fine-grain level of node activations in the Rete network, there is neither a requirement nor an opportunity for you to manage the parallelism in your application program. Techniques such as partitioning productions among processors are irrelevant. The system extracts and exploits the parallelism automatically in the queuing and scheduling of the tasks corresponding to the node activations. You can, however, affect the performance of your program by designing your program and writing your productions in a style that maximizes the parallelism available for the system to exploit. Programming style and design issues are discussed in Section 6.

3. System Distribution, Installation, and Maintenance

This section explains how to obtain and install the ParaOPS5 system. Also discussed here are the system requirements for running ParaOPS5.

3.1. System Distribution, Documentation, and Maintenance

The ParaOPS5 system can be obtained by contacting us directly:

The PSM Project **psm-requests@centro.soar.cs.cmu.edu**
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

The system is distributed on a 1600 or 6250 bpi mag tape written using the Unix[2] **tar** utility. Please specify the tape density required with your request.

Besides this document, the *OPS5 User's Manual* [2] is an essential reference. It describes the OPS5 language and the basic production system architecture and should be used in conjunction with this document. Additional references and papers related to the PSM project are listed at the end of this document.

ParaOPS5 has been developed and used exclusively as a research tool for the PSM project. Our use of the system has been focused mainly on performance and measurement of a select set of benchmark programs rather than active development of application programs. While we have attempted to make ParaOPS5 robust and provide most of the functionality of the Lisp-based versions of OPS5, we expect that bugs will be uncovered and features missing may prove essential to others. To report bugs and other deficiencies, we ask that you write to the PSM project or send electronic mail addressed to **psm-bugs@centro.soar.cs.cmu.edu**. Suggestions for new features or improvements are welcome too. We would also appreciate receiving any improvements you make to the system as well as ports of ParaOPS5 to other machines. We will attempt to consolidate these and make them available to others when subsequent releases of the system are built.

3.2. System Requirements

Currently, ParaOPS5 can be configured to run on Encore Multimax and Vax computers. The Encore Multimax is a shared memory multiprocessor that provides up to 20 processors and is based on the National Semiconductor NS32332 processor. Since the ParaOPS5 system can be run in uniprocess mode, it also runs on Vax workstations and other uniprocessor Vax machines

---

[2]Unix is a registered trademark of AT&T.

such as the 11/780 in addition to Vax multiprocessors such as the 8800 and 11/784.

ParaOPS5 runs on Carnegie Mellon's Mach operating system which supports BSD 4.3 Unix. The system is not heavily dependent upon Unix and Mach system calls. It uses Mach only to provide for shared memory when configured in the multiprocess mode. It uses Unix primarily to just fork processes in the multiprocess configuration.

In principle then, the system can be easily ported to run in uniprocess mode on any 32000 or Vax based machine which supports C. In order to run in multiprocess mode, the operating system must provide a mechanism to support the allocation of shared memory.

3.3. System Installation

The instructions for system installation assume that you have 4.3 BSD Unix running on top of Mach. However, for systems running a version of Unix without Mach, advice is also given for installing the system in a uniprocess configuration. For Mach-based Unix systems, your **.login** file should include environment directives to provide search paths for Mach libraries and include files. The following should be specified:

```
source /usr/mach/lib/machpaths
```

To install the ParaOPS5 system, create a working directory on your machine into which the ParaOPS5 files will be copied and move to that directory. Mount the tape and remove the file **getcmds**. This is a command file which you then execute in order to unload the tape. It will set up three subdirectories: **lhs**, **rhs**, and **programs**. It will also deposit two other command files, **setupENCORE** and **setupVAX**, in your working directory. Select one of those two command files that corresponds to your machine and execute it in order to configure ParaOPS5. A typical sequence of Unix commands and actions to install ParaOPS5 on an Encore machine follows:

```
% mkdir paraops5
% cd paraops5
% <<issue command to mount the tape>>
% tar xv getcmds
% getcmds
% <<issue command to dismount the tape>>
% setupENCORE
```

After this is executed you will have the three subdirectories in your working directory. The **lhs** subdirectory contains the ParaOPS5 compiler, named **cops5**, along with the files used to construct the compiler. The **rhs** subdirectory contains the files that comprise the run-time system for ParaOPS5 and which are ultimately linked together with your application program. Both of these subdirectories contain Unix **make** files which specify the dependencies among the files within each subdirectory. The third subdirectory is **programs** which provides you with some examples of OPS5 application programs you can compile and run.

In Section 4, we will explain the steps involved in getting your OPS5 application programs to run and use one of the sample programs to illustrate. Before going on to that, you may wish to delete some of the files in the **lhs** and **rhs** subdirectories in case you are short on disk space. In the **lhs** subdirectory, the only essential file is **cops5**, the executable file for the ParaOPS5 compiler. In the **rhs** subdirectory, the only files required are the **.o** binary files that are necessary to link together with your application program. All other files in those two subdirectories can now be removed.

If your machine is running a version of Unix without the support of Mach to provide shared memory among processes, then you can probably configure ParaOPS5 to run in uniprocess mode without too much difficulty. Go to the **rhs** subdirectory and edit the file **version.h** to uncover the definition for the non-shared memory version and to remove the definition for the Mach shared memory version. The result of the editing should be:

```
#define  NON_SHARED_MEMORY_VERSION  1
/* #define  MACH_SHARED_MEMORY_VERSION  1 */
```

Then invoke the Unix **make** utility to recompile the files in **rhs**.

4. Using the System

This section describes how to prepare and run programs under ParaOPS5. Run-time options and the user interface are also discussed.

4.1. Overview of the System

From an implementor's point of view, the ParaOPS5 system consists of four components. The first component is the code for the Rete network which performs the match. The second component consists of the routines that manage the queuing and dispatching of the node activation tasks in the Rete network. Both of these are written in assembly code, produced by the ParaOPS5 compiler when it compiles the LHSs of your productions. Because we exploit parallelism at the fine grain of node activations in the Rete network, it is essential that the network code and the task management routines be fast and have low overheads.

The third component is the threaded code representation produced by the ParaOPS5 compiler for the RHSs of the productions. At run-time, when a production is fired, the threaded code segment corresponding to the production is interpreted in order to execute the RHS actions of the production. Since the cost of production firings is small compared to the match, interpreted execution of the RHS is adequate.

The fourth component is the run-time system for ParaOPS5. Written in C, it includes the routines that are invoked by the threaded code to implement the RHS actions. The run-time system also contains the routines that start up your program, create and activate the match processes, control the basic OPS5 recognize-act cycle, and provide the user interface. Also included in this component are the OPS5 data structures such as working memory, the conflict set, and the Rete network node memories along with the routines to manage these resources.

The ParaOPS5 compiler, **cops5**, in the **lhs** subdirectory, takes an OPS5 program as input and produces an assembly language file (a **.s** file) that contains the first three components: the Rete network code, the task management code, and a data segment consisting of the threaded code for the RHSs of your productions. The assembly language file also includes symbol table information from the program's declarations. This file is then assembled and linked together with the fourth component, the run-time system, to produce a working ParaOPS5 program. The routines and data for the run-time system are contained in the **rhs** subdirectory.

As mentioned in Section 2.3, there is no special preparation required for your OPS5 program in order to manage the parallelism. The parallelism is extracted and exploited automatically by the ParaOPS5 system.

4.2. Running a Sample Program

The **programs** subdirectory, created by the installation procedure, contains some OPS5 application programs and some command files that can be used to create a working program. To see how that is done, go to the **programs** subdirectory and execute the command file, **cmd**, with the argument, **mab**, as follows:

```
% cmd mab
```

The argument, **mab**, refers to the OPS5 program, **monkeys and bananas**, which is contained in the file **mab.l**. The command file performs three actions:

1. ```
   ../lhs/cops5 mab.l
   ```
   compiles **mab.l** with the ParaOPS5 compiler into an assembly language file **mab.s**.

2. ```
   as -j -n mab.s -o mab.o
   ```
   invokes the Unix assembler (on the Encore) to produce the object file **mab.o**. (The **-j -n** switches are replaced by the **-J** switch on the Vax.)

3. ```
   cc -g mab.o ../rhs/extern.o ../rhs/conres.o
               ../rhs/gensymbol.o ../rhs/match.o
               ../rhs/matchasm.o ../rhs/rhsrtn.o
               ../rhs/utility.o ../rhs/wmemory.o
               ../rhs/shmem.o ../rhs/wminput.o
               ../rhs/y.tab.o ../rhs/lex.yy.o
               -ll -lmach -o mab
   ```
   invokes the Unix linker/loader to form the executable image **mab**. Linked together are **mab.o** along with the object files for the ParaOPS5 run-time system in **rhs** and library routines from the Unix and Mach libraries.

After these three steps, the run-file **mab** can be executed with

```
% mab
```
```

4.3. Requirements for Running Your Program

As shown in Section 4.2, there are three steps involved in getting an OPS5 program ready to run:

1. Compile it with the ParaOPS5 compiler.
2. Assemble it with the Unix assembler.
3. Link it together with the run-time system and libraries.

Your own OPS5 program must satisfy a few requirements before it can be compiled. All declarations and productions must be contained in a single file with all the declarations at the beginning of the file. You must have a **start** production, that is, a production with a condition element **(start)**. Usually the **start** production will have an RHS containing **make** actions that initialize working memory to begin your program execution. The **start** production in the example program is:

```
(p start_production
        (start)
    -->
        (make monkey ^at 5-7 ^on couch)
        (make object ^name couch ^at 5-7 ^weight heavy)
        (make object ^name bananas ^on ceiling ^at 2-2)
        (make object ^name ladder ^on floor ^at 9-5 ^weight light)
        (make goal   ^status active ^type holds ^object bananas)
)
```

When execution of your program begins, the ParaOPS5 run-time system will automatically make a WME of class **start** and add it to working memory. In the example above, this action would instantiate **start_production** making it eligible to fire and create the WMEs specified in its RHS. You may, however, want the **start** production to be just a dummy production (i.e., with an empty RHS) if you intend to initialize working memory by another means that will be described in Section 4.4 and Section 4.5.

The output of the compiler is a single assembly language file. That file must then be assembled with the Unix assembler. The switches, **-j -n** (**-J** on the Vax), given to the assembler in the example from Section 4.2 instruct the assembler to use long branches to resolve jumps when byte-displacement branches are insufficient. The **-o mab.o** switch instructs the assembler to put the object code into the file **mab.o**. You should consult the documentation for the Unix assembler on your system to determine the required switches.

Having the object file (the **.o** file) produced by the assembler, your last step is to link it together with the object files that constitute the run-time system and the required library routines. Referring again to the **mab** program example in Section 4.2, you see several switches specified to the Unix linker/loader. The **-o mab** switch instructs the linker to put the executable image it produces into the file **mab**. The **-g** switch includes some information for the debugger. The **-ll** and **-lmach** switches specify the Unix library **libl.a** and the Mach library **libmach.a** respectively. (For those porting to a non-Mach based Unix system, the **lmach** switch is omitted.)

4.4. Program Switches

Program switches set up or change the default run-time environment for your program. For example:

```
% mab -w1
```

instructs the system to set the OPS5 **watch** to level 1. This causes a trace of the productions fired to be printed to the terminal as the program runs. A number of other switches can be specified and are explained in the following paragraphs.

1. **-a**n n >= 0

   This switch causes the PSM control process to ask you after every n production firings whether you want to continue or quit your program. The default is n=0 which is used to mean just run the program and never ask.

2. **-c**

   This switch causes the PSM control process to enter an interactive mode that provides you with a command interface to more directly control the execution of your program. This command interface is especially useful in debugging your program. It is described in Section 4.5.

3. **-d**

   This switch turns on the printing of some information we used in debugging the system. It shows some of the background activity involved in allocating memory, forking processes, etc.

4. **-f**file file is a filename

   This switch instructs the system to get program switches from an input file instead of just the command line. It is useful if you want to use the same switch settings repeatedly. Go to the **programs** subdirectory and execute:

   ```
   % bam -fbam.switches
   ```

5. **-h** or **-?**

   This prints a help message to the terminal which gives a brief explanation of the available program switches.

6. **-i**file file is a filename

   This switch allows you to specify an input file to be used to load a set of initial WMEs to working memory. The WMEs are specified in the file in the same way as **make** actions in a production's RHS. For example:

   ```
   (make monkey ^at 5-7 ^on couch)
   ```

   The **makes**, however, may use only constant symbols and numbers, the operator ^, and literalized atoms. Variables and functions are not permitted. Loading initial working memory from a file is convenient when you want to run your program with several different data sets. If instead you use the **start** production to load your data set, then you must edit the **start** production each time you run a different data set. This in turn requires that you compile, assemble, and link your program each time, which may be time consuming for large programs. The **programs** subdirectory contains a modified version of the **monkeys and bananas** program called **bam.l** which illustrates this feature. Do the following to run it:

```
% cmd bam
% bam -ibam.input
```

7. **-m**n n > 0

This specifies the size in kilobytes of a contiguous block of shared memory to be allocated at the beginning of your program. Static data structures like the task queues, hash tables, and symbol table that must be shared among all processes are allocated from this block. Dynamic data objects such as WMEs, node memory tokens, etc. are allocated as needed from this block by ParaOPS5 during program execution. The default is n=8184, i.e., 8184 Kbytes. You can set it to be more or less, depending upon the needs of your program and the size of your memory. (For the NON_SHARED_MEMORY_VERSION, this block is allocated using **malloc**.)

8. **-p**n 1 <= n <= 32

This specifies the number of processes used. The default is n=1 which means that no match processes are forked when your program is run. In this single process mode, the PSM control process (see Section 2.3) performs the match in addition to selecting and firing productions in the recognize-act cycle. For n>1, n-1 PSM match processes are forked by the PSM control process. The match processes run continuously looking for Rete node activation tasks to perform. The PSM control process does not participate in the match for n>1.

9. **-q**n n = 1, 2, 4, 8, 16, 32

This specifies the number of task queues to use to hold Rete node activation tasks. The default is n=1. A single task queue can become a bottleneck when several match processes are contending for the lock on the queue in order to deposit or remove tasks to or from the queue. Multiple task queues reduces this contention. The effect of using multiple task queues is reported in [7, 8].

10. **-r**

This switch has an effect only when multiple task queues are used. It instructs the PSM control process to deposit root node activations of the Rete network in a round-robin fashion among the task queues. Root node activations correspond to WMEs added/deleted to/from working memory by the firing of a production's RHS actions. The default is to simply place all root node activations in the first queue. Using this switch does not seem to produce a significant performance improvement in program execution.

11. **-s**strategy strategy is mea or lex

This switch sets the OPS5 conflict resolution strategy. The default strategy is mea, unlike the Lisp-based OPS5.

12. **-t**

This switch instructs ParaOPS5 to touch all the memory pages in the block of shared memory allocated at program initialization. The effect of this on a machine with sufficient physical memory is to page in the entire shared memory block before program execution begins. This avoids the system overhead time for paging while the program runs. This is used primarily to factor out the system paging overhead when accurate timing measurements of program execution are desired.

13. **-v**

    This switch causes the ParaOPS5 version number to be printed to the terminal.

14. **-w**n 0 <= n <= 3

    This specifies the OPS5 **watch** level. The default is n=0 which prints no **watch** information. The other **watch** levels are described in the *OPS5 User's Manual* [2].

4.5. The Interactive Command Interface

The ParaOPS5 interactive command interface corresponds to the top level interface described in Section 8 of the *OPS5 User's Manual* [2]. This section describes the differences between the OPS5 top level interface and the ParaOPS5 command interface. As indicated in Section 4.4, the ParaOPS5 command interface is invoked by specifying the **-c** switch when your program is run. In the interactive mode, you can exert greater control over the execution of your program and use the commands available to monitor the progress of your program and examine its state at intermediate points. This is helpful in program debugging.

Except for **back**, **excise**, and **pm**, ParaOPS5 supports all the OPS5 top level commands. These commands are,

call	**make**	**remove**
closefile	**matches**	**run**
cs	**openfile**	**strategy**
default	**pbreak**	**watch**
exit	**ppwm**	**wm**

All the commands are implemented as described in the *OPS5 User's Manual* [2] except for the **strategy** command. The **strategy** command differs only in that the default conflict resolution strategy is **mea** instead of **lex**. Unlike the OPS5 top level interface, commands are given without enclosing parentheses and commands are terminated by the end of line character.

ParaOPS5 supports the following additional commands not provided by the standard OPS5 top level:

1. **ask** n

   This command performs the same function as the **-a** program switch described in Section 4.4. When invoked without an argument, it shows the current setting for **n**.

2. **help** or **?**

   This prints a help message showing the commands available and giving a brief description of each.

3. **loadwm** inputfile

   This command is similar to the functioning of the **-i** program switch described in Section 4.4. It allows you to load working memory from a file holding **make** actions. The **-i** switch provides the opportunity to load working memory only at the beginning of your program. The **loadwm** command gives you the opportunity to also load working memory at other points in your program's execution.

4. **resolve**

This command shows the next production that will be selected to fire by conflict resolution when your program is resumed by the **run** command.

5. **showtime**

This command prints the timing statistics accumulated for your program's execution. It shows the time spent at the beginning to initialize the ParaOPS5 system and data structures. It also gives a breakdown of the time spent in the recognize-act cycle on performing the match and executing the RHS actions. Conflict resolution time is included in the RHS time. The time is given in terms of Unix user time and system time.

6. **version**

This prints the ParaOPS5 version number.

7. **zerotime**

This command resets the timing statistics to zero. This allows you to record time for executing parts of your program. For example, you can time the execution of 50 cycles in the middle of your program with the command sequence:

```
enter cmd> zerotime
enter cmd> run 50
enter cmd> showtime
```

To try using the command interface, go to the **programs** subdirectory and run the **bam** program as follows:

```
% bam -c
  enter cmd> wm
  enter cmd> loadwm bam.input
  enter cmd> wm
  enter cmd> watch 1
  enter cmd> run 5
  enter cmd> cs
  enter cmd> run
  enter cmd> exit
```

5. System Features, Limitations and Differences with OPS5

This section points out the differences between ParaOPS5 and the Lisp-based OPS5 described in the *OPS5 User's Manual* [2]. Some of the differences related to the top level user interface have already been discussed in Section 4 and will not be repeated here. Also discussed here are some of the features, limitations, and weaknesses of the ParaOPS5 system.

5.1. Representation of OPS5 Atoms

The scalar values in OPS5 are either symbolic or numeric atoms. In ParaOPS5 these atoms are represented uniformly as 32-bit quantities defined as type OpsVal:

```
typedef long  OpsVal;
```

The kind of atom represented is indicated by the low order bit - it is 0 for numeric atoms and 1 for symbolic atoms. For numeric atoms, the remaining 31 bits represent an integer in twos complement form. This allows a range of integers from -2^{30} through $2^{30} - 1$ inclusive. For symbolic atoms, the top 31 bits contain a unique symbol ID assigned to represent the symbol. The run-time symbol table maintains a mapping of symbol IDs to the character strings they represent.

ParaOPS5 does not do case-folding. Thus the symbolic atom **p** is different from the atom **P**. Also, unlike standard OPS5, ParaOPS5 does not support floating point numbers. ParaOPS5 supports only integers which the *OPS5 User's Manual* [2] refers to as **fixed point numbers**. They consist of an optional sign followed by one or more decimal digits and an optional point. Some legal examples are **3**, **3.**, **-3.**, and **+3**.

5.2. Call Actions and User Defined Functions

Section 7 of the *OPS5 User's Manual* [2] describes the relationship between OPS5 and external routines that can be accessed through the OPS5 **call** action or the **user defined function call**. OPS5 provides a mechanism to pass parameters and results between it and external routines. Part of this mechanism is a set of interface functions that external routines can call to communicate with OPS5. In OPS5 these interface functions have names that begin with the character **$**. Since this character is not permitted in C identifiers, it is replaced in the function names by the word **dollar** followed by an underscore character. Thus, for example, the interface function, **$parameter**, is specified as **dollar_parameter** in ParaOPS5. The interface functions are,

dollar_parameter	**dollar_reset**
dollar_parametercount	**dollar_ifile**
dollar_assert	**dollar_ofile**
dollar_tab	**dollar_litbind**
dollar_value	

As the arguments and return values of the interface functions can be OPS5 atoms, OPS5 also provides an additional set of routines for external routines to use to process atoms. This allows an external routine, for example, to convert a numeric atom it receives as a parameter to a number so that it can use it in a computation. The standard processing routines provided are,

dollar_eql
dollar_symbol
dollar_cvna
dollar_cvan
dollar_intern

ParaOPS5 provides one additional routine, **dollar_cvas**, which is the counterpart to **dollar_intern** and allows a symbolic atom to be converted to the character string it represents.

It can be a bit confusing to keep straight the type required for parameters and return values for the interface functions and to realize when conversions need to be made. To make all of this more explicit, the procedure headers for all the interface functions and scalar processing routines are given in Appendix I. The headers show the type specifications for parameters and return values and give a brief summary of the function performed. The Appendix should be used in conjunction with Section 7 of the *OPS5 User's Manual* [2].

ParaOPS5 also restricts **user defined function calls** to a maximum of 50 parameters. Another difference is that **dollar_litbind** will return the number 0 when its argument is not a symbolic atom that has been bound by **Literal** or **Literalize**. OPS5 would return the argument unchanged in this case. However, the ParaOPS5 representation for atoms precludes doing this since the 32-bit quantity representing the atom might coincidentally be the same as the atom's numeric binding.

5.3. Weaknesses, Limitations, and Bugs

Some of the weaknesses and limitations of the ParaOPS5 system are related to the originally intended use of the system as a research tool for investigating parallelism in production systems. Our use of the system has centered around benchmarking existing OPS5 programs rather than new program development. As a result, there are some weaknesses and limitations that are enumerated below:

1. The ParaOPS5 compiler was created without much concern for compiler error messages and error recovery. The compiler is unforgiving when errors occur and gives only the line number where the error occurred and some context information before it exits. For those of you who find this unbearable, you can use one of the Lisp-based OPS5 systems[3] to get all syntax errors out of your program before using ParaOPS5.

2. The compiler does not support any features for separate compilation. All declarations and productions must be contained in a single file. This can be somewhat annoying when making changes to productions during the development of a large program.

3. The system does not support the **build** action.

A more serious concern is the potential existence of bugs in both the compiler and the run-time system. Features such as I/O, files, user defined functions, and much of the user interface have received modest use and large pieces of these were added only recently to provide for functionality needed for real program development. Also most of the system has not been subjected to exhaustive and systematic testing procedures to uncover errors.

On a positive note however, another research group at Carnegie Mellon has recently moved a system with over 600 productions onto ParaOPS5 without uncovering any bugs in ParaOPS5. Their application makes use of some of the less used OPS5 features just mentioned.

---

[3]Franzlisp and Common Lisp versions, **vps2.l** and **vps2.cl** respectively, are on the release tape and can be removed with **tar xv vps2.l vps2.cl**.

6. Programming Style and Design

This section provides some insights into OPS5 program design that can have a significant impact on the parallelism available for ParaOPS5 to exploit in a program. Program design issues are presented here in the context of a detailed analysis [8] of program execution behavior in ParaOPS5. The analysis has revealed three bottlenecks that can limit the parallelism in the current ParaOPS5 implementation. These bottlenecks and suggestions for preventing them are described below:

1. *Small Cycles:* Small cycles are recognize-act cycles which contain less than about 50 tokens in them. In ParaOPS5, a token in the Rete network corresponds to a node activation task. When the number of tasks in a cycle is small, ParaOPS5 does not achieve good speed-ups on that cycle since there is not enough work to keep the match processes busy and the processing overheads at the beginning and end of the cycle have a greater impact.

   If possible, instead of having a system with only 15 tokens/cycle generated in 5 different cycles, it is better to have a system with 150 tokens generated in one cycle. Though the number of tokens is doubled here, ParaOPS5 will get good speed-ups; and thus make up for the inefficiency.

   It is difficult to give a precise method for designing a program to minimize short cycles. However, a general rule of thumb is to prefer productions which have a large impact. This implies favoring productions that have a larger number of RHS actions to affect working memory or that have RHS actions that affect a greater number of other productions.

2. *Long Chains:* Productions with a large number of CEs can be a bottleneck due to the long chain effect [5]. The long chain effect is due to a long chain of dependent node activations in which a node activation causes an activation of its successor node which in turn causes an activation of its successor node, and so on. The chain imposes a serial processing order which can have a serious impact on parallelism when the chains are too long. In general, more than 15-20 CEs in a production can lead to a long chain effect. If possible, such productions should be avoided.

3. *Cross Products:* A cross product refers to the tokens that get generated by a 2-input node whenever it is activated. When a token flows into a 2-input node and activates it, the node computes the cross product by examining the tokens stored in the node's opposite memory (see Section 2.2) in order to find tokens which satisfy the predicates in the CEs that the 2-input node serves to join. If the number of tokens in the opposite memory is large, then the work required to compute the cross product is also large.

   When the CEs joined by the node share common variables bound with the predicate for equality, then ParaOPS5 is able to greatly reduce the amount of work required to compute the cross product. This is because ParaOPS5 uses hashing [8] based on the *equality variables* along with the node's ID to discriminate among the tokens in the opposite memory that need to be examined. But if there are no equality variables to provide such discrimination, then all of the tokens in the opposite memory must be tested. In this case, the processing required for the node activation to compute the cross product can present a bottleneck that limits parallelism.

Therefore, use of equality variables common among CEs is recommended for programs to be run on ParaOPS5 in order to take advantage of the hashing. Some programs, however, do not readily admit the use of such variables. Consider, for example, the production **cross:product** shown below which is used to generate all the 4-permutations of a set of, say, 25 numbers:

```
(p cross:product
        (Number <n>)
        (Number {<k1> <> <n>)
        (Number {<k2> <> <k1> <> <n>})
        (Number {<k3> <> <k2> <> <k1> <> <n>})
 -->
        (make Permute4 ^slot1 <n> ^slot2 <k1>
                                  ^slot3 <k2> ^slot4 <k3>)
)
```

This production will have large cross products to compute at each of its 2-input nodes since it has no equality variables common across CEs. Also by the nature of the problem, it is not possible to introduce any such variables either. In cases such as this, it may be possible to break up the problem into several pieces and replace a single production with several. This usually requires introducing some explicit knowledge about the domain. In our example, suppose we know in advance that the set of numbers is {1, 2, 3, ..., 25}. Then we can replace the single production above with 25 more specific productions:

```
(p cross:product:case-1
        (Number 1)
        (Number {<k1> <> 1)
        (Number {<k2> <> <k1> <> 1})
        (Number {<k3> <> <k2> <> <k1> <> 1})
 -->
        (make Permute4 ^slot1 1 ^slot2 <k1>
                                ^slot3 <k2> ^slot4 <k3>)
)

(p cross:product:case-2
        (Number 2)
        (Number {<k1> <> 2)
        (Number {<k2> <> <k1> <> 2})
        (Number {<k3> <> <k2> <> <k1> <> 2})
 -->
        (make Permute4 ^slot1 2 ^slot2 <k1>
                                ^slot3 <k2> ^slot4 <k3>)
)
.
.
.
(p cross:product:case-25
        (Number 25)
        (Number {<k1> <> 25)
        (Number {<k2> <> <k1> <> 25})
        (Number {<k3> <> <k2> <> <k1> <> 25})
 -->
        (make Permute4 ^slot1 25 ^slot2 <k1>
                                 ^slot3 <k2> ^slot4 <k3>)
)
```

> This new scheme does not reduce the amount of work required. It does, however, provide the opportunity to process the 25 subparts in parallel, thus reducing the execution time for the program. Again, the method requires incorporating specific knowledge about the domain. Some problems do not present such an opportunity for incorporating domain knowledge to increase the available parallelism. The PSM group is currently investigating domain independent solutions to the problem of large cross products.

7. Acknowledgements

This research was sponsored by Encore Computer Corporation, Digital Equipment Corporation and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the Air Force Avionics Laboratory. Anoop Gupta is supported by DARPA contract MDA903-83-C-0335 and an award from the Digital Equipment Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Encore Computer Corporation, Digital Equipment Corporation and the Defense Advanced Research Projects Agency or the US Government.

Appendix I.

This section provides the routine headers for the standard OPS5 interface routines and scalar processing routines that may be called by external routines invoked through the OPS5 **call** action or **user defined function call**. The interface routines allow external routines to communicate with OPS5 in order to obtain parameters and return results. The scalar processing routines allow external routines to convert between OPS5 atoms and their actual representation in C as numbers or strings. The routine headers show the type specification for parameters and return values and give a brief summary of the function performed. The interface routines are discussed in Section 7 of the *OPS5 User's Manual* [2] and in Section 5.2 of this document.

```
#define  TRUE  1
#define  FALSE 0


typedef  long  OpsVal;



int
dollar_eql(atom1, atom2)
   OpsVal atom1, atom2;
/*------------------------------------------------------------------------------
 *
 * Abstract:
 *    Test if 2 atoms are the same.
 *
 * Parameters:
 *    atom1, atom2 - the atoms to test.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    TRUE if the same, FALSE otherwise.
 *
 * Called by:
 *    User defined routines.
 *
 *----------------------------------------------------------------------------*/
```

```
int
dollar_symbol(atom)
   OpsVal atom;
/*----------------------------------------------------------------------------
 *
 * Abstract:
 *    Test if the atom is a symbolic atom or numeric atom.
 *
 * Parameters:
 *    atom - the atom to test.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    TRUE if a symbolic atom and FALSE otherwise.
 *
 * Called by:
 *    User defined routines.
 *
 *--------------------------------------------------------------------------*/
```

```
OpsVal
dollar_cvna(num)
   int num;
/*----------------------------------------------------------------------------
 *
 * Abstract:
 *    Convert a regular integer number to an OPS5 numeric atom.
 *
 * Parameters:
 *    num  - the number to convert.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    The numeric atom representing the number.
 *
 * Called by:
 *    User defined routines.
 *
 *--------------------------------------------------------------------------*/
```

```
int
dollar_cvan(atom)
   OpsVal atom;
/*-----------------------------------------------------------------------------
 *
 * Abstract:
 *    Convert a numeric atom to a regular number.
 *
 * Parameters:
 *    atom - the numeric atom to convert.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    The number represented by the numeric atom.
 *
 * Called by:
 *    User defined routines.
 *
 *---------------------------------------------------------------------------*/
```

```
OpsVal
dollar_intern(str)
   char *str;
/*-----------------------------------------------------------------------------
 *
 * Abstract:
 *    Convert a string to a symbolic atom and return the symbolic atom.
 *    Also enters it into the global symbol table if it is not already
 *    there.
 *
 * Parameters:
 *    str - the string to convert.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    The symbolic atom representing the string symbol.
 *
 * Called by:
 *    User defined routines.
 *
 *---------------------------------------------------------------------------*/
```

```
char*
dollar_cvas(atom)
   OpsVal atom;
/*----------------------------------------------------------------------------
 *
 * Abstract:
 *    Convert a symbolic atom into a string.
 *
 * Parameters:
 *    atom - the symbolic atom to convert.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    A pointer to the string represented by the symbolic atom.
 *
 * NOTE:
 *    This is not a standard OPS5 routine but seems like it might be
 *    useful to have available.
 *
 * Called by:
 *    User defined routines.
 *
 *---------------------------------------------------------------------------*/
```

```
OpsVal
dollar_parameter(findex)
   int findex;
/*----------------------------------------------------------------------------
 *
 * Abstract:
 *    Get the value (i.e., atom) held in the indicated field of the result
 *    element wme.
 *
 * Parameters:
 *    findex - the index of the required field of the result element.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    The atom from the indicated field. If the field was never given a
 *    value, the symbolic atom for "nil" is returned.
 *
 * Called by:
 *    User defined routines.
 *
 *---------------------------------------------------------------------------*/
```

```
int
dollar_parametercount()
/*----------------------------------------------------------------------
 *
 * Abstract:
 *    Determine the number of the last field in the result element wme that
 *    received a value. For call actions, this corresponds to the number of
 *    parameters supplied to the called user function.
 *
 * Parameters:
 *    None.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    The index of the last field assigned a value.
 *
 * Called by:
 *    User defined routines.
 *
 *---------------------------------------------------------------------*/
```

```
dollar_assert()
/*----------------------------------------------------------------------
 *
 * Abstract:
 *    Copy the result element wme into working memory.
 *
 * Parameters:
 *    None.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    Nothing.
 *
 * Called by:
 *    User defined routines.
 *
 *---------------------------------------------------------------------*/
```

```
dollar_tab(atom)
   OpsVal atom;
/*----------------------------------------------------------------------------
 *
 * Abstract:
 *    Advance the pointer into the result element wme to the field indicated.
 *    This is where the next value will be inserted into the result element
 *    when "dollar_value" is next called.
 *
 * Parameters:
 *    atom - a numeric or symbolic atom that specifies the field index; a
 *           symbolic atom here must represent an attribute symbol that
 *           received a numeric binding via a Literalize or Literal
 *           declaration.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    Nothing.
 *
 * Called by:
 *    User defined routines.
 *
 *--------------------------------------------------------------------------*/
```

```
dollar_value(atom)
   OpsVal atom;
/*----------------------------------------------------------------------------
 *
 * Abstract:
 *    Insert a symbolic or numeric atom into the result element wme.
 *
 * Parameters:
 *    atom - the atom to insert.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    Nothing.
 *
 * Called by:
 *    User defined routines.
 *
 *--------------------------------------------------------------------------*/
```

```
dollar_reset()
/*----------------------------------------------------------------------
 *
 * Abstract:
 *    Clears out all values in the result element wme.
 *
 * Parameters:
 *    None.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    Nothing.
 *
 * Called by:
 *    User defined routines.
 *
 *---------------------------------------------------------------------*/
```

```
FILE *
dollar_ifile(fileatom)
   OpsVal fileatom;
/*----------------------------------------------------------------------
 *
 * Abstract:
 *    Get a file descriptor to access a file previously opened by an
 *    "openfile" RHS action. Access is for reading.
 *
 * Parameters:
 *    fileatom - a symbolic atom that is associated with an open file,
 *               the symbolic atom was associated by a prior "openfile".
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    The file descriptor or NULL if the fileatom is not associated with
 *    a file open for input.
 *
 * Called by:
 *    User defined routines.
 *
 *---------------------------------------------------------------------*/
```

```
FILE *
dollar_ofile(fileatom)
   OpsVal fileatom;
/*----------------------------------------------------------------------------
 *
 * Abstract:
 *    Get a file descriptor to access a file previously opened by an
 *    "openfile" RHS action. Access is for writing.
 *
 * Parameters:
 *    fileatom - a symbolic atom that is associated with an open file,
 *               the symbolic atom was associated by a prior "openfile".
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    The file descriptor or NULL if the fileatom is not associated with
 *    a file open for output.
 *
 * Called by:
 *    User defined routines.
 *
 *--------------------------------------------------------------------------*/



int
dollar_litbind(atom)
   OpsVal atom;
/*----------------------------------------------------------------------------
 *
 * Abstract:
 *    Check if a symbolic atom has received (via a Literalize or Literal
 *    declaration) a numeric binding (which represents an attribute index
 *    to a field in a wme).
 *
 * Parameters:
 *    atom - the symbolic atom to check.
 *
 * Environment:
 *    This routine is used by user defined functions and call actions to
 *    communicate with the OPS5 interpreter.
 *
 * Returns:
 *    The numeric binding assigned to the symbol if it’s an attribute.
 *    Returns 0 otherwise.
 *
 * NOTE:
 *    This functions differs from the one defined in the standard OPS5
 *    manual in that it returns 0 if the string symbol has no binding.
 *
 * Called by:
 *    User defined routines.
 *
 *--------------------------------------------------------------------------*/
```

References

[1] Brownston, L., Farrell, R., Kant, E., and Martin, N.
Programming Expert Systems in OPS5: An introduction to rule-based programming.
Addison-Wesley, Reading, Massachusetts, 1985.

[2] Forgy, C. L.
OPS5 User's Manual.
Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, July, 1981.

[3] Forgy, C. L.
Rete: A fast algorithm for the many pattern/many object pattern match problem.
Artificial Intelligence 19(1):17-37, 1982.

[4] Forgy, C. L.
The OPS83 Report.
Technical Report CMU-CS-84-133, Computer Science Department, Carnegie Mellon University, May, 1984.

[5] Gupta, A.
Parallelism in Production Systems.
PhD thesis, Computer Science Department, Carnegie Mellon University, March, 1986.

[6] Gupta, A. and Tambe, M.
Suitability of message passing computers for implementing production systems.
In *Proceedings of the National Conference on Artificial Intelligence*, pages 687-692. August, 1988.

[7] Gupta, A., Forgy, C. L., Kalp, D., Newell, A., and Tambe, M.
Parallel OPS5 on the Encore Multimax.
In *Proceedings of the International Conference on Parallel Processing*, pages 271-280. August, 1988.

[8] Gupta, A., Tambe, M., Kalp, D., Forgy, C. L., and Newell, A.
Parallel implementation of OPS5 on the Encore Multiprocessor: Results and analysis.
International Journal of Parallel Programming 17(2), 1989.

Table of Contents

List of Figures