An Investigation of an OR Parallel Execution Model for Horn Clause Programs

by

Khayri A.M. Ali and Milton Wong

SICS R88017
Research Report
ISSN 0283-3638

An Investigation of an OR Parallel Execution Model for Horn Clause Programs

Khayri A.M. Ali

Swedish Institute of Computer Science
Stockholm, Sweden

Milton Wong

Department of Telecommunication and Computer Systems
The Royal Institute of Technology
Stockholm, Sweden

(September 1988)

ABSTRACT

We present a model for OR parallel execution of Horn clause programs on a combined local and shared memory multiprocessor system. In this model, the shared memory only contains control information that guides processors requesting a job to independently construct the environment required to get a new job. Each processor has a local memory containing its own binding environment. This reduces the traffic to the shared memory and allows each processor to process its job with high performance. Each processor is almost the same as Warren's Abstract Machine (WAM). A method for nonshared memory multiprocessor architectures is outlined. We also present some preliminary results of an experimental investigation of the model.

1. Introduction

Many researchers have proposed schemes for OR parallel execution of Prolog programs. OR parallelism allows different clauses of a predicate to be tried in parallel.

However, many of the approaches that have been suggested for OR parallel execution involve either sharing or copying large data structures among the processors. This gives rise to heavy memory or communication contention. Therefore, several researchers [1,5,7] have devised methods for OR parallel execution which reduce the sharing or copying of data by doing some duplicate computation (recomputation). In our model [1], processors only share some control information that guides processors requesting a job to independently construct the environment required to get a new job. Each processor has a local memory containing its own binding environment. A copy of the program is stored in either the global memory or in each local memory. Thus, the traffic to the shared memory is reduced by doing some recomputation. Each processor is a modified WAM [8], where the semantics of the TRY, RETRY, TRUST and FAIL

instructions has been changed.

Several variants of the model have been implemented and evaluated [9,10]. The implementation was made on a Sequent Balance 8000 multiprocessor system with 10 processors [11], and is based on SICStus Prolog, a sequential Prolog system developed at the Swedish Institute of Computer Science (SICS) [3].

In the next section, our model is presented. Section 3 discusses some implementation issues. Section 4 presents the experimental investigation of the model. Section 5 outlines a method for nonshared memory multiprocessors. Section 6 concludes the paper and discusses our findings and outlines some possibilities for future work.

2. The model

In this section, a short description of our model will be given. A complete description of the model is given in [1].

In the model, processors only share some control information. This information specifies the shortest path from the root of the search tree to the current state of each processor. It also gives the active choice points, the untried branches of those choice points, and which processors that share the points.

Processors are in either **computation mode** or **recomputation mode.** A processor is in recomputation mode if it is following another processor. Otherwise, the processor is in computation mode.

When a processor p finishes its current job, it backtracks until it finds a choice point with an untried branch or a choice point shared by another processor in computation mode. If p finds an untried branch, it takes the branch. If there is no untried branch at the choice point, but there is a computation mode processor q (other than p), p switches to recomputation mode and follows q.

If processor p is following processor q, and p gets to a choice point where there is an untried branch, p stops following q, switches to computation mode, and takes the branch.

Each processor keeps its binding environment in its own local memory. Bindings are not copied or shared among processors. Instead, each processor constructs its own binding environment. This implies that some work is duplicated (recomputation).

Each processor is a WAM, with a local memory containing the data areas of WAM: the local stack, the trail stack and the heap.

Figure 1 shows an example of a state during the execution of a program. Assume that Pi, Pj and Pk are the only processors working in the subtree with root Y. Suppose that processor Pk finishes its current job. It will then backtrack to choice point Y, where it finds that processors Pi and Pj are in computation mode and below Y. Since there are no untried branches at Y, Pk will then switch to recomputation mode and follow either Pi or Pj to choice point X, where there is an untried branch. At X, Pk switches to computation mode and takes the untried branch. (We assume that Pi and Pj are still below X on the first and second branch, respectively, at this time)

3. Implementation issues on the model

In this section, we discuss some implementation issues on the model presented in Section 2. Section 3.1 presents the global data areas of the model. Section 3.2 discusses the start phase. Section 3.3 discusses how to report solutions. Section 3.4

discusses different strategies to select a processor to follow. Section 3.5 discusses different strategies for locking global data.

3.1. The global data areas of the model

In addition to each processor's local (WAM) data areas, the model includes a number of global data areas. These are (Figure 2): one **History-Path stack** for each processor, one **split frame** for each active choice point, one **recomputation flag** for each processor, a **recomputation matrix** , one **stop flag** for each processor, and one variable for each processor containing the number of recomputing processors that are following it.

3.1.1. History-Path stacks

Each processor is associated with a stack, called the **History-Path stack (HP stack)** , containing the shortest path from the root of the search tree to the current branch being processed. Each entry of the stack consists of two parts: *branch number* and *frame address*. The frame address part points to a global frame (split frame) that keeps information about a choice point and the branch number specifies the current branch of that point being processed. An entry is pushed on the stack when a choice point is reached by the processor and popped on backtracking to the previous choice point. The branches of each choice point are numbered from left to right, starting with 1.

Figure 3 shows the numbering of branches of a search tree and four HP stacks associated with four processors processing branches of the tree. Each stack contains a unique sequence of branch numbers identifying the respective branch currently being processed. xi points to a global frame that contains information specifying the choice point Ci.

3.1.2. Split frames

Each active choice point is associated with a global frame called a **split frame** , which contains information that specifies the name of the processors that share it, and the untried branches of the choice point. Branches of each point are selected from left to right.

At most, there is one split frame for each active choice point. A new frame is allocated when a choice point is processed by the first processor that processes it. The frame is deallocated when it becomes unaccessible by any processor.

Figure 4 shows the split frames associated with choice points C1, C2, C3 and C4 (referred to by x1, x2, x3 and x4) of the example shown in Figure 3.

If a choice point is processed by only one processor, the corresponding choice point frame (CP frame) and split frame is deallocated when the last branch of that point is selected. In this case, the value of the frame address field of the corresponding HP stack entry becomes NOSF (no split frame). Thus the number of split frames is less than or equal to the number of active choice points in the system.

3.1.3. Recomputation flags

As mentioned before, a processor is in either computation mode or recomputation mode. There is one **recomputation flag** for each processor. This flag is true if the processor is in recomputation mode, false otherwise.

3.1.4. Recomputation matrix

The **recomputation matrix** of size n*n consists of one **recomputation vector** for each processor in the system (n=the number of processors). Each recomputation vector has n bits, where each bit corresponds to one processor. Bit j of the recomputation vector of processor i is 1 if processor j is following processor i, 0 otherwise.

The recomputation matrix is needed because when a processor q in computation mode backtracks to a point which some processor p has passed through, and p is following q, q has to stop p. So each processor in computation mode has to know which processors, if any, are following it.

3.1.5. Stop flags

Each processor is associated with a **stop flag.** When a processor q is going to stop a processor p, it sets p's stop flag. P reads its stop flag when it arrives at a choice point. When p detects that its stop flag is set, it stops following q.

3.2. Start phase

We have suggested two different starting strategies. In the first strategy, one of the processors, say p1, starts in computation mode; the others start in recomputation mode, following p1.

In the second strategy, all processors start in computation mode. The processor that gets to the first choice point first in time, creates the corresponding split frame and takes the first branch. Processors that get to the first choice point later on take one of the untried branches, if there are any, or switch to recomputation mode and follow one of the processors that have got a branch of that split frame.

3.3. Reporting results

Only processors in computation mode report results.

3.4. Selection strategies

In the model, when a processor p backtracks to a choice point with no untried branches, and there is another processor, say q, at that point, and processor q is in computation mode, p follows q. This is because there may be untried branches further down the subtree which q is in.

But what if there is more than one processor in computation mode which p could possibly follow? Which one should be chosen in this case?

In one implementation, the computation mode processor with the lowest processor identification number is chosen. This tends to distribute the processors rather unevenly over the search tree. In general, it appears that a more balanced selection strategy would give better results. This is also confirmed by the results of the experiments, which show better performance for a more balanced selection strategy.

In the original specification of the model, if a processor p is following a processor q and p gets to a choice point where there are no untried branches, and q is below that point, p continues to follow q, even though there may be other processors in computation mode below that point. It seems to be better to use the selection strategy that is used after backtracking in this case, too. Permitting p to follow someone else than q makes it possible to distribute processors more evenly over the search tree. This belief

is also supported by the results of the experiments.

A balanced selection strategy may not only reduce the amount of recomputation done, it may also reduce the time spent waiting for mutually exclusive access to shared data, especially if a locking strategy is used which allows parallel access to global control information associated with different parts of the search tree.

3.4.1. Follow the processor with the fewest followers

One way of achieving a more balanced distribution is to follow the computation mode processor with the fewest followers. If there is more than one such processor, select one of them arbitrarily.

3.4.2. Search cyclically at each choice point

The operation of selecting the computation mode processor with the fewest followers requires a large amount of accesses to shared data. All computation mode processors at the choice point have to be found. For each computation mode processor, its number of followers has to be read and compared to the smallest number of followers previously found during the search. If one finds a computation mode processor with no followers at all, the search can stop.

Another selection strategy, which requires less shared data accesses and which reduces the search time, is to search cyclically the bit vector in the split frame recording which processors that are below that choice point. Each split frame then contains an index variable kv which contains an index to the bit vector. The search for a processor to follow starts at the element of the bit vector given by the value of kv. If that bit is reset or the corresponding processor is not in computation mode, the search continues with the next element of the vector. The index variable kv is incremented for each processor that is examined. If the end of the bit vector is reached, the search continues at the beginning of the vector. The first computation mode processor with its bit set that is found is selected.

3.4.3. Which selection strategy is the best one?

The best choice of selection strategy depends on the relative costs of accesses to shared and unshared information as well as on properties of the programs that are to be executed by the machine. Following the processor with the fewest followers implies a relatively expensive selection operation in terms of shared data accesses, but also less recomputation and fewer useless try and fail operations executed. The cyclic search strategy, on the other hand, implies a faster selection operation with fewer shared data accesses, but also more recomputation and more useless trys and fails.

It should be noted that regardless of which selection strategy that is chosen, only those processors that are below the choice point are examined. Only if there are no untried branches and no computation mode processors at a point does the processor backtrack and look at a larger part of the search tree.

3.5. Locking strategies

Some operations require mutually exclusive access to shared data. We have proposed a number of locking strategies with different degrees of restriction of the parallelism.

In **the first locking strategy (L1)** , only one processor at a time manipulates the global control information.

In the following locking strategies, parts of the global data can be manipulated in parallel. The global control information can be divided into two sections: "shared" and "unshared" sections. The "shared" section is the part of the global control information related to the parts of the search tree which are processed by more than one processor. The "unshared" section is the part of the global control information corresponding to the parts of the search tree that are processed by only one processor. Figure 5 shows the part of a search tree that is associated with "shared" control information.

The second locking strategy (L2) is as follows. Whenever no processor is manipulating control information of the "shared" section, processors can manipulate control information of the "unshared" section in parallel. When a processor is manipulating a part of global control information of the "shared" section, no other processor is allowed to manipulate any part of the global control information.

In **the third locking strategy (L3)** , many processors can manipulate control information in the "unshared" section in parallel with one processor at a time manipulating control information in the "shared" section.

Even L3 may be too restrictive. In L3,
only one processor at a time may access control information in the "shared" section. In the following locking strategies, we allow processors to access "shared" control information in parallel.

Processors may be allowed to access "shared" split frames and HP-entries related to **different** choice points in parallel. This would require a separate lock for each choice point. Let "shared" information that is not associated with a particular choice point (recomputation flags, number of followers, stop flags, recomputation matrix) be accessed by only one processor at a time. There would then be one lock for such choice point independent "shared" information. This strategy will be referred to as **L4.**

A strategy that would be easier to implement is the following: let any number of processors manipulate control information in the "unshared" section in parallel. In the "shared" section, let one processor at a time manipulate split frames and HP-entries at level i, for all i. Let one processor at a time access CP independent "shared" information. This is less restrictive than L1, L2 and L3, allowing processors to access "shared" split frames and HP-entries at different levels in parallel. This strategy will be referred to as **L5.**

A variant of locking strategy L5 uses a fixed number of locks for CP related "shared" information. The levels would then be mapped into the finite set of locks. This strategy will be referred to as **L6.**

The same principle can be applied to L4. Use a fixed number of locks for CP related "shared" information. A mapping f from levels and branch addresses into the finite set of locks would then be used. One possible way of organizing the locks is as a two-dimensional matrix. Call the resulting strategy **L7.**

Of course, it is possible to devise locking strategies that are even less restrictive than strategy L4. Such strategies will be more complicated, like the locking used in the Aurora OR parallel Prolog system [6].

4. An experimental investigation of the model

In this section, we describe the experimental investigation [9,10] of the model.

4.1. First experiment: evaluation of selection strategies

This section presents some of the results of the first phase of the experimental investigation [9]. Three different variants of the model, with different selection strategies, were tested.

4.1.1. The benchmark programs

A set of benchmark programs has been run on the implemented machine in order to evaluate the performance of the model. The set of benchmark programs consists of a subset of a set of programs presented by Ciepielewski et al [4] together with two abstract programs having specified execution trees.

In [4], the results of measurements of certain properties of the presented benchmark programs, e.g. degree of parallelism, are given. However, these results are not directly applicable to the model, since the results are only valid under certain assumptions, which do not hold for the model. For instance, the effects of indexing are not taken into account in [4]. When indexing is used, the degree of parallelism for some of the benchmark programs becomes significantly lower than indicated in [4]. This is not a disadvantage, since the work that has been eliminated is of no use. Indexing also reduces the percentage of short branches. In the implementation of the model, indexing has been used. Another implicit assumption in [4] is that the execution time between each pair of adjacent nodes in the search tree is the same. Nevertheless, the data given in [4] for the presented benchmark programs gives a rough idea of the behaviour of the programs. This facilitates the interpretation of the results of running these benchmark programs.

In order to evaluate specific aspects of the model, we have constructed two benchmark programs that have specified search trees.

4.1.2. Implemented selection strategies

In the first phase of the investigation, three implementations were made, using locking strategy L1, with different selection strategies. In **implementation 1,** the selection strategy is to follow the computation mode processor with the lowest processor identification number.

In **implementation 2** and **implementation 3,** the strategy is to search cyclically at each choice point. In **implementation 3** only, the selection strategy is also used by a processor p when it is following a processor q and p gets to a choice point without untried branches, and q is below that point. This means that p might then choose another processor to follow, if there is a computation mode processor other than q below that point. In the **implementations 1 and 2,** p would continue to follow q in this case.

4.1.3. Results

In this section, some of the results of the experiments are given. For each benchmark program, there is the code, some of the properties of the search tree listed in [4], and the measured performance. The performance is given as speed-up as a function of the number of processors, where

speed-up(n) = sequential execution time / exec. time with n processors,

where sequential execution time = the time taken by the original, unaltered (sequential) SICStus Prolog WAM.

Thus, both the relationships between the implementations and the relationship between the sequential, unmodified SICStus Prolog WAM and the parallel implementations may be determined from the figures.

4.1.3.1. permute

The code for permute, which is taken from [4], is given in Figure 6.

Some of the properties of the search tree of permute listed in [4] are:

maximum depth:	10
mean degree of parallelism:	8
maximum branch length:	5
percentage of branches with length 1:	62

In the implementations of the model, as in SICStus Prolog, indexing on the principal functor of the first argument is used, which implies that the degree of parallelism and the percentage of branches with length 1 is reduced in comparison with the values listed above.

The performance of permute on the three implementations is shown in Figure 7. The figure shows speed-up as a function of the number of processors. The best maximum speed-up is 0.55 at 2 processors for implementation 2.

4.1.3.2. qsort2

The code for qsort2, which is taken from [4], is given in Figure 8.

Some of the properties of the search tree of qsort2 listed in [4] are:

maximum depth:	28
mean degree of parallelism:	3
maximum branch length:	4
percentage of branches with length 1:	59

The performance of qsort2 on the implementations is shown in Figure 9. This is a program with little parallelism.

4.1.3.3. mutation

The code for mutation, which is taken from [4], is given in Figure 10.

Some of the properties of the search tree of mutation listed in [4] are:

maximum depth:	24
mean degree of parallelism:	78
maximum branch length:	13
percentage of branches with length 1:	39

The performance of mutation is shown in Figure 11. The behaviour of mutation is typical of highly parallel programs. Speed-up reaches 1.0 at 4 processors for implementation 1.

4.1.3.4. numbers

The code for numbers, which is taken from [4], is given in Figure 12.

Some of the properties of the search tree of numbers listed in [4] are:

maximum depth:	23
mean degree of parallelism:	41
maximum branch length:	11
percentage of branches with length 1:	59

The performance of numbers is shown in Figure 13. The performance of numbers is not quite as good as the performance of mutation, speed-up reaching 0.70 at 3 processors for implementation 3.

4.1.3.5. sort

The code for sort, which is taken from [4], is given in Figure 14.

Some of the properties of the search tree of sort listed in [4] are:

maximum depth:	31
mean degree of parallelism:	435
maximum branch length:	9
percentage of branches with length 1:	58

The performance of sort is shown in Figure 15. The best maximum speed-up is 0.78, reached by implementation 3 at 3 processors. Here, implementation 3 is the best one for large numbers of processors, with implementation 2 in second place and implementation 1 being the worst.

4.1.3.6. fast

We have constructed this program to test the effects of changing the ratio between the time spent accessing global control information and the time spent working on local data. The search tree of fast has 90 identical, deterministic branches from the root. By varying the length of these branches, the ratio global time/local time is varied. The code for fast is given in Figure 16. 7 versions of fast with different branch lengths were run: fast10000, fast1000, fast100, fast75, fast50, fast25 and fast10.

The performance of fast10000/fast100/fast50/fast25/fast10 is shown in Figure 17a/b/c/d/e. Performance deteriorates from an almost linear speed-up for fast10000 to a mutation-like performance for fast10.

4.1.3.7. select

This program was constructed to compare the different selection strategies. The search tree is highly asymmetrical: the call p(X), where X is bound to an integer greater than 0, matches 2 clauses of p/1; the first clause leads to a long, (almost) deterministic branch, whereas the second clause leads to a subtree with more than X branches. The code for select is given in Figure 18.

The performance of select is shown in Figure 19. Implementation 1 performs very poorly, giving a constant speed-up of 1.8 when the number of processors is equal to or greater than 2. The best performance is given by implementation 3, but it only achieves logarithmic speed-up. The reason for this is that regardless of selection strategy, when a processor does not find any untried branch and follows another processor along a branch, there may not be any work for it to do on that branch.

4.1.4. Summary of the first experiment

Figure 20 is a brief summary of some of the results of the measurements (max sp = maximum speed-up, @ = number of processors at which maximum speed-up was reached, @9 = speed-up at 9 processors, mean par = mean degree of parallelism). Parentheses indicate that the maximum speed-up was reached at 9 processors, so that increasing the number of processors beyond 9 may possibly increase the speed-up further.

The programs from Ciepielewski's benchmark [4] can be roughly divided into two groups: "parallel programs", which reached their best maximum speed-up when the number of processors was greater than 1 ; and "sequential programs", which reached their best maximum speed-up at 1 processor. The parallel programs have mean degrees of parallelism greater than 8; the sequential programs have mean degrees of parallelism less than or equal to 8.

The programs of the group "parallel programs" all give quite similar performance. The speed-up increases to about 0.8 at 3-4 processors, and then decreases again as the number of processors increases. For sort, the performance is improved significantly at higher numbers of processors by using a more balanced selection strategy. For the other programs in this group, the differences between the selection strategies are small.

The "sequential programs" are also very similar to each other. The speed-up for these programs decreases from about 0.5 at 1 processor to about 0.1 at 9 processors.

The performance of fast10000/fast100/fast50/fast25/fast10 ranges from an almost linear speed-up for fast10000 to a very mutation-like performance for fast10. The performance of the "fast" programs is independent of the selection strategy used.

The performance of "select", on the other hand, is highly dependent on the selection strategy, as expected. But even a more balanced selection strategy only gives a logarithmic speed-up. A possible solution to this problem may be to limit the number of processors allowed to follow each processor.

A point worth noting is that for all the programs that give significantly different performance with different selection strategies, implementation 3 performs best, with implementations 2 and 1 in second and third place, respectively. Thus it appears to be better to search cyclically at each choice point than to follow the computation mode processor with the lowest processor identification number. Also, a processor p following a processor q should use the selection strategy when p gets to a choice point without untried branches, and q is below that point.

When a locking strategy is used that permits processors to access different parts of the global control information in parallel, the advantage of distributing the processors evenly over the active search tree will become greater. But even when there is only one single lock for all global control information, a more balanced selection strategy gives better performance.

4.1.5. The importance of locking strategies

The choice of locking strategy strongly affects the performance of the model. The simple locking strategy that was used in implementations 1, 2 and 3, i.e. with one lock for all global control information, severely limits the performance.

That the performance is limited by the processors having to wait for mutually exclusive access to the global control information is also indicated by the results of the

measurements. Even those programs from Ciepielewski's benchmark [4] that have a high degree of parallelism only reach a speed-up of about 0.8. Furthermore, the abstract program "fast" shows that when the percentage of the time spent accessing global control information is increased, the performance gradually deteriorates and becomes very similar to the performance of the highly parallel programs from Ciepielewski's benchmark. With a low percentage of time spent accessing global control information, on the other hand, "fast" gives an almost linear speed-up.

4.1.6. Limiting the number of followers

When a processor p does not find any untried branch and follows another processor down a branch, there may not be any work for it to do on that branch. So it may be a good idea to limit the number of processors allowed to follow each processor. If no untried branch is found, and there is at least one processor in computation mode below the choice point, and all processors in computation mode that are below the choice point have the maximum allowed number of followers, the processor p has to wait.

By limiting the number of followers, the amount of useless recomputation is reduced. But in some cases, this may limit the parallelism of the computation. However, variants of the model with different constant limits on the number of followers are definitely worth studying.

4.2. Second experiment: evaluation of some locking strategies

This section presents some of the results of the second phase of the investigation [10]. In the second phase, several different locking strategies were implemented, and their performance was measured. With the best of the tested locking strategies, speed-ups of up to about 2 over sequential WAM were achieved for highly parallel programs, using 7 processors. In [10], we also derived a simple performance model for our machine.

4.2.1. The benchmark programs

The set of benchmark programs used in the second phase consists of the set that was used in the first phase [9] and a new abstract program, select2.

4.2.2. Implemented locking strategies

Three new implementations of the model were made. All of them use the same selection strategy as implementation 3 of the first phase of the investigation [9]. Implementation 3 will also be called **Old.**

The first new implementation, which will be referred to as **New,** has one lock for all global control information, but has been improved from implementation 3 (**Old**). An attempt has been made to reduce the time that this single lock is locked. The improvements of **New** are also included in the other two new implementations.

In the second new implementation, which will be called **E2,** locking strategy L2 is used. Whenever no processor is manipulating control information of the "shared" section, processors can manipulate control information of the "unshared" section in parallel. When a processor is manipulating a part of global control information of the "shared" section, no other processor is allowed to manipulate any part of the global control information.

In the third new implementation, which will be called **E2c,** a hybrid of locking strategy L2 and locking strategy L6 is used. At any time, either

1) No processor is manipulating any part of the global control information,

or

2) No processor is manipulating control information of the "shared" section, and one or more processors are manipulating control information of the "unshared" section in parallel,

or

3) No processor is manipulating control information of the "unshared" section, and one or more processors, that are at different levels of the search tree, are manipulating CP-related control information of the "shared" section in parallel. Only one of them at a time may manipulate non-CP-related control information.

4.2.3. Results of the second experiment

In this section, some of the results of the experiments are given. For each benchmark program, the measured performance is given as speed-up as a function of the number of processors.

Since speed-up is given relative to sequential WAM, the figures tell us how much faster the different benchmark programs can be executed with the implemented versions of the model than with a sequential machine that is equivalent to the parallel implementations except for the behaviour at choice points.

4.2.3.1. permute

The performance of permute is shown in Figure 21. For this program, **New** is 40% slower at 1 processor than sequential WAM. Performance improves at 2 and 3 processors, where speed-up is 0.75. Above 3 processors, the performance deteriorates, finishing at a speed-up of 0.30 at 9 processors.

E2 and **E2c** start at a speed-up equal to 0.55 at 1 processor, and gradually catch up with **New** as the number of processors increases, giving the same performance as **New** at 9 processors. The degree of parallelism of the program is not high enough for the more parallel locking strategies to pay off.

4.2.3.2. qsort2

The performance of qsort2 is shown in Figure 22.

4.2.3.3. mutation

The performance of mutation is shown in Figure 23. The behaviour of mutation is typical of highly parallel programs. Speed-up reaches 1.7 at 6 processors for **E2c.** For this program, **E2c** gives the best performance, with **E2** in second place and **New** third. Note that when the number of processors is lower than 5, **New** gives the best performance. This is because the benefits of the more parallel locking strategies do not outweigh the overhead of those strategies until the number of processors is fairly large.

4.2.3.4. numbers

The performance of numbers is shown in Figure 24. **New** gives the highest maximum speed-up, about 1.0 at 3 processors. When the number of processors becomes greater than 4, **E2** and **E2c** give slightly better performance than **New.** However, the maximum speed-up reached by **E2** and **E2c** is only about 0.9, at 3 processors.

4.2.3.5. sort

The performance of sort is shown in Figure 25. Maximum speed-up is 1.8 at 7 processors. Here, the best maximum speed-up is given by **E2c** , with **E2** and **New** in second and third place, respectively.

4.2.3.6. fast

The performance of fast10000/fast100/fast50/fast25/fast10 is shown in Figure 26a/b/c/d/e. fast10000 gives an almost linear speed-up, independent of the locking strategy. As the branches are shortened, performance deteriorates, and the best maximum speed-up for fast10 is 1.7, reached by **New** at 4 processors.

For fast10000, the differences between the locking strategies are small. For fast10, fast25, fast50 and fast100, **New** gives the best performance, with **E2** second best and **E2c** in third place. One reason for this behaviour is that the search tree of fast has (almost) only one choice point (the root), which means that (almost) nothing can be gained by letting processors work at different choice points in parallel.

4.2.3.7. select

The performance of select is shown in Figure 27. The differences between the locking strategies are small.

4.2.3.8. select2

This abstract program is a variant of select with very short left branches. The code for select2 is given in Figure 28.

The performance of select2 is shown in Figure 29. For a small number of processors, **New** gives better performance than **E2** and **E2c** , whereas **E2c** gives the best performance when the number of processors is greater than 4.

The best maximum speed-up is given by New (0.46 at 1 processor). This is a program with a low degree of parallelism. It is also a program where a large percentage of the time is spent accessing global control information, so that even **New** is 54% slower at 1 processor than sequential WAM.

4.2.4. Summary of the second experiment

Figure 30 is a brief summary of some of the results of the measurements (max sp = best maximum speed-up, @ = number of processors at which the best maximum speed-up was reached, impl = the name of the implementation that gave the best maximum speed-up, break = greatest number of processors for which **New** gives the best performance, mean par = mean degree of parallelism, E2c@1 = speed-up of **E2c** at 1 processor). Parentheses indicate that the best maximum speed-up was reached at 9 processors, so that increasing the number of processors beyond 9 may possibly increase the speed-up further.

The programs from Ciepielewski's benchmark [4] can be divided into two groups: "parallel programs", which reached their best maximum speed-up when the number of processors was greater than 1; and "sequential programs", which reached their best maximum speed-up at 1 processor. This gives almost the same division as in Section 3.3.4. The "parallel" programs have mean degrees of parallelism greater than or equal to 8; the "sequential" programs have mean degrees of parallelism less than 8.

The "parallel" programs give maximum speed-ups at 2 - 7 processors. When the number of processors increases further above the point of maximum speed-up, the speed-up decreases. When the number of processors is large, **E2c** gives the best performance. For small numbers of processors, **New** gives the best performance. The breakpoint above which **New** no longer gives the best performance is indicated in the column "break" of Figure 30. The column "impl" indicates which implementation that gives the best maximum speed-up. This information can be derived from columns "@" and "break": if "@" is greater than "break", then the best maximum speed-up was reached at a number of processors above the breakpoint, so the best maximum speed-up was given by **E2c.** Otherwise, the best maximum speed-up was reached by **New.**

For "sequential" programs, the best maximum speed-up is reached at 1 processor; as the number of processors increases, the speed-up decreases. At 1 processor, **New** gives the best performance. When the number of processors increases, **E2** and **E2c** gradually catch up with **New** , and at 9 processors, their performance is about the same as that of **New.**

fast10000 gives an almost linear speed-up, independent of the locking strategy. As the branches are shortened, the performance deteriorates and the differences between the locking strategies grow. **New** gives the best performance, with **E2** second best and **E2c** in third place.

The performance of select is practically independent of the locking strategy, as expected. select2, which is a variant of select with shortened branches, shows a behaviour typical for programs with a low degree of parallelism. It is also a program where a large percentage of the time is spent accessing global control information.

For most of the highly parallel programs, the locking strategy of **E2c** gives the best maximum speed-up, with **E2** second best and **New** in third place. The best maximum speed-up for these programs reaches 1.1-1.8 at 4-7 processors. These programs then run 3-5 times faster than at 1 processor. However, **New** always gives the best performance when the number of processors is small, so it gives the best maximum speed-up for programs with a low degree of parallelism. The sequential programs run 20-50% slower on **E2c** at 1 processor than on sequential WAM. The parallel programs run 30-60% slower on **E2c** at 1 processor than on sequential WAM. All three new implementations, **New, E2** and **E2c** , give better performance than **Old** (implementation 3). The locking strategy of **Old** , with one lock for all global control information, and also the use of this lock at times when it was not necessary, was the main factor limiting performance in **Old.**

To sum up, **New** gives better performance than **Old** for all benchmark programs and for any number of processors. The more parallel locking strategies of **E2** and **E2c** improve the performance for large numbers of processors, but give worse performance than **New** when the number of processors is small. For highly parallel programs, **E2c** gives the best maximum speed-up, whereas **New** is better for sequential programs. **E2c** executes highly parallel programs up to 1.8 times faster than sequential WAM, but

sequential programs run at down to 0.5 times the speed of sequential WAM. By improving the locking strategy, it may be possible to improve the performance further. Limiting the number of followers may also improve performance.

4.3. A performance model

We now present a performance model [10] for the machine.

Assume that the time taken by sequential WAM to execute a program is *wseq*.

When the program is executed by our machine with one processor, the execution will take longer time than with sequential WAM, because of the **manipulation of global control information** at choice points. Assume that the execution by our machine with one processor takes *w* time units. Then *(w - wseq)* is the time spent manipulating global control information at choice points, and *(w - wseq)/w* is the percentage of the time that is spent accessing global control information. Also, *wseq/w* is the speed-up for our machine with one processor.

Now, assume that the program is executed in parallel by more than one processor of our machine. Two factors that have to be taken into consideration are **contention for access to global control information** and the **degree of parallelism** of the program. Let *mp* be the mean degree of parallelism of the program, and *n* be the number of processors. Assume, as an approximation, that the degree of parallelism is constant and equal to *mp* during the whole execution of the program. If *n>mp* , then only the first *mp* processors contribute to the speed-up. Adding more processors than *mp* does not improve the performance further. Assume that there is one lock for all global control information, and that it is locked whenever any part of the global control information is manipulated. Now, *((w-wseq)/w) * n* is the percentage of the time that the global control information is being accessed. If *((w-wseq)/w) * n <= 1* , then the execution time is *w/min(n,mp)*. If *((w-wseq)/w) * n > 1* , then the contention for access to the global control information limits the performance, and the execution time is *(((w-wseq)/w) * n) * (w/min(n,mp))*.

So far, we have assumed that there is one single lock, and that it is locked whenever any global control information is accessed. But what if the lock is locked only part of the time that global control information is accessed? This can be modeled by introducing a variable *k*, representing the fraction of the time spent accessing global control information that the lock is locked (0 <= *k* <= 1). The factor *(w-wseq)/w* is then replaced by *k * ((w-wseq)/w)* in the formulas given above.

The effects of allowing different parts of the global control information to be accessed in parallel may be represented as a reduction of the fraction *k*. Note that the value of *w* is a function of the locking strategy used.

This simple performance model represents the behaviour of the implementations of the model quite accurately.

Figure 31 shows the measured performance of qsort2 executed by **Old** , together with the performance predicted by the model for *k* =1 (real = real performance, pred(3) = predicted performance for mp = 3, pred(1) = predicted performance for mp = 1). The best agreement between predicted and real performance is reached by letting *mp* =1. The mean degree of parallelism listed in [4] is 3. There are several reasons why the effective mean degree of parallelism is lower than the value stated in [4]. First, indexing was used in my measurements, but not in [4]. Second, in [4] the execution time between each pair of adjacent nodes in the search tree is the same. Third, the results

of [4] are based on the assumption that there is always a processor available when a new branch is reached, and that the processor can start working on the new branch without delay. In reality, there may not be any processor available for the new branch, and if there is one, it may take some time for it to reach the new branch.

Figure 32 shows the performance of sort executed by **Old** (real = real performance, pred(435) = predicted performance for mp = 435). k =1 and mp =435 gives a good agreement between predicted and real performance. Note that the effective mean degree of parallelism may well be lower than 435, even though the value cannot be determined by the results of the measurements for n<=9.

Figure 33 shows the performance of qsort2 executed by **E2** (real = real performance, pred = predicted performance). k =1 and mp =1 gives a good agreement between predicted and real performance. For this program, **E2** does not reduce k.

Figure 34 shows the performance of sort executed on **E2** (real = real performance, pred(k=1) = predicted performance for k = 1, pred(k=0.5) = predicted performance for k = 0.5). k =0.50 and mp =435 gives the best prediction. Here, the improved locking strategy of **E2** has reduced the value of k.

Finally, let us look at the most advanced of the tested locking strategies, that of implementation **E2c.** Figure 35 shows the performance of qsort2 on **E2c** (real = real performance, pred(k=1) = predicted performance for k = 1, pred(k=0.65) = predicted performance for k = 0.65). k =0.65 and mp =1 gives the best agreement. The value of k is lower than for the other implementations.

Figure 36 shows the performance of sort on **E2c** (real = real performance, pred(k=1) = predicted performance for k = 1, pred(k=0.45) = predicted performance for k = 0.45). k =0.45 and mp =435 gives the best agreement between real and predicted performance. The value of k is lower than for the other implementations.

To sum up, a simple performance model for our machine was derived that takes into consideration the extra work of manipulating global control information at choice points, contention for access to global control information, the degree of parallelism of the program, and the locking strategy. This model represents the behaviour of the implementations of our machine quite accurately. The effective mean degree of parallelism was found to be less than the value listed in [4] in some cases. The effects of indexing, of different distances between nodes of the search tree, of scheduling algorithms, and of the costs of backtracking and recomputing to get a new job, affect the effective mean degree of parallelism. The fraction of the time spent accessing global control information that the lock is locked (k) was reduced from **Old** to **E2** , thanks to the more parallel locking strategy of **E2.** The locking strategy of **E2c** gave the largest reduction of k. Figure 37 gives the values of k and mp that give the best predictions for qsort2 and sort, for the implementations **Old, E2** and **E2c.** The figure also gives the values of the mean degree of parallelism listed in [4], and the measured values of wseq/w.

5. A method for nonshared memory multiprocessors

We have outlined a method for OR parallel execution on nonshared memory multiprocessors, based on the same principle as the model for shared memory multiprocessors described above. This method combines the history path idea from [1] with the multisequential machine idea [2].

Assume a system with a number of processing elements (PE) connected by some medium that provides communications from every PE to all other PEs. Every PE has a local memory for holding its environment and a copy of the program. Every PE is a WAM with an additional HP-stack.

The basic principles of the method are:

1. All PEs in the system start simultaneously processing the top-level query.

2. Every PE independently selects a part of the search tree for processing exactly the same as in [2].

3. Every PE maintains its current path from the top-level query to the current branch in its History-Path stack.

4. When a PE p has completed processing its part of the tree and there is another PE q with an unprocessed job, q sends a path to p specifying an unprocessed branch. The PE p compares the received path from q with its History-Path stack. In general, there is a common part starting at the root. The PE p backtracks to the common part, then it processes the received uncommon part.

Notice that PEs communicate only information specifying a path and not a computation state. Communications occur only when there are idle PEs and busy PEs with unprocessed jobs. The required communication capacity is not high. The performance of every PE is almost as the WAM.

6. Conclusions and future work

We have presented a method for OR parallel execution of Horn clause programs on a combined local and shared memory multiprocessor system. The shared memory contains only control information that guides processors requesting a job to independently construct the required environment, in its local memory, to get a new job from another processor. This reduces the traffic to the shared memory and allows each processor to process its job with high performance. Each processor is almost the same as WAM. The extra overhead in comparison with the sequential WAM is the overhead of maintaining the shared control information.

The model has been implemented on a Sequent Balance 8000 [10], using an existing sequential Prolog system (SICStus Prolog [4]).

Two experiments have been made. The first experiment investigates the effects on the performance of different strategies for selecting a processor to follow, when a simple locking strategy is used. The second experiment investigates the effects on the performance of different strategies for locking shared control information.

The result of the first experiment is that a more balanced selection strategy gives better performance. The result of the second experiment is that the more parallel locking strategies improve the performance for large numbers of processors, but give worse performance when the number of processors is small.

A simple performance model for our machine was derived that takes into consideration the extra work of manipulating global control information at choice points, contention for access to global control information, the degree of parallelism of the program, and the locking strategy. This model represents the behaviour of the implementations of our machine quite accurately.

Investigating a general locking strategy that has one lock on each shared choice point, as in Aurora [6], is our next goal. We believe that with such a locking strategy

and with a balanced selection strategy, the model will give significant speed-ups on a large number of processors for programs with sufficient parallelism. Using an Aurora-like scheduler, cut and ordered output as in standard Prolog could be supported. This is because the topology of shared choice points is known to the scheduler.

A general model for nonshared memory architectures, for a large system, based on the same principle is outlined. The general model is similar to Shapiro's model [7] and Clocksin's model [5]. The three models are developed independently.

An interesting experiment we are going to look at in the near future is to distribute History Path information over choice points and to use the Aurora scheduler, using recomputation instead of sharing environments, and compare the results with the aurora results, to come to the conclusion of the usefulness of the recomputation methods.

We believe that the main reason for the unsatisfactory speed-ups in our two experiments is that we have used too restrictive locking strategies, using locking even when a processor works on a local job.

7. Acknowledgements

The research reported in this paper has partially been done at the Department of Telecommunication and Computer Systems of the Royal Institute of Technology in Stockholm. We would like to thank Professor Lars-Erik Thorelli for providing the environment that enabled us to do this work. Also thanks to Rassul Ayani and Handong Wu for many valuable comments.

References

[1] Ali, K.A.M. OR parallel execution of Horn clause programs based on WAM and shared control information. Research report R 88010, Nov 1986. Swedish Institute of Computer Science.

[2] Ali, K.A.M. OR parallel execution of Prolog on a multi-sequential machine. International Journal of Parallel Programming, Vol. 15, No. 3, pp. 189-214 (June 1986).

[3] Carlsson, M. SICStus preliminary specification. Draft, Dec. 29, 1986. Swedish Institute of Computer Science.

[4] Ciepielewski, A., Hausman, B., and Haridi, S. Initial evaluation of a virtual machine for OR-parallel execution of logic programs. Dept of Computer Systems, Royal Institute of Technology, Stockholm, Sweden, 1986.

[5] Clocksin, W.F., and Alshawi, H., 1986. A method for efficiently executing Horn clause programs using multiple processors. Technical Report, Computer Laboratory, University of Cambridge.

[6] Lusk, E., Warren, D.H.D., Haridi, S., Butler, R., Calderwood, A., Disz, T., Olson, R., Overbeek, R., Stevens, R., Szeredi, P., Brand, P., Carlsson, M., Ciepielewski, A., Hausman, B. The Aurora Or-Parallel Prolog System. Proceedings of FGCS'88, Tokyo, Japan, 1988.

[7] Shapiro, E. An OR parallel execution algorithm for Prolog and its FCP implementation, 1987 International Conference on Logic Programming, Melbourne, Australia, May 1987.

[8] Warren, D.H.D. An abstract Prolog instruction set. Technical Note 309, October 1983. SRI International.

[9] Wong, M. Initial evaluation of an OR parallel execution model for Horn clause programs. TRITA-TCS-8803B. Department of Telecommunication and Computer Systems, Royal Institute of Technology, Stockholm, Sweden, 1988.

[10] Wong, M. Second evaluation of an OR parallel execution model for Horn clause programs. TRITA-TCS-8807. Department of Telecommunication and Computer Systems, Royal Institute of Technology, Stockholm, Sweden, 1988.

[11] Balance 8000 system technical summary. MAN-0110-00. December 12, 1985. Sequent Computer Systems, Inc.

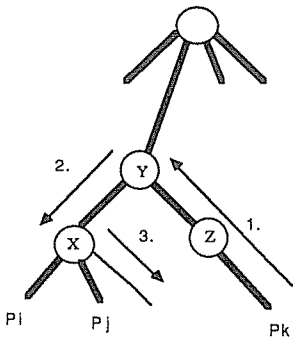


Figure 1. The model

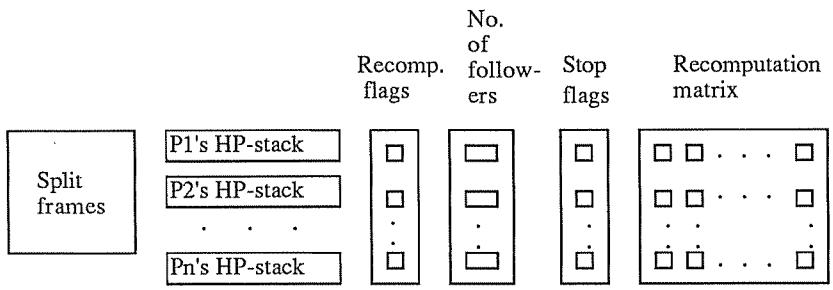


Figure 2. The global data areas of the model

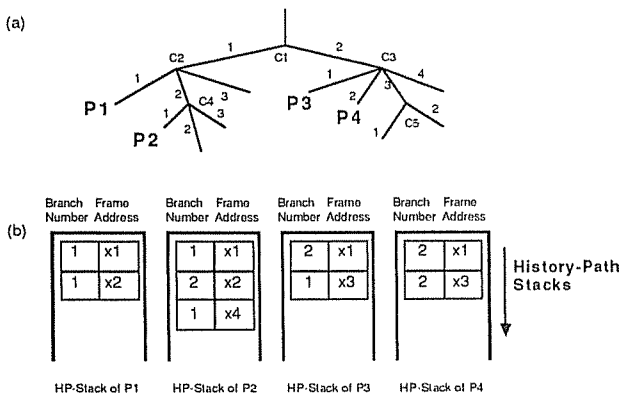


Figure 3: (a) Numbering of branches (b) History-Path stacks

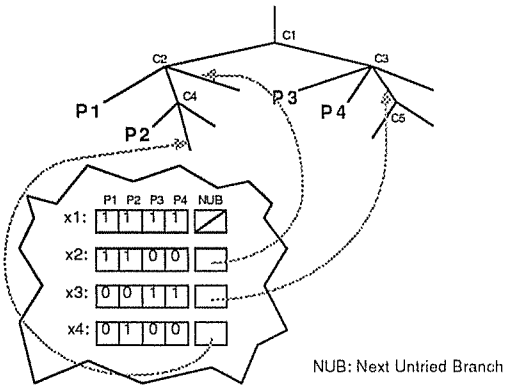


Figure 4: Split frames associated with the active choice points

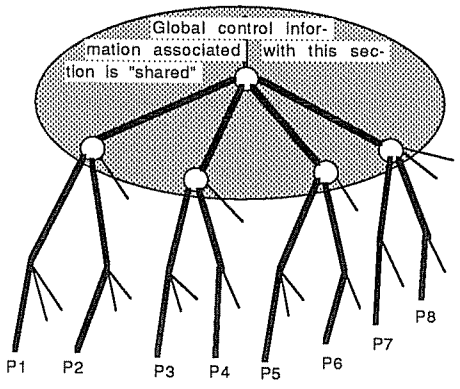


Figure 5. Section of a search tree associated with "shared" control information

```
?- permute([a,b,c],Y).

permute([],[]).
permute([X|Y],[U|V]) :- delete(U,[X|Y],Z),permute(Z,V).

delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|W]) :- delete(X,Z,W).
```

Figure 6. Code for permute

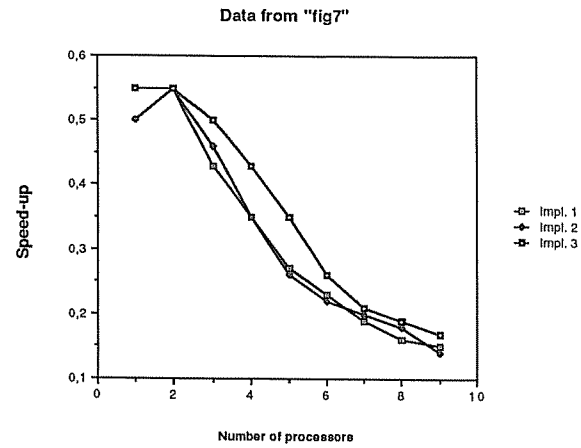


Figure 7. Performance of permute

```
?- qsort2([4,5,2,1,3,6],A).

qsort2(Unsorted,Sorted) :- qsort(Unsorted,d(Sorted,[])).

qsort([],d(Rest,Rest)).
qsort([H|Unsorted],d(Sorted,Rest)) :-
        split(H,Unsorted,Smaller,Larger),
        qsort(Smaller,d(Sorted,[H|Sorted1])),
        qsort(Larger,d(Sorted1,Rest)).

split(_,[],[],[]).
split(X,[H|T1],[H|U1],U2) :- H=<X, split(X,T1,U1,U2).
split(X,[H|T1],U1,[H|U2]) :- H>X, split(X,T1,U1,U2).
```

Figure 8. Code for qsort2

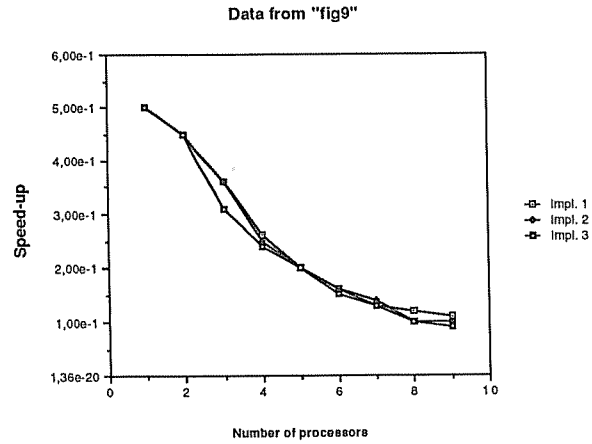


Figure 9. Performance of qsort2

```
?- mutation(A).

mutation(X) :- animal(N1), animal(N2), create(N1,N2,X).

create(N1,N2,New) :- append(Y1,Y2,N1), Y1\== [],
        append(Y2,Z2,N2), Y2\== [], append(Y1,N2,New).

animal([a,l,l,i,g,a,t,o,r]).
animal([t,o,r,t,u,e]).
animal([c,a,r,i,b,o,u]).
animal([o,u,r,s]).
animal([c,h,e,v,a,l]).
animal([v,a,c,h,e]).
animal([l,a,p,i,n]).

append([],X,X).
append([H|T],X,[H|Y]) :- append(T,X,Y).
```

Figure 10. Code for mutation

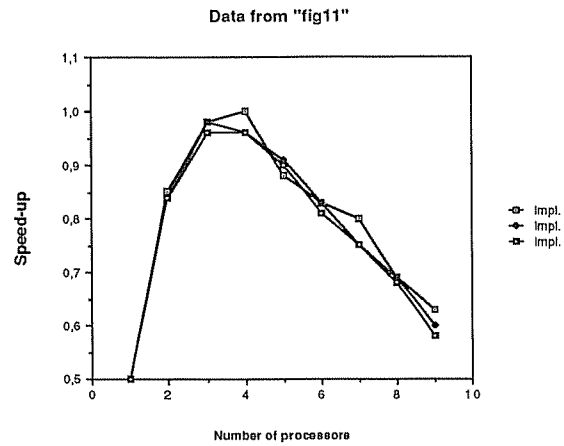


Figure 11. Performance of mutation

```
?- numbers(A,B,C,D,E,F,G).

numbers(1,B,C,D,E,F,G) :- List = [1,2,3,4,5,6,7],
        delete(1,List,L1),
        delete(G,L1,L2),
        delete(C,L2,L3),
        delete(E,L3,L4),
        Ag is 1+G, Ag is C+E,
        delete(F,L4,L5),
        delete(B,L5,L6),
        Ce is C+E, Ce is F+B,
        delete(D,L6,L7).

delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|W]) :- delete(X,Z,W).
```

Figure 12. Code for numbers

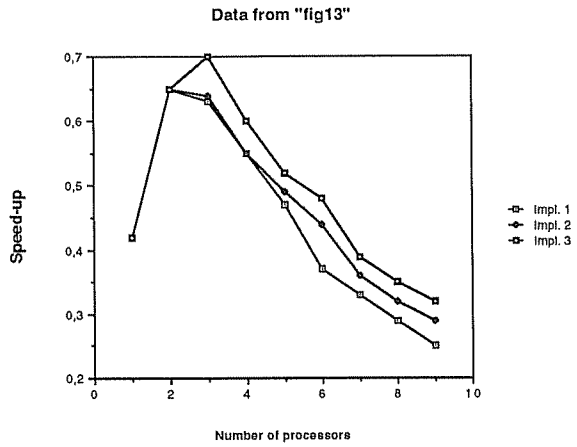


Figure 13. Performance of numbers

```
?- sorti([4,5,2,1,3,6],A).

sorti(X,Y) :- permute(X,Y), sorted(Y).

sorted([X]).
sorted([X,Y|Z]) :- X =< Y, sorted([Y|Z]).

permute([],[]).
permute([X|Y],[U|V]) :- delete(U,[X|Y],Z),
                        permute(Z,V).
delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|W]) :- delete(X,Z,W).
```

Figure 14. Code for sort

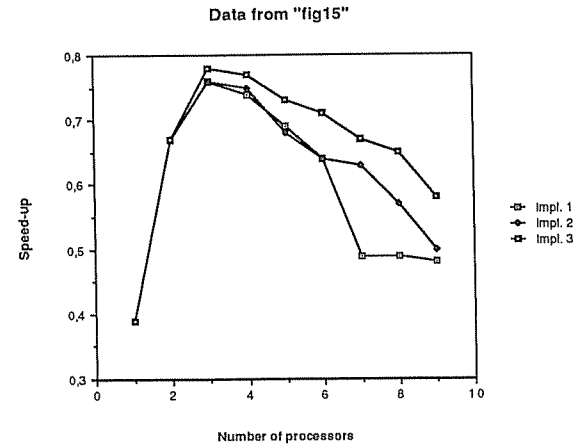


Figure 15. Performance of sort

```
fast10000:
        ?- p.

        p :- count(10000).
             .
             .          (p/0 has 90 identical clauses)
             .
        p :- count(10000).

        count(0).
        count(N) :- N>0, N1 is N-1, count(N1).

fast1000:
        as fast10000, except that count(1000) is substituted
        for count(10000).

fast100,fast75,fast50,fast25,fast10:
        defined analogously.
```

Figure 16. Code for fast

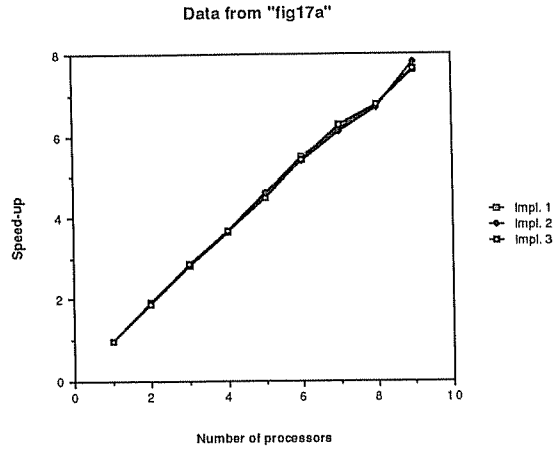


Figure 17a. Performance of fast10000

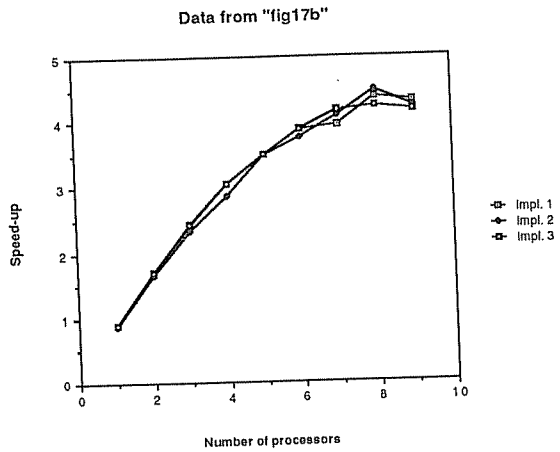


Figure 17b. Performance of fast100

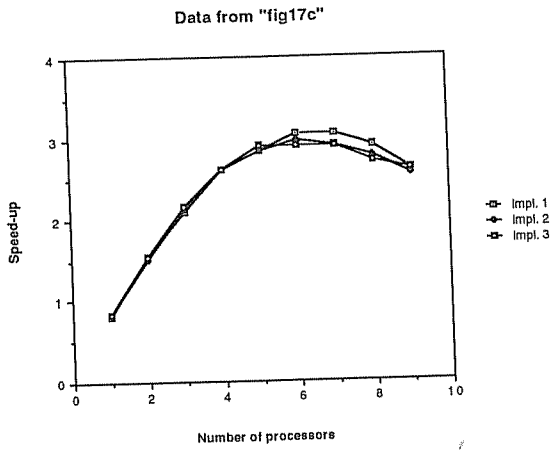


Figure 17c. Performance of fast50

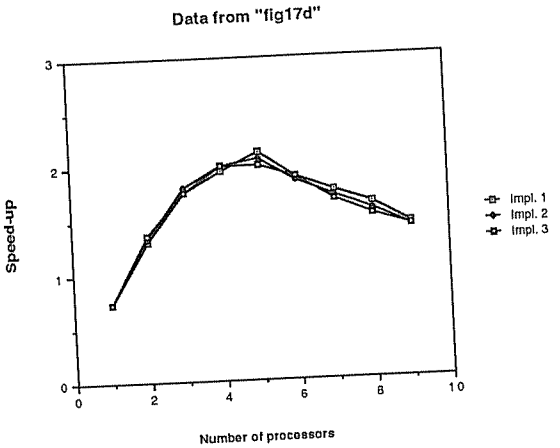


Figure 17d. Performance of fast25

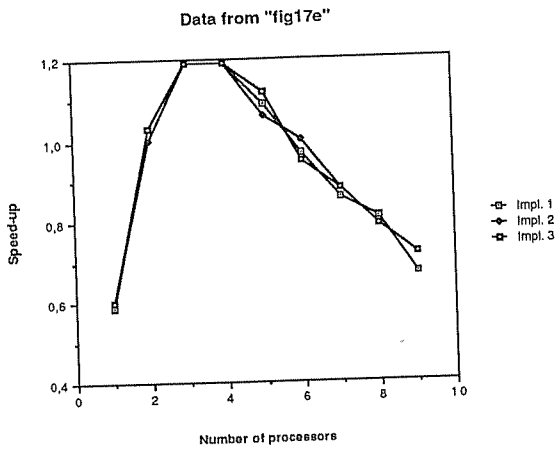


Figure 17e. Performance of fast10

```
?- p.

p :- p(100).

p(N) :- N>0, count(10000).
p(N) :- N>0, N1 is N-1, p(N1).
p(0).

count(0).
count(N) :- N>0, N1 is N-1, count(N1).
```

Figure 18. Code for select

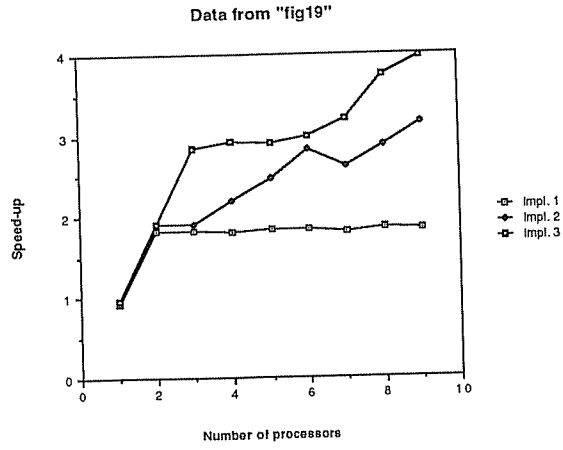


Figure 19. Performance of select

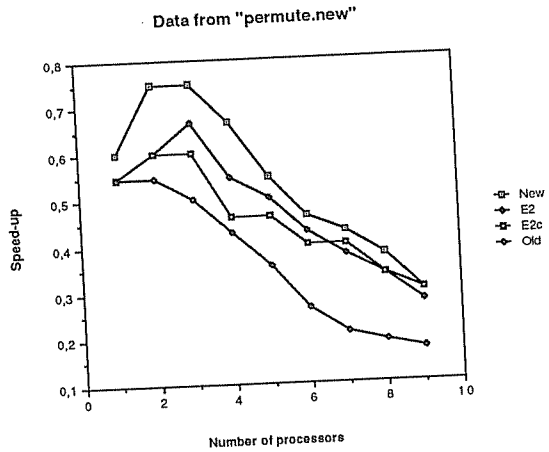


Figure 21. Performance of permute

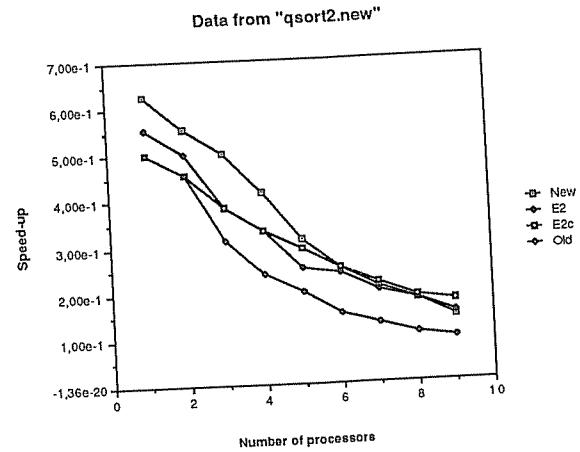


Figure 22. Performance of qsort2

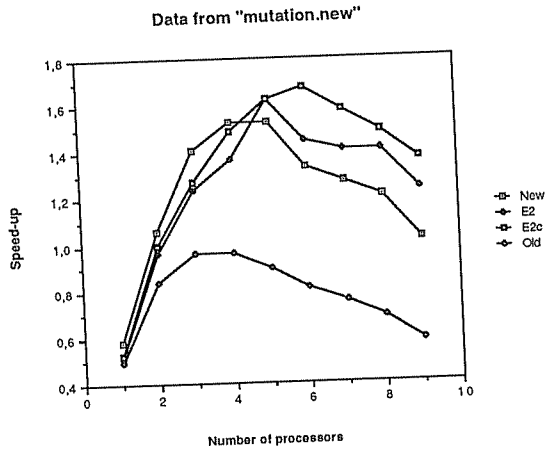


Figure 23. Performance of mutation

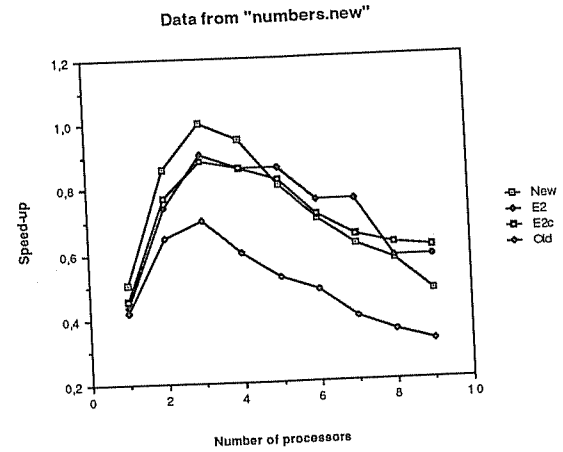


Figure 24. Performance of numbers

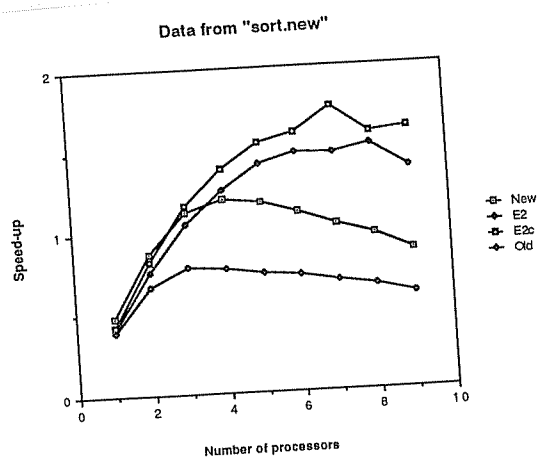


Figure 25. Performance of sort

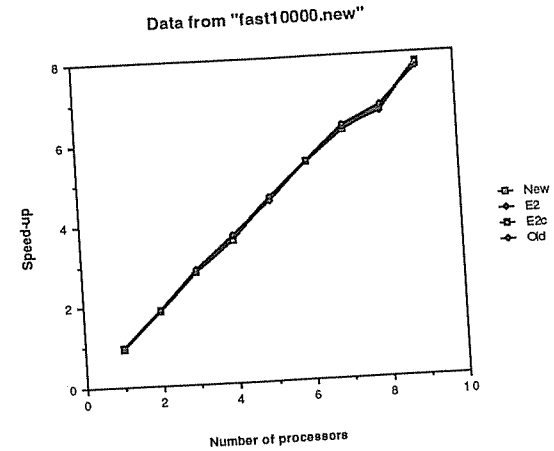


Figure 26a. Performance of fast10000

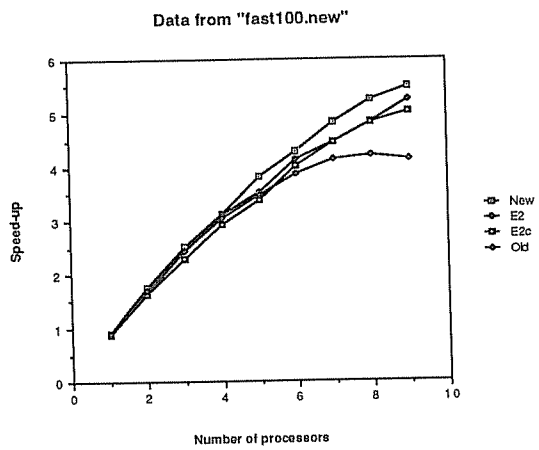


Figure 26b. Performance of fast100

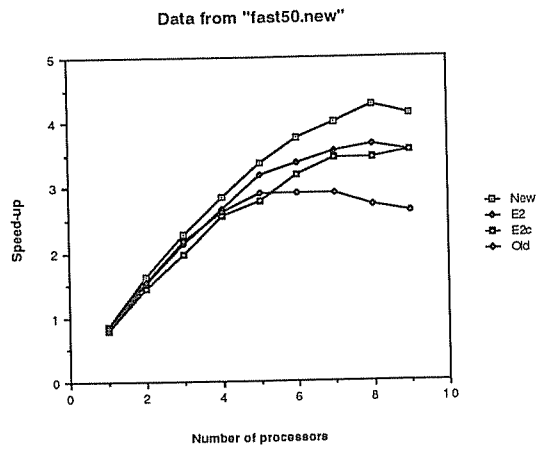


Figure 26c. Performance of fast50

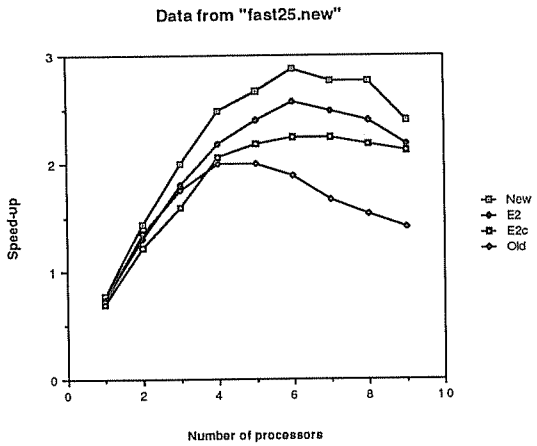


Figure 26d. Performance of fast25

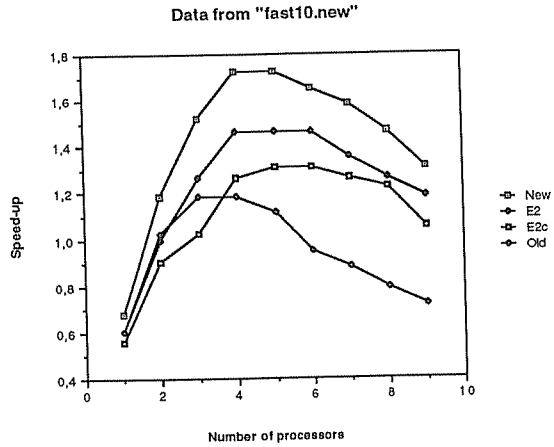


Figure 26e. Performance of fast10

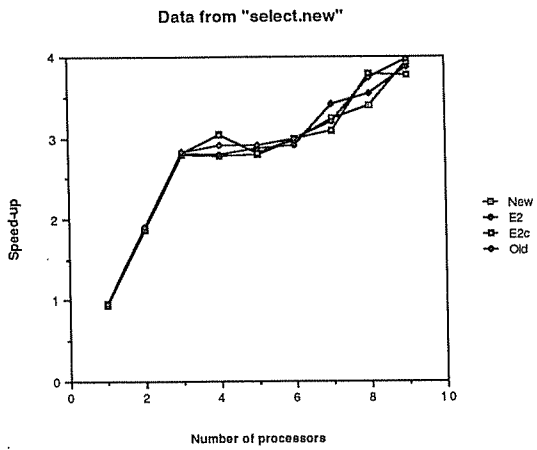


Figure 27. Performance of select

Name	max sp	@	@9	mean par
"Abstract programs:"				
fast10000	(7.8)	9	(7.8)	90
fast100	4.4	8	4.3	90
select	(4.0)	9	(4.0)	100
fast50	3.0	6	2.6	89
fast25	2.1	5	1.4	89
fast10	1.2	3	0.72	87
"Parallel programs:"				
mutation	1.0	4	0.63	78
sort	0.78	3	0.58	435
numbers	0.70	3	0.32	41
"Sequential programs:"				
permute	0.55	1	0.17	8
qsort2	0.50	1	0.11	3

Figure 20. Brief summary of some of the results.

```
?- p.

p :- p(100).

p(N) :- N>0.
p(N) :- N>0, N1 is N-1, p(N1).
p(0).
```

Figure 28. Code for select2

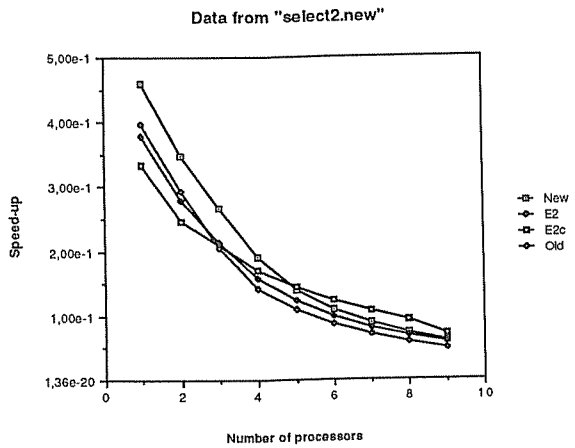


Figure 29. Perormance of select2

Name	max sp	@	impl	break	mean par	E2c@1
"Abstract programs:"						
fast10000	(7.8)	9	---	---	90	0.97
fast100	(5.5)	9	New	9	90	0.87
fast50	4.3	8	New	9	89	0.79
select	(3.9)	9	---	---	100	0.96
fast25	2.9	6	New	9	89	0.70
fast10	1.7	4	New	9	87	0.56
select2	0.46	1	New	4	2	0.33
"Parallel programs:"						
sort	1.8	7	E2c	2	435	0.43
mutation	1.7	6	E2c	4	78	0.53
numbers	1.0	3	New	4	41	0.46
permute	0.75	2	New	8	8	0.55
"Sequential programs:"						
qsort2	0.63	1	New	5	3	0.50

Figure 30. Brief summary of some of the results.

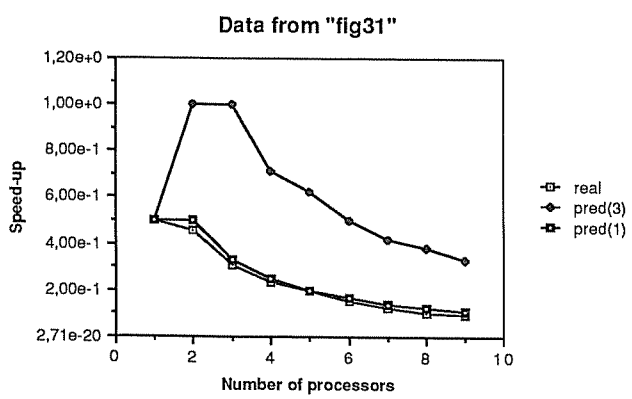


Figure 31. Real and predicted performance of qsort2 on Old

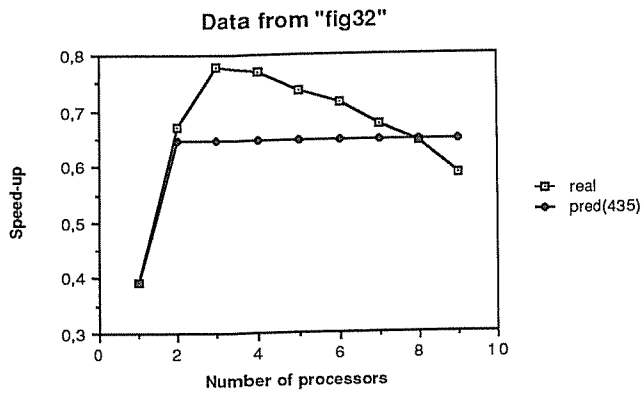
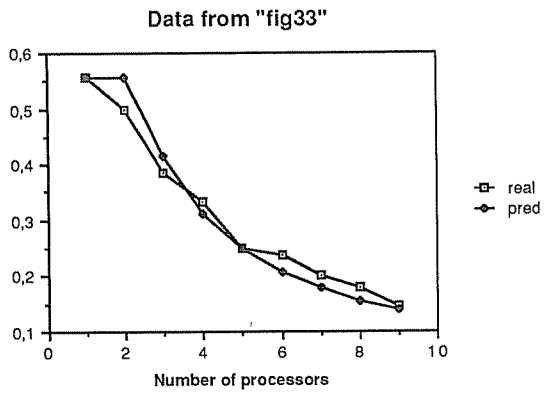


Figure 32. Real and predicted performance of sort on Old



;ure 33. Real and predicted performance of qsort2 on E2

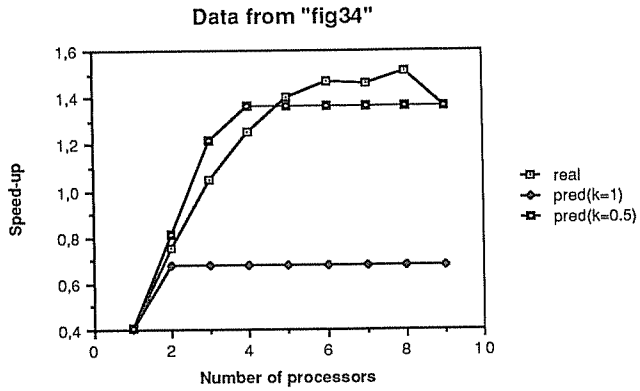
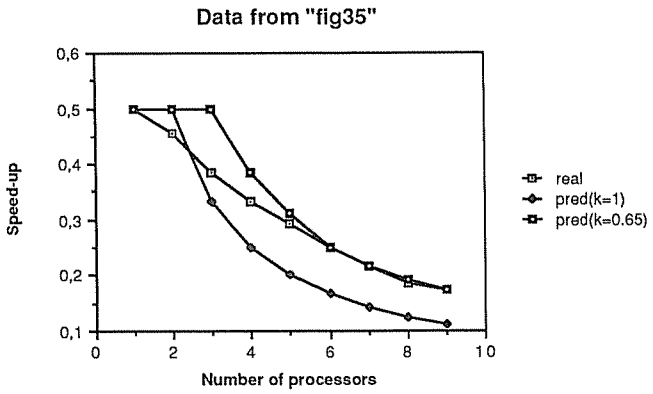


Figure 34. Real and predicted performance of sort on E2



gure 35. Real and predicted performance of qsort2 on E2c

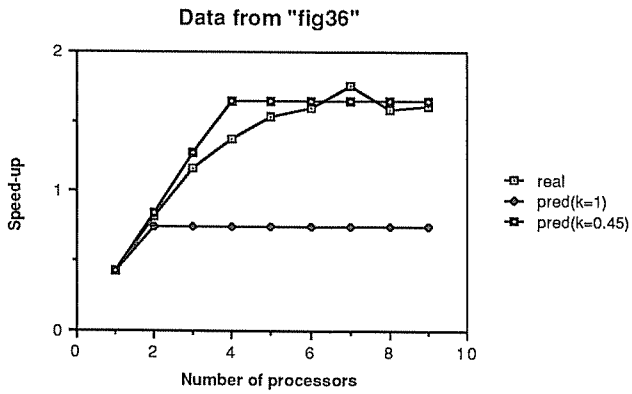


Figure 36. Real and predicted performance of sort on Ec2

		k	mp	mean par	wseq/w
qsort2:	Old	1	1	3	0.50
	E2	1	1	3	0.56
	E2c	0.65	1	3	0.50
sort:	Old	1	>= 9	435	0.39
	E2	0.50	>= 9	435	0.41
	E2c	0.45	>= 9	435	0.43

Figure 37. Parameters of the performance model for different programs and different locking strategies.