Design and Implementation of a Distributed Parallel Programming System

Z. Lin S. Zhou
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A1

W. Li
Department of Computer Science
University of Maryland
College Park, Md 20742

Abstract

A parallel programming system is presented in this paper. The system supports a specification language, parScript, which can conveniently encode a fork-joint graph by which many parallel applications are representable. The system consists of a parScript-to-C compiler, and a runtime kernel. The compiler translates parScript specification directly into C code executable either on a sequential workstation or in a workstation cluster. In this system, a parallel program comprises a parScript specification which advises the system how tasks can be decomposed and coordinated, and a number of C procedures which perform the actual tasks. We study the feasibility of such an approach by building a prototype system both on sequential UNIX workstations and in workstation clusters. Experiments are performed with numerical as well as symbolic processing programs to study the effectiveness of the method. Results shows that programs in parScript are efficient and portable across different architectures. When executed in parallel, many of the programs achieve reasonably good speed-up compared to the corresponding sequential C counterpart.

Keywords: Parallel Programming, Compilation, Parallelism, Scheduling

1 Overview

A parallel programming system is presented in this paper. The system supports a specification language called parScript which overlaps Prolog, and contains additional features such as the SET operator for parallel control. ParScript is capable of describing, in a concise and intuitive way, fork-joint graphs in which a large number of parallel applications are representable. A program in this system consists of a task decomposition specification in parScript and a number of sequential C procedures, or it may be written entirely in parScript as a normal logic program.

Parallel computers are generally considered more difficult to program compared to traditional uniprocessor systems. The process of programming, debugging and performance tuning of a parallel application is more often than not found to be a fun-filled adventure full of surprises, due largely to the primitive and unstable software tools available. It has gradually been realized that this problem may probably be the most important factor preventing parallel computers from being broadly accepted as practical general purpose computing machines. Providing a convenient programming environment for developing portable yet efficient programs remains a big challenge [15]. This motivates the development of the system being presented in the paper.

Most existing parallel programming methods fall into the category in which a runtime library containing parallel control functions is used to supplement the host language such as C (C++) and Fortune [2]. While this may sometimes be the only way to exploit the full potential of the underlying hardware, the approach certainly has limitation: it can easily lead to the interweaving of sequential and parallel control codes, which would be unnecessary for many applications as evidenced by our results shown later in this paper. De-coupling sequential procedures and parallel control code offers the advantages of 1) achieving a program whose parallel control part and sequential procedures can be developed and debugged separately; 2) hidding the lower level message passing and/or synchronization routines from the programmer; 3) making it easier to port the program to other parallel architectures as usually only the parallel control part needs to be modified. With a proper parallel control specification, the modification can be taken care of by a compiler. We will explain the reasons further later in the paper.

We believe that the idea of de-coupling sequential procedures and parallel control code is not new, and it could be implemented in different ways. What we are advocating in this paper is that the model be applied conscientiously, and we propose an experimental system that facilitates the application of the idea.

For the parallel control specification, we opt for a logic oriented approach. This needs some explanations as it does not appear to be an obvious choice.

- a Horn clause program, in conjunction with a SET operator, to be defined in the next section, can describe a fork-joint graph conveniently
- parallelism in a logic program can be extracted effectively as demonstrated by many previous studies [1, 4, 16, 17, 11], due partly to the fact that there is no hidden dependency among alternative clauses in a logic program
- a logic program can be very concise, and sometimes they are declarative as well

However, to satisfy the demand for efficiency, in terms of speed and space, by users who are accustomed to imperative languages, logic programming alone is hardly adequate. The reason for that is a logical language lacks the capability to describe the fine details of certain computation, which may be necessary to attaining satisfactory performance in a general purpose computer [6].

We believe that this limitation can be alleviated. The belief is based on the observation that it is neither necessary nor beneficial to force a logical language to carry out tasks for which procedural languages are known to be more suitable. Sometimes it may just be natural to encode the problem in a procedural manner, sometimes it is necessary to provide detailed control instructions to the machine in order to achieve good performance. This leads to the idea of a hybrid solution, one that maintains the logic orientation at the high level of a program while allowing procedural code at lower levels if desired. It is not clear at this point where to draw the line between the logical and procedural parts of a program. Our conjecture is that such a line exists in many applications where control coding is imperative at the low level procedures. But for the upper level specification, a logical language is sufficient and probably preferred due to its cleanness and possible parallelism. In addition, for the purpose of distributed parallel programming, being a "coarse-grained" language may not necessarily be a disadvantage given the fact that fine-grained tasks could easily overwhelm the task scheduler in a loosely-coupled architecture [12].

We have implemented a prototype system both on sequential workstations, and in workstation clusters connected by the Ethernet. The system includes a parScript-to-C compiler, and a runtime kernel. The compiler can translate parScript rules into a C program for sequential machines. It can also translate the rules into a C program executable in a workstation cluster, with necessary parallel control instructions inserted automatically. The runtime kernel provides support for the execution of nondeterministic rules in the specification. The kernel also serves as a parallel task coordinator when the system is running in parallel. The task coordinator is *programmable*, in the sense that it executes the specification written in parScript which essentially tells the system how tasks can be decomposed and how they are related to each other. There is also a built-in task scheduler in the kernel which is responsible for load balancing among cooperating processors.

The PVM [7] package is used to provide communication support for the network version.

The system is operational on UNIX workstations and workstation clusters. The following are highlights of our accomplishments:

- As a stand-alone logic programming system, sequential parScript programs (Horn clause programs) run faster than the compiled execution with Sicstus Prolog 0.6 #18, a well-known non-commercial implementation of Prolog.
- In a DEC5000 workstation cluster, good speed-up is achieved for medium- to coarse-grained programs, including symbolic processing as well as numerical computation programs such as LU-decomposition and matrix multiplication.

The paper is organized as following: section 2 presents background materials; section 3 describes the parallel control mechanism; section 4 briefly outlines the implementation; section 5 gives experimental results; and section 6 concludes our work.

2 Background

2.1 Logic Programs

We consider a logic program to be a set of Horn clauses written as $H :- B_1, B_2, ..., B_n$ where H, the head of the clause, is a positive literal and the B_i's, the body of the clause, are either positive or negated literals (possibly empty). The intuitive interpretation of such a clause, or rule, is that if all B_i's are solved then H is considered solved. Evaluation of the program starts with a query written as $:- Q$, and uses resolution [13] to derive an empty clause should one exist. There may be multiple selections of rules at each resolution step, provided by rules with a matching head. All solutions to the query can be found by exhausting every possible alternative in the program. The answer to the query can be *true* or *false*. But at most of the time, the interesting solution to the query is the bindings on the variables in Q.

A practical logic programming language, Prolog for instance, usually contains symbols, built-in functions, and language constructs other than just Horn clauses. We are not able to give a full description of the Prolog language here. Interested reader may find out more in [18]. Only Horn clause programs are used in this paper so that the examples are understandable even without full knowledge about the Prolog language.

We note that the execution of a logic program can be viewed as constructing a search tree [13]: with branches corresponding to multiple rules selected for expansion at one resolution step. In the next subsection we will define a SET operator which joins results from different branches in the tree. Thus, the search tree is generalized to a graph, i.e., with arcs inciding into some nodes to represent joint operations dictated by the SET operator.

2.2 The SET Operator

A *set* operator is used in Parlog [5] to compute all solutions to a predicate. The *bagof* operator in Prolog has similar function. We implemented a similar operator, SET, whose format is as follows:

$$SET(finalset, template, goal(input, template))$$

The SET operator evaluates the predicate *goal* with input variable *input*, and then returns all solutions with regard to the variable *template* in the variable $finalset$.

In addition to adhering to the above semantics, we make explicit the following:

If there are more than one processes computing the set, every process will wait at a barrier associated with the set after it completes its share of the computation, results from different processes are joined, and the final set is made available to all the processes.

Here the term "join" means combining the results from individual processes in such a way that $finalset$ resembles the one obtained from a sequential execution. In our implementation, however, ordering of elements in the $finalset$ may not be preserved in the same way as in sequential execution, due to consideration of efficient implementation.

The proposed condition forces the SET operator to behave similar to that of a fork-join pair in procedural languages, with all the communication details encapsulated. Note that the SET operator still computes all solutions to the predicate that are inside the set. Therefore, the logical reading of a program with SET operators does not change in any way.

Figure 1 is an example showing how the SET operator is used in a merge program to achieve parallelism in the *msort* predicate. Notice that SET is invoked recursively.

The presence of SET operators in a program does not, in principle, alter the way we view the execution of a program: we still consider it a process of constructing a search tree. The tree now has two types of nodes: regular Or-nodes and the SET nodes. The new type of SET nodes are defined as following:

```
msort([X],[X]).
msort(Xs,Ys):-    split(Xs,Lts,Bgs),
                  SET([(1,Ls),(2,Bs)],LL,p_msort((Lts,Bgs),LL)),
                  merge(Ls,Bs,Ys).
p_msort((L,B),(1,LL)) :- msort(L,LL).
p_msort((L,B),(2,LL)) :- msort(B,LL).
```

Figure 1: A merge sort program

Definition 2.2.1 SET Node: a SET node in a search tree represents the whole subtree generated by the predicate being computed in the set. A SET node can occur in any place in the tree just as if SET were a predicate.

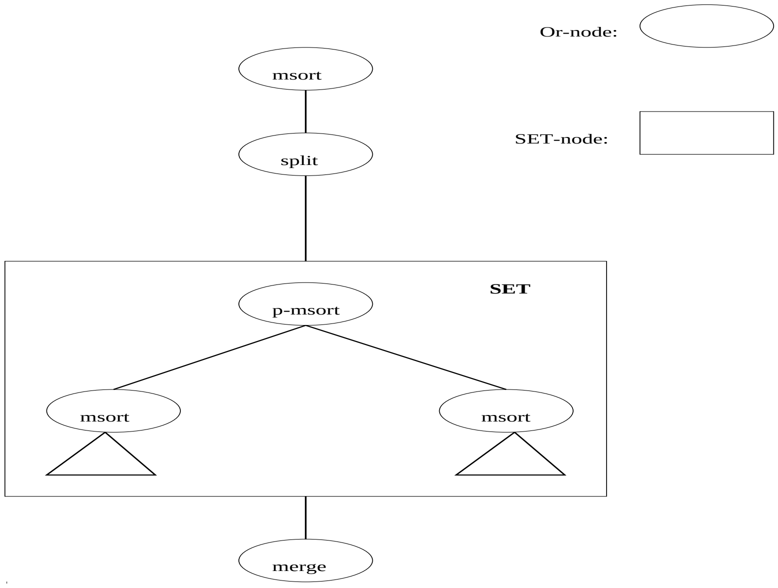


Figure 2: The search tree for the merge sort program.

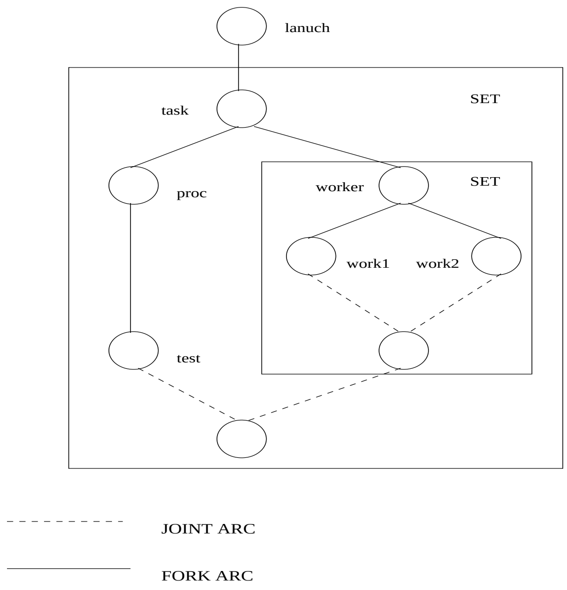
Figure 2 depicts the search tree generated by the *msort* program. A search tree that contains SET nodes will be refered to as an OS-tree, in order to distinguish it from an Or-tree.

With the SET operator, one can easily encode parallel processes that are often represented as a fork-join graph. The coding is very simple, as illustrated in Figure 3. This allows parallel computation that observes the fork-join model to be conveniently written without exposing the programmer to the tedious communication protocols at the join phase[1]. On the other hand, the SET operator can still be understood as computing all solutions to a given goal, should one choose not to use the notion of (concurrent) processes. Programs written with the SET operator can be developed and verified on a sequential machine to eliminate logical errors before having them executed in parallel.

2.3 ParScript Programs

ParScript is a language based on Horn clause syntax and the SET operator defined above. But unlike Prolog, facts in a parScript program, i.e., those clauses with an empty body, can be defined as external C procedures. These facts are expressed in the format: predicate_name(X,Y) :- foreign_c. The parScript specification is translated into a

[1]Deciding which processes to join with is not a trivial problem. Interested readers are encouraged to trace a recursive program involving the SET operator.




```
:- launch.

lanuch :- SET(Y,X,task(X)).
task(X) :- proc(X),test(X).
task(X) :- SET(X,Z,worker(Z)).
worker(X) :- work1(X).
worker(X) :- work2(X).
```

Figure 3: A sample fork-join graph and a parScript program describing the graph.

C program by the compiler. One can link the target C program with any external C modules using existing linking facilities.

A parallel program thus consists of a task decomposition specification in parScript, and sequential C codes that carry out the actual work. The parScript specification may be considered a program to the built-in task coordinator which is responsible for task creation, scheduling, and coordination. Communication among tasks is implicitly specified in the parScript program. The sequential C codes usually do not directly interact (i.e., sharing variables or passing messages) with each other, but it is not enforced by the system. The advantage of this is that it leaves open opportunities for certain performance optimization that may otherwise be impossible. We do not, however, advocate such use (or abuse) as it can easily destroy the readability and portability of a program, the objective parScript strives to achieve. Figure 4 depicts the bi-level concept of a program. The picture shows a non-recursive merge sorting phase (at the bottom level of the recursion) taken from the merge sort program shown in the previous subsection.

3 Parallel Execution

An interesting problem in executing parScript programs in parallel is how to decompose and schedule tasks. We shall make clear that in our system, the parScript program only tells how tasks *can* be decomposed. The suggestion is taken as advice by system. How they *are* actually decomposed is decided by the system at runtime. The decision is based on the number of processors available, and an input parameter provided by the user when the program is invoked. The parameter indicates the number of processes to run on each processor. At present, we do not assume knowing the size of the tasks in a program or any part of it before hand. But there is an implicit assumption that the degree of parallelism, i.e., the number of tasks that can be executed simultaneously, as specified in the parScript program, is ample relative to the number of processors being used. The objective of the scheduler is to balance the need for controlling task granularity and balancing load distribution so as to minimize response time. Below, we describe the scheme used the current implementation.

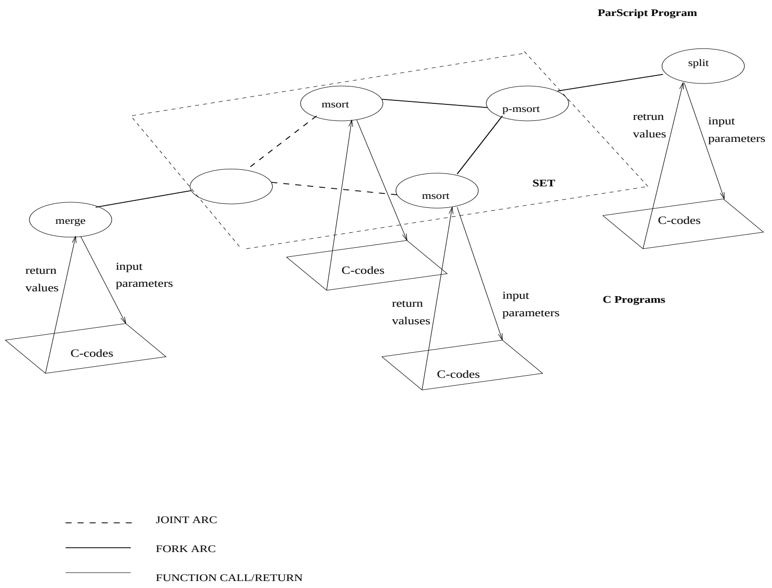


Figure 4: The bi-level view of parallel sorting program.

3.1 Self-Organizing Task Scheduling

Task scheduling in a parallel system usually involves moving computation from one processor to another in order to balance the workload. We call this process dynamic task migration. Dynamic task migration incurs overhead, which may offset the gain from maintaining a balanced load distribution. In a loosely-coupled system such as a cluster of workstations, the performance bottleneck is likely to be the interconnection network. This dictates that only coarse-grained tasks, i.e., those with a relatively high ratio of computation time to communication time, can effectively utilize the system's CPU power while avoiding the communication bottleneck. In other words, dynamic task migration must not occur frequently.

Self-organizing scheduling (SOS), proposed in [10], has been shown to be effective in attaining a relatively large computation granule for a given program, compared to existing scheduling schemes. The basic idea of SOS is to allow each process[2] to decide, independently, its share of a task belonging to a group of processes by following an universal task distribution rule. No shared work queue or interprocessor communication is required by SOS.

Consider the simple case of a search tree (an Or-tree) that is implicitly defined by a program and a given query. Finding all solutions to the query is viewed as an exhaustive search of the tree. Suppose there are N processes participating in the search. SOS works as follows:

1. every process starts with the original goal, i.e., the root of the tree;

2. at each node, a processor expands all children of the node and claims those belonging to it according to an universal rule;

3. a process repeats step 2, until all alternative branches it claims are explored.

The universal task distribution rule is a mapping between a set of processes and a set of tree nodes. It is a function F: process id $\rightarrow$ a set of nodes. A tree node is said to be claimed by a process when it is considered a part of the task for the process.

[2] In previous reports the term "processor" was used. Recent studies suggested that an additional layer of mapping, one that maps virtual processes to physical processors, can yield even better result. This leads to the use of "process" instead of "processor" in our present discussion. Also, we note that a process is not necessarily an operating system process. In the current implementation, they are more like light-weight threads, manipulated solely by our system.

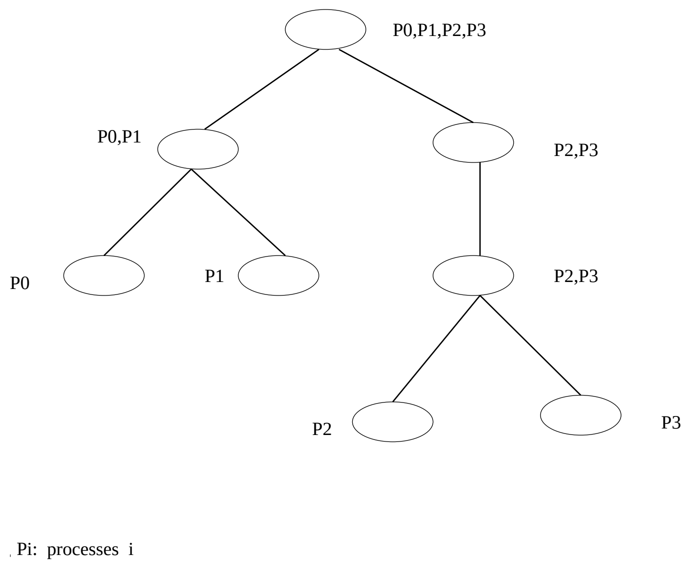


Figure 5: Example of SOS: four processes are cooperating to explore the search tree. Process IDs are listed next to the node visited by the processes.

Step 2 in SOS can be repeated because of the following: at the very beginning every process knows that there are N processes visiting the root, and, each process can compute the number of children, k, of the root. The mapping from N processes to the k children is governed by the function F, which is known to all processes. Therefore, a process can determine:

1. which children shall be allocated by itself for further processing, and

2. which children shall be allocated by any other process.

In other words, a process is fully aware of which other processes (if any) will visit the same child nodes it claims. Furthermore, this information is obtained without any dialogue with other processes. Note that a process is uniquely identified by a process ID throughout the execution. Figure 5 illustrates the idea of SOS. The task distribution rule used in this example evenly divides the children of a tree node among processes visiting the node.

SOS can be extended for a tree containing SET nodes. As stated in section 2, the presence of SET operators in a program do not alter the way we view the execution of a program: we still consider it a process of constructing a search tree. The only difference is that the tree now contains SET nodes. When multiple processes visit a SET node, they simply apply SOS to the tree embeded inside the SET node. Note that these processes are synchronized at a barrier before they leave the SET node. There is overhead incured running the barrier. Optimization such as overlapping the waiting time with other computation is possible. Several different strategies are being evaluated, we will not discuss them in this paper.

With SOS, load balancing is achieved by:

1. restructuring the program to obtain a better balanced search tree;

2. assigning processes to tasks with a statistically even task distribution rule. In our case, we use the equi-partitioning rule which was shown [10] to be optimal under the condition that nothing is known about the tasks before hand, and

3. allowing a greater number of processes than that of physical processors so that the load balancing problem can partially be taken care of by carefully matching processes to processors.

The first two approaches were studied in [11]; results indicate that many generate-and-test programs can indeed be restructured to obtain a better balanced tree, hence achieving a better balanced load distribution. The equi-partitioning rule was proven valid for tasks whose size follows certain probabilistic distributions. The intuition for the third approach is as follows: by assigning an equally large number of processes onto every processor, the weak law of large numbers [19] will take effect evening out the workload on each processor. In addition, some mapping function allows a processor to execute processes that are assigned to tasks (represented by one or more subtrees in the search tree) at different parts of the search space, realizing the effect of sampling the shape of the search tree. One of these functions, for example, is the division function: Processor_id = Process_id *modulo* Processor Number. This function is used in the current implementation. To illustrate how the mapping works, consider the example in figure 5. Suppose we have two processors. By the above mapping function, the first processor (id=0) will execute processes P_0 and P_2, the second processor (id=1) will execute P_1 and P_3. Despite the uneven shape of the tree, each processor still obtains a more or less even share of the workload.

One may observe that there is redundant computation involved with SOS. In fact, the top portion of the tree is computed by many processes. As shown in [11], the more the redundant computation is, the better the scheme can adapt to the shape of the search tree, or in other words, the more robust SOS will be in terms of balancing workload. The issue with SOS is thus the trade-off between *redundant computation* vs. *load balancing*, as opposed to *communication overhead* vs. *load balancing* as in conventional schemes. Given the fact that the difference in speed between CPUs and communication links is increasing (at least for the near future), we believe that the issue of redundant computation vs. load balancing is probably an easier one to deal with. For some architectures, dynamic task migrations may simply be too costly to perform. Such architectures include workstation clusters, in which the interconnection network is slow compared to the CPU, and massively parallel multiprocessors, in which a large number of processors are involved and the scheduling algorithm may take time proportional to the number of processors in the system to decide from where and in what way to migrate a task. For these systems, we believe that SOS is a better alternative, despite the redundant computation involved. On the other hand, dynamic tasks migration would probably have advantages in a tightly-coupled architecture. Logic programming systems using dynamic scheduling methods can be found in [1, 14, 17].

4 Implementation

The system consists of two major components, a compiler and a runtime kernel. Figure 6 depicts the compilation process. A brief description of these two components is given below.

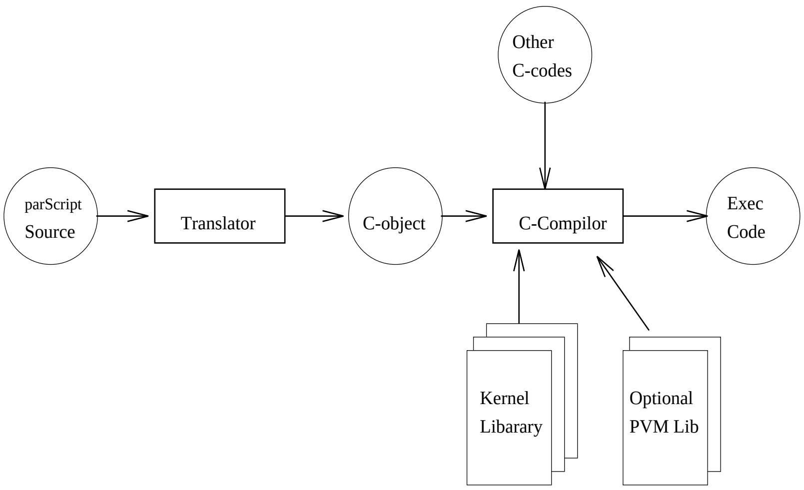


Figure 6: Program Compilation.

4.1 The Compiler

The input to the compiler is a parScript program, and the original goal one wants to solve. The output of the compiler is a set of C procedures. For every predicate, the compiler produces a procedure called OR procedure. An OR procedure selects one clause for the predicate upon an invocation. A standard OR procedure is of the format:

```
predicate-name(argument-array, body-conj)
{
    record-environment();
    switch(clauseNumber)
    {...
    case i: ret-val=predicate-clause-i(argument-array, body-conj);
            if(ret-val ==SUCCESS) return SUCCESS;
            else {
                    recover-environment();
                    next-clause(&clauseNumber);
                 }
    ...
    }
    return FAILURE;
}
```

Record-environment() records the state of the system (e.g., saves registers, stack pointers, etc.). Recover-environment() resets the stack pointers and undoes the variable bindings which occured in predicate-clause-i(). Predicate-clause-i() is the procedure for the ith clause of this predicate. For every alternative clause the compiler produces a procedure, called an AND procedure, which does unification and creates the body conjunction of the clause. A body conjunction is a list of procedures that corresponds to the body of the clause. The format of a standard AND procedure is:

```
predicate-clause-i(args, body-conj)
{
    ...
    allocate-ith-head-args(headarg[i]);
    if(unify(headarg[i], args[i]) == FAILURE) return FAILURE;
    ...
    allocate-body-conjunction(body-conj);
    return SUCCESS;
}
```

When the head unification succeeds, allocate-body-conjunction() generates the clause's body conjunction. The address of the conjunction is put in body-conj. The AND procedure then returns SUCCESS. The OR procedure that invokes this AND procedure also return SUCCESS. Control is thus turned back to the runtime driver, which inserts the body-conj to a candidate procedure pool. Then, the driver selects the next literal to invoke. For deterministic predicates, the allocate-body-conjunction() function is replaced by:

```
    ...
    allocate-args-for-ith-literal(args);
    if (call-ith-literal(args) == FAILURE) return FAILURE;
    ...
```

Deterministic predicates are executed on-line: call-ith-literal() invokes the procedure for the ith (deterministic) literal in the clause body. Note that if all the body conjuncts are deterministic the procedure's body is a serial of C function calls, that is, there would be no involvement of the kernel.

The query of a program is translated into a special procedure with the name QUERY, which creates an initial process conjunction in the same way that we create the body conjunction of a clause. The QUERY procedure is the entry point of the parScript program.

4.2 The Runtime Kernel

The runtime kernel includes the following components: a search driver (task coordinator), unification primitives, memory management primitives, and system built-in predicates.

The search driver obtains the initial conjunction by calling the QUERY procedure which returns the initial query conjunction. The driver then repeatly executes the following:

1. push the new conjunction into the work stack;
2. select a literal to invoke;
3. if the literal returns FAILURE then backtrack, otherwise repeat step 1 or 2 depending on whether or not a new body conjunction is created by the literal.

The iteration terminates when the initial conjunction is completed. The search driver uses a depth-first search strategy as the default.

SOS, described in the previous section, is implemented as the primary task scheduling mechanism. Dynamic task migration is not supported in the current implementation.

Unification primitives perform unification without the occur check. Some optimization has been incorporated. The compiler discovers special cases that require less extensive runtime check on variable types and produces special codes for them. For example, unification between constants is treated as a simple term matching test without calling the general unification routine. A separate stack (trail) is used to record bindings that need to be recovered upon backtracking. Each stack frame contains an address of a bound variable. Upon backtracking, bound variables are reset to free variables. Again, bindings to variables created in a deterministic predicate need not be trailed, as no backtracking would ever occur for the predicate.

A third stack (a heap to be precise) is used for general storage including space for lists and structures. In addition, a portion of the heap is reserved for the SET operator to store results.

5 Experimental Results

A set of benchmark Prolog programs (recall that parScript overlaps Prolog) were run to compare the uniprocessor performance of our system with Quintus and Sicstus Prolog 0.6[3]. This experiment is done on a Sparc-II IPX workstation when the load is light. The machine has 24 MB memory installed. Timing statistics are in terms of seconds. All source codes can be found in [11]. Most of programs tested are small symbolic processing programs. The *msort* (mergesort), *mm* (matrix multiplication), and *lu* (LU-decomposition) programs are numerical comutation programs. They are used to demonstrate how to incorporate C procedures in a program in order to deal with problems for which a logic program alone is probably not suitable. Their source programs in parScript are listed in the appendix.

5.1 Performance on a Uniprocessor

System	fib(26)	9-Queens	11-Queens	Cube	Zebra	Pattern	Turtle
ParScript	3.8	3.3	83.9	4.4	0.9	0.4	2.8
Scistus	10.4	4.4	121.5	4.6	1.1	0.6	1.8
Quintus	3.0	2.2	58.5	1.7	0.5	0.2	1.1

Table 1: Run time comparison with Scistus 0.6 and Quintus with small Prolog programs.

[3] Quintus is the best-known commercial Prolog system. Sicstus Prolog 0.6 is probably the fastest non-commercial implementation of Prolog reported so far. We do not have access to any other systems and we feel that the above two systems are representative.

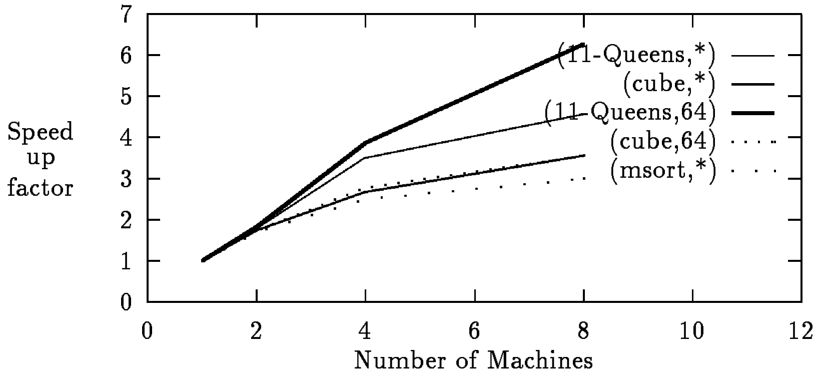


Figure 7: Speed-up on a DEC5000 workstation cluster. Line labels have the format (program_name,process_num). A "*" in the second field indcates using the default number of processes, i.e., the number of machines.

Table 1 shows the sequential run times[4] of the small test programs. The timing is averaged over five consecutive runs. Variance in execution time appears to be small and is not listed in the table. Our system generally yields an average runtime in between that of Sicstus 0.6 and Quintus Prolog. The slower *turtle* parScript program is dued to the unification routine in our system that is not yet optimized for the kind of list structure used in the *turtle* program.

5.2 Performance in a Workstation Cluster

We select a few sufficiently large programs (i.e., programs with a sequential execution time > 5 seconds) which have a chance to be substantially sped up by running in a cluster of workstations to test the parallel execution mechanism in our system. In our system, a user program is started with a startup program, *pstart*, which takes as command line arguments the name of the user program, number of machines, and optionally the number of processes to run. Recall that with SOS, one may have an arbitrary number of processes running. The default number of processes is the same as the number of machines. Note that a PVM daemon process [7] is running in the background in every machine in the cluster when *pstart* is invoked. The elapsed time taken in the experiment includes the time to start up the user program, but does not include the time to start the daemon process. Figure 7 gives speed-up curves from the experiment. Speedup factor is computed as the average sequential elapsed time divided by the average parallel elapsed time (on DEC5000 workstations). All of the DEC5000 stations have 16 MB main memory. The sequential runtime on a DEC5000 is generally slower (approx. 10%) than that on a Sparc IPX which has 24 MB memory. These curves indicate that reasonably good speed-up can be achieved. Figure 7 also shows the difference between having one process running per machine, and having a large number of processes per machine. The latter helps balance the workload on different machines as explained in section 3. Note that the difference is quite substantial with the *11-Queens* program using 8 machines. The *cube* program, however, has a very shallow search tree (24 potential branches at each node) and is not sensitive to this strategy. Poor speed-up by the *cube* program is attributed in part to its short sequential runtime, which is about 5 seconds on a DEC5000. We note that its takes at least 1 second to start up all 8 processors. *Msort* is the mergesort program used as an example in section 2. The program sorts a list of 12,000 randomly generated integers. Merge sorting is not a very good parallel algorithm (due to the merge phase being sequential), speed-up of a factor of 3 appears to be

[4] That is the CPU time. Elapsed time is close to the CPU time as few I/O operations, including paging activities, are involved. We do not use elapsed time here as it is hard to measure in the other two systems which we are comparing to. Sicstus is its version 0.6 #18 and Quintus is its Release 3.1.1

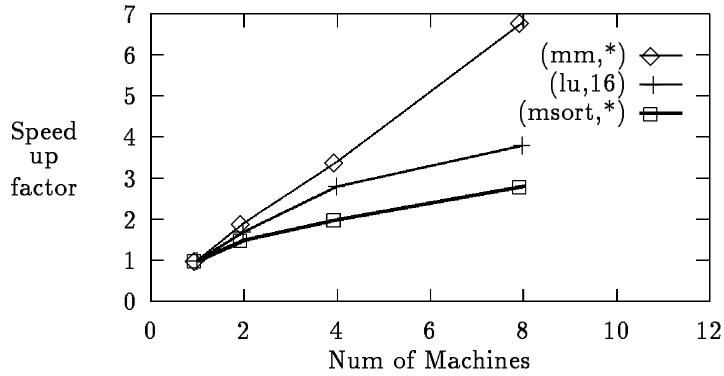


Figure 8: Speed-up on a DEC5000 workstation cluster. Line labels have the format (program_name,process_num). A "*" in the second field indcates using the default number of processes, i.e., the number of machines.

reasonable for sorting a relatively small number of elements.

5.3 Performance of ParScript Programs with External C Modules

The following are parScript programs that use external C functions to perform the actual work[5].

1. *msort*: sorting 100,000 4-bytes records. Using the merge sort algorithm in Section 2. The split() and merge() predicates are defined as C procedures. Also, a quicksort() procedure in C is called once the input array is split into 16 partitions.

2. *lu*: perform LU-decomposition on a dense 960x960 matrix. Problem decomposition is specified in parScript (see appendix). The rest are done by C procedures.

3. *mm*: perform matrix multiplication on a dense 640x640 matrix. Problem decomposition is specified in parScript (see appendix). The rest are done by C procedures.

Figure 8 shows the speed-up curve for the above three programs running in a cluster of DEC5000 workstations. Speed-up is computed against the runtime of the corresponding sequential C programs, which are 22.0, 690.0, 104.0 seconds for the *msort*, *lu* and *mm* programs, respectively. A mixed parScript and C program has a uniprocessor runtime about 5% slower than that of its pure C counterpart. The reason for the performance loss being so small is that the parScript portion of the program is executed very infrequently. The overhead incurred can mainly be attributed to the cost of manipulating multiple choices in parScript rules. We shall note that timing fluctuation is severe with the *msort* and *lu* programs, due to the fact that they require massive transfering of data among processors through the network. The numbers shown represent the average on the best 3 runs out of at least 10 runs. As can be seen, near linear speed-up is achieved by the matrix multiplication program, which needs to communicate very little. The other two programs show moderate speed-up. Given the relatively slow network used, these speed-up factors are deemed reasonable. In fact, the speed-up by the *lu* program is comparable to a hand-written LU-decomposition program which uses PVM calls and is running in a similar environment.

[5] As one may notice, the programs listed all operate on arrays. There is a new kind of variables, generic variables, supported in parScript [9]. These variables can be passed around in a parScript program, but they can only be manipulated by external C codes.

5.4 Related Parallel Logic Programming Systems

Existing high performance parallel Prolog implementations are mostly on shared-memory multiprocessors. Among them the Muse system [1], and the Aurora [14] system are the most popular, partially because they are successful parallelization of Sicstus Prolog 0.6 which is by far the fastest research implementation of Prolog. Performance data [3, 1] shows that the two systems both achieve good speed-up on the Sequent Symmetry machine. The Muse system also performs well on the BBN butterfly TC2000 machine [1]. Our system is different from the above systems in that 1) our system runs faster than Sicstus 0.6 on uniprocessor; 2) we use C as the target language; 3) our system operates on network; 4) we provide a very efficient SET operator that can practically be used to achieve AND-parallelism [9]; and 5) we do not support full Prolog. Comparable performance data are not available as experimental results are taken from so drastically different parallel platforms.

The Reduced-Or parallel logic programming system [17] is implemented on a variety of parallel machines including Sequent Balance, Symmetry, Encore Multimax, Intel iPSC/2, etc. The system supports both AND- and OR- parallelism. It is built on top of a supporting kernel, Charm [8], which provides message passing and light-weight process management facilities for many parallel applications including the Reduced-Or logic programming system. Portability appears to be the main objective for the Reduced-Or system. The drawback seems to be that its uniprocessor speed was very slow compared to Sicstus Prolog 0.6. Again, there is no comparable data from running in a workstation cluster.

6 Summary

A parallel programming system is described in the paper. The system is implemented both on sequential Unix workstations and in workstation clusters connected by the Ethernet. The system encourages conscientious separation of parallel control from sequential codes that perform the actual computation. We show the potential of this approach by empirical studies on the experimental programming system we implemented.

Contribution of this research is summerized as follows:

- implemented a prototyping system which facilitates experimenting with the idea of separating parallel control from sequential codes. Obtained quantitative results to evaluate the merit of the approach
- studied and experimented with techniques to generate efficient C object code from a Horn clause specifications
- demonstrated that parScript (with C modules) can be very efficient, and that they are portable across different architectures.

Experience with the system also suggests that the advantage by being able to separately debug the parallel and sequential parts of a program can significantly shortens the program development time. All the mixed parScript and C programs shown in the paper were developed by first writing a sequential C program, and a parScript program which is debugged through simulation runs by substituting dummy rules for the extern C function calls, then merging the two parts together.

We have not addressed the problem as to what happens if a fork-joint graph turns out to be a not good formula for the application at hand. We will look into this in the future. General solutions that are effective under all circumstances are yet to be invented. But there is no doubt that one should write clean and portable programs whenever possible. ParScript is one small step towards the goal.

Work is under way to port the system to a NUMA machine. We expect better speed-up results on a tightly-coupled multiprocessor. In the future, we will investigate the possibility of using other languages for parallel specification. We are also considering building real world applications such as full text search and natural language processing applications with parScript.

Acknowledgments: This research was supported in part by NSERC (Canada) CRD grant, and by the National Science Foundation under grant IRI-89-16059.

References

[1] K. Ali and R. Karlsson. The Muse or-parallel Prolog model and its performance. In *Proceeding of the North American Conference of Logic Programming, 1990*. MIT press, 1990.

[2] B.N. Bershad, E. D. Lazowska, and H. M. Levy. Presto: A system for object-oriented parallel programming. *Software - Practice and Experience*, 18(8), August 1988.

[3] R. Butler, T. Disz, E. Lusk, R. Overbeek, and R. Stevens. Scheduling or-parallelism: an Argonne perspective. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium on Logic Programming*. MIT press, 1988.

[4] M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, Swedish Insitute of Computer Science, 1990.

[5] K. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Transaction on Programming Languages and Systems*, 8(1):1–49, Jan 1986.

[6] K. Fuchi *et el.* Launching the new era. *Communication the ACM*, 36(3), March 1993.

[7] G. A. Geist. Network based concurrent computing on the pvm system. Technical report, Oak Ridge National Lab., 1991.

[8] L.V. Kale. A tutorial introduction to charm. Technical report, University of Illinois at Urbana Champaign, TR-92-6, 1992.

[9] W. F. Li and Z. Lin. A parallel prolog compiler and its implementation. Technical report, University of Toronto, CSRI-TR-284, 1993.

[10] Z. Lin. Self-organizing task scheduling for parallel execution of logic programs. In *Proceeding of the 1992 Conference on Fifth Generation Computer Systems*, Japan, 1992.

[11] Z Lin. *Task Scheduling for Parallel Execution of Logic Programs*. PhD thesis, Department of Computer Science, University of Maryland, 1992.

[12] Z. Lin and J. Minker. A distributed load balancing scheme for parallel execution of logic programs. In *Proceedings of the 21st International Conference on Paralllel Processing*, Chicago, August 1991.

[13] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[14] E. Lusk, Warren H. D., S. Haridi, and et al. The aurora or-parallel Prolog system. Technical report, Argonne National Laboratory, 1989.

[15] W. F. McColl. General purpose parallel computing. Technical report, Programming Research Group, Oxford University. Internal Report, 1992.

[16] S. Mudambi. Performance of aurora on numa machines. In *Proceeding of the 8th International Conference on Logic Programming*. MIT Press, 1991.

[17] B. Ramkumar and L. V. Kale. Compiled execution of the reduced-or process model on multiprocessors. In *Proceeding of the North American Conference of Logic Programming, 1989*. MIT press, 1989.

[18] L. Sterling. *Art of Prolog: Advanced Programming Techniques*. MIT Press, 1986.

[19] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*.

A ParScript Programs

1: Matrix multiplication program in parScript

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Matrix Multiplication:                                                    %
% C(row,col) = A(row,x) * B(x,col) for all x                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
matrix(Mc) :-
        get_matrix(500,500,Ma),         % obtain Ma
        get_matrix(500,500,Mb),         % obtain Mb
        multiply(2,500,500,Ma,Mb,Mc).   % Mc = (Ma X Mb), 4^2 way parallel

multiply(Depth,N,M,Ma,Mb,Mc) :-         % Mc is a product of Ma(N,M) and Mb(M,N)
        Depth < 1,                      % if bottom level
        prod(2,N,M,Ma,Mb,Mc).           % compute Mc = Ma X Mb
multiply(Depth,N,M,Ma,Mb,Mc) :-
        Depth => 1,                     % otherwise
        SUB(Depth,1,Ndepth),            % go another level deeper
        DIV(N,2,Nn),                    % Nn = N/2
        split(0,N,M,Ma,2,Ma0,Ma1),      % split Ma by row
        split(1,N,M,Mb,2,Mb0,Mb1),      % split Mb by column
        SET(Mm,R,p_multi(Ndepth,Nn,M,Ma0,Ma1,Mb0,Mb1,R)), % fire p_multi
        eq(Mm,Mc).                      % output Mm to Mc

% 4-way parallel, recursive.
p_multi(P,N,M,Ma0,Ma1,Mb0,Mb1,Mc00) :- multiply(P,N,M,Ma0,Mb0,Mc00).
p_multi(P,N,M,Ma0,Ma1,Mb0,Mb1,Mc01) :- multiply(P,N,M,Ma0,Mb1,Mc01).
p_multi(P,N,M,Ma0,Ma1,Mb0,Mb1,Mc10) :- multiply(P,N,M,Ma1,Mb0,Mc10).
p_multi(P,N,M,Ma0,Ma1,Mb0,Mb1,Mc11) :- multiply(P,N,M,Ma1,Mb1,Mc11).

eq(X,X).                                % everything equals to itself

% external calls
get_matrix(Row,Col,Matrix) :- foreign_c.
split(Flag,Row,Col,Matrix,Part,SubM0,SubM1) :- foreign_c.
prod(Part,Row,Col,SubM0,SubM1,M) :- foreign_c.

% query: compute all solution to matrix(R)
? SET(M,R,matrix(R))
```

2: LU-decomposition program in parScript

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% LU-decomposition                                                          %
% lu(K,N,Matrix,Lm,Um,R):                                                   %
% decompose Matrix(N,N) into a Lower matrix Lm and a Upper matrix Um.      %
% Using the kij-algorithm (row-major), parallelize the second loop         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

decompose(N,[Matrix,Lm]) :-
```

```
        get_matrix(N,N,Matrix),
        initLower(N,Lm),
        initFstrow(N,Fst),
        lu(1,N,Matrix,Lm,Fst).
lu(N,N,_,_,_).                         % terminate when K=N
lu(K,N,Matrix,Lm,Fst) :-
        K < N,                         % otherwise
        ADD(K,1,NextK),                % NextK = K + 1
        rows(16,N,K,NextK,Matrix,Lm,Fst,Nrow),  % compute LU, 16 bands
        lu(NextK,N,Matrix,Lm,Nrow).     % next K

rows(H,N,K,I,Ma,Lm,Orow,Nrow) :-
        SET(Fs,R,band(H,N,K,I,Ma,Lm,Orow,R)),   % fire bands
        cc_convert([Nrow],Fs).                  % set Nrow to first row

% Naive specification for bands. May be turned into recursion
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(0,H,N,K,I,Ma,Lm,Fst,R).      % band 0
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(1,H,N,K,I,Ma,Lm,Fst,R).      % band 1
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(2,H,N,K,I,Ma,Lm,Fst,R).      % band 2
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(3,H,N,K,I,Ma,Lm,Fst,R).      % band 3
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(4,H,N,K,I,Ma,Lm,Fst,R).      % band 4
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(5,H,N,K,I,Ma,Lm,Fst,R).      % band 5
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(6,H,N,K,I,Ma,Lm,Fst,R).      % band 6
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(7,H,N,K,I,Ma,Lm,Fst,R).      % band 7
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(8,H,N,K,I,Ma,Lm,Fst,R).      % band 8
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(9,H,N,K,I,Ma,Lm,Fst,R).      % band 9
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(10,H,N,K,I,Ma,Lm,Fst,R).     % band 10
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(11,H,N,K,I,Ma,Lm,Fst,R).     % band 11
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(12,H,N,K,I,Ma,Lm,Fst,R).     % band 12
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(13,H,N,K,I,Ma,Lm,Fst,R).     % band 13
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(14,H,N,K,I,Ma,Lm,Fst,R).     % band 14
band(H,N,K,I,Ma,Lm,Fst,R) :- cc_lu(15,H,N,K,I,Ma,Lm,Fst,R).     % band 15

% external calls
cc_lu(B,H,N,K,I,Ma,Lm,Fst) :- foreign_c.
cc_convert(X,Y) :- foreign_c.
initLower(N,Lm) :- foreign_c.
initFstrow(N,Fst,Nrow):- foreign_c.
get_matrix(N,N,M) :- foreign_c.

% query: compute all solution to decompose(960)
? SET(X,Y,decompose(960,Y))
```