A PRAM oriented programming system

COROMOTO LEÓN , CASIANO RODRÍGUEZ , FÉLIX GARCÍA AND FRANCISCO DE SANDE

Dpto. de Estadística, I. O. y Computación, Facultad de Matemáticas y Física, Universidad de La Laguna, c/Astrofísico Francisco Sánchez, s/n 38271 La Laguna, S/C de Tenerife, Spain

SUMMARY
A 11-oriented programming language called 11 and its implementation on transputer networks are presented. The approach taken is a compromise between efficiency and simplicity. The 11 language has been conceived as a tool for the study, design, analysis, verification and teaching of parallel algorithms. A method for the complexity analysis of 11 programs called PRSW is introduced. The 11 compiler guarantees the conservation of the PRSW complexity of the algorithms translated. Furthermore, the computational results illustrate the good behaviour of the system for PRAM algorithms. ©1997 by John Wiley & Sons, Ltd.

1. INTRODUCTION
The majority of existing massively parallel computers belong to the category of multi-computers: a group of processors, each with its own local memory and interconnected according to a certain network. Technological and economical reasons constrain the number of edges at each node (e.g. at most four if we select the transputer as node processor). Message-passing seems to be the natural programming model for this kind of machine. In this context, one of the most common computational models is CSP (communicating sequential processes)[1]. The 0ccam programming language is based upon this model. 0ccam allows the efficient implementation of a wide class of parallel algorithms. The 0ccam programmer controls the allocation of processes to processors and also the interprocessor communication and synchronization. It is his/her responsibility to guarantee the absence of deadlocks and starvation in the program. The lack of recursion and dynamic memory management are serious limitations of 0ccam. Usually, the abstraction level required for the representation of parallel algorithms is higher than the one provided by existing versions of the language. Despite these shortcomings, the situation is changing, as we can see by the research on more sophisticated versions of 0ccam, such as the virtual channel router[2] and the development of a new 0ccam standard[3,4].

On the other hand, most of the theoretical papers and parallel algorithms published from the 1970s to nowadays use in their presentation the PRAM (parallel random access machine) programming model. The PRAM is defined as a parallel shared-memory synchronous MIMD machine[5]. In this model, the time required for a memory access is the same for each processor, and it does not depend on the number of simultaneous accesses. The synchronism of the model and the presence of shared memory allow the designers of algorithms to use a high level of abstraction without having to deal with the problems of communication and synchronization among processes. The large number of algorithms published has generalized the use of a certain kind of pseudo-code for the presentation of parallel algorithms. The model has also given birth to specific techniques for analysis and design of parallel algorithms. The most important technical challenge in the practical
realization of the PRAM model lies in the construction of a scalable shared memory, providing constant access time for an arbitrary number of processors. An important number of publications\cite{6-11} assert the existence of theoretically efficient simulations of the PRAM model through the use of more realistic parallel computers.

Languages based on message-passing, such as occam, are not appropriate for the expression of PRAM algorithms. An example of a PRAM-oriented language is the FORK language. Hagerup, Schmidt and Seidl define FORK in \cite{12}. Parallelism is expressed in FORK through the use of two different statements:

\begin{verbatim}
start[<Expr>..<Expr>]
\end{verbatim}

and

\begin{verbatim}
fork[<Expr>..<Expr>]
\end{verbatim}

The \texttt{start} statement can be used to readjust the number of processors which are available, while the \texttt{fork} construction creates independent subgroups of the processors for different tasks. With an appropriate combination of the \texttt{fork} and \texttt{start} statements it is possible to achieve in FORK the same effect as the construct:

\begin{verbatim}
for i:=1 to n pardo ...
\end{verbatim}

used in the presentation of PRAM algorithms\cite{13,14}. FORK allows the expression of nested parallelism and the combination of parallelism and recursion. The development of the FORK language is related to a research project in the Computer Science Department at Saarbrücken University\cite{15} which deals with the construction of a PRAM model prototype computer. The objective of the authors of FORK\cite{12} is the implementation of the language on this prototype.

Philippsen and Tichy\cite{16} extend the Modula-2 language with two new \texttt{forall} statements that make it adequate for the expression of PRAM algorithms. They have developed compilers for this new language, called Modula-2*, for a Workstations network (medium-sized MIMD), a CM2 Connection Machine, a 16K MasPar (massively parallel SIMD) and a SparcStation 1 (SISD). The programmer can help in the distribution of data in the processors through the use of some directives in the declaration of arrays. These directives indicate how the elements must be distributed among the processors. Modula-2* establishes no differences between private and shared variables. The authors of Modula-2* do not tackle the problem of conservation by the compiler of the PRAM complexity of the algorithms.

The pm2 language\cite{17-19} is also a PRAM-oriented language based on Modula-2. In pm2 shared variables must be declared in the main block of the program, and although the current implementation presents some minor problems with nested parallelism, pm2 is a useful tool for teaching PRAM algorithms. The pm2 compiler produces, as output, an assembler program (the pm2 system can be downloaded from anonymous ftp at cs.joensuu.fi).

In this paper we present the 11 programming language (pronounced ‘eleven’), a PRAM-oriented programming environment. The 11 language is a superset of Pascal\cite{20}. The system is being developed by the authors\cite{21,22} in the Centro Superior de Informática at La Laguna University. Currently we have two implementations: one for transputer networks and another for IBM-PC computers.

In 11 the presentation of PRAM algorithms, even those that combine parallelism and
recursion, is quite simple and straightforward. During the life of the system (since 1990), our students in the Centro Superior de Informática have programmed a considerable number of algorithms, taken from the existing literature regarding the PRAM model\[13,14]. The experience we have acquired during these years shows us that users with some practice in sequential programming find the 11 language easy to learn, simple to use, and flexible enough to express the PRAM algorithms. Debugging programs in 11 is as easy as it is in sequential languages. These features make the 11 language a very appropriate tool for study, design, analysis, verification and teaching of parallel algorithms. The results we have obtained using the transputer network implementation show that 11 is also a valid tool for improving performance using parallel systems.

2. THE 11 PARALLEL PROGRAMMING LANGUAGE

In this work we use a slight variant of the PRAM model. To attain a better understanding of the 11 system, we briefly explain the modifications introduced. Each PRAM processor owns a segment of the shared memory, a private memory, a replica of the program, a function mode bit and a control bit. When the mode bit is on, the processor works synchronously with its siblings in the activation tree and its control bit determines its participation in the current step. Figure 1 shows the PRAM shared memory. Each memory address \( d \) is replicated on each processor so that the address \( d \) is represented by the set of cells in the column labelled \( d \). A processor \( P_i \) performing a read access simply reads from the cell containing its replica of that address. When a value is stored by a processor \( P_i \) in its replica of the shared address \( d \), the receiver cell transmits the incoming value to the other replicant cells using broadcast link \( d \). In general, the time spent in the broadcasting will be a function \( BT(p) \) of the number of processors. For example, when the replicant cells belonging to the same address are connected according to a tree topology, the broadcasting time will be \( BT(p) = \log(p) \). When we assume \( BT(p) \) to be constant, the model just described has the same computational power as the classic PRAM model.

2.1. The parallel statement for processor activation

We distinguish between a virtual or logical processor (process) and a physical or real (hardware) processor. A physical processor has the capacity to simulate a certain number of logical PRAM processors.

The parallel statement in 11 activates a set of virtual processors, all of them working concurrently. The syntax of the parallel statement is:

\[
\text{PARALLEL } <\text{Identifier}> \text{ IN } <\text{Expression}1> \ldots <\text{Expression}2> \text{ DO}<\text{Statement}>
\]

The \(<\text{Identifier}>\) in the parallel statement must be declared of type integer. Let \( R = \{a_0, a_1, \ldots, a_m\} \) denote the set of active virtual processors in the moment of execution of the parallel statement. Let \( s \) be the index of the active virtual processor \( a_s \). If we designate by \( l(s) \) the result of the evaluation of \(<\text{Expression}1>\) by the virtual processor \( a_s \) and by \( k(s) \) the result of the evaluation of \(<\text{Expression}2>\), then the following are the steps that take place when processor \( a_s \) performs the parallel statement:

1. For each \( s \in \{0, \ldots, m\} \), the number of virtual processors that must be activated
by the $a_s$ processor is calculated. Let $n(s) = k(s) - l(s) + 1$ be this number. A set $V^s = \{v^s_0, v^s_1, \ldots, v^s_{n(s)-1}\}$ with $n(s)$ virtual processors becomes active. Each virtual processor $v^s_t$ in $V^s$ has an <Identifier> variable with value $l(s) + t$.  

2. After all the processors in $R$ have completed the first step, the set $\bigcup_{s=0}^{m} V^s$ is the new group of processors that synchronously perform the instructions of <Statement>. The total number of active virtual processors is:

$$\sum_{s=0}^{m} (k(s) - l(s) + 1)$$

3. At the end of <Statement> the old set $R$ of active processors is restored.

The virtual processors $V^s$ may access the ‘shared data space’ associated with the $a_s$ virtual processor that has activated them. This means that any shared variable accessible by processor $a_s$ is also accessible by all the processors in $V^s$.

Each 11 variable has two attributes: its scope and its shared property. These two attributes are independent and ought not to be confused. Each variable declared at a point of the program belongs to the processor $a_s$ which is active at the moment of the declaration. (In contrast to Pascal-, 11 allows one to declare variables at the beginning of a compound statement). Furthermore, the variable will be accessible to all the processors $V^s$ that become active within the scope of the variable if the SHARED qualifier is used in the declaration.

The parallel statement can be used with the THRESHOLD qualifier. Its function is to determine the number of virtual processors per physical processor. This qualifier has no effect on the semantics of the program, but affects its efficiency. The complete syntax of
the parallel statement is

\[
\text{PARALLEL } <\text{Identifier}> \text{ IN } <\text{Expression}1>..<\text{Expression}2>
\]

[THRESHOLD <Constant>] DO

<Statement>

Consider once again the set of active virtual processors: \( R = \{a_0, a_1, \ldots, a_m\} \). Let \( H = \{p_0, p_1, \ldots, p_n\} \) (\( H \) for hardware) denote the set of physical processors of the object machine. Let \( p \in H \) denote a physical processor. Let us fix an instant in the computation, and denote by \( R_p \) the subset of virtual processors of \( R \) whose simulation is assigned to \( p \) (it may happen that for some \( p \) the set \( R_p \) is empty). Evidently, \( R \) is the union of \( R_p \) with \( p \in H \). The number of processors in \( R_p \) may be modified because more virtual processors may be required. If this occurs the actions that take place are the following:

1. The physical processor \( p \) computes, as we have indicated before, the set \( R'_p \) of processors required by the \( a_i \in R_p \) processors.
2. If the number \( |R'_p| \geq \text{<Constant>} \) value specified in \text{THRESHOLD}, then \( p \) shares the simulation of the \( R'_p \) processors among the most idle processors of \( H \). Otherwise, the processor \( p \) will deal with the whole simulation of \( R'_p \).

The present implementation of the 11 system sets the threshold value to three if no value is specified.

Figure 2 illustrates these actions for a parallel statement with threshold 5 when executed on a machine made up of three physical processors \( H = \{p_0, p_1, p_2\} \). Before the activation, four virtual processors \( R = \{a_0, a_1, a_2, a_3\} \) are active. Processor \( p_0 \) is idle (\( R_{p_0} \) is empty), \( p_1 \) is in charge of two \( R_{p_1} = \{a_0, a_1\} \) logical processors, and \( p_2 \) is simulating another two \( R_{p_2} = \{a_2, a_3\} \) logical processors. After the activation, processor \( p_2 \) simulates the total number \((n(2) + n(3) = 4) \) of virtual processors \( R'_{p_2} \) demanded in \( p_2 \), since this amount is smaller than the threshold. Meanwhile, processor \( p_1 \) distributes between \( p_1 \) and \( p_2 \) the simulation of the set \( R'_{p_1} \) of five logical processors demanded by \( R_{p_1} \).

Evidently it is necessary to allow the nesting of the processor activation statements to support the modular programming paradigm. The 11 language permits nesting of parallel statements.

### 2.2. The assignment statement

The syntax of the assignment statement in 11 is:

\[
<\text{Variable}> : [\text{:] := <Expression> [\text{:]}}
\]

A synchronous assignment statement like

\[
<\text{Variable}> := <\text{Expression}>
\]

is executed by the set of active virtual processors as follows:

1. The processors calculate the left value \(<\text{Variable}>\).
2. After finishing the address evaluation phase, they evaluate the expression.
3. After all the processors finish the previous phase, the results of the expression evaluation are stored in the memory addresses calculated in the first step.
If the <Variable> is shared, there will be a synchronization point after the evaluation of the right side of the statement and another after the values are stored. The 11 programmer can suppress the first and the second synchronization points using, respectively, the first and the second optional token `:'. The aim of this option is to allow the programmer to help the compiler in the task of minimizing the number of synchronizations. If the variable is private (not shared) no synchronization will take place.

In 11, concurrent write operations of the same value in the same memory address are considered correct. When different values are assigned to the same memory address, the different replicant cells constituting that address (Figure 1) will contain some of the different written values. In such cases the coherency of the implied address cannot be assumed.

2.3. Call statements

If a subroutine may be called from a parallel context, it must be declared using the qualifier SHARED. If the routine is going to be called only from a sequential context, it is declared using the keyword PROCEDURE. There is a differentiation between ‘tasks’ and ‘procedures’. This difference eases the generation of code by the compiler. When a parameter passed by reference is used, the shared property attribute of the formal parameter must coincide with that of the actual parameter.

In 11 there is an extension of the Pascal concept of calling statement which allows multiple parallel calling of the same task. The syntax of a parallel call is

\[
\text{[\text{THRESHOLD} <\text{Constant}>]}
\text{\[} t(x_1^0, x_2^0, \ldots, x_n^0) || t(x_1^1, x_2^1, \ldots, x_n^1) \ldots || t(x_1^{k-1}, x_2^{k-1}, \ldots, x_n^{k-1}) \text{]}\
\]

where \( t \) is a task identifier and the \( x_i \) denote different expressions.
2.4. Conditional statement

The syntax of a conditional statement is

\[
\text{IF } <B> \text{ THEN } <\text{Statement1}> \ [\text{ELSE } <\text{Statement2}>]
\]

The steps that take place when an if-else conditional statement is executed are:

1. Initially, the set of active processors is \( R = \{r_0, r_1, \ldots, r_m\} \). Each processor in \( R \) evaluates the Boolean expression \(<B>\). The values of the condition divide \( R \) into two complementary subsets:

\[
S = \{r_i/B_i = \text{TRUE}, i = 0, \ldots, m\}
\]

\[
S' = \{r_i/B_i = \text{FALSE}, i = 0, \ldots, m\}
\]

where \( B_i \) represents the result of the evaluation of \(<B>\) by the processor \( i \).

2. After all the processors in \( R \) have evaluated \(<B>\), the set of processors \( S \) for which the evaluation of the condition has become true execute \(<\text{Statement1}>\). When the execution of \(<\text{Statement1}>\) by all the processors in \( S \) finishes, the processors in the other subset \( (S') \), execute \(<\text{Statement2}>\).

3. When all the virtual processors in \( S' \) finish the execution of \(<\text{Statement2}>\), the active processors set is the same as it was at the beginning \( (R) \).

There are two synchronization points: one after the evaluation of the Boolean expression and another before the execution of \(<\text{Statement2}>\).

2.5. Loops

The syntax of the while statement is

\[
\text{WHILE } <B> \text{ DO } <\text{Statement}>
\]

Its meaning could be recursively described by the following statement sequence:

\[
\text{IF } <B> \text{ THEN}
\]

\[
\text{BEGIN}
\]

\[
<\text{Statement}>;
\]

\[
\text{WHILE } <B> \text{ DO } <\text{Statement}>
\]

\[
\text{END}
\]

The execution of a while statement by a set of active elemental processors \( R = \{r_0, r_1, \ldots, r_m\} \) flows through the following steps:

1. All the processors in \( R \) evaluate the logical expression \( B \). Once the evaluation has been completed by all the processors in \( R \), the subset of processors from \( R \) for which \( B \) is false become inactive with respect to the while statement. In other words, the new set \( R \) of active processors is

\[
R = \{r_i/B_i = \text{TRUE}\}
\]
2. If $B$ yields false for all the processors, that is, $R = \phi$, then the execution of the \texttt{while} statement ends and the execution continues in the fourth step. Otherwise, the processors in $R$ execute the instructions in \texttt{<Statement>}. 

3. When all the processors in $R$ have finished the execution of \texttt{<Statement>}, control is returned to the first step.

4. The old pattern of active processors is recovered.

The \texttt{while} statement has one synchronization point after the evaluation of the boolean expression. Conventional \texttt{repeat} and \texttt{for} statements are also present in 11.

2.6. The \texttt{RELAX} qualifier

The \texttt{RELAX} qualifier can be used in any statement:

\begin{verbatim}
RELAX < Statement >
\end{verbatim}

In that case, all the synchronization points in \texttt{<Statement>} are eliminated, making the execution of \texttt{<Statement>} totally asynchronous.

2.7. Parallelism and recursion in 11

The provision of nested parallelism, in combination with the power of recursion, allows a simple and elegant formulation of a large number of parallel algorithms. (See, for example, [14], pp. 59, 381, 382, 466, 467). Let us consider the 11 code in Figure 3 that solves enumeratively the 0–1 knapsack problem. Given a set $O$ of $N$ objects whose weights are $w[1], \ldots, w[N]$ and with benefits: $p[1], \ldots, p[N]$ and a knapsack of capacity $C$, the problem consists of finding a subset $S$ of objects from $O$ providing an optimum benefit $B$ and fitting into the knapsack.

The problem is solved using the divide-and-conquer programming technique. For each object $i$ in $O$, we must decide whether or not $i$ will be included in the knapsack. Two knapsack subproblems with a set of objects $O-\{i\}$ arise. These new subproblems can be solved recursively in parallel following the same method. When this \textit{NP-complete} problem is solved using the above technique, the execution time of the algorithm measured according to the PRAM model grows linearly with the number of objects. Unfortunately, the number of processors used grows, in the worst case, exponentially with the number of objects. Since no synchronization is needed the 11 code to call the solution task is:

\begin{verbatim}
B := 0;
RELAX Knapsack(n, C, B);
\end{verbatim}

3. THE TRANSPUTER IMPLEMENTATION

The 11 compiler[21] uses the predictive recursive descendant syntax analysis method and generates code for a stack oriented PRAM machine. The instruction set of the PRAM machine[22] is an extension of the instruction set of the P-machine described by Brinch Hansen[20]. We have developed implementations of the 11 language for transputer networks and IBM-PC computers. Transputer networks are the cheapest parallel computers
1: SHARED PROCEDURE knapsack(i: INTEGER; C: INTEGER;
    var B: SHARED INTEGER);
2: CONST
3: NO = 0;
4: YES = 1;
5: TYPE
6: parameters = ARRAY [NO..YES] OF INTEGER;
7: VAR
8: benefit, capacity : SHARED parameters;
9: fits : BOOLEAN;
10: BEGIN
11: IF (i > 0) AND (C > 0) THEN
12: BEGIN
13: benefit[NO] := B;
14: capacity[NO] := C;
15: fits := (C >= w[i]);
16: IF fits THEN
17: BEGIN
18: benefit[YES] := B + p[i];
19: capacity[YES] := C - w[i]
20: END
21: ELSE benefit[YES] := -INFINITY;
22: PARALLEL name IN NO..ord(fits) DO
23: knapsack(i - 1, capacity[name], benefit[name]);
24: B := benefit[NO] MAX benefit[YES]
25: END
26: END (* knapsack *);

Figure 3. The knapsack problem

and they are also the most widely used in European universities. This is the reason we
chose them as the target machine for our development.

The topology selected for the transputer network executing the code generated by the
11 compiler is the tree. The root processor deals with the execution of the sequential part
of the program. Through the computation, each transputer simulates a certain number of
virtual processors. When a processor activation statement is performed (i.e. a parallel
statement or a parallel call statement), transputers executing it proceed (depending on
the threshold value) to share their load among their descendents.

Each non-leaf transputer is the owner of a segment of addresses from the shared memory.
Transputers in the same level of the tree share the same address segment. The working
memory of a transputer constrains the size of any object declared in 11. It follows that any
memory access in a transputer will take place in its own private memory or in the memory
of some ancestor in the tree.

A very important feature of the implementation is that any transputer has a replica of the
shared memory of each one of its ancestors. Therefore, any transputer has not only its own
work memory and its piece of shared memory but also a copy, kept perfectly updated at all
times, of the shared memory of its ancestors.
Let’s consider, as an example, a ternary tree with 13 transputers distributed in three levels and 4 Mbyte per transputer. We assume that to run a certain program we need 132Kbyte of shared memory. Half of this memory is accessed by all the processors, while the remainder is shared only by processors belonging to groups executing the same parallel task. A suitable distribution is the allocation of 64Kbyte of shared memory to the root transputer and the remaining 64Kbyte to the first level transputers. Leaf transputers do not need to have shared memory. Figure 4 shows the property relations that are established among the different transputers and the shared memory address space.

The advantage of this simple design is that any read memory access takes unit time. The disadvantage of the scheme is that any write access to the address of a shared variable must be propagated to all the transputers descending upon the owner of the address. This process, necessary to keep updated copies of shared addresses, takes $O(p)$ time. Here $p$ denotes the number of processors taking part in the write access. Without loss of generality, let us assume that the number $p$ of virtual processors coincides with the number of physical processors. According to the current implementation four kinds of PRAM instructions are distinguished by their costs:

1. Instructions dealing with write operations in the shared memory, with a cost in the implementation of the same order as the number $p$ of processors participating in the variable access.
2. Synchronization instructions which are implemented with a logarithmic cost in the number of processors used.
3. Processor activation instructions, with a logarithmic cost in the maximum number of processors required.
4. The remaining instructions (read operations, sum, product, comparison, write operations in private memory cells, etc.) having constant cost.

A very important advantage arises from the fact that the links of a transputer processor have the ability to operate in parallel with their CPU. When a shared memory write instruction is asynchronous (e.g. using the RELAX qualifier in 11) the transputer CPU can continue running while the propagation of the write access takes place. If the overlap between communication and computation is complete, the shared memory write operation is finished without increasing the computing time.

Many theoretical works have proved the feasibility of simulating a PRAM machine using hypercubic processor networks. It seems that the most reasonable choice for the implementation of a PRAM-oriented system on a transputer network would be to start from one of those proposed simulation algorithms. To discuss the advantages and disadvantages of both approaches, let us take the algorithm described in [10], pp. 703–709, as a representative of those theoretical works. In that algorithm, the deterministic simulation of each
memory access in a PRAM with $N$ processors by an $N$-node butterfly machine takes time $O(\log M \log N \log \log N)$, where $M$ represents the size of the shared memory.

The total number of steps $T_\lambda$ spent by a PRAM algorithm $A$ is the sum of three components: the steps $T_c$ spent in computation, the number $T_w$ of write accesses and the number of read accesses $T_r$:

$$T_\lambda = T_c + T_w + T_r$$

Hence, the total number of steps $T_\lambda^1$ spent by algorithm $A$ when memory requests to the shared memory are simulated according to [10] is

$$T_\lambda^1 = T_c + C \log M \log N \log \log N (T_w + T_r)$$

where $C$ denotes the constant associated with the algorithm. Let $q = T_r/T_w$ denote the ratio between the numbers of read and write operations. Then $T_\lambda^1$ becomes

$$T_\lambda^1 = T_c + C \log M \log N \log \log N (1 + q) T_w$$

Since in the implementation proposed read requests take constant time and the propagation algorithm for write requests $C'N$, the number of steps consumed by algorithm $A$ when memory requests are simulated using 'total replication' is

$$T_\lambda^2 = T_c + C'NT_w + T_r = T_c + (C'N + q)T_w$$

the difference between times $T_\lambda^1$ and $T_\lambda^2$ is marked by the factors $f_\lambda^1$ and $f_\lambda^2$ that multiply $T_w$.

$$f_\lambda^1 = C \log M \log N \log \log N (1 + q)$$

and

$$f_\lambda^2 = (C'N + q)$$

A detailed reading of [10] will persuade the reader that the constant $C$ involved is huge (the order of thousands) and far larger than $C'$ (which is the order of several units). When the ratio between read and write requests $q$ is large and the number of processors $N$ is moderate (several hundreds), the adding component $(C \log M \log N \log \log N)$ $q$ in $f_\lambda^1$ makes this factor greater than $f_\lambda^2$. Trivially, the following lemma holds.

3.1. Lemma

If $q > N$ then $T_\lambda^1 > T_\lambda^2$.

3.1.1. Proof

$$f_\lambda^2 = C'N + q < C'q + q = (C' + 1)q < C \log M \log N \log \log N q < f_\lambda^1$$

As shown in the next Section, there is an entire class of PRAM algorithms where the quotient $q = T_r/T_w$ tends to infinity as the problem size grows.
4. THE PRSW COMPLEXITY ANALYSIS MODEL

In the analysis of the complexity of parallel algorithms in the PRAM model, unit cost is supposed for any instruction, even one that handles memory access. The analysis of an algorithm written in $\text{II}$ language using the model we call PRSW (parallel read sequential write PRAM) is similar to the analysis using the PRAM model. The PRSW complexity of any $\text{II}$ statement is calculated as in the PRAM model, except for the assignment statement. The difference stands in the following assertion: the execution cost of an assignment to a shared variable in $\text{II}$, when it is accomplished in parallel, must be considered equal to the number of processors that share the variable and which are being used in the moment of the assignment.

In the PRSW model we have an extension of the concurrent write concept. In the classic PRAM model, concurrent write is considered to happen when several processors write simultaneously to the same memory address. In the PRSW model, concurrent write results when several processors write simultaneously to the same variable.

4.1. Example: Addition in the PRSW model

If we assume the declarations in lines 1–7 of Figure 5, where $\sqrt{N}$ denotes the square root of $N$, the $\text{II}$ code that follows solves the problem of summing the $N$ elements of the array $a$ with a $O(N^{\frac{1}{2}})$ PRSW time, using $N^{\frac{1}{2}}$ processors.

The loop in lines 15–16 takes $O(N^{\frac{1}{2}})$ time since the sum variable is private, and so the assignment in line 16 takes constant time. The assignment to the shared variable aux in line

1:   TYPE
2:     vector = ARRAY [0..N-1] of INTEGER;
3:     smallvector = ARRAY [0..sqrtN-1] OF INTEGER;
4:     VAR
5:       a : SHARED vector;
6:       aux : SHARED smallvector;
7:       result : INTEGER;
8:       ....
9:     PARALLEL i IN 0..sqrtN-1 DO
10:    VAR sum : INTEGER; first, last : INTEGER;
11:   RELAX BEGIN
12:     first := sqrtN * i;
13:     last := first + sqrtN - 1;
14:     sum := 0;
15:    FOR k := first TO last DO
16:      sum := sum + a[k];
17:     aux[i] := sum;
18:   END (* RELAX *);
19:   result := 0;
20: FOR k := 0 TO sqrtN-1 DO
21:   result := result + aux[k];

Figure 5. The sum of the elements of an array
Type  vector = array[1..N] of integer;
matrix = array[1..N] of vector;

Var  a, b, c : shared matrix;

1: parallel i in 1..N do
2: parallel j in 1..N do
3: var sum : integer;
4: begin
5: sum := 0;
6: for k := 1 to N do
7: sum := sum + a[i][k] * b[k][j];
8: c[i][j] := sum
9: end

Figure 6. Matrix product

17 consumes O(N^4) time in the PRSW model because N^4 processors write to aux. The assignment to private variables in lines 12, 13 and 14 take unit time. Finally, the loop in lines 20-21 is O(N^4). Therefore, the algorithm runs in O(N^4) PRSW time. The product

(Parallel time) x (Number of processors)

equals the time of the sequential algorithm, O(N).

4.2. Example: Matrix product in the PRSW model

Analysing the algorithm in Figure 6, which calculates the matrix product, using the PRSW model, we observe that assignments in lines 5 and 7 take constant time. The assignment to the shared variable c in line 8 is performed by N^2 processors and therefore is O(N^2). Consequently, the algorithm runs in time N^2. This result contrasts with the O(N) complexity obtained when using the PRAM model. However, it is possible to develop an optimal algorithm for the PRSW model with the alternative algorithm presented in Figure 7.

The algorithm uses N processors and performs the matrix product in time O(N^2). In fact, the loop 3–10 runs N times. The time consumed in each loop iteration results from the sum of the time spent in the assignment in line 6, the interior loop in lines 7–8 and the assignment in line 9. The assignment to the private variable in line 6 demands constant time. The interior loop runs N times. The assignment in line 9 to the shared variable c is performed by N processors, so it takes time O(N) in the PRSW model. Therefore, the PRSW time for the algorithm is O(N^2). Again the product N x O(N^2) equals the time of the sequential algorithm, O(N^3).

4.3. Example: The knapsack problem

If we study carefully the algorithm in Figure 3 solving the knapsack problem, we will realize that in line 23 the parameter B, passed by reference, is replaced by benefit[name]. Consequently, the only assignment to a shared variable occurs in line 24,
The variable $B$ is shared at most by both processors which have been activated by the parallel statement in line 22, and therefore the time consumed in the assignment is constant. Thereby, the PRSW time for the algorithm matches the $O(N)$ time obtained in the PRAM model.

It is a well-known fact that the sum of $N$ elements can be calculated in $O(\log N)$ time using $N/\log(N)$ processors in the PRAM model. Furthermore, the matrix product in the PRAM model can be achieved in $O(\log N)$ time using $N^3/\log(N)$ processors. It is common to find algorithms for the PRAM model providing exponential time reductions: the sequential time results from an exponential function of the parallel time. These speedups are in some cases fictitious: when it is feasible to port these algorithms to realistic machines without loss of efficiency, the behaviour of the algorithm is strongly dependent on some special input data distribution in the network, and the algorithm leaves its results in a particular distribution. An explanation of how data reached that initial location, and whether the final distribution is suitable for subsequent routines, is usually avoided in the literature. In some situations, especially when all the input/output in the parallel machine takes place through a host computer, to move these data from their initial distribution (i.e. the host) to the start distribution takes time which is far from logarithmic. This proliferation of efficient algorithms with exponential reductions is not true in the PRSW model: the time reductions we can expect are, in general, polynomial. In fact, the size of the output of a PRSW algorithm is a lower bound for the PRSW complexity of the algorithm. However, the PRSW model provides a benefit: it is easily and efficiently accomplished in bounded degree processor networks. The present version of the 11 compiler guarantees the conservation of the PRSW complexity of the analysed algorithm.

```plaintext
1: RELAX
2: PARALLEL i IN 1..N DO
3:   FOR j := 1 TO N DO
4:   VAR sum : INTEGER;
5:   BEGIN
6:     sum := 0;
7:   FOR k := 1 TO N DO
8:     sum := sum+a[i][k]*b[k][j]
9:   c[i][j] := sum
10: END;
```

Figure 7. Another implementation of the matrix product

5. COMPUTATIONAL RESULTS

The experiments have been accomplished on 32 T800-20MHz transputers, distributed on eight quadputer boards. Each transputer has 4Mbyte of memory. The reconfigurability of the network is achieved with the use of a linkputer board, having four C004 crossbar chips. The links operate at 10Mbyte/s. The C004 chips impose some delay in the communications. Quadputer boards are located on the expansion bus of an IBM-PC compatible machine.
Table 1. Computational results

<table>
<thead>
<tr>
<th>Problem</th>
<th>Processors</th>
<th>Size</th>
<th>$T_S$</th>
<th>$T_{11-IC}$</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAT</td>
<td>1024</td>
<td>1024</td>
<td>2.60</td>
<td>6.21</td>
<td>3.92</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>2048</td>
<td>5.77</td>
<td>5.90</td>
<td>5.48</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>4096</td>
<td>9.81</td>
<td>4.34</td>
<td>5.07</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>8192</td>
<td>19.99</td>
<td>3.79</td>
<td>5.19</td>
</tr>
<tr>
<td>SHP</td>
<td>3600</td>
<td>60</td>
<td>38.17</td>
<td>3.44</td>
<td>15.09</td>
</tr>
<tr>
<td></td>
<td>4900</td>
<td>70</td>
<td>70.36</td>
<td>3.43</td>
<td>15.16</td>
</tr>
<tr>
<td></td>
<td>6400</td>
<td>80</td>
<td>104.66</td>
<td>3.43</td>
<td>15.31</td>
</tr>
<tr>
<td></td>
<td>8100</td>
<td>90</td>
<td>148.63</td>
<td>3.43</td>
<td>15.42</td>
</tr>
<tr>
<td>MAT</td>
<td>3600</td>
<td>60</td>
<td>4.51</td>
<td>3.47</td>
<td>13.27</td>
</tr>
<tr>
<td></td>
<td>4900</td>
<td>70</td>
<td>7.13</td>
<td>3.46</td>
<td>15.29</td>
</tr>
<tr>
<td></td>
<td>6400</td>
<td>80</td>
<td>10.62</td>
<td>3.46</td>
<td>14.21</td>
</tr>
<tr>
<td></td>
<td>8100</td>
<td>90</td>
<td>15.09</td>
<td>3.45</td>
<td>15.88</td>
</tr>
<tr>
<td>KNP</td>
<td>8100</td>
<td>14; 1000</td>
<td>1.47</td>
<td>6.27</td>
<td>11.77</td>
</tr>
<tr>
<td></td>
<td>8100</td>
<td>15; 1000</td>
<td>2.89</td>
<td>6.15</td>
<td>14.14</td>
</tr>
<tr>
<td></td>
<td>8100</td>
<td>16; 1000</td>
<td>5.26</td>
<td>5.60</td>
<td>14.49</td>
</tr>
<tr>
<td></td>
<td>8100</td>
<td>17; 1000</td>
<td>10.00</td>
<td>5.32</td>
<td>14.69</td>
</tr>
<tr>
<td>SUM</td>
<td>1024</td>
<td>131,072</td>
<td>1.65</td>
<td>4.36</td>
<td>12.69</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>131,072</td>
<td>1.73</td>
<td>4.59</td>
<td>12.99</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>131,072</td>
<td>1.81</td>
<td>4.78</td>
<td>11.92</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>131,072</td>
<td>1.81</td>
<td>4.78</td>
<td>9.45</td>
</tr>
</tbody>
</table>

A transputer connected to the PC behaves as an ordinary I/O server. The network is configured as a tree, where the root and the three first level processors have three children each. Each of the nine transputers on the second level has two children. There are 18 transputers in the third and last level.

Tests have been performed on five PRAM efficient algorithms: the SUM algorithm described in [13], pp. 13, 14, following the balanced binary tree method; the KNP algorithm that solves the knapsack problem (Figure 3); the BAT, Batcher’s sorting algorithm, written using the recursive presentation given in [14], pp. 178, 179, and Kucera’s SHP shortest paths algorithm for graphs ([13] pp. 25, 26). We have also implemented sequential versions of these algorithms in 11 and C. The data for the input problems were randomly generated.

In Table 1, the first column identifies the problem. The second column shows the maximum number of virtual processors that participate in the parallel computation. The third column is the size of the problems: the size of the vector of integer values for Batcher’s sorting algorithm, the number of vertices for the shortest path problem, the dimension of the matrices, the number of objects and the capacity in the knapsack problem and the size of the vector of integer values for the aggregation algorithm. The next column, labelled $T_S$, represents the time in seconds for the sequential algorithm coded in 11. The fourth column shows the quotient $T_{11-IC}$ between $T_S$ and $T_{IC}$, where $T_{IC}$ is the time for the same sequential algorithm coded in Inmos C[23]. This relation shows the performance of the 11 system in comparison with the Inmos C. The last column shows the speedups obtained with the 11 system using 18 leaf transputers.

The algorithms were programmed straightforwardly, without any special optimizations. The results show the good behaviour of the implementation for efficient PRAM algorithms. Only Batcher’s algorithm does not approximate linear speedup.
6. LIMITS AND FUTURE DEVELOPMENT

The 11 programming environment is still an experimental system. Char, real, file, pointer and variant record types have not been implemented. We also intend to provide vector segmentation in the language. At present, input and output operations can take place only in sequential execution. One of the most important drawbacks of the current implementation is that transputers located in a middle layer of the tree remain only as simple routers and synchronizers when the number of processors demanded is large. Another important drawback is that the 11 language is interpreted. As a consequence, the actual performance is poor when compared with the inmos C toolset. All these limitations will be overcome in the new versions of the system. A second implementation of the 11 system for a butterfly topology following the classic probabilistic simulation algorithm ([10], pp. 700, 701) is currently being developed.

REFERENCES


