LANGUAGE SUPPORT FOR PARALLEL PROGRAMMING

Teodor Rus
Department of Computer Science
University of Iowa
Iowa City, Iowa 52242
(319) 335-0742, rus@cs.uiowa.edu

Abstract

This paper discusses rationale for standardizing language support for parallel programming. The evolution of the concepts handled by parallel programming and the actual state of the art are presented in the introduction. The intuitive meaning of the terms parallel algorithm, parallel process, parallel programming, are informally discussed in section 2. Section 3 is devoted to an algebraic formalization of these terms. Section 4 is dedicated to the discussion of a minimal language support to be provided by programming languages in order to facilitate parallel programming. A set of constructs are developed in this respect. These constructs have as the basis both semantic model of parallel processes discussed in section 3 and parallel libraries provided by the actual multiprocessor machines to support parallel programming.

General terms: algorithm, algebra, process, processor, programming language.

Additional key words and phrases: algebraic system, lock data type, mutual exclusion, parallel algorithm, parallel process, parallel programming.

1. Introduction

> *Technology without standards is chaos, standards without rationale is disaster.*

Programming concerns the activity of mapping algorithms (expressed in human oriented languages) into expressions (called programs) of some programming languages such that the computation contents of the algorithms are preserved when their expressions are evaluated by a given machine. Thus, programming activity regards the handling of sophisticated mathematical objects (algorithms, expressions, programming languages, machines) which are not always well understood. Therefore, in order to become convenient, computers are equipped with tools which hide the complexity of the objects handled by their programmers. Among the most sophisticated such tools are those regarding parallel execution of the computations encapsulated into an algorithm using two or more streams of control. Since an algorithm might solve certain problem, interaction between the computations performed by its parallel streams of control must be provided and controlled. The major problems posed by programming such parallel computations have been discussed in early 1970-s, [12], [25], [17], [10]. However, no actual machines providing hardware support for parallel programming were available at that time.

Parallel architectures [15], [27] have been studied with the goal of developing machines that allow efficient execution of various classes of parallel algorithms. The major problems posed by such architectures regard the information flow between processors while carrying out parallel streams of control of a parallel algorithm [27], [34]. Meanwhile, models of parallel computations (not necessarily stemmed from the mathematical concept of computation) have been developed [12], [14], [18], [19], [21], [22], [23], [25], [35], and continue to be developed [38].

The high technology provided by the VLSI allowed parallel architectures evolve to the commercial multiprocessor machines (Encore, Alliant, Balance, Cube, etc.) providing real hardware support for parallel programming. However, the models of parallel computation did not mature into a theory of parallel programming. Parallel programming as discussed in the most of today texts [28], [36] do not stem from a formal concept of parallel program. It rather mimics the period of time when sequential programming has been developed regarding its objects, the programs, as binary expressions executable by a given machine [1] and not as abstractions representing the behavior of actual objects. Various vendors offering multiprocessor machines provide their computers with various packages supporting parallel programming without concern about the portability of their products among the available machines or the development of a standard language support for parallel programming. Parallel programming is regarded as a discipline which allows a programmer to fit his or her algorithm on certain type of parallel architecture [36]. Some current research [29] timidly approaches parallel algorithms conceptually, however characterizing them in terms of their performance more than their logical development. In these terms a parallel algorithm has an architecture dependency and thus, knowledge of the actual architecture on which the algorithm is implemented are required in order to develop a parallel program. This way of approaching parallel programming excludes automatically most of the today programmers from writting parallel programs. However, parallel computations are developed in the real world, outside the machine, and their purpose is to solve problems which are independent of the machine on which they are solved. Moreover, the natural parallelism implied in many computations is not necessarily determined by the speed of their execution. For example, keeping dynamic systems in balance (such as dining philosophers, reader/writers, consumer/producer, robot movement and control, etc) implies

parallel computations in a natural way without regard to the performance of the algorithm execution. Yet, today almost entire research on parallel programming [28], [30], [9], [3], [27],[8] is oriented towards machine dependent algorithms and its only reason seems to be to improve the algorithm performance. Programmer's accessibility to the machine is addressed only through sophisticated packages whose use require even more than understanding parallel architectures and algorithms [33], [6].

This paper provides rationale for standardizing language support for parallel programming. The essential issues posed by parallel programming are approached. These issues concern the development of parallel algorithms independent of the machine designed to execute them and the mapping of parallel algorithms into parallel programs of a given programming language. The fundamental concepts used in parallel programming such as parallel algorithm and parallel process are developed as mathematical objects [26]. While a parallel algorithm is a linguistic expression a parallel process is an abstraction [10] that carry out computations provided into a parallel algorithm. Thus, the separation of the issues that regard program development by the programmer from the issues that regard program execution by the system are approached and standards for their language expressions are proposed. Since a parallel algorithm expresses operations carried out by a parallel process both parallel algorithm and parallel process need to be developed on a common basis. This conceptualization of parallel algorithms development and of the process of their execution allow us to discuss a minimal language support required by parallel programming.

2. Concepts

Better fuzzy standards for conceptualization than fuzzy concepts for standardization.

Intuitively an algorithm is a finite ordered set of computation rules. When the order relation among these rules is total the algorithm is called sequential. When the order relation among these rules is partial the algorithm is called parallel. Notice that an algorithm is a static entity which can be represented as a linguistic expression. The language that allow us to represent an algorithm symbolically is called an algorithmic language. By itself the expression of an algorithm does not imply any dynamic activity of computation. The dynamic activity of performing the computations specified by an algorithm is called the process of algorithm execution. If the algorithm is parallel the process of its execution is called a parallel process. If the algorithm is sequential the process of its execution is called a sequential process. Finally, the concept of a computation in our terminology is specified by the set of computing processes performing it. In other words, a computation is an element of an algebra of processes [26]. A parallel computation handles parallel processes and a sequential computation handles sequential processes. Thus, the keyword in our discussion of parallel computations is the concept of a process.

Intuitively a process is a unit of computation. It is characterized by a static expression representing the algorithm performed by the process and a dynamic behavior representing the computations encapsulated in the algorithm. The dynamic behavior of a process requires an agent to carry it out. The agent performing the computation task provided by the static expression of a process is called a processor. Hence, the intuitive concept of a process imply a language to express the algorithm performed by the process and a processor to carry out the computations specified by the algorithm.

3. Formalizations

Mathematics well applied illuminates rather than confuses.

We use the hierarchy of concepts *algebraic system*, *processor over an algebraic system*, and *algorithm over an algebraic system* in order to formalize the concept of a process and to develop an algebraic theory of parallel computations.

3.1. Algebraic System

The concept of an algebraic system is needed here in order to provide the formal concept of an operation to be used as the computation rule of the algorithms we are dealing with. The following definition will be used in this respect:

Definition_1: *An Algebraic System, $\mathcal{A}$, is a pair $\mathcal{A} = \langle A, \mathcal{O} \rangle$, where A is a set, (or a family of sets indexed by a given set of integers), called the carrier of the algebraic system, and $\mathcal{O}$ is a set of operations and relations over A.*

Observation_1: If there are only operations in $\mathcal{O}$ the algebraic system is called an algebra. If there are only relations in $\mathcal{O}$ the algebraic system is called a model.

The algebraic support for programming languages is provided by the *heterogeneous algebraic systems*, HAS, [37], which are defined as follows:

1. A finite set of integers called index set, I, is given. The elements $i \in I$ are used as domain and range selectors for the heterogeneous operations.
2. A finite set of symbols, S, over a given alphabet is also given. The elements $s \in S$ are called generators of operation names.
3. A finite set of operation schemes, Σ, defined in terms of S and I is also given. An operation scheme $\sigma \in \Sigma$ is a triple $\sigma = \langle\ n,\ s_0 s_1 ... s_n,\ i_1 i_2 ... i_n j \rangle$ where:
   - *n is the arity of the operation, i.e., the number of operands mapped by the operation into a result.*
   - *$s_0, s_1, ..., s_n \in S$ and $s_0 s_1 ... s_n$ is the operation symbol distributed over operands.*
   - *$i_1, i_2, ..., i_n \in I$ are domain selectors and $j \in I$ is the range selector.*
4. A function $F : \Sigma \longrightarrow \mathcal{O}(A = (A_i)_{i \in I})$, where $\mathcal{O}(A)$ is the set of all operations and relations defined on A, is also given.

The triple $\mathcal{B} = \langle I, S, \Sigma \rangle$ is the basis (or the signature) specifying a class of heterogeneous algebras. The triple $\mathcal{A} = \langle A = (A_i)_{i \in I}, \Sigma, F(\Sigma) \rangle$ is a heterogeneous algebra specified by the base $\mathcal{B}$ in terms of the user defined family of sets (or types) A.

Observation_2: Since the carrier of a heterogeneous algebraic system can always be enriched with the true value set $\{true, false\}$ any n-ary relation $\circ$ can be represented by an appropriate heterogeneous operation $\circ : A_1 \times A_2 \times \ldots \times A_n \longrightarrow \{true, false\}$. Thus, no distinction need to be made between models and algebras. Therefore, we use the terms algebra and algebraic system as synonyms.

3.2. Processor over an Algebraic System

In this context a processor is identified with an abstract agent capable of executing any of the operations, functions, predicates, characterizing a given algebraic system, $\mathcal{A}$. Processors are denoted further by Pr, accompanied by indices if necessary.

Example of Processors:

1. The hardware processor of a computer installation is a processor over the algebraic system defined by the pair $\langle Memory, Instruction\_set \rangle$ where $Memory = \{Location_t | \ t \in T\}$. T is a finite set of hardware recognized location types such as byte, word, longword, etc. $Location_t$ is the class of locations of type t.
2. The processor performing Pascal programs is defined over the algebraic system defined by the Pascal data structures together with the Pascal operations, i.e., by the pair $Pascal = \langle Data\_Structures, Operators \rangle$.

Observation_1: The operations of an algebraic system are provided with composition rules which allow new operations to be generated from the given ones. Thus, an operation of an algebraic system is either primitive (i.e., predefined), or composed (i.e., defined). The rules for operation composition are defined by generic composition schemes acting on already defined operations. Thus, the composed operations are expressed in terms of the predefined and/or already defined operations.

Observation_2: The carrier of an algebraic system is provided with composition rules which allow new objects to be generated from the given ones. Thus, an object of the carrier of an algebraic system is either a primitive (i.e., predefined) data or a composed (i.e., defined) data. The rules of data composition are defined by generic composition schemes acting on already defined data and generating new and more complex data called data structures. Therefore, composed data are expressed in terms of the predefined and/or already defined data.

Observation_3: In defining a processor over a given algebraic system the assumption is that the predefined operations defining the algebraic system are atomic operations according to the processor. Likewise, the predefined data are implicitly recognized by the processor.

3.3. Algorithm over an Algebraic System

An algorithm over an algebraic system, AS, is specified by a finite set of data which belong to the carrier of the algebraic system, a finite set of operations from the operations defining the algebraic system, and a relation which specifies the order in which the operation defining the algorithm are executed on its data. Formally an algorithm can be defined as follows:

Definition_2: *An algorithm, Alg, over a given algebraic system $\mathcal{A}$ is specified by a triple $Alg = \langle D, O, R \rangle$ where:*

1. *D is finite set of data defined in the carrier of $\mathcal{A}$.*
2. *O is a finite set of operations, functions, predicates, among the operations, functions, predicates, supported by the $\mathcal{A}$.*
3. *R is a relation showing the order of execution of operations in O on the data in D.*

Sequential algorithm: If the ordering relation R is linear (or total) on O then the algorithm is called a sequential algorithm.

Parallel algorithm: If the ordering relation R is partial on O then the algorithm is called a parallel (or concurrent) algorithm.

The actor model of parallel computations [21], [22], [23], is a good illustration of this concept of a parallel algorithm. However, all the models of parallel computations developed so far can be formally defined in terms of a concept of parallel algorithm defined as above.

Since the set O of operations specifying an algorithm is finite, it can always be decomposed into the subsets O_1, O_2, ...,O_k such that each O_i, $i = 1, 2, \ldots, k$ is a linear ordered set of operations by the order of the operation execution. Let us denote by R_i the linear relation defined by the order of operation execution of the operations in O_i.

If for each $i, j, i \neq j, i, j = 1, 2, ..., k$ the data $D_i \subseteq D$ on which operations in O_i are acting are disjoint of the data $D_j \subseteq D$ on which operations in O_j are acting, then the algorithm $Alg = \langle D, O, R \rangle$ gives rise to the family of sequential algorithms

$$Alg = \{Alg_i = \langle D_i, O_i, R_i \rangle; i = 1, 2, \ldots, \}$$

which can be performed in parallel and preserve the consistency of the computation specified by the original algorithm, Alg. However, the problem of decomposing data and operations of a given algorithm into sequential components such that the consistency of the original algorithm is preserved is more general. Even if D_i and D_j are not disjoint the computation performed by the sequential components of the algorithm could still be preserved in some conditions. When the algorithm Alg solves a given problem its components Alg_i, $i = 1, 2, \ldots, n$ can be regarded as the asynchronous actions performed by Alg during the process of problem solving. Consistency of the process of problem solving (if data on which algorithms Alg_i, $i = 1, 2, \ldots, n$ act are not disjoint) and/or communication requirements might impose synchronization of the actions performed by Alg_i, $i = 1, 2, \ldots, n$. Therefore, the algebraic system supporting Alg must be provided with operations capable to perform the synchronization of the actions of sequential components of a parallel algorithm. These operations together with the data on which they are defined are called further a *lock data type.*

3.4. Process over an Algebraic System

The concept of a process over an algebraic system is defined by a pair $Process(\mathcal{A}) = \langle Pr(\mathcal{A}), Alg(\mathcal{A}) \rangle$ where $Pr(\mathcal{A})$ and $Alg(\mathcal{A})$ are defined over the algebraic system $\mathcal{A}$. The

algebraic system over which a process is defined can be factored out and we shall drop it from further discussion. It will be mentioned only when necessary.

When the algorithm in the definition of the process P is sequential, the operations executed by the processor can be seen as a sequence $\circ_{\pi_1} \rightarrow \circ_{\pi_2} \rightarrow \ldots \rightarrow \circ_{\pi_k}$ for certain permutation $(\pi_1, \pi_2, \ldots, \pi_k)$ of its operations $\circ_1, \circ_2, \ldots, \circ_k$. Hence, a sequential algorithm defines a sequential process, i.e., a process $P = \langle Pr, Alg \rangle$ will be called sequential if the algorithm Alg defining it is a sequential one.

When the algorithm in the definition of the process P is parallel, it can be decomposed into a given number of sequential algorithms, $Alg = \{Alg_i\}, i = 1, 2, \ldots, r$. The operations executed by Alg_1, Alg_2, ..., Alg_r can be seen as the sequences:

$$Alg_1 : \circ_{11} \rightarrow \circ_{12} \rightarrow \ldots \rightarrow \circ_{1p_1}$$
$$Alg_2 : \circ_{21} \rightarrow \circ_{22} \rightarrow \ldots \rightarrow \circ_{2p_2}$$
$$\ldots$$
$$Alg_r : \circ_{r1} \rightarrow \circ_{r2} \rightarrow \ldots \rightarrow \circ_{rp_r}$$

If Alg_1, Alg_2, ..., Alg_r have an independent (conditionally or unconditionally) functional behavior [19] and if a number of r processors Pr_1, Pr_2, ... , Pr_r over $\mathcal{A}$ are available, then obviously the pairs

$$P_1 = \langle Pr_1, Alg_1 \rangle$$
$$P_2 = \langle Pr_2, Alg_2 \rangle$$
$$\ldots$$
$$P_r = \langle Pr_r, Alg_r \rangle$$

define r sequential processes which can perform in parallel. In this case the process $P = \langle Pr, Alg \rangle$ is a parallel process. In general, a process $P = \langle Pr, Alg \rangle$ will be called parallel (or concurrent) if the algorithm Alg defining it is a parallel algorithm.

Observation_1: The complexity of the computations performed by the sequential components of a parallel algorithm and the frequency of their interaction during the parallel process life lead to the concept of parallel algorithm granularity [4]. We consider in this paper only medium coarse and coarse parallel algorithms.

Observation_2: A partial or total ordered finite set can always be decomposed into a finite set of total ordered subsets. However, the problem of designing parallel algorithm regards the consistency of the computation expressed by the algorithm. Therefore, only those decompositions of the ordered set of operations defining an algorithm which preserve the consistency of its computation are of interest here.

A parallel process gives rise to a parallel computation providing that a set of processors are given over the algebraic system defining the respective process, and the sequential algorithms defined by the order relation within the algorithm are (conditionally or unconditionally) functional independent. Thus, the main problems possed by the theory and practice of parallel computations using multiprocessor machines concern the process management. The essential operations of a process management system are process creation, process manipulation (scheduling), and process deletion. The definition and the implementation of these operations require a data representation of the concept of a process.

3.5. Process Data Representation

In order to handle processes a process data representation is required. We call it a *Process Control Block*, **PCB**, and define it as follows:

Definition_3: *The* **PCB** *of a process* $P = \langle Pr, Alg \rangle$ *is a data structure of two components,* **Pr_REP** *and* **Alg_REP**. **Pr_REP** *is the data representation of the processor performing the process and* **Alg_REP** *is the data representation of the algorithm defining the computations performed by the process.*

We assume that the behavior of a processor performing an algorithm can be specified by a finite set of primitive data structures representing the computation state of the processor [10]. Likewise, we assume that the data representation of an algorithm is specified by a finite set of data structures specifying the linguistic expression of the algorithm, its access rights, and its capability list [31].

The algebra of processes over which we develop computations has as the carrier a data structure having **PCB**-s as elements. We call this data structure the *Process Data Structure*, **PDS**. The operations of this algebra handle **PCB**-s. A process in the **PDS** can be in one of the following states according to the processor specified in its **PCB**:

1. The processor specified in the **PCB** performs the computations specified by the algorithm represented by the **PCB**. This state is called the *Running* state.
2. The algorithm specified by the **PCB** is ready to be executed by the processor but the processor does not execute it. This is called the *Ready* state.
3. The processor specified by the **PCB** did execute the algorithm, but for some reason it was interrupted before its completion. This is called *Suspended* state.

The life of a process **PCB** $\in$**PDS** starts by its creation using a *create* operation and evolves on the *Process State Transition Diagram* given in figure 1.

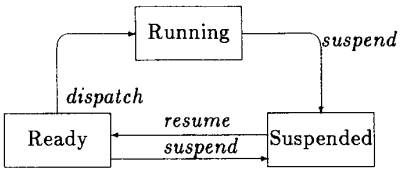

Figure 1: Process State Transition Diagram

At this point we assume that the computer system designed to perform a computation benefits of more than one processor. There is a process state transition diagram for each processor provided in the computer installation. However, more than one process in the **PDS** can have the same physical processor provided in the system as its processor component. Thus, the component processor of the process represented by a **PCB** specifies an abstract processor. Since this abstract processor is similar to a physical one, the *dispatch* operation on the process state transition diagram maps the abstract processor specified by a **PCB** into a physical one by loading the physical registers of the processor with the data representation of the abstract pro-

cessor specified by that **PCB**. The sequence of operations performed by a processor in order to switch its processing state from one process to another is called a *switch context* operation.

The operating system provided on a computer installation handles parallel computations by maintaining the state transition diagram of all physical processors in the system using switch context operations. The programmer needs to specify in his or her program the process creation, the process execution, and the process termination, according to the process of problem solving performed by his algorithm. A *create* process operation issued in the program will be implemented by the system by creating a new **PCB**, updating its components with the data representations of the algorithm and processor specified by *create* operation, and then sending this **PCB** into the **PDS**. The operation *execute* process is implemented by assigning the process to a processor and providing it with the ready state. When that processor becomes free and its scheduling policy makes this process the next one to be dispatched on the state transition diagram, the process will be dispatched and then its execution starts. So, an operation *execute* process issued by a program does not necessarily mean that the process starts executing immediately. During its execution a process may be suspended by the system for various reasons. An operation *terminate* process will suspend the process and an operation *remove* process will remove the process specified by that **PCB** from the **PDS**. The component of the operating system that performs these operations is the **P**rocess **M**anagement **S**ystem, **PMS**. The activity of the **PMS** is not visible to the programmer. We focus further only on the aspects of a parallel computation that are visible to the programmer.

4. Expressing Parallel Processes

> *One uses the standards one has, when one hasn't the standards one needs*

Assume now that we know how to split the sequence of operations defining an algorithm into (conditionally or unconditionally) functional independent subsequences which define functional independent sequential processes. Then these processes can be executed in parallel. From the programming viewpoint the first problem to be solved concerns the development of a suitable language support that facilitates the expression of such parallel processes. The construct used in a conventional programming language to express a sequential computation unit is that of a statement. Therefore, we assume that the sequential components of a parallel algorithm are expressible as statements. A parallel algorithm could be expressed as a well formed construct of a programming language using an appropriate parallel composition operation defined on statements [26]. Let us denote by S (short for statement) the generic expression of a sequential computation. The computation expressed by a parallel algorithm might require that the sequential components of the algorithm to interact. The interaction of the sequential components of a parallel algorithm can be achieved by letting them operate on common variables and/or by receiving and sending information (messages) between them.

The processor executing a parallel algorithm requires each sequential process to be created and represented by an appropriate **PCB** that specifies an algorithm and its processor requirements. As long as only one processor is available in a hardware system there exists only one actual process executing and this process can be handled in a standard way by the operating system. However, the programmer must be allowed to create and schedule sequential processes performing the task of his or her algorithm explicitly, according to the problem requirements, rather then implicitly, according to the operating system conveniency. Moreover, since the interaction between sequential components of a parallel process is not transparent to the operating system the programmer needs to control it at the level of the language expression of the parallel algorithm. Hence, the minimal support that a programming language should provide to facilitate parallel programming consists of:

1. Constructs that allow explicit definition, declaration, and use of shared resources.
2. Constructs that allow explicit process creation, process execution, and process termination.
3. Constructs that allow explicit control of processes interaction (synchronization and/or message passing primitives).

The minimal language support for parallel programming suggested in this paper is provided in a way or another by the actual multiprocessor computers available today. However, the standardization of this support by the traditional programming languages is somehow rendered more difficult by their historical development.

4.1. Fork and Join

Historically the first programming language construct supporting parallel programming is **fork** [7]. Its general linguistic expression is **fork Label** ... **join**[*count*]. In the context of a program it can occur as follows:

```
...
   S1;
fork L;
   S2;
go to L1;
L: S3;
L1: join [count];
   S4;
```

The meaning of this construct is:

- Program is executed sequentially until **fork** is encountered.
- At the point in the program where **fork** occurs the program execution is split in two parallel components. One component (the parent) executing further (S_2 in the above expression) and another component (the child) starting the execution of the statement labeled by L (S_3 in the above expression).
- The sequential components executing S_2 and S_3 are joined again at the label L1.

The parameter *count* of the primitive **join** allows a variable number of processes to be joined at the label L1. This is achieved by the following actions performed by each process arriving at the **join** *count* statement:

```
count = count -1;
if (count > 0)
   quit;
else
   continue;
```

If quit means "terminate" then the above construct tells that all processes which arrive at **join** *count* are terminated except the last one which continues, executing sequentially.

The following critics are pertinent to this construct:

- Process execution is implicitly specified by process creation (by **fork**). Operating system executes the process following a standard policy.
- No provisions for resource sharing and/or controlling process interaction are provided.

A variation of the **fork** construct has been implemented in the UNIX operating system as the system call *fork()*. In this version the forking does not mention a label and the **join** is provided by the system call *wait()*. In order to understand the meaning of the UNIX *fork()* and its use in a C language program we observe that any executing program under UNIX system is composed of three different parts called segments, the code segment, the data segment, and the stack segment.

1. *fork()* creates a child process which inherits the entire memory image of the parent. The code segment is shared while the data segment and the stack segment are duplicated.
2. If *fork()* succeeds to create a child process *fork()* returns a positive integer representing the child identification number called *child_id* in the parent and 0 in the child. If *fork()* fails to create the child process it returns -1 in the parent.
3. The child process once created executes further the same code as the parent does.

The allocation of the disjoint data and stack segments to the parent and the child removes the main difficulties and dangers regarding computation consistency performed by the two parallel streams of control. It is however inefficient in both time and resources. The main difficulties for the programmer result from the fact that parent and child are identical and executes the same code. No provisions are made for explicit specification of different sections of code to be performed by the two sequential processes in parallel. The *child_id* needs to be used in this respect. Moreover, there are no explicit provisions for parallel processes to interact (by sharing resources and/or sending messages) and primitives for controlling their interaction. Such primitives can be implemented however, using other UNIX system calls.

4.2. Cobegin and Coend

Using **fork ... join** the program gets an ugly structure. Therefore another construct which keeps the program structure clean has been introduced by Dijkstra [12] under the name **cobegin ... coend**. This construct allows the programmer to specify explicitly the sequential portions of his or her program performed by the parallel sequential processes. In addition, it allows an arbitrarily number of sequential processes to be specified as acting in parallel. However, the task of process creation and process scheduling for execution is still implicitly provided by the operating system. No provisions are made for process interaction. If processes that are defined by the statements enclosed in the parenthesis **cobegin ... coend** interact their code need to be provided with locks which synchronize their actions.

The linguistic expression of a parallel program using **cobegin ... coend** is:

cobegin S_1; S_2; ... ; S_n **coend**

The meaning of this construct is:

1. Create the new processes P_1, P_2, ..., P_n, one for each computation encapsulated within the statements S_1, S_2, ..., S_n.
2. The processes P_1, P_2, ..., P_n are executed further in parallel.
3. When all the processes P_1, P_2, ..., P_n terminate, the original computing activity evolves according to the text of the program enclosing the **cobegin ... coend** construct.

4.3. Minimal Language Support

The minimal language support suggested in this paper for expressing parallel algorithms results from the above discussion and is inspired from the parallel library provided in the Multimax system implemented on the Encore machine [13] and Concurrent Pascal [19].

4.3.1. Sharing Resources

In order to preserve the consistency of the computations while sequential components of a parallel process operate on shared resources the shared resources need to be accessed in mutual exclusion. This means that shared objects need to be recognized as such and only one sequential process to be allowed to operate on such an object at once. So, the first problem is to provide a shared object type as a special type supported by the programming language, using a special type constructor. The shared object type needs be defined by the programmer and the shared objects of the shared object type need be declared by declarations supported by the programming language. Thus, we consider three type of constructs that regard shared objects:

1. Constructs which allow shared type definition.
2. Constructs which allow shared object declarations.
3. Constructs which allow safe operations on shared objects.

Using the **typedef** constructor provided in the C language, a shared type can be defined by the scheme:

typedef shared *type_constructor type_name*;

where *type_constructor* is a given type expressed in terms of already constructed types and *type_name* is the name of the new shared type. Example of such type definitions are:

typedef shared int shared_int;

```
typedef shared struct my_object
        {
         char First;
         float Matrix[lines][cols];
         float vect1[lines],vect2[cols];
         BARRIER barr;
         LOCK lock1, lock2;
         SEMAPHORE sem1, sem2;
         char Last;
        } shared_object, *to_shared;
```

The declaration of a shared type object should be provided in a programming language following its usual declaration schemes. For example, the declaration of shared objects of the type defined above could follow the C language conventions as follows:

```
shared_int i, j, k; /* i, j, k are shared integers */
shared_object block; /* block is the form above */
to_shared p1, p2; /* p1 and p2 are pointers */
```

The conventions to reference shared objects are the same as the references of other objects in the languages. However, the operations on shared object need to be protected by appropriate lock code.

The shared object constructs are not explicitly provided by most of the languages providing support for parallel programming. However, the language support for parallel programming implemented on the actual multiprocessor machines [13], [33] are provided with parallel libraries or macro facilities in which shared objects are explicitly provided. The parallel library provided in Multimax system uses three different functions to create shared objects. They are:

1. *share(Address, Size)* where *Address* is a pointer to an object previously defined and *Size* is the size of that object. It returns a pointer to a shared copy of this objects. The *fork()* system call used for process creation does not duplicate such an object letting it in a global area which is shared by the parent and the child. Example of use of this function is:

   p1 = share(&block.First, &block.Last-&block.First)
2. *share (0, size(to_shared))* has the same effect as the *share(Address, Size)*. However, the compiler allocates the shared object at the end of the shared data segment.
3. *share_malloc_init(Size)* declares that a region of memory of size *Size* needs to be allocated as a shared memory between the parent and its children. Locations in this region can be dynamically handled by the functions *share_malloc* and *free_malloc*. A call of the form *p1 = share_malloc(sizeof(to_shared))* allocates a chunk of shared memory of size *sizeof(to_share)* in the shared memory area provided by *share_malloc_init* and returns a pointer to it.

The advantage of this sort of parallel libraries is that they observe the structure of the language they are meant to extend. However, the programs which use such parallel libraries are not portable among machines supporting the same language. To solve the problem of program portability macro operations can be used which are expanded by a common macro processor supported by the operating system of the machines on which parallel programs are intended to execute. The experience provided in [33] is significant in this sense. However, since such macro operations do not belong to the language in which the programmer expresses his or her program, the programming task becomes more difficult.

4.3.2. Process Management

Few of the languages providing support for parallel programming have constructs that allow explicit process creation, process execution, and process termination. The operation of process creation is considered an operation performed by the operating system. The language constructs such as **fork ... join, cobegin ... coend**, etc., assume that processes executing parallel statements specified by them are created, executed, and terminated automatically by the operating system. This means that such processes are created and start their execution when the process executing the main program enters the parallel construct and are terminated when the parallel construct is exited. In other words, no distinction is made between process creation, process execution, and process termination. The distinction between these three stages in a process life need to be transparent to the programmer since the programmer develops the parallel program.

Versions of the UNIX system extended with parallel libraries are used as operating systems for most of the new multiprocessor machines (Encore, Alliant, Balance Sequent, Cub, etc.). The most difficult problem faced by programmers developing parallel programs using these parallel libraries regard the parallel execution environment in which process creation and process execution are implicitly provided by the same construct. Since the only difference between the parent and the child created by *fork()* is the value returned by *fork()*, the expression of the parallel processes has the standard (sometime undesired) form:

```
...
Id_fork = fork();
switch (Id_fork)
      {
       case '0':{Child's code}
       case '-1':{Parent's code when fork fails}
       default: {Parent's code when fork succeeds}
      }
...
```

We advocate here that a programming language supporting parallel programming needs to be provided with explicit constructs that allow the programmer to create processes, initiate process execution, and terminate process execution. Mimicking the C language such constructs could be specified as follows:

1. The process creation can be provided by an operation denoted by a language construct of the form:

   create(Pr, Statement);

Pr shows the processor provided in the hardware which will execute the process and *Statement* shows the algorithm (the section of the program) executed by this process. The effect of the operation *create* will be a new **PCB** in the **PDS**. It should return a user *Process_id* of the process thus created which can be further used for explicit process scheduling for

execution. The type of processor executing a process is not necessarily the same for all processes. See the execution modes used by the Alliant multiprocessor in this respect [2].

2. The process execution can be explicitly specified by an operation denoted by a language construct of the form:

execute(Process_id);

Process_id is obtained by a *create* operation Such a construct is strongly advocated by the development of parallel execution environments in which parallel programs can be debugged by debuggers developed for sequential program debugging.

3. The process termination can be explicitly specified by operations denoted by language constructs of the form:

terminate(Process_id);
remove (Process_id);

The *terminate* operation terminates the process initiated by an *execute* if active, and the *remove* operation removes the process from the **PDS**.

The designers of the Multimax system [13] observing that *fork()* is both inconvenient and expensive provided in the parallel library supporting parallel programming similar constructs under the name *tasking environment.*

4.3.3. Controlling Processes Interaction

Parallel processes created and activated as shown in previous section carry out computing tasks that might contribute asynchronously to perform a given computation. The process cooperation for achieving this computation is obtained by means of three generic communication relations, process synchronization, sharing resources, and sending messages. Process synchronization is the relation which restricts the activity of parallel processes to not proceed beyond given points until some events happen. (For example, other processes have reached other given points). Sharing resources is the relation by means of which two or more sequential processes can operate on the same physical or logical resource. In order to ensure the consistency of their activity the processes sharing resources must obey the protocol of mutual exclusion. Sending messages is the relation by which a process is allowed to send and/or receive data to/from another process.

The language constructs that allows process synchronization and the implementation of the protocol of mutual exclusion are known as lock data type. The first such data type has been proposed by Dijkstra [12] under the name *semaphore.* A definition of the semaphore data type follows:

1. The carrier of the semaphore data type are non-negative integers.
2. The operations defined on the semaphore data type are of two types: atomic operations (*wait* and *signal*) and non-atomic operations (*assignment*). Assuming that S is a variable of type semaphore these operations can be defined as follows:

```
wait(S): if (S <= 0)
            {
             Enqueue(Itself, S_queue);
             Suspend(Itself);
            }
         else
             S = S - 1;

signal (S): if (S_queue != Null)
            {
             Process_id = Dequeue(S_queue);
             Resume(Process_id);
            }
          else
             S = S + 1;

assignment: S = Expression;
```

The parallel library implemented on the Encore machine under Multimax operating system diversifies this lock data type providing four types of locks: *spin_lock*, which is a sort of binary semaphore, *event*, which is a semaphore which frees all processes waiting on it when the event occur, *barrier* which is a semaphore that opens after a certain given number of processes arrive at it, and the usual semaphore as defined above.

Most of the research on the language support for parallel programming provided by conventional languages regards the computation power of their lock data type constructs and the encapsulation of this computation power in convenient high level language constructs.

The language constructs that allow sequential processes computing in parallel to interact by message passing are *send* and *receive.* They are usually provided as operating system functions and are well formalized by the input-output channels discussed by Hoare, [26]. However, they are rarely provided as constructs of conventional programming languages.

References

[1] Abbott, R. J., *Knowledge Abstractions*, Comm. ACM Volume 30, No. 8, August 1987, pp. 664-671.

[2] —— *Using the Alliant FX/8*, ANL/MCS-TM-69, Rev. 1, Technical Memorandum No. 69, Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue, Argonne, IL 60439-4844, September 1986.

[3] André, F. Herman, D., Verjus, M. P., *Synchronization of Parallel Programs*, The MIT Press, 1985.

[4] Bell, C. G., *The Multi - A New Computer Class*, in [13], pp. 5-8, February 1987.

[5] Bell, C. G., *Multis: A New Class of Multiprocessor Computers*, Science 228, April 1985, pp. 462-467.

[6] Boyle, J., Butler, R., Disz, T., Glieckfeld, B., Lusk, E., Overbeek, R., Patterson, J., Stevens, R., *Parallel Programs for Parallel Processors*, Holt, Rinehart, and Winston, Inc., 1987.

[7] Conway, M., *A Multiprocessor System Design*, Proceedings of the AFIPS Fall Joint Computer Conference, 1963, pp. 139-146.

[8] —— *CSNET NEWS*, Spring 1987, No.13, Summer 1987, 14.

[9] Denning, P. *Parallel Computation*, American Scientist 73, 1985, pp. 322-323.

[10] Dennis, J. B., Van Horn, E. C., *Programming Semantics for Multiprogrammed Computations* Comm. ACM, Volume 9, No.3, March 1966, pp. 143-155.

[11] Dijkstra, E. W., *Solution of a Problem in Concurrent Programming Control*, Comm. ACM, Volume 8, No.9, (September 1965), pp. 596.

[12] Dijkstra, E. W., *Cooperating Sequential Processes*, in Programming Languages (Ed. F. Genuys), Academic Press, New York 1968.

[13] —— *Using the Encore Multimax*, ANL/MCS-TM-65, Rev.1, Technical Memorandum No. 65, Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue, Argonne, Ill 604-4844, February 1987.

[14] Filman, R. E., Friedman, D. P., *Coordinated Computing - Tools and Techniques for Distributed Software*, McGraw-Hill, Inc., 1984

[15] Flynn, M. J., *Very High-Speed Computing Systems* Proceedings IEEE Volume 54, 1966, pp. 1901-1909.

[16] Hansen, P. B., *The Nucleus of a Multiprogramming System*, Comm. ACM, Volume 13, No. 4, April 1970, pp. 238-250.

[17] Hansen, P. B., *Structured Multiprogramming*, Comm. ACM, Volume 15, No. 7, July 1972, pp. 574-578.

[18] Hansen, P. B., *Concurrent Programming Concepts*, Computing Surveys, Volume 5, No. 4, December 1973, pp. 223-245.

[19] Hansen, P. B., *Operating System Principles*, Prentice Hall, Inc., 1973.

[20] Hansen, P. B., *Architecture of Concurrent Programs*, Prentice-Hall, Inc., 1977.

[21] Hewitt, C., Baker, H., *Actors and Continuous Functionals*, Proceedings of IFIP Working Conference on Formal Description of Programming Concepts, August 1977, pp. 369-387.

[22] Hewitt, C., Baker, H.,*Laws of Communicating Parallel Processes*, Proceedings of IFIP Congress, August 1977, pp. 987-992.

[23] Hewitt, C., Atkinson, R., *Synchronization in Actor Systems*, Proceedings of the Conference on Principle of Programming Languages, January 1977, pp. 267-280.

[24] Hoare, C. A. R., Monitors: An Operating System Structuring Concept Comm. ACM Volume 17, No.10, October 1974, pp. 549-557.

[25] Hoare, C. A. R., *Communicating Sequential Processes*, Comm. ACM Volume 21, No. 8, August 1978, pp. 666-677.

[26] Hoare, C. A. R., Communicating Sequential Processes, Prentice-Hall, Inc., 1985.

[27] Hwang, K., Briggs, F. A., *Computer Architectures* McGraw-Hill, Inc., 1984.

[28] Jamiesen, L.H., Gannon, D. B., Douglass R. J., Editors, *The Characteristics of Parallel Algorithms*, The MIT Press, 1987.

[29] Jamiesen, L. H., *Characterizing Parallel Algorithms*, in [28] pp. 65-100.

[30] Karp, A., *Programming for Parallelism*, Computer 20, No.5, pp. 43-57.

[31] Levy, H. M., *Capability-Based Computer Systems*, Digital Press, 1984.

[32] Lipovski, G. J., Malek, M., *Parallel Computing - Theory and Comparisons*, John Wiley and Sons, 1987.

[33] Lusk, L. E., Overbeek, R. A., *A Minimalist Approach to Portable Parallel Programming*, in [Jami87a], pp. 351-362.

[34] Mikloško, J., Kotov, V. E., Editors, *Algorithms, Software and Hardware of Parallel Computers*, Springer-Verlag, 1984.

[35] Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1979.

[36] Quinn, J. M., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, Inc., 1987.

[37] Rus, T., *Data Structures and Operating Systems*, John Wiley and Sons, 1979.

[38] Yonezawa, A., Tokoro, M., (Editors), *Object-Oriented Concurrent Programming*, The MIT Press, 1987.