# Programming Languages For High Performance Computers

H.J. Sips *Delft University of Technology, Delft, the Netherlands*

Abstract

In this lecture we present high performance computing from a programmers perspective. Several approaches to programming high-performance computers are reviewed. Examples are given of extensions to FORTRAN for programming shared-memory systems as well as distributed memory systems.

Keywords: High Performance Computing, Parallel Languages

## 1 Introduction

Rapid advances in hardware technology have made possible computers which depart dramatically from the traditional von Neumann model of computation in that they exhibit very significant degrees of parallelism and very high levels of cost-effective “raw” performance. Such machines are now commercially available. One of the barriers to the widespread usage of these machines is the lack of user-friendly and efficient programming tools, compilers, and debuggers for developing applications on these high performance systems.

There are many new and difficult problems associated with programming high-performance computers, which do not concern those who develop software for conventional computers. Examples of such problems are:

- how to extract parallelism from a program description?
- given a target parallel machine, how should one represent and decompose a problem so as to obtain maximal performance?

In this paper, we will focus on the problem of making *programs* for high performance computers. This means that we view high performance computers through the eyes of programmers of these systems.

### 1.1 Basic approaches

Basically, there are three approaches to programming high performance computers:

- take an existing language and let all the work been done by the compiler;
- extend an existing (sequential) language with new constructs to represent parallelism;
- design a completely new language that incorporates parallel features.

The first approach is called the *implicit* approach. Indeed from a user perspective this would be ideal. Existing codes could run immediately on high performance platforms, without users having to worry about changing and optimising programs. This can lead to substantial savings in development costs and is also very attractive for vendors of high performance computers. Although there has definitely been some success in this area, there are some problems.

The first problem is that the full parallel potential of the problem cannot always be exploited. This is due to the fact that certain information known to the programmer is lost when coding the problem in a computer language or that the programmer has introduced spurious dependencies in the program, which inhibit the compiler from doing the necessary transformations.

The second problem is related to the architectural characteristics of the underlying system. The performance of a chosen algorithm strongly depends on the architecture on which the algorithm is executed. Many of the current algorithms are based on the characteristics of computer systems designed a decade or more ago. These systems had small memories and hence algorithms were favoured which put a limited burden on the memory capacity. Storage capacity of current

memories is not effectively used by these old algorithms. So, to profit from the advances in hardware technology, it turns out that these algorithms need to be replaced by newer algorithms.

In the second approach, existing languages are extended with new language constructs as to express parallelism. We call this the *explicit* approach. The clear advantage of this approach is that users have already been trained in using the base languages. To get their programs running efficiently, only a limited set of extra functions need to be applied. The problems with this approach concentrate on the sometimes difficult interaction between the base sequential language and the parallel features (debugging). Moreover, there is a lack of standardisation, meaning that a number of extensions has been developed with similar functionality, but different appearance.

The last approach is to develop a new language as to present a coherent approach to programming for high performance machines. This approach clearly implies recoding of the application software, which can be very costly and has many risks as well. In designing a new language one can choose an *implicit*, an *explicit*, or a *hybrid* approach.

## 2 Compiler-based detections and transformations

The first attempts to make programs ready for vector and parallel computers were directed to inspection and automatic transformation of complete programs. The argument for this approach was the existence of an enormous base of "dusty deck" Fortran programs, which represented a significant investment made in the past. Automatic analysis would protect these investments and would put little burden on the application developers. Let us take a look into some more detail at the ideas behind this approach.

**Vectorisation versus Parallelisation** The first commercial high performance computers were *vector computers*. The basic instructions in these computers are defined in terms of operations on vectors instead of scalars, while in languages like FORTRAN 77 the basic operations are defined on scalars. To use these vector computers efficiently, the scalar operations in a program written in such languages must be transformed into vector operations. To give a simple example, the program

```
DO i = 1, 3
   a(i) = b(i) + c(i)
END DO
```

must be translated in `a(1:3)=b(1:3)+c(1:3)`, which has the following semantics: Take from `b` and `c` the elements 1 to 3 (denoted as 1:3), add them element-wise, i.e. `a(1)+c(1)`, `b(2)+c(2)`, …, and assign the results to the elements 1 to 3 of `a`, respectively. We call this transformation *vectorisation*. We can see from this example that all element-wise operations can be performed independently of each other.

In vector computers, this operation is implemented by loading both `b(1:3)` and `c(1:3)` from memory, feeding them one element after the other to a pipelined adder and getting – after a few clock ticks delay – the results out element by element, which are then finally stored back in memory.

In vector computers, the data elements are processed in a stream by the operational units. An alternative way of getting speed up is to use a *parallel computer*. In a parallel computer, each processor is executing a part of the program. To be able to use a parallel computer, a compiler must identify which parts of a program can be executed concurrently. It is easy to see that each body instance of the above loop can be independently executed. Hence, each index can be assigned to a different processor without any order restrictions. We could assign `a(1)=b(1)+c(1)` to Processor 1, `a(2)=b(2)+c(2)` to Processor 2, etc., but any other order is also allowed.

To indicate the (possible) parallel execution of a loop, the keyword `DO` can be replaced by the keyword `PAR DO`, indicating that the iterations can be processed independently.

In essence, vectorisation is a very restricted form of parallelisation. Vectorisation can only be applied to single statements containing the basic operators `*` and `+` (some machines also allow division, logical, and character operations in vector mode). So the statement `a(i) = b(i)*c(i)` can be vectorised, but the statement `a(i) = SIN(b(i))` cannot be directly vectorised, because it includes the sine function.[1]
The reason for this restriction is that the hardware of a vector unit is only able to pipeline the basic operations `*` and `+`. Hence, vectorisation can be defined as an operation on one-dimensional data objects with a restricted set of operators.

**Automatic parallelisation.** So far the assertion that a loop can be vectorised or parallelised was done on a purely intuitive basis, that is to say we have assumed that "it is easy to see" that parallelisation or vectorisation can take place. However, a compiler is a machine and machines cannot work on such a basis. Therefore, algorithms are needed to determine whether or not transformations of the code are allowed. In doing so, it must be ensured that the semantics of the program remain unchanged by the transformations, i.e. the program produces the same results. This analysis is called *dependence analysis* [9].
The compiler must analyse the program in order to find the dependences in the program. It would be of great help if this were an easy task. However, it is not. In principle, the dependence problem can be formulated as finding the integer solutions to a linear programming problem. In practice, linear programming is too costly a technique for solving the dependence problem. Hence, in actual compilers faster approximation techniques are applied to detect the more simple cases. This implies that sometimes a conservative approach is taken by the compiler, meaning that a dependence is assumed, where there might not be one, if a more sophisticated analysis were applied.

The above suggests that if we were to do our best, the dependence problem can be solved. However, a more serious problem is that in a number of cases the program does not contain sufficient information to decide on dependences.
Concluding, fully automatic parallelization of programs is currently beyond the state of the art.

## 3 Languages with parallel features

To express parallel algorithms, some sequential base languages have been extended with parallel constructs. In this section, we will treat a number of models of parallel computation and show how parallel language constructs can be used to create parallel programs according to these models. We will use an example algorithm (MVP: Matrix-Vector Product) to illustrate various approaches.

### 3.1 Models of parallel computation: a conceptual view

In the world of sequential programming, there are many ways to code an algorithm. It took a while before people realised that to obtain well-structured and maintainable code, programs must be structured according to certain principles. This was called structured programming. For most application areas this was not enough. Higher level models were needed to keep the software engineering process under control. Nowadays, several design methodologies are available to support the application developer.

Parallel programming is now where sequential programming was two decades ago. We realise that models of parallel computation are needed to structure the design process of parallel

[1] However, library functions implementing these non standard operations might use vector instructions themselves.

programs. After having decided which model of parallel computation suits our algorithms best, we can choose an appropriate parallel language to implement the parallel algorithm.

As outlined before, models of parallel computation can be defined on many levels of abstraction. We will present two levels of abstraction here; one on the conceptual level and, in the next section, one on the systems level.

On the conceptual level, three commonly used models can be identified: the *Farmer/Worker* model, the *data parallelism* model, and the *function parallelism* model. The three models are described below.

**Farmer/Worker:** The idea behind the Farmer/Worker model of computation is that in a number of applications the work can be split in a number of jobs with identical programs, but with a different value of certain parameters (e.g. an index). The basic idea is to have a single program which is replicated and brought into execution through a number of identical processes, which are each assigned a piece of the work by a farmer process. When a worker process is finished, it asks the farmer process for more work. If no more work is available the computation is finished.

The number of worker processes can be smaller than the amount of work available and more than one worker process can be assigned to a processor. We can even bring up a Farmer/Worker model on a single workstation. What is chosen is purely a matter of efficiency and availability of hardware resources. For example, if each worker process has to perform a lot of I/O, it could be profitable, in terms of execution speed, to assign a number of worker processes to a single processor.

The Farmer/Worker model is very suited to express parallel computations in which the individual computational tasks are equal in terms of functionality and relatively independent of each other. In Figure 1a the Farmer/Worker model is illustrated.

**Data parallelism:** In the data parallelism model of computation, the subdivision of the data structures is the major driving force for obtaining parallelism. The programmer using this model, first makes a subdivision of all relevant data structures and then allocates each substructure to a processor (or process). The computations subsequently follow this subdivision and allocation, meaning that an assignment to an element of a data structure is done on the processor (or process) to which that element was allocated (i.e. which 'owns' the element). This assignment includes the computation of the right-hand side of that assignment statement. The principle is shown in Figure 1b.

**Functional parallelism:** In the functional model of parallel computation (also called Task parallelism), the basic computational blocks are functions acting on arguments and providing output to other functions (Figure 1c). Usually the computation is expressed as an acyclic task graph. Arguments can be scalar values or complete structures, such as arrays. The mapping of such a task graph onto processors can be done in different ways. The differences follow different imposed execution semantics. For example, two semantically different execution models are *data driven* and *demand driven* execution. In data driven execution, a function is executed as soon as all its input arguments are available. In demand driven execution, a function is only executed if its output argument is needed as input in a subsequent function.

### 3.2 Models of parallel computation: a systems view

The models of parallel computation of the previous section must somehow be implemented onto a high-performance computing system. One way of doing this is to define a parallel language in which these models are reflected, i.e. which allows a straightforward and easy way to program

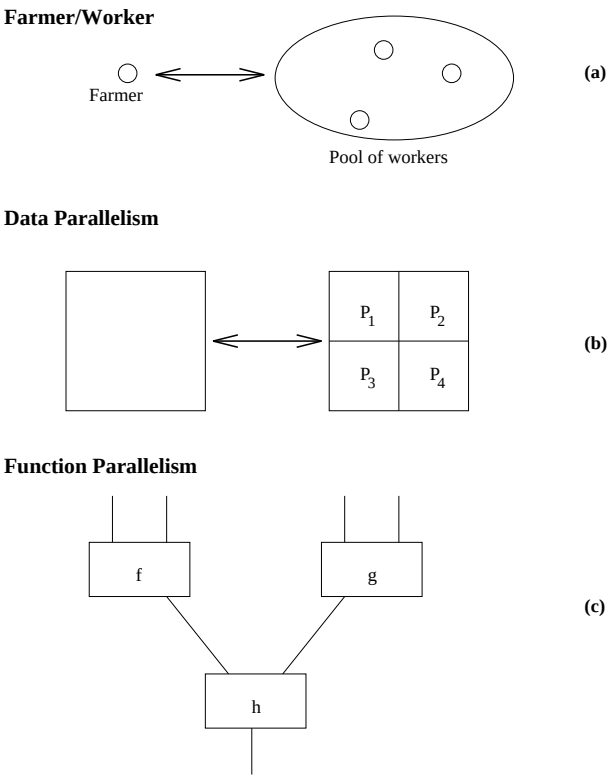


**Figure 1:** *(a) Farmer/Worker, (b) Data parallelism, (c) Function parallelism.*

according to these models. Some parallel languages indeed offer such a functionality. We will point this out when treating a number of parallel languages in the coming sections.

Another way of implementation is to use a lower level model of parallel computation. Usually, this model is defined on the systems level of high performance computers. To get some insight, it would be very handy to have a description of the elements of the model of computation on this level, to be able to identify the major characteristics of a high-performance system from an implementation point of view.

There already exists a generally accepted model classification of parallel machines [3]. In this model, four classes (SISD, SIMD, MISD, and MIMD) are distinguished according to the properties of machines having single or multiple instruction and data streams. However useful this classification might be, it gives too much a *machine view* on parallelism.

**Programs and Data.** Here we will try to make a similar classification, but now with a specific *systems* point of view. On the system level, we can distinguish model elements such as (logical) data spaces, programs, and communication and synchronisation methods, the latter defining the bindings between data spaces and programs. Considering only data spaces and programs, we define a model of parallel computation on the systems level with the following element definitions:

| | | |
|---|---|---|
| SD | : | Single Data space (no distribution) |
| MD | : | Multiple Data spaces (distribution) |
| SP | : | Single Program |
| MP | : | Multiple (different) Programs |
| * | : | Replication operator (multiple copies) |

A specific model of a parallel system is then defined by combining a program and a data space qualifier and an optional *. The data space qualifier indicates if data is considered to reside in a single common memory (such as in shared-memory systems) or is distributed among logically different memories (such as in distributed-memory systems).

The program qualifier tells us if the model allows for a single program or (functionally) different programs in execution. The * qualifier indicates if multiple copies of programs or data are allowed.

What does this mean in practice? To start with the most simple case: a sequential model is defined as SPSD in this system. A model defined as MPSD, allows the creation of multiple, functionally different programs acting on a single, non-distributed data space. Another classification we will encounter is SP*MD, meaning multiple, identical programs acting on multiple data spaces. Along these lines SPMD means a single program acting on multiple data spaces. The classical SIMD computer, such as the CM-2, conforms to this model, where a single program residing in a single controller acts on multiple (simple) processing elements, each storing a single data element.

**Communication.** The above models do not contain a specification of communication and synchronisation methods. A concise classification is hard to give, because many mechanisms have been proposed and implemented at various levels of abstraction. Regarding communication of variables, a rough subdivision is:

- shared variables
- communicated variables

In shared variables communication, programs have direct access to shared variables stored without interference of other programs. A number of methods are known to prevent concurrent update of shared variables from different programs, ranging from basic semaphores to critical sections and monitors [10]. Shared variable communication is not to be confused with shared-memory. Shared variables is a logical concept and shared variables may well reside in local memories.

In communicated variables, programs can only communicate through interference with other programs. Each program has a local set of variables. They can be made available by explicit communication between programs.

**Message passing.** A popular method is for implementing the communicated variables method is *message passing*. In message passing programs communicate by sending each other messages. In general, messages have the form:

```
send( <destination>, <variable_name> )

receive( <source>, <variable_name> )
```

where <destination> and <source> are process or processor identifiers and <variable_name> is the variable to be communicated.

For message passing, there exist two popular libraries: PVM (Parallel Virtual Machine) [7] and MPI (Message Passing Interface) [12], which can be linked to application programs.

# 4 Examples of Parallel Programming Languages

## 4.1 Shared-Memory programming

Almost all shared-memory computers, such as from CRAY and CONVEX, have facilities for multi-programming. However, most vendors offer proprietary extensions to standard programming languages. In an attempt to provide a common solution, a group of people

consisting of vendors, application developers, and researchers made a draft for a standard. This group was called the Parallel Computing Forum (PCF). After their work was finished the draft was handed over to the X3H5 committee of ANSI [4], where it is still under discussion.

Although the Parallel Fortran Standard (PCF-Fortran for short) is not yet widely implemented, it gives a good model for many of the currently available proprietary language extensions.

**Main features and execution model.** Basically, PCF-Fortran is an extension to FORTRAN 77 with the following features:

- Parallel do loops
- Parallel sections
- Synchronisation methods (e.g. Locks, Critical Sections)

Some of the features are only directives, i.e. they have no effect on the semantics of the program when omitted. Other features do change the semantics of the program and cannot be omitted without reconsidering the program.

A PCF-Fortran program begins executing like a normal sequential program. Execution proceeds until a *parallel construct* is encountered. Encountering a parallel construct represents an opportunity to begin parallel execution. A compiler might create at this point additional processes to execute the block of code defined by the parallel construct. The base process that encountered the parallel construct will resume execution when the parallel execution is completed.

Within a parallel construct, *work sharing constructs* are used to indicate work that is eligible for concurrent execution. Within a parallel construct, statements that are not contained within a work sharing construct are executed by all processes redundantly. When a base process encounters a parallel construct, a number of worker processes are started. Each worker process executes the same code until a work sharing construct is encountered. This work sharing construct hands out work to available worker processes. Some worker processes might finish earlier then others, or even have no work at all (are idle), dependent on the amount of available work and the number of workers created.

The basic parallel constructs in PCF-Fortran are the `PARALLEL DO` construct and the `PARALLEL SECTIONS` construct.
The `PARALLEL DO` construct is used to specify parallelism among the iterations of a block of code. It combines a parallel and work sharing construct. A prerequisite is that all iterations must be independent. Several options can be added to this construct. Adding `MAX PARALLEL` enables a programmer to control the number of worker processes. For example, `PARALLEL DO MAX PARALLEL = 5` in the loop body header limits the number of worker processes to 5.
The `PARALLEL SECTIONS` construct allows the specification of independent execution of subprograms. The `PARALLEL DO` construct clearly complies with the Farmer/Worker model, because each worker executes the same program, but on a different value of the index space. On the other hand, with the `PARALLEL SECTIONS` construct we can define a limited form of functional parallelism. The example with the two parallel sections shows that both models can be used intertwined.

Besides the two parallel constructs, PCF-Fortran has a number of synchronisation constructs to be able to enforce synchronisation among worker processes and to communicate variables. Synchronisation constructs include Locks, Events, Ordinals, and Critical Sections.

The use of explicit synchronisation constructs is error prone and need to be applied with great care. One can even make programs which deadlock or obtain non-deterministic results.

**MVP example in PCF.** How would a parallel version of a matrix-vector product (MVP) look like in PCF-Fortran? This is very simple. Parallelising the outer loop already gives a parallel program:

```
REAL a(m,n), y(m), x(n)
PARALLEL DO (MAX PARALLEL = 3) i = 1, m
   DO j = 1, n
      y(i) = y(i) + a(i,j)*x(j)
   END DO
END PARALLEL DO
```

In this example, each of the three worker processes is given a vector-vector dot product.

**Implementation of PCF.** PCF-Fortran lends itself for creating Farmer/Worker type of programs and programs with limited function parallelism, mainly in a shared-memory environment. The inclusion of low-level synchronisation methods in PCF also allows the creation of SP*SD or MPSD programs. As already remarked, programming by using these synchronisation methods must be done with great care.

## 4.2 Distributed-Memory programming

The last years an increasing number of high performance computers with non-uniform memory access times (often referred to as distributed-memory systems) and a large number of processing elements have appeared on the market. Most machines in this class use message-passing as the main means for communication. A programmer of these systems can make programs by writing a number of communicating sequential programs. Any data declaration is local to the program it is declared in and hence local to the processor the program is executed on. This model of computation can be classified as MPMD, since the user can develop multiple independent and functionally different programs which act on multiple different data spaces.

A disadvantage of this in principle powerful model is that programming is a tedious task with a high degree of complexity. This complexity can only be mastered by using a more structured programming model, such as Farmer/Worker or data parallelism. Especially data parallelism is generally viewed as a model very suited to program these highly parallel machines. Data parallelism can be made operational through an SP*MD model of parallel computation. An SP*MD program consists of a number of identical copies of a single program, each acting on a different subset of the data[2]. These data subsets are created by distributing the data objects among the available processors. SP*MD programs can be made by hand (composition) or generated from a higher level data parallel language (decomposition). Here, we will describe the approach from a higher level language.

Until now there was no generally accepted data parallel programming language. In January 1992, a group of people from industry and academia formed the High Performance Fortran Forum (HPFF) with as goal to develop a number of data parallel extensions to Fortran. In May 1993, the Forum came with a draft of what is called High Performance Fortran (HPF) [5] (now called HPF-1).

**Main features and mapping model.** The main features of High Performance Fortran are:

- a binding to Fortran 90
- data distribution features
- parallel statements

HPF is, with a few exceptions, based on the addition of *directives* or *annotations* to a normal Fortran 90 program. These directives do not change the semantics of the program and are merely hints to the compiler for optimal execution of the program. Ignoring the directives leaves a normal sequential program. Each directive is preceded by `!HPF$` or `CHPF$`.

[2] In literature, our SP*MD model is usually referred to as just SPMD, omitting the replication *.

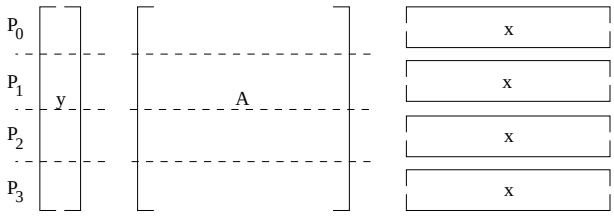


**Figure 2:** *MVP example in HPF on* 4 *processors.*

There are a few extensions which are no pure directives. One such extension is a new parallel construct: the FORALL construct.

The basic mapping mechanism is a specification of the distribution of the data structures of a program onto an abstract processor space. It consists of a three phase mapping. Arrays are first aligned to other arrays. The purpose of alignment is locality: two aligned array elements are guaranteed to reside on the same processor.
The top array is distributed on an abstract multi-dimensional *abstract processor space*. This abstract processor space is in turn mapped onto a *physical processor space*. HPF only concerns the first two mappings. The mapping from abstract processor space to physical processor space is left to the compiler implementer. The best way to give an idea about programming in HPF is through a simple example.

**MVP example in HPF.** In HPF there are also many ways to code the MVP operation. Here, we will use a simple block-wise placement of both y and a, and a replication of x. The program can be stated as follows:

```
      REAL a(m,n), y(m), x(n)
!HPF$ PROCESSORS p(4)
!HPF$ ALIGN a(:,*) WITH y(:)
!HPF$ ALIGN x(*) WITH y(*)
!HPF$ DISTRIBUTE y(BLOCK) ONTO p
      DO j=1,n
        FORALL(i=1:m) y(i) = y(i) + a(i,j)*x(j)
        END FORALL
      END DO
```

In this program, we declare a processor space p of size 4. We align each row of a to an element of y, so row 1 of a resides on the same processor as y(1), etc. The vector x is replicated over all processors (for exact syntax definitions, see the HPF standard). Then y is distributed in equal consecutive portions over the 4 processors. In Figure 2 the placement of the arrays is illustrated. The actual parallelism is achieved by a FORALL statement in the inner loop. In a FORALL loop, the body instances can be evaluated in any order, but the complete righthand side of expression must be evaluated before any assignment can take place. Hence, the semantics of a FORALL statement are different than the PARALLEL DO construct in the previous section (see HPF standard).

**Implementation of HPF.** PCF-Fortran has an intuitive execution model based on the process notion. HPF does not come with such a model. The main reason for this is that it is extremely difficult to pin down exactly how HPF should execute on a specific machine, without giving up possible compiler, run-time system, and operating system optimisations.

In a sense, the directives for data alignment and distribution in HPF are nothing more than an assertion of the programmer about the locality of operations on that data. How the underlying implementation system makes use of this, is a matter for the implementer and vendor of the system. Because of this, nothing can be said about the conformance of a compiler to the HPF

definition, except for the fact a compiler is HPF conforming if the directives are parsed correctly. The real test can only come from comparative benchmarks.

Still, with the above remarks in mind, one can give some indications of possible implementation models. On distributed-memory machines, a viable model is SP*MD, in combination with the *Owner Computes* rule. In this model, the data is distributed among the local memories along the specified distribution functions (hence MD). The code part of the program is replicated, with the premise that assignment to a variable and the calculation of the righthand side of the associated assignment statement is done on the processor to which that variable is allocated by a distribution statement.

## 5 Other developments and further reading

Besides Fortran, also other base sequential languages have been extended with parallel constructs. Examples are C and C++ (for references to these languages see [1]). The use of sequential base languages gives its own problems and therefore alternatives have been sought. Examples of languages which specifically have been designed with parallelism in mind are Occam [6] and Linda [8]. However, the success of introducing new languages has been very limited as up to now.

For books on programming high performance computers, the reader is referred to [1, 2, 10, 11].

There are a number of scientific journals devoted to the subject of parallel and distributed processing. Among these are the IEEE Transactions on Parallel and Distributed Processing, Parallel Computing, Journal of Parallel and Distributed Computing, Journal of Supercomputing, and Concurrency: Practice and Experience. For the user community an attractive magazine is Parallel and Distributed Technology from the IEEE.

## References


1 I. Foster, Designing and Building Parallel Programs, Addison Wesley, 1995.

2 A. van der Steen (Ed), Aspects of Computational Science: a textbook on high performance computing, NCF, the Hague, 1995, ISBN 90-70608-33-2.

3 M.J. Flynn, “Some computer organizations and their effectiveness,” IEEE Transactions on Computers, Vol.21, September 1972.

4 ANSI X3H5, FORTRAN 77 Binding of X3H5 Model for Parallel Programming Constructs, Draft Version, September 1992.

5 -, HPFF, High Performance Fortran Language Specification, Version 1.0, Rice University, May 1993.

6 C.A.R. Hoare, Communicating sequential proceses, Prentice Hall, Englewood Cliffs, NJ,1985.

7 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM 3 user’s guide and reference manual, report ORNL/TM-12187, May 1993.

8 N. Carriero, D. Gelernter,“How to write parallel programs: a guide to the perplexed”, Communications of the ACM, Vol. 21, no.3, 1989.

9 M. Wolfe, High Performance Compilers for Parallel Computing, Addison Wesley, 1995.

10 R. Perrott, Parallel Programming, Addison Wesley, 1987.

11 T.G. Lewis, H. El-Rewini, Introduction to parallel computing, Prentice Hall, 1992.

12 Message Passing Forum, “MPI: A message passing interface standard,” International Journal of Supercomputer Applications, Vol.8, no.3/4, 1994.