Parallel computing - an approach for scientific computing

ION T. CĂRSTEA
Faculty of Automation, University of Craiova, Craiova
Str. Doljului 14, bl C8c, sc. 1, apt. 7, CRAIOVA
ROMANIA

Abstract: - This paper presents an overview of parallel computing as an approach for high-performance scientific computing. We present some aspects in development of the parallel algorithms and parallel programs with emphasis on SIMD and MIMD computers. Some fundamental concepts are presented in the context of development of parallel algorithms and programs with emphasis on the programs efficiency. A comparison of different strategies is analysed.

The fundamental problems of the parallel computing and their solutions are presented. These aspects are treated in the general case related to the portability of algorithms across various parallel architectures

Key-Words: - Parallel computing; Engineering education.

1 Introduction

Why parallel computing? Because the sequential computers are not normal models of the real-life activities. A computer is an electronic model of the real world and the real world is dominated by parallel activities. The people, the actors of the world scene are not isolated actors. They enter in competition and they cooperate in some moments. From the activities of the big companies to the family activities the people do the work together and communicate for a common goal. For example, in a kitchen you say to your wife: “while I cut the bread, please prepare the salad or soup”. Many activities are done simultaneously and the parallel computers are models of these activities. Man thinks sequentially but in the society his activity is done simultaneously with the activities of other people. We must not forget that man has some organs that work in parallel. An illustrative example is the human eye. This is a real motivation to study the normal activities and their models reflected by the new parallel architectures and the problems arised by the programming of these architectures.

With the general-purpose parallel systems available on the market and the extremely short life cycles of computer hardware, the development of software products for the new architectures is a necessity. The commercial products must respond to some requirements: the utility, compatibility and portability. The utility is a natural requirement. The compatibility is another important requirement in the sense that a parallel version of a problem solution must return the same results of a serial version but in a significantly shorter period of time. The portability ensures a quick and easy migration of a software product to system in use and permits the software product to keep pace with evolution of the hardware products.

Parallel computing is currently an area of intense research activity, motivated by a variety of factors [7]. Some of them are the followings.
- the need for the solution of very large computational problems
- the technological progress.

1.1. Concurrent processes

A fundamental problem on any conventional computer is the concurrent execution of a number of processes [11]. This problem appears in parallel computers because the number of processors can be smaller than the number processes of a program. This must be done by interleaving the execution of the processes, so that one runs for a short time, then another runs for a short time, and so on. The problem splits into two parts [10]:
- Process switching - the mechanism by which the currently running process is suspended and another process started.
- Scheduling - the algorithm which decide when to switch processes and which process to run next.

Mutual exclusion is one of the most important and fundamental synchronisation problem [2]. To understand the problem we must introduce some concepts.

A critical section is part of a process that must not be executed concurrently with a critical section of another process, belonging to the same working set [5].
A class of critical sections is a set of critical sections, all of which must be mutually exclusive with all the others in the same class.

We say that mutual exclusion applies between critical sections of the same class when one process has entered its critical section, another process may not enter its critical section until the first process exits its critical section. Critical sections from different classes need not be mutually exclusive.

There is a large literature for the algorithms that implement mutual exclusion and it isn’t the goal of this work to present them [4]. The readers are invited to read the rich literature in this area.

A resource is an abstract concept. It can be anything that can block a process if it does not exist or it is not available. Resource can be a physical object (memory, processor, input-output devices etc.) or a logical object (time of the processor, memory space etc.).

2 Models of parallel computation

Two distinct classes of parallel machines arise from two different execution strategies: SIMD and MIMD, discussed in a precedent chapter. In addition to these two standard classes of parallel computers, another class is used for designing parallel algorithms: systolic machines. It is realised in very large-scale integration (VLSI) technology.

The conventional computers are implementations of the same computation model called John Von Neumann model. The model is sometimes called the RAM model (Random Access Machine). A parallel computer consisting of a large number of processors to solving a single problem at a time is based on different model of computation.

The underlying model of computation used in a machine has a fundamental role in design and analysis of a parallel algorithm because the design and performance of the algorithm depends very much on the architecture. In serial computation, there are standard models of computation such as RAM model and Turing machine. For parallel algorithms there are a considerable number of models related to the three classes presented above. In this section we present some basic models for parallel computation.

When we talk about parallel computation we must consider four levels of analysis: machine, operating system, language and algorithm. So that we have four levels of parallel models: architectural model, operating system model, high-level language model and algorithmic model.

Architectural models are used to include a parallel computer in one of the categories of parallel computers. Flynn’s taxonomy is widely used but there are other criteria to classify the computers [10]. For example, using the interprocessor communication there are two large classes of architectural models: shared-memory model, and distributed-memory model. Programming models are very close to architectural models. They are used to present the problem parallelism and sometimes are used instead of the algorithmic models. Algorithmic models are useful in analysis of parallel algorithms although there is no universal model. In general the algorithms are developed for a particular class of architectures.

We limit our discussion to some architectural models based on Flynn’s taxonomy so that we have two classes of parallel computers: SIMD and MIMD. Different models of computation are used in each class obtained by choosing different interprocessor communication strategies [11].

SIMD class. The most popular models for this class are: shared memory, mesh connected, cube connected, perfect shuffle and tree connected. Let us consider see the shared memory SIMD computer. It can be viewed as a parallel variant of the RAM; for this, the model is referred to as Parallel Random Access Machine or PRAM model.

MIMD class. The MIMD models can be characterised by three important attributes: (1) the number of processors - fixed or unlimited; (2) the processor operation that may be synchronous or asynchronous; and (3) the interconnection network. In general, MIMD machines may be divided into two distinct classes: MIMD machines with shared-memory (or tightly coupled machines), and MIMD machines with distributed-memory (or loosely coupled MIMD machines).

In the class of MIMD with shared-memory there are some variants of model computation as: (1) the models with uniform memory access; (2) the models with nonuniform memory access, and (3) the models with memory access by cache (Cache Only Memory Access).

In general, the algorithmic models were developed for shared-memory machines. The common global memory is a useful concept from the programmer’s viewpoint, especially in the first phase of parallel algorithm design. The algorithmic models are used as general frameworks for describing and analysing parallel algorithms. These models must satisfy the following requirements:

- Simplicity. This requirement is useful for performance analysis of an algorithm. More, the model should be hardware-independent.
• Implementability. The model must be easily implementable on parallel architectures.

2.1. Programming models
There are several recognised paradigms used for the development of efficient sequential algorithms [9]. These paradigms are not algorithms, rather they are the problem-solving strategies that are used in developing efficient algorithms. The purpose of identifying paradigms in parallel computation is:

• to have an algorithmic skeleton for a large class of problems
• to have a structured management of parallel computation.

In sequential computation, the paradigms generally encapsulate information about useful patterns of data reference. In parallel computation the paradigms generally encapsulate information about useful communication patterns. In this way the paradigms are useful for a comparison of different communication patterns.

Directed acyclic graph (or DAG represents a general class of models of computation in short). A DAG can represent a linear algorithm (without branches). The model of computation is independent of architecture.

Some well-known paradigms for parallel computation are binary-tree paradigm, growing by doubling paradigm; spanning tree for graphs paradigm, divide-and-conquer, computer-aggregate-broadcast, systolic paradigm etc [12].

A. Binary-tree paradigm. This paradigm can be applied to performance computation on N data items (n1, n2, ... nN). If the computation is such that it can be performed by combining the results of the same computation on data items (n1,n2, ...n(N/2)) and (n(N/2)+1, ... nN). Assuming that the data lies on the leaves of a binary tree, the computation proceeds from the leaves to the root of the tree and the entire computation is terminated when the processor at the root performs its computation.

B. Growing by doubling paradigm. The paradigm is used for graph theoretical problems. Let us consider a problem represented by a binary tree structure. The binary tree paradigm can be viewed in another way. At each step a processor doubles the number of elements for which it has performed the required computation, that is the growth is achieved by doubling the number of data elements at each step.

C. Spanning tree for graphs paradigm. This is a common technique for constructing parallel solution to graph theoretic problems by using an arbitrary spanning tree.

D. Divide-and-conquer. In this strategy a problem is divided into more subproblems. Each of these subproblems is solved independently and their results are combined to obtain the overall result. In a parallel computation, the subproblems can be solved at the same time if sufficient processors are available. If the subproblems are smaller instances of the original problem, it is possible to obtain a recursive solution.

E. Computer-aggregate-broadcast. This strategy consists of three basic phases:

• compute phase that performs some basic computation;
• aggregate phase that combines local data;
• broadcast phase that returns the combined data to each computing process.

The compute phase depends by the algorithm. The aggregate phase is usually binary-tree-like computation that combines all the values at the leaves into a single value at the root of tree. The broadcast phase broadcasts this combined data by means a broadcast message or a shared memory location to all other processes.

F. Systolic paradigm. The systolic strategy is used for nonshared memory parallel computation. It consists of decomposition of the entire computation into subcomputations that are assigned to dedicated processors. The data flowing through the processors visit an appropriate subset of processors to complete the computation for particular input data. Systolic algorithms have some properties like (1) locality of communication, and (2) a regular communication structure.

3 Parallel algorithms
There is no general design method for developing the parallel algorithms to any specific problem. The act of algorithm design is considered an art and may never be fully automated. Every possible algorithm provides a clear specification of the architecture used the data structures chosen and the mode of synchronisation and communication as well as the computation and communication complexities. The most known strategies of design are [4]: divide and conquer, greedy method, dynamic programming, search and traversal method for problems involving graphs and trees, backtracking method, branch and bound method, algebraic transformation, random trial methods, approximation methods, etc.

A basis on which it can choose among algorithms for the same problem is the mathematical computational model that enables to assign a cost to
executing a given algorithm on a specific problem instance. It must not forget that the best algorithm is not fastest, in other words given a good algorithm for a problem it is tempting to design a special-purpose computer containing just the hardware needed to implement the algorithm effectively. The debate of this aspect of the special-purpose vs. general-purpose question is opened because many optimal algorithms have not yet been discovered for most problems.

A main issue is the choice of a suitable architecture for implementing a given algorithm. The kind of parallelism of the computer, namely it is SIMD or MIMD, essentially influences the nature of the algorithm. The standard architectures that lead to two classes of parallel algorithms, are grouped in two major categories [11]:

(i) synchronous computers that have either a central control (SIMD machines) or a distributed local control mechanism (as systolic processors);

(ii) asynchronous computers. These computers are MIMD machines, composed of a number of independent processors sharing the primary memory by a communication network.

In the multiprocessor systems, the performance of the computation is influenced by some factors as:

- the type of problem parallelism;
- the methods used for decomposing a problem into tasks;
- allocation of tasks to microprocessors;
- granularity of the tasks;
- the possibility of overlapping communication with processing;
- data-access method;
- the architecture of the host system;
- the speed of processors, memories and interconnection work.

The performance of a computer program depends both on the method used to solve a specific problem -known as the algorithm, and on the skill with which the algorithm is implemented on the computer by the programmer or compiler during the operation of coding. The best way to obtain high-performance code is obtained if the parallelism in the algorithm matches the parallelism of the architecture so that the details of programming for any particular architecture are issues that must be considered in an efficient algorithm.

3.1. Problem parallelism

Problems vary widely in the degree to which parallelism is evident in the problem itself. Thus, we can have some categories of parallelism [11]: obvious, discovered and implicit parallelism.

Obvious parallelism is evident in the problem by its nature and it is encountered in the problems involving large regular data spaces. If the problem is inherently parallel, it is simpler and more natural to follow the parallelism in its solution.

Discovered parallelism. In other problems we must seek the parallelism in an approach to a solution. Typically, the algorithm for problem solution is chosen for specific computer architecture.

The implicit parallelism is detected and extracted automatically by the compilers. The user programs are developed in a language without explicit parallel features. The advantage of this type is the portability.

The parallel implementations of the applications can be included in one of the following categories:

- data parallelism or homogeneous multitasking;
- code (function, algorithmic) parallelism or heterogeneous multitasking;
- irregular parallelism.

In the first category the parallelism is implemented by data partitioning. In the latter, the parallelism involves the generation of parallel code. The implementation can be realised either in design-time and is called static parallelism, either in run-time and is called dynamic parallelism. In the third category the dominant aspect of the problem is the irregular structure so that the program structure evolves dynamically as the program executes and cannot be known a priori. There are some common solution paradigms for the each category of the parallelism and some of them we present in brief.

Farm parallelism. This paradigm applies to problems whose data space is partitioned into many pieces, called packets, that can be process independently. A master, or farmer distributes the packets to each of several identical worker processes and the results of processing are returned to master. The communications are only between the master and workers.

Geometric parallelism. If there is a geometric structure of the data space, it is naturally to divide this space in regions and to assign each region to a different processor. This is the case when computation for a region may depend on that of their immediate neighbours. Usually the same program is run on each processor and it isn't necessary that the processor topology must match the problem dimensionality.

If the structure of the data space is complicated, we have a long-range parallelism. Each processor (home processor) sends a packet to all processors carrying the information from the home processor and the packet is returned with updated state by accumulating the effects of all processors.

Algorithmic parallelism. Not only the data space can be partitioned but also the program can be split
into many tasks. This parallelism has an unregular structure.

The first two types represent the data parallelism and can be realised, like the algorithmic parallelism, either in design-time, either in run-time.

3.2. Parallel algorithms
An algorithm is a step-by-step method for solving a problem. One of the most important requirements of the algorithm is to complete in a finite time. Since the computation time depends on the performance of the host computer, a main objective was and is to increase the speed of the computers. The current demands of high-speed computing cannot be satisfied only by using faster hardware, but the parallel computers, that is computers that have a number of processors that cooperate.

In the last few years a new classification of the algorithms has become more important: if the algorithm is serial or parallel. The difference between such algorithms has become highly significant because of the development of parallel computers. The basic difference is that the programs using p processors should be p times faster than programs using only one processor although the experience and theory prove that the real speedup is less. The development of parallel computers involves the development new efficient algorithms. The structure of these algorithms and their software are inherently dependent on the architecture of the computer system used and vice versa.

The new architectures present a great challenge to sequential-algorithms designers: they must design parallel algorithms. In this area there are three possible scenarios:

1. To detect and exploit any inherent parallelism in an existing sequential algorithm.
2. To invent a new parallel algorithm.
3. To adapt another parallel algorithm that solve a similar problem.

Obviously, the first strategy is attractive for the user but it is not effective. We must not forget that for a sequential algorithm, the running time and the memory space are the main measures of complexity. These two measures are also important for parallel algorithms but there are some additional measures of complexity for parallel algorithms, one of them being the number of processors required.

Some sequential algorithms have no obvious parallelization so that a parallel algorithm made from a sequential algorithm will exhibit poor speedup. The third scenario is often possible because there are a lot of general techniques that can be used in the solution of similar problems.

Introduction of parallelism in computation enhances the speed of execution but the degree of speedup is limited. New problems appear in performance analysis of parallel algorithms: communication, synchronisation and other restrictions (for example, a limited number of processors available).

4. How to design parallel algorithms
A fundamental issue of both theoretical and practical importance is related to the portability of algorithms across various parallel architectures. In a graph approach of the design, analysis of portability naturally leads to the simulation or embedding of one set of graphs by another. The two parameters characterise the quality of an embedding [12]: expansion and dilation. Expansion relates to the hardware overhead in terms of the number of processors needed, and dilation to the communication overhead involved in simulating one graph by the other.

As we have mentioned, the parallel machines may be divided into two categories: SIMD machines, and MIMD machines [10].

Different models of these computer categories are obtained by choosing different interprocessor communication strategies and these models influence the design of a parallel algorithm.

The synthesis and analysis of a parallel algorithm can be carried out in the following assumptions [12]:

• bounded parallelism with the computational model consists of p processors
• unbounded parallelism when we can use an unlimited number of processors.

In practice the first case is preferable because in many applications the number of processors is limited and is no more than the problem size. However the unbounded parallelism is of a great theoretical interest and becomes useful if it can be transformed to bounded parallelism.

Transformation of the unbounded parallelism in bounded parallelism can be done in two ways:

• decomposition of the problem
• decomposition of the algorithm

In the first approach, splitting it into smaller subproblems, each of which is then solved by the original algorithm using a smaller number of processors, decomposes a problem. When we use decomposing the algorithm, each of its steps is decomposed into substeps in such a way that each of them can be executed using a smaller number of processors.
4.1. Techniques for improving the efficiency

In executing any parallel algorithm there is always a trade-off between the computation time and the number of processors used by the algorithm.

A parallel algorithm is said to be adaptive or self-reconfiguring if for subsets of processors available the algorithm can be executed and the cost of the algorithm is constant, that is the product of time and number of processors remains constant [12].

The following results are presented without proof ([11], [12]):

T1. Any parallel algorithm of time complexity O(T) with p processors must have at most O(Tp) elementary operations.

The result is a consequence of the fact that a parallel algorithm that solves a given problem in O(T) time with p processors, yields a serial algorithm that solves problem in time O(Tp).

T2. (Brent’s theorem). Let us suppose that an algorithm that consists of e elementary operations can be performed in time T with a sufficiently number of processors. Suppose that ei denotes the number of operations performed at step i. In these conditions, with p processors, the computing time satisfies the following relation

\[ \sum_{i=1}^{T} \left( \frac{e_i}{p} \right) + T \leq \sum_{i=1}^{T} (\left( \frac{e_i}{p} \right) + 1) \leq \left\lceil \frac{e}{p} \right\rceil + T \]

An immediate consequence is the following:

T3. Any parallel algorithm of time complexity O(T) using a sufficiently large number of processors that consists of O(e) elementary operations can be implemented by p processors with a time complexity of O(\([e/p]+T\)).

By \([\cdot]\) we understand the round function.

The following result gives a way to make algorithms adaptive when the number of processors available decreases.

T4. Any parallel algorithm of time complexity O(T) with p processors can be implemented by \([p/k]\) processors, \(1 \leq k \leq p\) in time O(kT).

In some cases, when the number of processors decreases, the reallocation of the computation to processors can be performed in such a way as to increase the efficiency of the new version of the algorithm. There are two approaches to increase the efficiency:

- to reduce the number of processors without increasing the time complexity;
- to reduce the time complexity without increasing the number of processors.

5. Techniques for parallel software design

Efficient parallel programs are more difficult to write and understand than efficient sequential programs, because the behaviour of parallel programs is nondeterministic. The parallel programs depend critically on specific architectural features of the underlying hardware (such as the way in which data sharing memory access, synchronisation and process creation is handled). They are, in general, less portable than sequential programs.

There are two possible approaches to implement parallel processing. The first and obvious approach is to develop completely new programs based on new algorithms that are suitable for parallel processing. Another approach that is attractive to users is to let a sophisticated compiler expose and exploit parallelism from an existing sequential program.

In first scenario it is difficult to write a program for a parallel computer. There are some motivations for this:

- Difficulty in exploiting parallelism. It is not an easy task for a programmer to divide a program into a set of tasks that are executed in parallel.
- Difficulty in utilising processors efficiently.

In writing a parallel program we must fulfil two requirements ([1], [3]):

- The processor load balance.
- The data communication time between processors. Processes should be allocated to processors so that the time to communicate data between processors is minimised.

5.1. Parallel execution of the sequential programs

A program can be regarded as a collection of statements, the ordering and scheduling of which are constrained by information dependence. There are two types of dependence: data and control dependence. Data dependence exists when statements compute data that are used by other statements. Control dependencies arise from the ordered flow of control in a program. On a computer that supports concurrent operations, dependencies can limit the full utilisation of the machine.

Serial programs written in an existing language may be analysed to discover how much parallelism they contain and how they can be transformed into programs for parallel computers. The task of analysing
sequential program is worthwhile because of the huge number of existing programs that have been written. There are two factors which influence the type of parallel processing: the type of machine on which we are going to be doing the processing and the basic data structures in the program. The problem is how best to exploit the parallelism found in a program and if this parallelism can be exploited to good effect on all types of parallel architectures.

A particular class of sequential algorithms is the class of numerical algorithms. The basis of numerical methods is the evaluation of arithmetic expressions. The path followed by the expression evaluation can be represented by graphs (trees). This approach can lead to the height reduction whose possible utilisation is dependent on the type of computer considered. In general this approach is a problem of mapping a given arithmetic expression into an equivalent expression that can be performed in a parallel computer.

5.2. Programmer's viewpoint

There are three possible scenarios for a programmer that have to get a program to run effectively on a parallel processor ([8], [11]):

1. Do nothing. In this scenario the programmer does nothing new at all and a parallelizing compiler extracts parallelism from programs written in existing serial languages, that is we have an "implicit parallelism".

2. Explicit parallelism. In this scenario, the programmer uses a serial language augmented with a few new constructs that allow the programmer to specify and properly coordinate the parallel execution of a number of parallel processes. These languages are extensions of the Von Neumann languages as Fortran, C, etc.

3. New language. Write the program in an entirely new language designed to make parallelism easier to detect and extract. This scenario involves a substantial relearning.

All three approaches are influenced by a single concept, namely "side effect". A language with side effects support the concept of a global variable that can be modified by many parts of the program other than the part in which it appears. This concept can limit the amount of parallelism can be extracted automatically in the first scenario. In the second scenario programmer helps the compiler to identify the side effects, and in the third scenario the side effects disappear.

In the development of parallel programs there is no universal strategy but some activities are common. Given a specification and target architecture, a programmer's task is to derive a program, select a mapping that maps program on a given architecture, and to evaluate complexity measures.

An algorithm for writing a parallel program consists in the following steps ([3],[11]):

1. choose the paradigm that is most natural for the problem;
2. write a program using the method that is most natural for that paradigm;
3. if it is necessary, transform it into a more efficient version that uses another method.

In this approach three paradigms can be used: result, agenda and specialist parallelism [1]. Result parallelism focuses on the shape of the final product, agenda parallelism focuses on the list of tasks to be performed, and specialist parallelism focuses on the make-up of the work team. The choice of the paradigm depends by the problem and in many problems we must use a combination of these paradigms.

6 Performance limits in parallel computing

The evaluation of parallel computer performance is quite complicated. Different problems have different possibilities to be parallelized and some problems can not be parallelized. The fact that the same problem can be solved in different ways and the programs for the same problem are different, it is necessary to find some measures to compare them. To define the performance measure is an open problem. When we must evaluate the performance of parallel computing we must keep in the mind three aspects: architecture of the host computer, algorithm and software [6].

There are a number of different ways to characterise the performance of both parallel computers and parallel algorithms [10]. Usually, the performance of a computer is expressed in units of millions of instructions executed per seconds (MIPS) or millions of floating point operations per second (MFLOPS). In practice, the realisable performance may be far lower than these measures. The actual realisable performance is a function of the match between algorithms and the architecture.

A practical approach to evaluate and compare different parallel computers is to run a large number of benchmark programs on them and note the difference in total time execution, memory use and other factors (hardware and software).

In the professional literature we meet the terms like complexity of algorithm, complexity of the program and the complexity of problem [6]. These
concepts are used both in analysis of the sequential programs where we use the terms as time and space complexity and in parallel programs. Suppose that our software product is a program. One way to measure its time efficiency is the running the program through a particular compiler, on a particular computer with a particular input, and the measure the actual processing time required. In this scenario, efficiency is regarded as an external product measure because we measure it indirectly in terms of the time of a particular process and depends on external factors. A real measure of the program performance is in terms of its underlying algorithm. In this case, the efficiency measure can be based on the number of operations required for a given input. For most problems, the inputs may be characterised by a single size parameter \( n \). In this case for an algorithm we must identify a small number of types of primitive arithmetic operations which are relevant for the algorithm. We could measure efficiency in terms of the number of operations required for a given input that is it depends by internal factors. Suppose that an algorithm for solving all instances of a particular problem requires \( f(n) \) computations. We say that algorithm is asymptotically optimal if for every other algorithm with complexity \( g \), which solves the problem, then \( f \) is \( O(g) \). We thus define the complexity of a given problem as big \( O \) of the asymptotically optimal algorithm for the problem’s solution.

In a parallel computer an important aspect in the performance analysis of an algorithm is the communication complexity. The problem complexity is complicated because in order to define it we must know all algorithms used in solving the problem, an impossible thing (many algorithms are unknown).

The complexity measures are introduced for evaluation of computational resources used by a parallel algorithm. Some useful measures are:

- the number of processors
- the execution time or the time complexity
- the communication time during the algorithm execution or communication complexity

We say that an algorithm is good if it exploits very well two resources: the time and the memory space. In other words the algorithm is good if the execution time and the memory space used are minimum. We talk about these aspects in terms of the time complexity and the space complexity. Communication problem can be viewed as a separate component of the algorithm complexity or can be included in the time complexity.

To define the performance of an algorithm we must define a measure of its quality [6]. In this area two important measures are well known: the speedup and the efficiency. Some remarkable results are presented in the professional literature ([5], [12]).

7 Conclusion

In this paper we presented practical aspects in development of parallel programs. Scientific computing of high-performance involves the use of parallel computers. We can not ignore the large computing power of the advanced architectures for solving large scientific problems. How to write parallel programs is an open problem so that a comparison between different strategies is necessary for a particular problem.

References: