33
1-25-91 JS ②

IS-M--657

CONF-9104 DE91 006674

A Structured Representation for Parallel Algorithm Design on Multicomputers

Xian-He Sun [*] Lionel M. Ni [†]

abstract

Traditionally, parallel algorithms have been designed by brute force methods and fine-tuned on each architecture to achieve high performance. Rather than studying the design case by case, a systematic approach is proposed. A notation is first developed. Using this notation, most of the frequently used scientific and engineering applications can be represented by simple formulas. These formulas constitute the *structured representation* of the corresponding applications. The structured representation is simple, adequate and easy to understand. They also contain sufficient information about uneven allocation and communication latency degradations. With the structured representation, applications can be compared, classified and partitioned. Some of the basic building blocks, called *computation models*, of frequently used applications are identified and studied. Most applications are combinations of some computation models. The structured representation relates general applications to computation models. Studying computation models leads to a guideline for efficient parallel algorithm design for general applications.

Index Terms:
Parallel processing, Multicomputer, Programming paradigm,
Structured representation, Computation models,
Communication complexity, graph

[*]Ames Laboratory, USDOE, Ames, IA 50011-3020
[†]Computer Science Dept., Michigan State University, E. Lansing, Michigan 48824

1 Introduction

Designing efficient parallel algorithms requires users to understand the performance characteristics of parallel machines and to modify their algorithm accordingly. These modifications are problem dependent. Therefore, parallel algorithms have had to be fine-tuned case by case to achieve higher performance. The painful, elusive design process has excluded casual users and restricted parallel computers to a rather small professional community. This situation needs to be changed to make parallel computers usable for other scientists.

We would like to reduce the burden of parallel algorithm design and make the design process more systematic. This raises the obvious questions: What are the techniques for developing efficient parallel algorithms? How could these techniques be used on a given application? To answer the first question, Nelson and Snyder [1] have proposed the concept of parallel paradigm and identified several paradigms. Paradigms provide good examples and may help users understand parallel computation. However, these paradigms are described verbally and are isolated from each other. How to connect these paradigms with general applications is unknown.

In our research, we approach these two questions from a different angle [2]. We study parallel algorithm design from a general point of view. First, a representation methodology, *structured representation*, is developed. With this representation, most of the frequently used scientific and engineering applications can be represented by simple formulations. These formulations are combinations of some simple data structures, called *parallel computation models*, and provide adequate information about performance degradations. Parallel computation models are the basic building blocks of structured representations. Since both parallel computation models and parallel paradigms are commonly used data structures, they share some similarities. A major advantage of parallel computation models over parallel paradigms is that computation models are based on mathematical formulations, and they are the constructing components of general applications. The design techniques used in computation models are the techniques needed for general algorithm design, and the design techniques are used in a similar way.

The parallel systems considered in this study are *multicomputers*. Multicomputers are message passing distributed-memory multiprocessors [3]. Multicomputers with hundreds or thousands of processors are available commercially. All first generation multicomputers adopt the store-and-forward communication mechanism and the hypercube topology. Second generation multicomputers have more advanced communication mechanisms. The structured representation proposed in this study aims at these new communication mechanisms. The generic architecture of multicomputers is depicted in Figure 1.

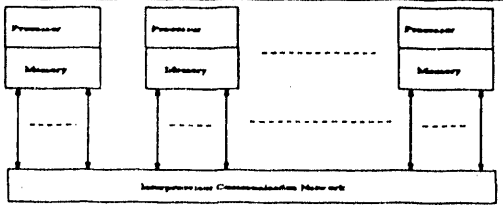


Figure 1: A Generic Multicomputer Architecture

2 Structured Representation

Parallelism can be achieved by dividing a given application into pieces, called *subtasks*, and solving these pieces concurrently. Ideally, these subtasks can be solved independently, where the exchange of intermediate result is negligible. Unfortunately, most applications do not have this easy parallelism property [4]. For most applications communication is necessary for exchanging data and coordinating activities. Although various asynchronous techniques have been designed to reduce the communication overhead, most communication must be achieved in a synchronous fashion, that is the receiving node must receive the communicated message before continuing. This synchronous communication requirement makes efficient algorithm design very difficult. The load needs to be balanced between synchronization and special care has to be taken to reduce the communication overhead. Figure 2 depicts the general parallel computation pattern with synchronous communication considered. It shows that the solving process consists of two phases, compute and data-exchange. These two phases occur alternatively and repetitively, and, therefore, form the *compute-exchange computation*. The data-exchange phase involves communication between compute phases. The communication patterns vary largely from application to application, and may be represented by a notation. To simplify this notation, we restrict ourselves to certain classes of communication, which are large enough for our purposes – describing the most frequently used scientific and engineering applications.

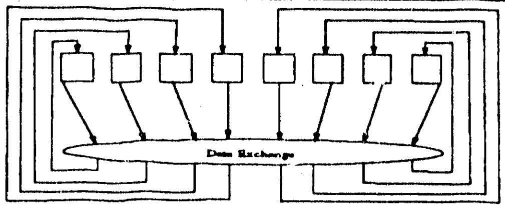


Figure 2: Compute-Exchange Computation

A processor sending a message in a communication is called a *sender*. A processor receiving message in a communication is called a *receiver*. A processor could be both sender and receiver in a communication. A *graph* $G(V, E)$ is a structure which consists of a set of *vertices* $V = \{v_1, v_2, ...\}$ and a set of *edges* $E = \{e_1, e_2, ...\}$ [5]. If we let processors in a communication be vertices in a graph and add directed edge (v, w) from v to w if processor v sends a message to processor w; a digraph (directed graph) is formed. This digraph is called the *communication digraph*. Following the notations of graph theory [5], the outdegree of a vertex v is the number of edges which have v as their start-vertex. In other words, the outdegree of a vertex v is equal to the number of destinations that v sends its message to. For this reason we also call the outdegree of a vertex the *degree of a sender*. The indegree of a vertex and the degree of a receiver are defined similarly. The *degree of a receiver* is the number of sources from which it receives messages.

Definition 1 *A regular communication is a communication in which all senders have the same degree and all receivers have the same degree.*

For a given undirected graph, if for every two vertices u and v there exists a path whose starting vertex is u and whose ending vertex is v, then the graph is connected. A connected subgraph $G(V', E')$ is a connected component if there is no other connected subgraph containing $G(V', E')$ as its proper subgraph.

The underlying (undirected) graph of a digraph is the graph resulting from the digraph if the direction of the edges is ignored. A connected component of a digraph is the corresponding subdigraph of the connected component of its underlying graph.

A connected component of the communication digraph is called a pattern of the communication.

Definition 2 *A regular communication is a regular data-exchange if it consists of one or more copies of the same pattern.*

By our definition, the communication requirement of a regular data-exchange is given by the number of patterns it contains. The pattern of a communication is described by the number of senders and the number of receivers in this pattern. The complexity of each sender and of each receiver is given by its degree. Thus, a regular data-exchange can be represented using five parameters as

$$P[S(D), R(d)], \tag{1}$$

where P is the number of instance of the pattern, S is the number of senders in each instance of the pattern, and D is the degree of the senders. Similarly, R is the number of receivers in each instance of the pattern, and d is the degree of each receiver. An example of using this notation for presenting communication is given in Figure 3. Notation (1) describes a communication by five parameters. Since messages must be sent one at a time. The number of times messages are sent and received is the dominant factor in communication cost. Notation (1) indicates the characteristics of a communication. More information may be needed when implementation is considered.

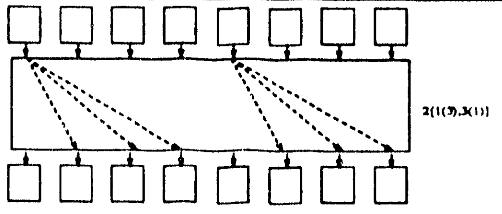


Figure 3: Multicast Data Exchange

The second class of data-exchange which we want to identify is called *conjunctive regular data-exchange.* We use the same five parameters to identify conjunctive regular data-exchange. The difference between regular data-exchange and conjunctive regular data-exchange is that in conjunctive regular data-exchange the patterns are not disjoint, they conjoin one another. Consider two special cases: conjunction at the sender side only and conjunction at the receiver side only. We have two general notations,

$$P[S(D), R_c(d)], \tag{2}$$

and

$$P[S_c(D), R(d)], \tag{3}$$

where the subscript, c, points out which side has conjunctions. An example of conjunctive regular data-exchange is given in Figure 4, in which the receiver side has conjunction.

A graph $G'(V', E')$ is a partition graph of $G(V, E)$ if $G'(V', E')$ is formed by splitting a subset of vertices $\{v_1, v_2, ... v_n\} \in V$ into two subsets of vertices as $\{v_1', v_2', ... v_n'\}$ and $\{v_1'', v_2'', ... v_n''\}$, where v_i', v_i'' are the vertices formed by splitting vertex v_i, and having edges (v_i', v_j') and (v_i'', v_j'') when edge (v_i, v_j) exists in graph $G(V, E)$. These divided vertices are called partition vertices. Figure 5 shows

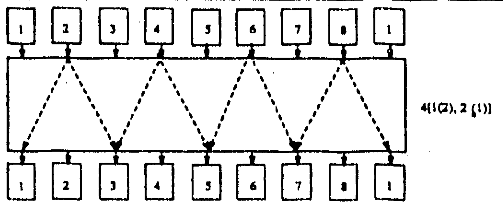


Figure 4: Conjunctive Data Exchange

two partition graphs of the given graphs. With this terminology, conjunctive regular data-exchange can be defined more mathematically as follows.

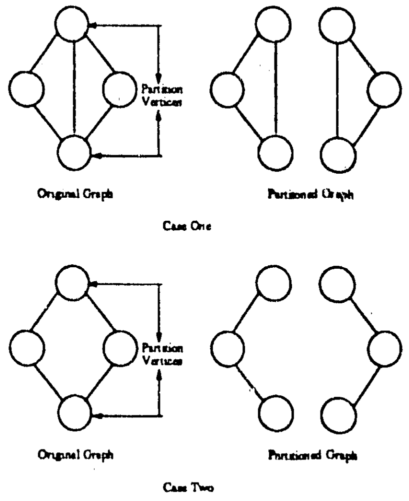
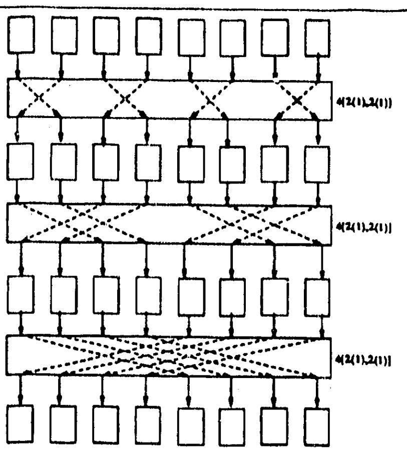


Figure 5: Partitioning of Graphs

Definition 3 *A regular communication is a conjunctive regular data-exchange if one of its partition graphs consists of one or more copies of the same pattern. If all partition vertices are senders, the conjunctive regular data-exchange is a sender conjunctive regular data-exchange. If all partition vertices are receivers, the conjunctive regular data-exchange is a receiver conjunctive regular data-exchange.*

Since a regular communication patterns could have more than one partition graph which consists of one or more copies of the same pattern, a conjunctive regular data-exchange could have more than one notation.

Once the data-exchange phase has been identified in an application, we can describe the application in terms of data-exchange. An application might have different data-exchange phases. Writing these data-exchange phases together in order by using the $\sum$ symbol and adding in the compute phases, we have a formula, called a *structured representation*, for each application. Figure 6 shows how to represent the Fast Fourier Transform (FFT) computation in terms of regular data-exchange. The communication divides the computation into layers. The total computation depends on the number of layers as well as the computation requirement at each layer. X_i is the computation work on each processor between data-exchange phase $i-1$ and i, if we have even allocation. X_i is the computation work of the processor which has the largest workload among all the working processors in the compute phase i, if we have uneven allocation.



$$\sum_{i=1}^{k}(2^{k-1}[2(1),2(1)] + X_i) \qquad k = log(N) = 3$$

Figure 6: FFT (Butterfly) Computation

We can see from Figure 6 that different communications could have the same data-exchange representation. The reason is that our notation is a high level notation. It provides the communication complexity for the data-exchanges. It does not contain detailed information about how the communication takes place.

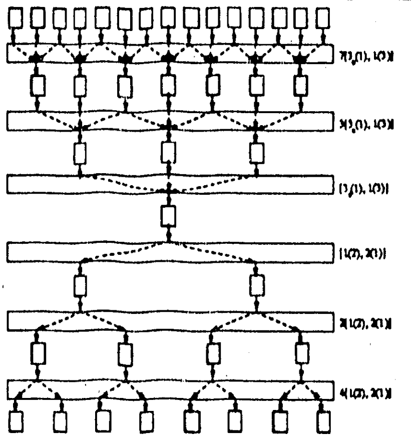
Odd-even cyclic reduction is a commonly used method for scientific applications. A well known parallel algorithm for tridiagonal linear systems is based on odd-even cyclic reduction [6]. From Figure 7 we can see that the odd-even cyclic reduction application contains two different structures. The upper half of Figure 7 is one structure and the lower half is another structure. This is a common phenomenon of scientific applications. Most of the frequently used scientific applications are combinations of a few simple structures, which we call *computation models*. The information in computation models can be used in general applications. Studying computation models will lead to a general algorithm design guideline for scientific applications.

3 Parallel Computation Models

Seven computation models are identified and studied in this section. They are *local computation model*, *global-exchange computation model*, *compute-aggregate-broadcast computation model*, *divide-and-conquer computation model*, *domain decomposition computation model*, *pipelined computation model* and *recursive doubling computation model*. The structured representations of these computation models are presented. We have found that various scientific applications are combinations of these models.

4 Conclusion

Traditionally, parallel algorithms have been designed by brute force method and fine tuned on each architecture to achieve high performance. A mathematical foundation is necessary in moving toward a systematic design methodology for parallel algorithms. In our study, a notation was first developed. Using this notation, most of the frequently used scientific and engineering applications



$$\sum_{i=1}^{k-1}((2^{k-i}-1)[3_c(1),1(3)]+X_i)+\ \sum_{i=k}^{2(k-1)}(2^{i-k}[1(2),2(1)]+X_i)$$

$$k=\lceil log(N)\rceil=4$$

Figure 7: Odd-Even Cyclic Reduction

can be represented by simple formulas. These formulas constitute the ***structured representation*** of the corresponding applications. The basic building blocks of these structured representation, called ***parallel computation models***, are identified and studied. Forming structured representation and acquiring computation models are the first step in developing such a foundation. Structured representations relate general applications to computation models. Studying computation models leads to a guideline for efficient parallel algorithm design for general applications.

References

[1] P. Nelson and L. Snyder, "Programming paradigms for nonshared memory parallel computers," in *The Characteristics of Parallel Algorithms* (L. Jamieson, D. Gannon, and R. Douglass, eds.), The MIT press, 1987.

[2] X.-H. Sun, "Parallel computation models: Representation, analysis and applications." Ph.D. Dissertation, Computer Science Department, Michigan State University, 1990.

[3] K. Hwang, "Advanced parallel processing with supercomputer architectures," *Proc. of the IEEE*, pp. 33–47, Oct. 1987.

[4] G. Almasi and A. Gottlieb, *Highly parallel computing.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[5] S. Even, *Graph Algorithms.* Computer Science Press, 1979.

[6] S. Johnsson, "Solving tridiagonal systems on ensemble architectures," *SIAM J. on SSTC*, vol. 8, pp. 354–392, May 1987.