UC Irvine
ICS Technical Reports

Title
Computer-aided programming for multiprocessing systems

Permalink
https://escholarship.org/uc/item/1481s0fv

Authors
Wu, Min-You
Gajski, Daniel D.

Publication Date
1988-06-30

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Archives
Z
699
C3
no. 88-19
c. 2

COMPUTER-AIDED PROGRAMMING FOR MULTIPROCESSING SYSTEMS

by
Min-You Wu
Daniel D. Gajski

Technical Report 88-19

Information and Computer Science
University of California at Irvine
Irvine, CA 92717

Abstract As both the number of processors and the complexity of problems to be solved increase, programming multiprocessing systems becomes more difficult and error-prone. This report discusses parallel models of computation and tools for computer-aided programming (CAP). Program development tools are necessary since programmers are not able to develop complex parallel programs efficiently. In particular, a CAP tool, named Hypertool, is described here. It performs scheduling and handles the communication primitive insertion automatically so that many errors are eliminated. It also generates the performance estimates and other program quality measures to help programmers in improving their algorithms and programs. Experiments have shown that up to a 300% performance improvement can be achieved by computer-aided programming.

COMPUTER-AIDED PROGRAMMING FOR MULTIPROCESSING SYSTEMS

Min-You Wu and Daniel D. Gajski

June 30, 1988

1. Introduction

High performance machines can be built using more than one processing element (PE). The main problem in multiprocessing, however, is not only how to build a system, but also how to use it. That requires development of parallel algorithms and programs that can be executed efficiently. There are three basic approaches to parallel program development. One school of thought believes that parallelization is a complex problem that can be only performed manually. However, programmers are error-prone and not very efficient in solving problems with hundreds of tasks. For example, system deadlock is the most common problem, and is difficult to detect once the program is developed. The second school of thought believes that a restructuring compiler will automatically restructure sequential programs into parallel programs. However, the parallelism revealed in this way is restricted by the algorithms embodied in the sequential programs. The third school believes in interactive program development with the assistance of computer-aided programming (CAP) tools. This approach recognizes that some tasks, such as algorithm design, are creative while others, such as task scheduling and synchronization insertion, are solved efficiently using CAP tools. These tools also generate performance estimates and quality measures to guide programmers in

improving their programs and algorithms. In this way, optimal performance can be obtained with increased productivity.

In this paper, we describe parallelization problems and a CAP tool for developing programs on message-passing systems. In section 2, we address some essential issues in parallelization and define several types of efficiency losses. The performance of a multiprocessing system is related to these efficiency losses. Two basic problems, namely, partitioning and communication, are discussed. Several methods to reduce communication overhead and processor suspension are presented. In section 3, we describe our program development methodology and the partitioning & merging (P&M) method. A CAP tool called Hypertool is introduced in section 4. Hypertool takes a partitioned program and generates parallel code for execution on hypercube machines. Hypertool also generates performance estimates and quality measures for the parallel code. A performance comparison of manually and automatically generated code is presented in Section 5.

2. Parallelization and Efficiency

2.1. Parallelization issues for multiprocessing systems

A multiprocessing system may be modeled as in Fig. 2.1. Several identical PEs are connected by a communication network with a certain topology. Each PE has a main processor (MP) and a memory module (MM). Some systems may have one or more communication co-processors (CCPs) in each PE to perform communication activities. Two types of multiprocessing systems can be identified by the types of communication schemes. In a shared memory system, communication between PEs is carried out via a shared memory. To exchange data, the producing processor stores data

into the shared memory, from which the consuming processor retrieves it. In a message-passing system, communication is performed by sending data directly from one PE to another, with no shared memory.

The success of multiprocessing technology depends on successful parallelization of application problems [GaPe85], which in turn, requires good partitioning of problems and minimization of communication between partitions. Parallelization problems include algorithm design, partitioning style selection, granularity determination, load balancing, and minimization of network contention.

The programmer must design an efficient parallel algorithm for the application problem. After algorithm development, the partitioning style and granularity are determined. For example, a matrix problem could be partitioned along matrix rows, columns, or blocks. The basic rule is to partition problems along dependencies in order to cut dependencies as little as possible and thus reducing the amount of communication between partitions. Granularity determines sizes of the partitions. Usually, fine granularity partitions have more parallelism and more dependencies, while crude granularity partitions have less parallelism and less dependencies. The choice of granularity is affected by the amount of dependencies and the communication overhead. Another source of performance degradation is unbalanced loading of PEs which causes some PEs to be suspended while waiting for data from other PEs. Finally, when the amount of communication exceeds a certain percentage of the network capacity, serious network contention may result. To reduce network contention, not only dependencies among tasks must be minimized, but also the tasks need to be mapped to PEs such that the network traffic is minimized. All of these parallelization problems cause efficiency losses.

2.2. Efficiency loss

For multiprocessing systems, the speedup is defined as $S = T_S \ / \ T_P$, where T_S is the execution time of the best sequential program, and T_P is the execution time of the parallel program. The efficiency is defined as $\mu = S \ / \ N$, where N is the number of PEs. The value of μ is usually less than one, indicating efficiency loss in parallelization. The efficiency loss may come from parallel algorithms, coding, and PE suspension.

a. **Efficiency loss from algorithm parallelization.** Although sequential algorithms may be applied to parallel systems, the best sequential algorithm may not be the best parallel algorithm. To exploit more parallelism, programmers should design new parallel algorithms instead of parallelizing sequential algorithms. A parallel algorithm may require more overall computation to solve a given problem than a sequential algorithm. The efficiency loss from algorithm parallelization is defined as $EL_A = T_A \ / \ T_S$, where T_A is the execution time of the parallel algorithm coded as a sequential program without any communication primitives.

For some algorithms, such as the Wave Equation and Matrix Multiplication [FJLO88], the same algorithm may be used for both sequential and parallel machines, and EL_A is equal to 1. However, some parallel algorithms are very inefficient. For example, the Jacobi parallel algorithm for Laplace equations is not as efficient as its sequential counterpart, the Gauss-Seidel algorithm [Jenn77]. The former converges much more slowly than the latter. Figure 2.2 shows the number of iterations for convergency of the two algorithms. Note that the number of iterations, and thus efficiency loss from the algorithm, may differ as much as one hundred or more.

b. **Efficiency loss from coding.** When a parallel algorithm is coded into a parallel program, overhead is introduced which causes efficiency loss. This overhead includes

communication overhead, PE initializations, selection statements, and duplication of operations. The efficiency loss from coding is defined as $EL_C = \left(\sum_{i=1}^{N} T_{R_i} \right) / T_A$, where T_{R_i} is the running time (not including the suspension time) of each PE.

Communication overhead includes time spent on message packing and initialization of message transfer on main processors. There are two kinds of message packing. One is to pack several elements of the same data type that are not stored contiguously. For example, Figure 2.3(a) shows the diagonal elements of a matrix being packaged and then sent. The other is to pack different types of elements. In Fig. 2.3(b), three different elements, force, position, and temperature are packaged and sent to another PE.

PE initialization includes getting identification parameters, setting topologies, and opening communication channels.

Selection statements are used frequently to select different code segments in each PE for boundary conditions. For example, each of the four code segments in Fig. 2.4 may or may not be executed on a particular PE.

Finally, to reduce communication overhead and suspension time, some operations may be duplicated on different PEs. For example, in Fig. 2.5(a), the statement $a = b * c$ is executed on one PE, and the result broadcasts to each PE. In Fig. 2.5(b), $a = b * c$ is executed on each PE, so that no communication is required.

EL_C for most problems increases with the number of PEs. If the parallel program is running on a single PE, EL_C is usually not equal to 1, since the overhead from PE initialization and selection statements still exists.

c. **Efficiency loss from processor suspension.** This efficiency loss is defined as $EL_P = NT_P \,/\, \sum_{i=1}^{N} T_{R_i}$, where T_P is the execution time of the parallel program on N PEs. Since $T_{R_i}+T_{B_i}=T_P$ (i = 1, 2, ..., N), where T_{B_i} is the total suspension times for PE i, $EL_P = 1+\left(\sum_{i=1}^{N} T_{B_i} \,/\, \sum_{i=1}^{N} T_{R_i}\right)$. Thus, the efficiency loss depends on the ratio of the total suspension time and total running time for all PEs. Processor suspension results from load imbalance and message dependencies. If the algorithm does not have enough parallelism, the efficiency loss from processor suspension may be great.

Load balance, in its normal sense, means an equal load for each PE. When the load of each PE is not balanced, a lightly loaded PE will become suspended. Even if all PEs are equally loaded, processor suspension can still occur due to message dependencies. For example, in Fig. 2.6(a), we divide a program into two equal parts without dependencies and load them onto two PEs, resulting a balanced load. However, if one part depends on the other, even though the load is "balanced", one of the PEs becomes suspended, as shown in Fig. 2.6(b). Therefore, the correct meaning of load balance should be that all PEs are running without any suspension, we call this *complete load balancing.* When the load is completely balanced, not only the load is equal for each PE, but also dependencies do not cause processor suspension.

Dependencies can lead to processor suspension in two ways. When PE i needs data to execute but the data has not been generated by PE j, PE i will be suspended. Even if the data has been generated by PE j and on the way to PE i, PE i may become suspended due to the message transmission time. In the latter case, we say the message transmission time is not tolerated.

In summary, all efficiency losses can be classified as one of the above efficiency

losses. Overall, the efficiency can be expressed as $\mu = 1 / EL_A EL_C EL_P$.

2.3. Methods for performance improvement

To increase the efficiency, we must design better algorithms, reduce computation and communication overhead, and reduce processor suspension time. A parallel algorithm must be designed to maximize parallelism and minimize dependencies, in order to achieve small efficiency loss. The problem has to be properly partitioned into several tasks. The dependencies among these tasks are the main reason for efficiency losses from processor suspension and coding. The dependencies on a shared memory system are the data, storage, and control dependencies [PeGW87], while in a message-passing system, the only dependency is the message dependency.

A parallel program usually consists of serial and parallel segments of codes, partitioned along its natural boundaries. The parallel parts can be further partitioned along iteration boundaries. If there are no dependencies between iterations, they can be executed on different PEs without communication. If the iterations are dependent with each other, the alignment method and minimum-distance method [PeGW87] may be used to reduce or even eliminate these dependencies. If dependencies still exist after applying these methods, tasks must exchange data. Problem data is to be partitioned to minimize dependencies among tasks and maximize parallelism, however, these two goals may contradict each other. Figure 2.7 shows an example of the Gauss-Seidel algorithm. When we partition the matrix into squares, we have less dependencies and less parallelism (Fig. 2.7(a)). On the other hand, if we partition it into strips, we have more parallelism and more dependencies (Fig. 2.7(b)).

We may use more memory space to eliminate some dependencies. For example, in matrix multiplication, where $C = A * B$, each PE can store a sub-block of matrix A

and a sub-block of B [FJLO88]. This algorithm must exchange data at each computation stage. On the other hand, if each PE stores the entire matrix B, the communication required above can be eliminated. This represents a trade-off between communication overhead and memory space. Also, communication can be reduced by duplicating operations. As previously stated, we must pay a price for duplication of operations in order to eliminate some message transfers.

Packing reduces the number of message transfers, and consequently eliminates some message initialization overhead. Whenever a PE sends several messages to another PE, these messages may be packed together to form a message package. However, the packing and unpacking themselves are overhead. Furthermore, message packing may delay certain message transfers since the earlier-generated message has to wait for the later message to be generated.

The efficiency loss from coding comes from both computation overhead and communication overhead. Efficient use of multiprocessing systems requires relatively crude granularity to reduce overhead. The overhead which dominates the performance of computation determines granularity [MJGS85]. If the major overhead is in initializing tasks, the *task granularity* is considered. The task granularity refers to the average amount of computation in a task. If the overhead from communication is dominant, we consider the *communication granularity*, which refers to the average amount of computation between two consecutive communications.

Fine granularity exploits more parallelism, but increases efficiency loss. Crude granularity, on the other hand, reduces efficiency loss but decreases parallelism [MJGS85]. Figure 2.8 shows the relationship between granularity, parallelism, efficiency, and speedup. If granularity is too small, large overhead leads to low efficiency and decreased speedup, although parallelism increases. On the other hand, when

granularity is too large, there is not enough parallelism for speedup.

The granularity is also determined by dependencies. When there are few dependencies, fine granularity may be used without increasing efficiency loss. When many dependencies exist, the overhead of communication becomes the dominate part of efficiency loss. If overhead is very large, crude granularity may be used with loss of parallelism. When overhead decreases, medium or fine granularity is possible [BrOP86]. The dataflow approach becomes practical when overhead is reduced to several operations. The spectrum of granularity is shown in Fig. 2.9.

In general, granularity should be comparable to overhead to keep efficiency loss small. Furthermore, granularity should be crude enough to tolerate the message transmission time. Current message-passing systems have relative large overheads so that only crude granularity is used in partitioning. Since the overhead is getting smaller on the new generation machines, medium granularity may be used in the near future.

3. Program Development Aid

3.1. Program development methods

The main goal of parallel program development is to reduce efficiency losses. One approach advocates sophisticated dependency analysis and extraction of parallelism from sequential programs by restructuring sequential programs into parallel programs [KKPL81]. However, since sequential programs were developed from sequential algorithms, the restructuring method cannot extract more parallelism than that available in those algorithms. If the extracted parallelism is not large enough, processor suspension

becomes the major source of efficiency loss.

Another approach focuses on developing parallel algorithms and partitioning data manually [Seit85], [FJLO88], [GuMo88]. This approach tries to partition a problem into subproblems of almost equal size to balance load for each PE, and to reduce the dependencies among these subproblems. The quality of programs developed is dependent on programmers' experience and problem regularity. For problems with irregular structures, the combined effect of dependencies and overhead is hard to estimate manually, causing poor granularity and load distribution. For example, load balancing may require some code segments to be moved from one PE to another, which may increase dependencies. These dependencies may lead to more processor suspension, resulting in even worse load imbalance. This approach may cause high efficiency losses from coding and processor suspension. Furthermore, since scheduling and communication is performed manually, debugging programs is difficult.

3.2. The CAP approach

The third approach uses friendly environments and automation to help programmers develop high quality programs with increased productivity.

Several research efforts have demonstrated the usefulness of program development tools for multiprocessing. There are two types of tools. One provides software development environment and debugging facilities. POKER [Snyd84] is a parallel programming environment for message-passing systems, which has been ported to the Cosmic Cube [SnSo86]. POKER provides a debugging environment and a graphic representation of communication structure. DAPP [ApMc85] accepts program code with inserted synchronization primitives and produces a report of parallel access anomalies, that is, pairs of statements that can access the same location simultaneously. Polylith

[PuRG87] was designed for prototyping parallel algorithms. It supports development of architecture-independent parallel programs. In this environment, task communications are specified by virtual connections, instead of physical processor connections, to simplify many data transmission issues.

The other type of tool performs some program transformation. Most tools of this type are based on the theory of program restructuring [PaKL80]. PTOOL [ABKP86] performs sophisticated dependency analysis, including advanced interprocedural flow analysis. It identifies parallel loops, extracts global variables, and provides a simple explanation facility. This information can be used to obtain more parallelism, eliminate some dependencies, and reduce efficiency losses. PTOOL does program transformations, too. For example, it transforms control dependences into data dependences. However, PTOOL only tests loops for independence and does not provide partitioning and synchronization mechanisms for non-parallel loops. CAMP [PeGa86] partitions both parallel and non-parallel loops, and reduces dependences by using process alignment and minimum-distance algorithms. Since it extracts more parallelism and eliminates many dependencies, efficiency loss from processor suspension is reduced. CAMP also inserts synchronization primitives, and estimates performance for each partitioning strategy. Brandes and Sommer [BrSo87] have introduced a knowledge-based parallelization tool. This tool performs dependency and anomaly analysis, as well as some execution order changing to obtain more parallelism and less efficiency loss.

The program development tool described in this paper, Hypertool, is based on the partitioning & merging approach to obtain proper granularity. Hypertool aims at increasing programming productivity and taking advantage of tedious tasks that computers do better than humans. It performs automatic scheduling and communication insertion, and generates parallel codes for target machines. An optimizing scheduler is

used to minimize communication overhead and processor suspension, so that efficiency losses are reduced. Hypertool also provides performance estimates and an explanation facility to help programmers improve their programs.

3.3. The partitioning & merging approach

In the partitioning and merging (P&M) approach, a problem is first partitioned into processes of small sizes. The partitioning strategy may be decided by a programmer or an automatic partitioner. A dataflow (or macro dataflow) graph is generated automatically by the dependency analysis of these processes. A scheduler is then used to merge processes into tasks. Since the scheduler takes care of dependencies and overhead, a high-quality solution can be obtained for problems with regular or irregular structures. Many scheduling algorithms may be used for this purpose. Most are based on the critical path algorithm [Hu61]. If scheduling is done before running the program, it is called static scheduling. Otherwise, it is called dynamic scheduling. The static P&M approach is also called grain packing [KrLe88].

After merging, communication primitives are inserted according to the remaining dependencies. Proper communication primitives are inserted automatically to avoid incorrect results and deadlock.

An simple atomic model is used for program development in the P&M approach. In this model, a computation may be considered as a set of processes among which there are dependencies. If each process is an indivisible unit of execution, a process can be expressed as an atomic node [BeRa81], [HuGo85], [Babb85]. An atomic node has one or more inputs and outputs. When all inputs are available, the node is triggered to execute and generates its outputs. An atomic node can be a procedure, an iteration, a statement, or an operation. We use a directed graph to represent the atomic model, in

which a set of nodes $\{n_1, n_2, ..., n_n\}$ are connected by a set of directed edges $\{e_1, e_2, ..., e_e\}$, as shown in Fig. 3.1. The weight of a node is equal to the process execution time. Since each edge corresponds to a message transfer from one process to another, the weight of the edge is equal to the message transmission time. If the nodes are operations, the graph is a *dataflow graph*, otherwise, it is a *macro dataflow graph*.

Dependencies may cause communication overhead and processor suspension. However, dependencies are not harmful as long as they are contained in the same task. Therefore, we may merge several processes into a task to reduce communication while maintaining as much parallelism as possible. In Fig. 3.2(a), when nodes 2 and 5, and 3 and 4, are merged, the number of dependences is reduced from 5 to 3, with reduction of parallelism. On the other hand, if nodes 1 and 3, and 5 and 6, are merged as shown in Fig. 3.2(b), the graph contains 3 dependences without reduction of parallelism.

In the P&M approach, the sizes of processes determine program performance. Usually, small-size processes lead to better performance. In the extreme case, a process might contain only a single operation. However, small-size processes can cause large amounts of scheduling work, which may exceed the capability of a static scheduler. When a dynamic scheduler is used, the scheduling itself becomes major overhead for a large number of processes. Therefore, from practical point of view, a process should consist of moderate size of operations. Thus, a careful choice of process sizes and use of a good scheduler will reduce the efficiency losses from coding and processor suspension.

4. Hypertool

4.1. System diagram

In this section, we present a version of Hypertool, which takes C programs as input, performs static scheduling, and generates parallel codes for the SIMON simulator. The performance estimation and quality measures also will be described.

The system diagram of our Hypertool is shown in Fig. 4.1. First, a user develops a proper algorithm, performs partitioning, and writes a program as a set of procedures. The program looks like a sequential program and can be run on a sequential machine for debugging purposes. This program is automatically converted into the parallel program for a hypercube target machine by parallel code synthesis and optimization. Hypertool then generates performance estimates, including execution time, communication time, and suspension time for each PE, and network delay for each communication channel. The explanation facility displays data dependencies between PEs, as well as parallelism and load distribution in any time interval [WuGa87]. If the performance is not satisfactory, the programmer can change the partitioning strategy and the size of the partitions using the information provided by the performance estimator and the explanation facility.

Figure 4.2 shows the organization of the program synthesis and optimization module. The lexer and the parser recognize data dependencies and user defined partitions. The graph generation submodule generates a macro dataflow graph, in which each node represents a process. The scheduling submodule assigns processes to tasks by minimizing the execution time for the graph. The mapping submodule maps each task to a physical PE in a given topology by minimizing network traffic. After scheduling and mapping are complete, the synchronization module inserts the communication

primitives. Finally, the code generator generates target machine code for each PE.

4.2. Code development

To facilitate automation of program development, we use a programming style in which a C program is composed of a set of procedures called from a main program. A procedure is an indivisible unit of computation to be scheduled on one processor. For example, a parallel Gaussian Elimination algorithm, which partitions a given matrix by columns, is shown in Fig. 4.3. The procedures FindMax and UpdateMtx are called several times. Procedure calls can be executed in any order, that is, the control dependencies can be ignored. Data dependencies are defined by the single assignment of parameters in procedure calls. Communications are invoked only at the beginning and the end of procedures. In other words, a procedure receives messages before it begins execution, and it sends messages after it has finished the computation. Data dependences among the procedural parameters define a macro dataflow graph.

This programming style has good modularity, which is necessary for developing general application programs. Also, it is system independent since communication primitives are not specified within the program.

4.3. Scheduling

A macro dataflow graph, which is generated directly from the main program, is a directed graph with a start and an end point. For example, Fig. 4.4 shows the macro dataflow graph of the program in Fig. 4.3. Note that only the parallel parts of Fig. 4.3 and the messages transferred among these procedures are shown in Fig. 4.4. Each node corresponds to a procedure, and the node weight is represented by the procedure

execution time. For example, nodes n_1, n_7, n_{12}, n_{16} in Fig. 4.4 correspond to the procedure FindMax, while the other nodes represent the procedure UpdateMtx. Each edge corresponds to a message transferred from one procedure to another, and the weight of the edge is equal to the transmission time of the message. In Fig. 4.4, for example, the edges connecting n_1, n_2, n_3, n_4, n_5, n_6, and n_7 correspond to a message called "vector" in the first iteration, and the edge connecting n_4 and n_9 to a message called "matrix". When two nodes are scheduled to a single PE, the weight of the edge connecting them becomes zero. The execution time of a node is obtained by running the corresponding procedure, while the transmission time is estimated by using system parameters. We assume that the given message transmission time is for neighbor communication. Non-neighbor communication takes a little more time. We also assume network traffic is not too heavy. Therefore, network contention is ignored in our model. The time for initiating a data transfer is assumed short enough to be ignored.

Next, we discuss static nonpreemptive scheduling of a macro dataflow graph for homogeneous multiprocessors. Critical-path scheduling has been addressed by Hu [Hu61]. The critical-path algorithm has been proved to be near optimal [AdCD74] [Kohl75]. This algorithm assigns a label to each node according to the longest path from this node to the end point. It performs well for a limited number of PEs. Ramamoorthy, *et al.* developed algorithms to determine the minimum number of PEs required to process a program in the shortest possible time [RaCG72] [KaNa84]. They used exhaustive search, which is not acceptable for large programs. Bussell, *et al.* proposed an alternative method to reduce the number of PEs [BuFL74], but the efficiency of the algorithm is still dependent on a bound estimate of the number of processors. More importantly, these algorithms did not model transmission time, that is, they assumed that the data transmission between PEs did not take any time. This is not true, however, for most message-passing systems. The data transmission time is a

significant factor that affects the overall performance of a system, which must be considered in modeling. Kruatrachue and Lewis have presented a model that assigned data communication time as weights of edges [KrLe88].

Wu and Gajski have introduced two scheduling algorithms, called the modified critical-path (MCP) algorithm and the mobility-directed (MD) algorithm, to minimize the execution time on a limited number of PEs and to minimize the number of PEs required to process a graph in the shortest possible time, respectively [WuGa88]. The MCP and MD algorithms first indicate the critical path, and calculate mobilities. The mobility is the time interval in which a node can be executed without delaying the execution of the critical path. Then, nodes are scheduled according to mobilities. These algorithms reduce dependencies between tasks, balance load for each PE, and minimize the efficiency losses from coding and processor suspension.

4.4. Mapping

A mapping algorithm should generate the minimum amount of communication traffic, reducing network contention and the efficiency loss from processor suspension. For best results, a traffic scheduling algorithm that balances network traffic should be used [BiSh87]. However, traffic scheduling requires flexible-path routing, which generates large overhead. If network traffic is not too heavy, simpler algorithms that minimize total network traffic may be used.

The mapping problem may be described as follows. Given a task graph, consisting of n_t nodes and e_t edges, which is generated by the scheduler. Each node in this graph corresponds to a task, and each edge corresponds to the message transferred between two tasks. The weight of the edge, $w(e_i)$, is the sum of transmission time of all messages between the two tasks. This task graph is to be mapped to a system

graph defined by a given topology. A system graph consists of n_s nodes and e_s edges, where $n_s \geq n_t$. Each node corresponds to a physical PE, and each edge to a connection between two PEs with weight 1. Each edge in a task graph or a system graph is bidirectional. These graphs can be simplified into an undirected graph since we are not performing traffic scheduling.

If the task graph can be mapped to the system graph and all communications are nearest-neighbor communications, no routing is necessary and the mapping is optimal. Otherwise, certain pairs of tasks connected by an edge will be mapped to two non-neighboring PEs. The corresponding message will be routed through the shortest path between the two PEs. The distance d of two PEs is defined as the number of hops on the shortest path from one PE to another. Our objective function is

$$F = \sum_{i=1}^{e_t} w(e_i) d_i$$

where, d_i is the distance of the two PEs to which the two tasks connected by e_i are mapped. Therefore, F stands for the total communication traffic.

As an example, Fig. 4.5(a) and (b) show a task graph and a system graph, respectively. When the task graph is mapped to the system graph as shown in Fig. 4.5(c), $F = 20$. A better mapping is shown in Fig. 4.5(d) with $F = 16$.

We may use the algorithms for the quadratic assignment problem to obtain a near optimal mapping. A heuristic algorithm presented by Hanan and Kurtzberg may be applied to minimize the total communication traffic [HaKu72]. It generates an initial assignment by a constructive method, which is then improved iteratively to obtain a better solution. Lee and Aggarwal described some experimental results for hypercube topologies and showed that the algorithm works well, even though it did not always guarantee an optimum solution [LeAg87].

4.5. Communication insertion

The communication primitives are used to exchange messages between processors. They must be used properly to ensure the correct sequence of computation. Since our programming model partitions programs into procedures and message exchanges only take place before and after the procedures, primitive insertion is easily performed automatically, reducing the programmer's load and eliminating insertion errors.

The communication primitive insertion is performed as follows. After scheduling and mapping, each node in a macro dataflow graph is allocated to a PE. If an edge exits from this node to another node that belongs to a different PE, the *send* primitive is inserted after the node. Similarly, if the edge comes from another node in a different PE, the *receive* primitive is inserted before the node. However, if a message has already been sent to a particular PE, the same message does not need to be sent to the same PE again.

The insertion method described above does not ensure that the computation sequence is correct. For example, two possible cases are shown in Fig. 4.6(a) and Fig. 4.7(a). In Fig. 4.6(a), the order of the *sends* is incorrect, and must be reordered as shown in Fig. 4.6(b). In Fig. 4.7(a), on the other hand, either the order of *sends* or the order of *receives* needs to be exchanged as shown in Fig. 4.7(b) or (c), respectively. Figure 4.8 shows the generated parallel code of Fig. 4.3 for two PEs. Note that only the main program for each PE is shown.

5. Experimental Results

Our Hypertool is currently running on a Sun workstation under UNIX. It takes 37 seconds to schedule a program with 162 processes to 8 PEs. Several examples have been tested on Hypertool.

By our experience, program development with Hypertool takes much less time than manual program development. Debugging is much easier, and we never have any deadlock in the programs developed on Hypertool. The results also show that Hypertool generates codes that execute faster than manually generated codes. For some problems, such as the Laplace Equation, Gaussian Elimination, and Dynamic Programming, up to 300% improvement in speed can be obtained by Hypertool (see Tables 5.1, 5.2, and 5.3). The Gauss-Seidel algorithm is used for the Laplace Equation since it has less efficiency loss. For the Gaussian Elimination algorithm, the matrix is partitioned by columns. The Dynamic Programming problem is partitioned by both rows and columns. These problems have less regular structures so that good load balance is difficult to obtain manually. For more regular problems, such as the Matrix Multiplication and Bitonic Sort, automatic scheduling gives performance similar to that of manual scheduling, as shown in Tables 5.4 and 5.5. However, even for these kinds of problems Hypertool still shows better performance when the size of matrix cannot be evenly divided by the number of PEs. For example, when the matrix sizes are 9*9 and 17*17, Hypertool-generated codes show better performance. In such case, manual scheduling usually leads to an unbalanced load distribution among PEs, while automatic scheduling moves some nodes from overloaded PEs to underloaded PEs and achieves a better load balance.

Since the execution time of nodes and the message transmission time are obtained by estimation, the performance affected by the difference between the estimated value

and the real value has been studied. Tables 5.6 and 5.7 show the results for Laplace Equation and Bitonic Sort, respectively. For the Laplace Equation, the node weight is estimated with matrix size of 4*4, and for the Bitonic Sort, the node weight is estimated with array length of 64. The results show that the difference between the estimated value and the real value has little effect on performance.

6. Conclusion

As both the number of PEs and the complexity of problems to be solved increase, programming multiprocessing systems becomes more difficult and error-prone. The optimal parallelization may be too complicated for all but simple problems. Actually, early experiments on programming hypercube systems has revealed that conceptualization of program execution is very difficult, and any further optimization of complex problems was discouraged. A program development tool that helps programmers to develop parallel programs by automating part of parallelization tasks and back-annotating some quality measures to programmers becomes a necessity.

The experimental results obtained by Hypertool show that the CAP methodology is better than the manual methodology in many respects. First, it increases the programming productivity by an order of magnitude. Programmers only define partitions without indicating the task allocation or communication primitives. Second, since communication primitive insertion is performed by Hypertool, many errors, such as incorrect computation sequence and deadlock, are eliminated. Moreover, most programming errors may be debugged by sequentially running the program. Finally, the program development tool generates better parallel codes since it uses good scheduling algorithms. This resulted in substantial performance increases as demonstrated in the

previous section.

Since the program development tool generates target machine codes automatically, the programs developed on the tool are portable. The programs may run on different message-passing systems, and even on shared memory systems. The tool can also be developed for a variety of languages to fit different applications.

Acknowledgments

This work was supported, in part, by National Science Foundation grant No. CCR-8700738. The authors wish to acknowledge the contributions of Wei Shu, for her programming work and Andrew Kwan, for carefully reading on the manuscript.

References

[ABKP86] R. Allan, D. Baumgartner, K. Kennedy, and A. Porterfield, "PTOOL: A Semi-Automatic Parallel Programming Assistant," *Proc.* 1986 *Int'l Conf. on Parallel Processing*, pp. 164-170, Aug. 1986.

[AdCD74] T.L. Adam, K.M. Chandy, and J.R. Dickson, "A Comparison of list scheduling for parallel processing systems," *Comm. ACM*, Vol. 17, No. 12, pp. 685-690, Dec. 1974.

[ApMc85] W.F. Appelbe, and C. McDowell, "Anomaly Detection in Parallel Fortran Programs," *Proc. Workshop on Parallel Processing Using the HEP*, May 1985.

[Babb85] R. G. Babb, "Programming the HEP with Large-grain Data Flow Techniques," *MIMD Computation: HEP Supercomputer and Its Applications*, Cambridge, MA: The MIT Press, pp. 203-227, 1985.

[BeRa81] E. Best and B. Randell, "A Formal Model of Atomicity in Asynchronous Systems," *Acta Information*, Vol. 16, pp. 93-124, 1981.

[BiSh87] R.P. Bianchini and J.P. Shen, "Interprocessor Traffic Scheduling Algorithm for Multiple-Processor Networks," *IEEE Trans. Computers*, C-36, No. 4, pp. 396-409, April 1987.

[BrOP86] J. D. Brock, A. R. Omondi, and D. A. Plaisted, "A Multiprocessor Architecture for Medium-Grain Parallelism," *Proc. 6th International Conf. on Distributed*

Computing Systems, pp. 167-174, 1986.

[BrSo87] T. Brandes and M. Sommer, "A Knowledge-based Parallelization Tool in a Programming Environment," *Proc. Int'l Conf. on Parallel Processing*, pp. 446-448, Aug. 1987.

[BuFL74] B. Bussell, E. Fernandez, and O. Levy, "Optimal scheduling for Homogeneous Multiprocessors," in *Proc. IFIP Congress* 74, North-Holland Publ. Co., Amesterdam, American Elsevier, N.Y., pp. 286-290, 1974.

[FJLO88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors*, Vol. I, Prentice-Hall, 1988.

[GaPe85] D.D. Gajski, and J.-K. Peir, "The Essential Issues in Multiprocessor Systems," *IEEE Computer*, Vol. 18, No. 6, pp. 9-27, June 1985.

[GuMo88] J. L. Gustafson and G. R. Montry, "Programming and Performance on a Cube-Connected Architecture," *IEEE COMPCON*, pp. 97-100, March 1988.

[HaKu72] M. Hanan and J.M. Kurtzberg, "A Review of the Placement and Quadratic Assignment Problems," *SIAM Rev.*, Vol. 14, pp. 324-342, April 1972.

[Hu61] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, Vol. 9, No. 6, pp. 841-848, 1961.

[HuGo85] P. Hudak and B. Goldberg, "Serial Combinators: Optimal Grains of Parallelism," *Functional Programming languages and Computer Architecture*, Lecture Notes in Comp. Sci., Vol. 201, Springer-Verlag, Berlin, pp. 382-388, 1985.

[Jenn77] A. Jennings, *Matrix Computation for Engineers and Scientists*, John Wiley & Sons, New York, 1977.

[KaNa84] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. Computers*, C-33, pp. 1023-1029, Nov. 1984.

[KKPL81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graph and Compiler Optimizations," *Proc. of 8th ACM Symp. Principles on Programming Lang.*, Jan. 1981.

[Kohl75] W.H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Trans. Computers*, C-24, pp. 1235-1238, Dec. 1975.

[KrLe88] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software*, pp. 23-32, Jan. 1988.

[LeAg87] S.Y. Lee and J.K. Aggarwal, "A Mapping Strategy for Parallel Processing," *IEEE Trans. Computers*, C-36, No. 4, pp. 433-442, April 1987.

[MJGS85] J. Mohan, A. Jones, E. Gehringer, and Z. Segall, "Granularity of Parallel Computation," *Proc. of 8th Annual Hawaii International Conf. on System Sciences*, pp. 249-256, 1985.

[PaKL80] D.A. Padua, D.J. Kuck, and D.L. Lawrie, "High Speed Multiprocessor and Compilation Techniques," *IEEE Trans. Computers*, Vol. C-29, No. 9, pp. 763-776, Sep. 1980.

[PeGa86] J.K. Peir, and D.D. Gajski, "CAMP: A Programming Aide for Multiprocessors," *Proc. Int'l Conf. on Parallel Processing*, pp. 475-482, Aug. 1986.

[PeGW87] J.K. Peir, D.D. Gajski, and M.Y. Wu, "Programming Environments for Multiprocessors," *Proceeding of International Seminar on Scientific Supercomputers*, pp. 47-68, Feb. 1987.

[PuRG87] J. Purtilo, D.A. Reed, and D. C. Grunwald, "Environments for Prototyping Parallel Algorithms," *Proc. Int'l Conf. on Parallel Processing*, pp. 431-438, Aug. 1987.

[RaCG72] C.V. Ramamoorthy, K.M. Chandy, and M. J. Gonzales, "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Trans. Computer*, C-21, pp. 137-146, Feb. 1972.

[Seit85] C.L. Seitz, "The COSMIC Cube," *Communications of the ACM*, Vol. 28, pp. 22-33, Jan. 1985.

[SnSo86] L. Snyder and D. Socha, "Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environments," *Proc. Int'l Conf. on Parallel Processing*, pp. 638-635, Aug. 1986.

[Snyd84] L. Snyder, "Parallel Programming and the POKER Programming Environment," *Computer*, pp. 27-36, July, 1984.

[WuGa87] M.Y. Wu and D.D. Gajski, "A Programming Aid for Message-passing Systems," *Proc. Third SIAM Conf. on Parallel Processing for Scientific Computing*, Dec. 1987.

[WuGa88] M.Y. Wu and D.D. Gajski, "A Programming Aid for Hypercube Architectures," *Journal of Supercomputing*, 1988.

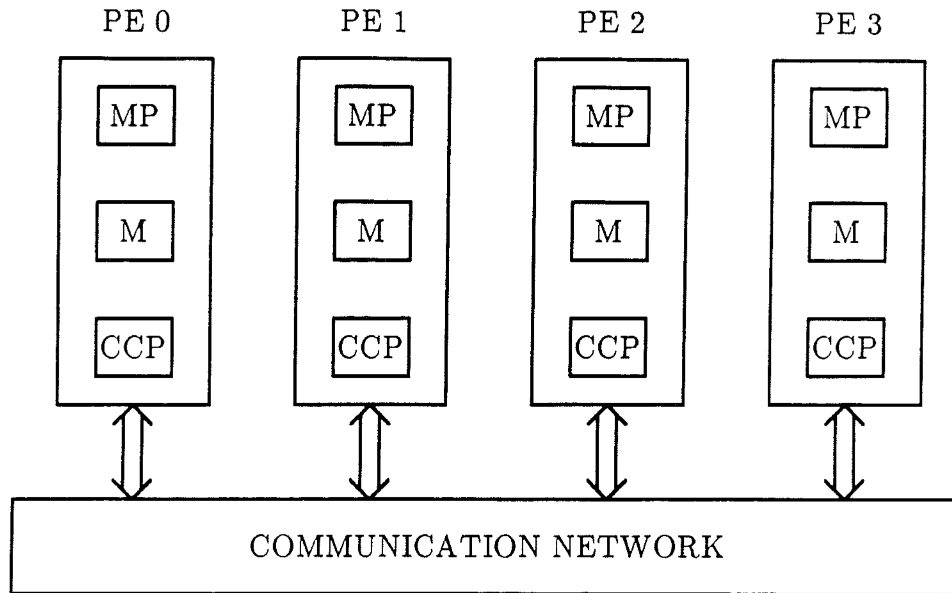


Fig. 2.1 Multiprocessor system model.

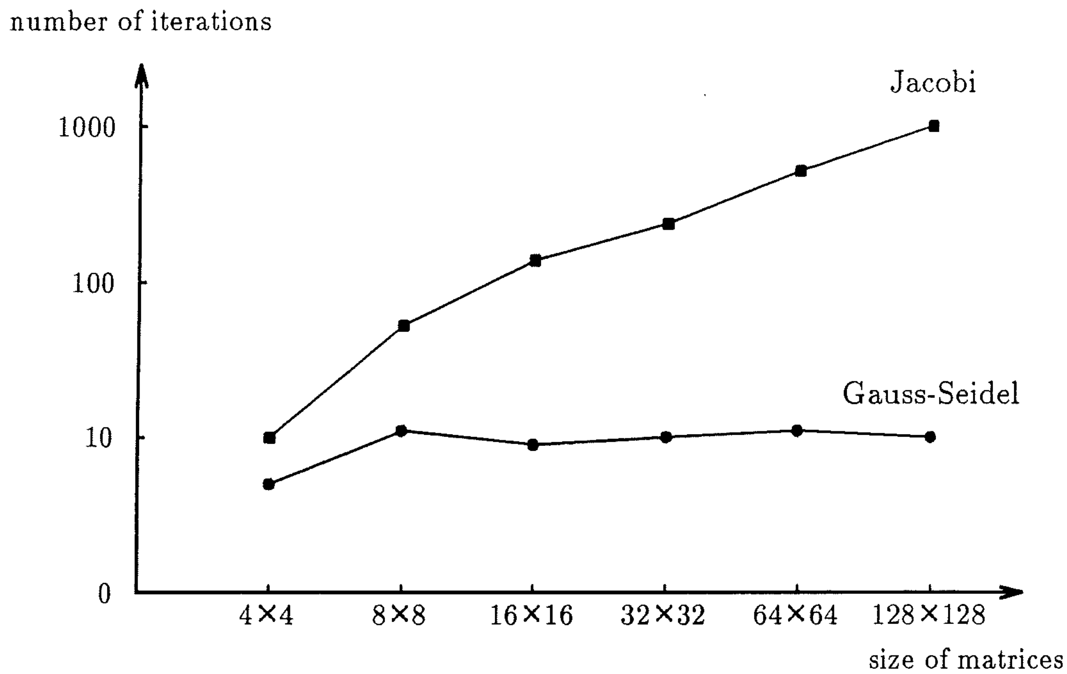
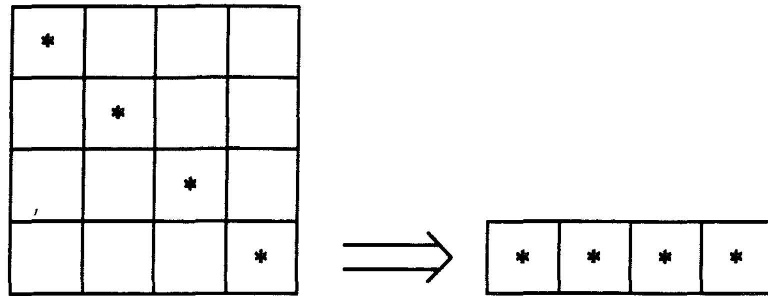
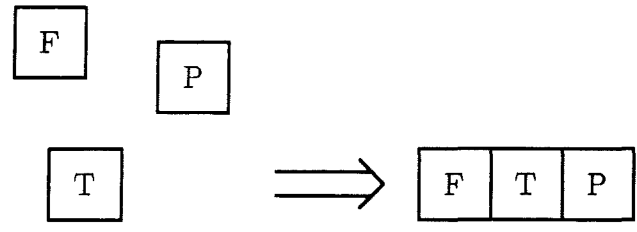


Fig.2.2 Number of iterations for convergency
of Jacobi and Gauss-Seidel algorithms.



(a)



F — Force
P — Position
T — Temperature

(b)

Fig.2.3 Packing: (a) the same data type; (b) different data types.

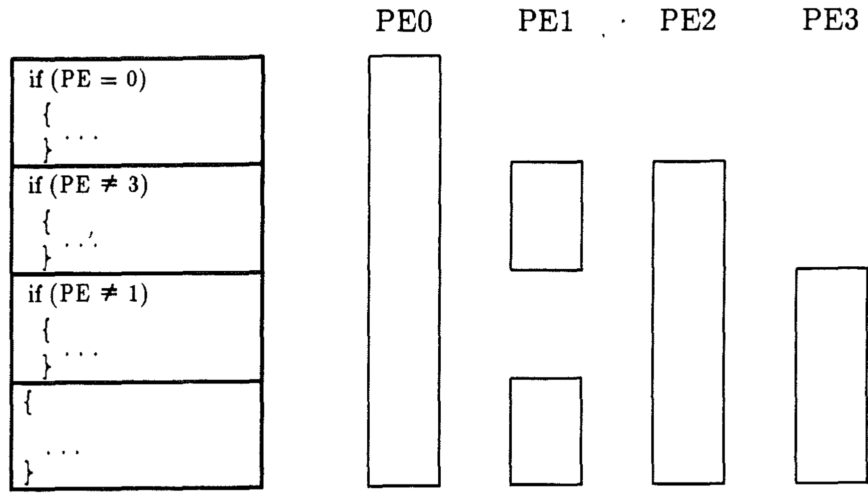


Fig.2.4 Conditional statements to switch code segments for PEs.

```
a = b * c               (I)
for i = 1 to 4 do
    x[i] = y[i] * a     (II)
```

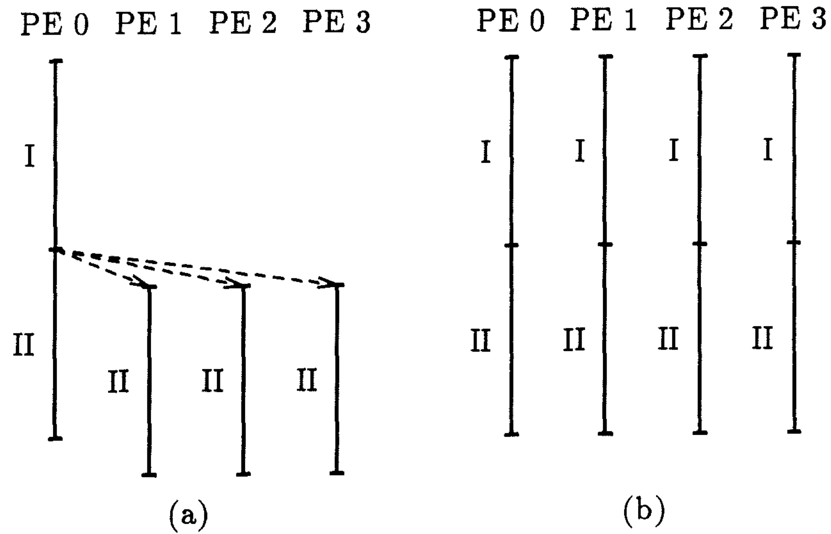


Fig.2.5 Duplication operations:
(a) PE 0 executes $a = b * c$ and broadcasts a;
(b) all PEs execute $a = b * c$.

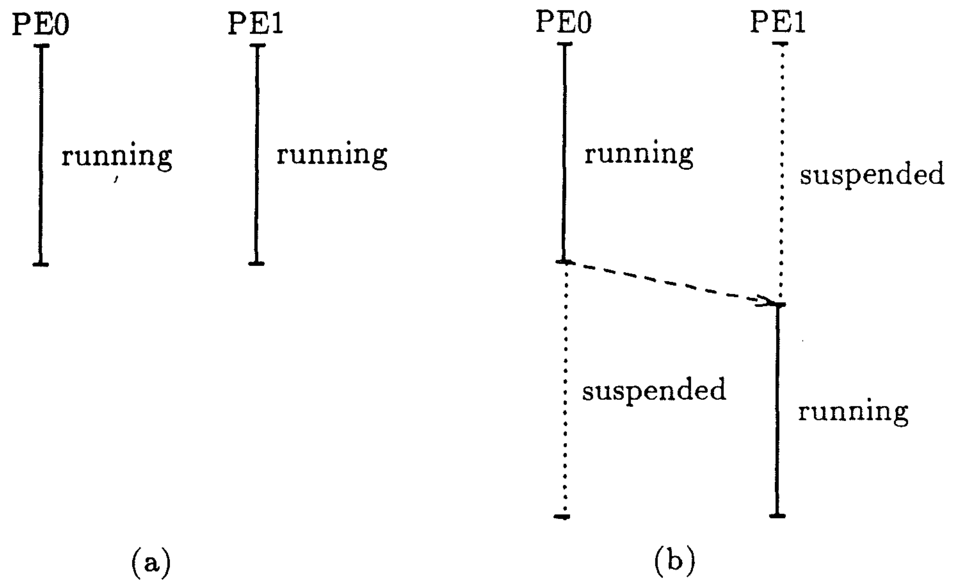
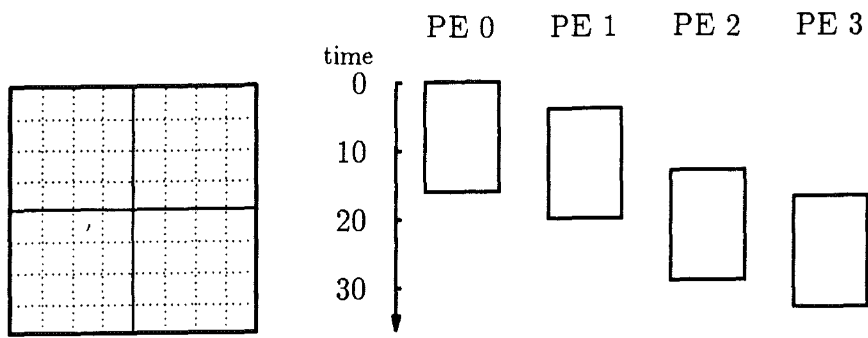
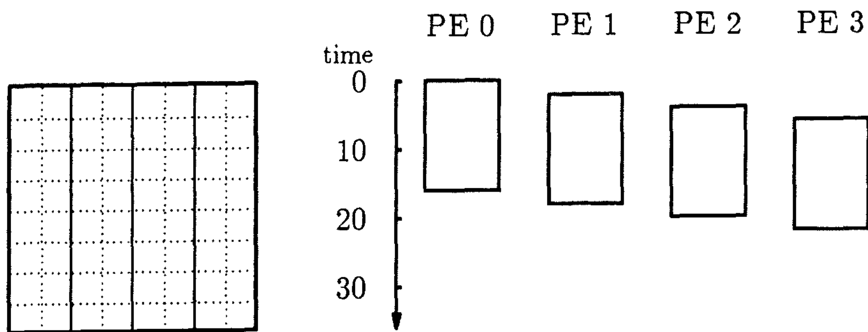


Fig.2.6 Load balance:
(a) without suspension; (b) with suspension.



(a)



(b)

Fig. 2.7 Different partitioning for Gauss-Seidel algorithm: (a) Square partitioning; (b) Strip partitioning.

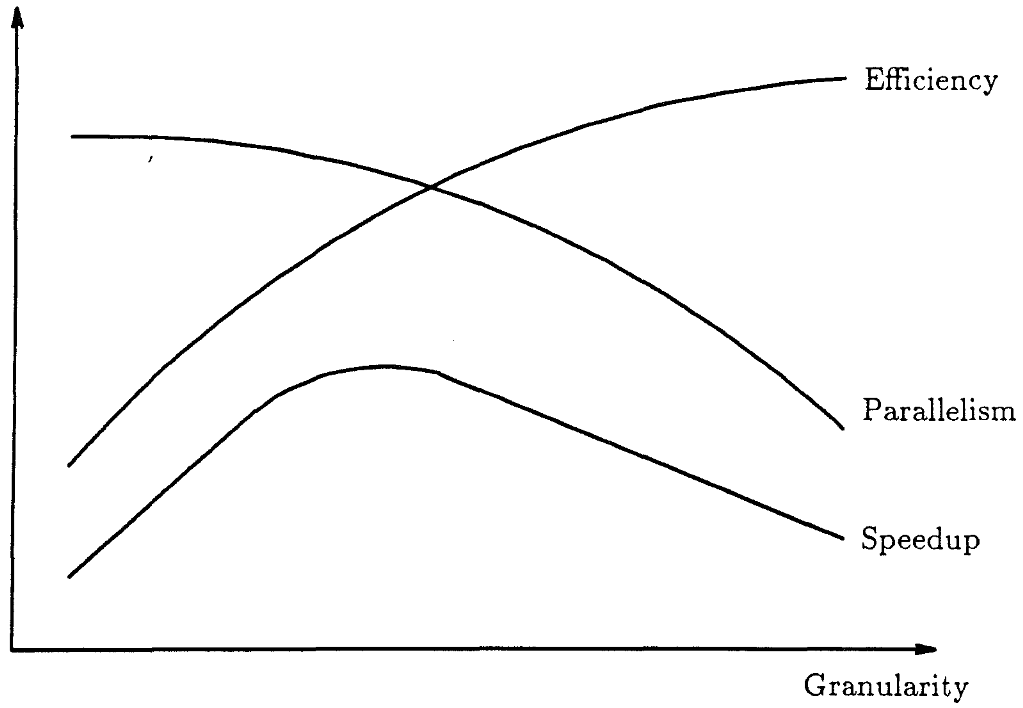


Fig. 2.8 The relationship between granularity, parallelism, efficiency, and speedup.

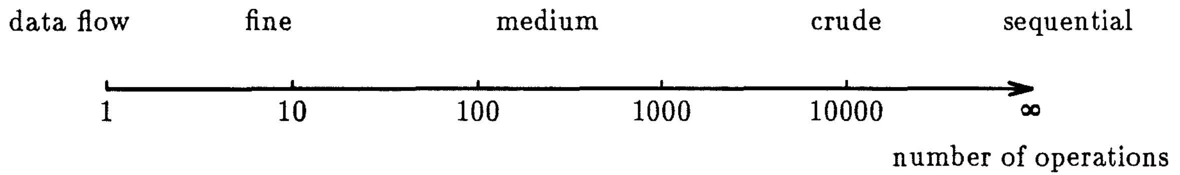


Fig. 2.9 Spectrum of granularity.

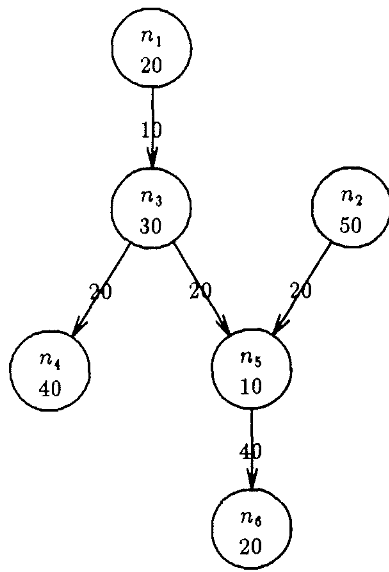
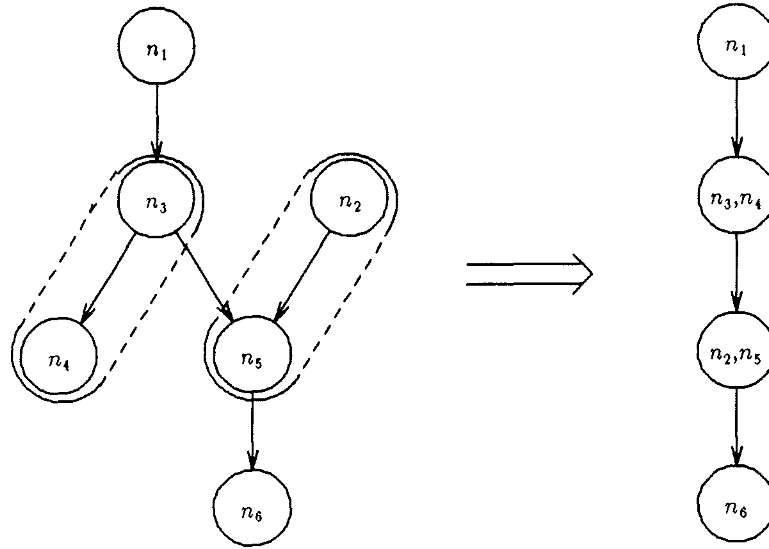
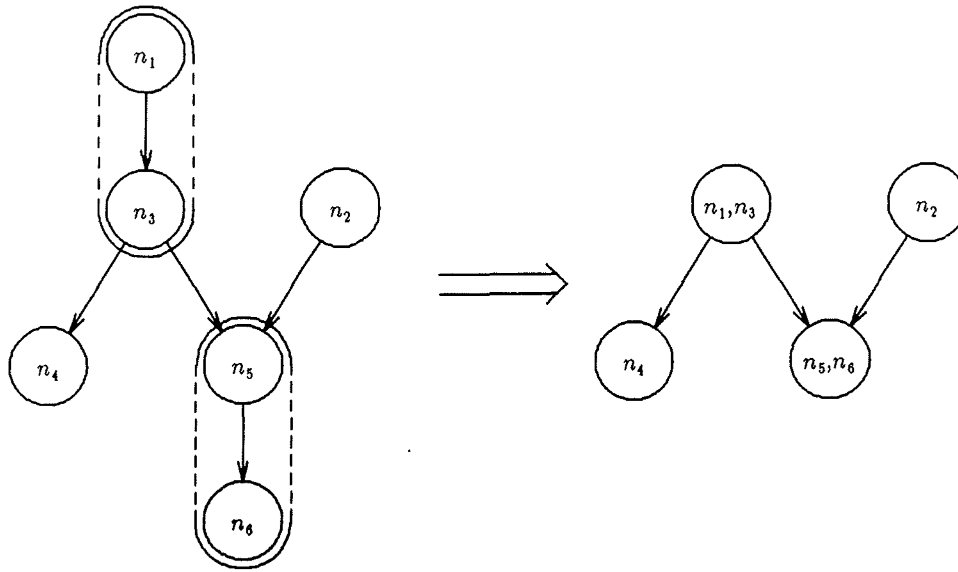


Fig. 3.1 A macro dataflow graph.



(a)



(b)

Fig. 3.2 Process merging: (a) with loss of parallelism; (b) without loss of parallelism.

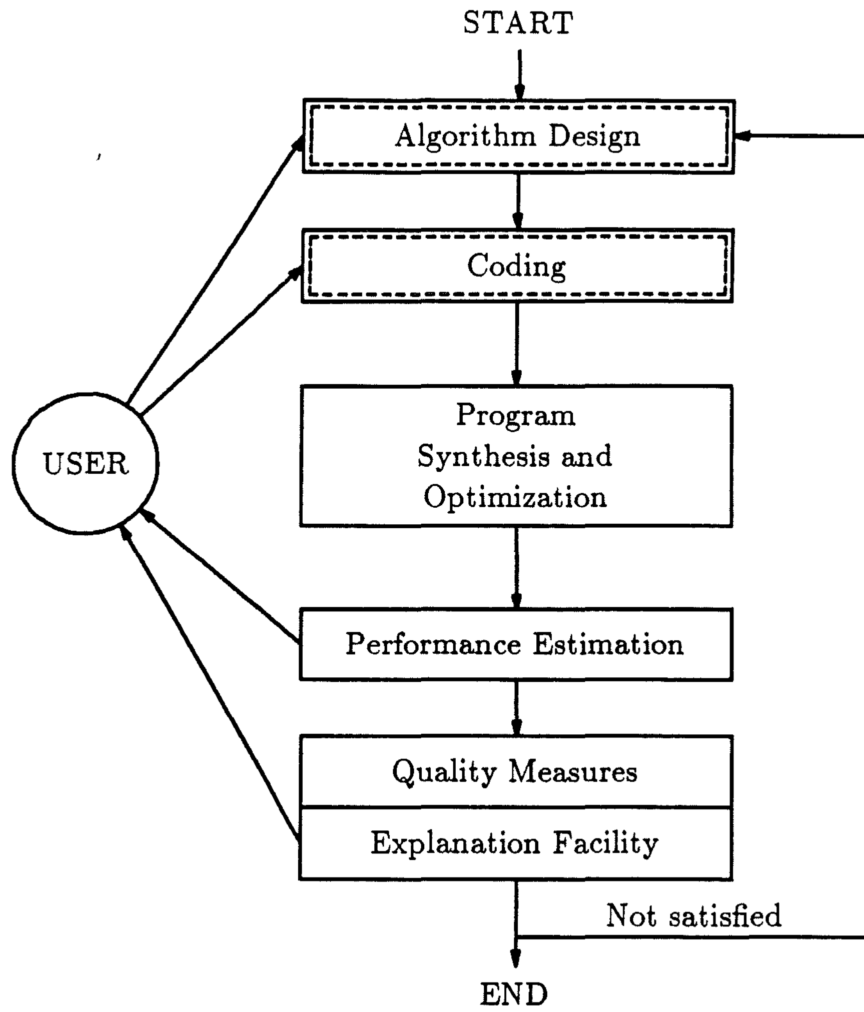


Fig. 4.1 The Hypertool.

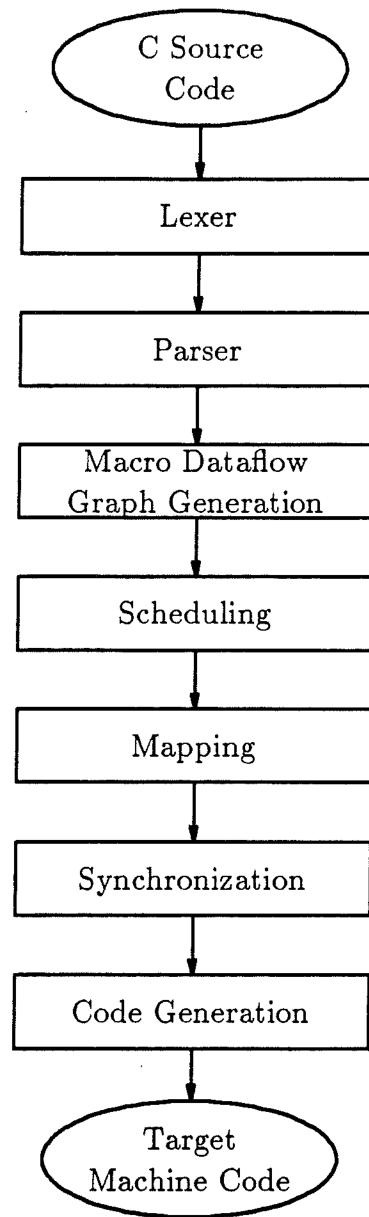


Fig. 4.2 Program synthesis and optimization.

```
Program GaussianElimination
        /*
        matrix[N+1][N][N+1]        stores single-assigned N*(N+1) matrix A and column of the equation Ax=y;
        vector[N+1].index[N]       stores single-assigned row permutation;
        vector[N+1].m[N]           stores single-assigned coefficients;
        */
    /**************************************** Main Program *****************************************/

        /* initialize matrix[0][N][N+1]; initialize vector[0].index[N]; (a serial part of computation) */
    call Initiation;

        /* perform N iterations in parallel */
    for i = 0 to N-1 do
      call FindMax(matrix[i][i], vector[i], vector[i+1], i);
        /* FindMax can be executed if matrix[i][i] and vector[i] are available */
        /* vector[i+1] becomes available at the end of this procedure execution */
        /* perform parallel operations on N-i+1 columns */
      for j = i to N do
        call UpdateMtx(matrix[i][j], matrix[i+1][j], vector[i+1], i);
        /* UpdateMtx can be executed if matrix[i][j] and vector[i+1] are available */
        /* matrix[i+1][j] becomes available at the end of this procedure execution */

        /* do back substitution   (a serial part of computation) */
    call BackSubstitution;
End
```

Fig. 4.3(a) A parallel Gaussian Elimination algorithm.

```
/**************************************** Procedure FindMax ***********************************/

Procedure FindMax(inColumn, inVec, outVec, k)
        /*
        Input:  inColumn    column k where max pivot will be found;
                inVec       permutation index and coefficients;
                k           iteration number;
        Output: outVec      vector of output values;
        */

        /* find maximum */
   max = inColumn[inVec.index[k]];
   n = k;
   for i = k+1 to N-1 do
      if max < inColumn[inVec.index[i]]
         max=inColumn[inVec.index[i]];
         n=i;

        /* copy inVec.index to outVec.index */
   for i = 0 to N-1 do
      outVec.index[i] = inVec.index[i];

        /* permute row index */
   if (n <> k)
      tmp=outVec.index[k]; outVec.index[k]=outVec.index[n]; outVec.index[n]=tmp;

        /* calculate multiplying factors */
   for i = k+1 to N-1 do
      j = outVec.index[i];
      outVec.m[j] = inColumn[j] / max;
End

/************************************** Procedure UpdateMtx **********************************/

Procedure UpdateMtx(inColumn, outColumn, inVec, k)
        /*
        Input:  inColumn    column to be updated;
                inVec       permutation index and coefficients;
                k           iteration number;
        Output: outColumn   column of output values;
        */

        /* copy inColumn to outColumn */
   for i = 0 to k do
      j = inVec.index[i];
      outColumn[j] = inColumn[j];

        /* update the column */
   pivot = inColumn[inVec.index[k]];
   for i = k+1 to N-1 do
      j = inVec.index[i];
      outColumn[j] = inColumn[j] - inVec.m[j] * pivot;
End
```

Fig. 4.3(b) A parallel Gaussian Elimination algorithm (continued).

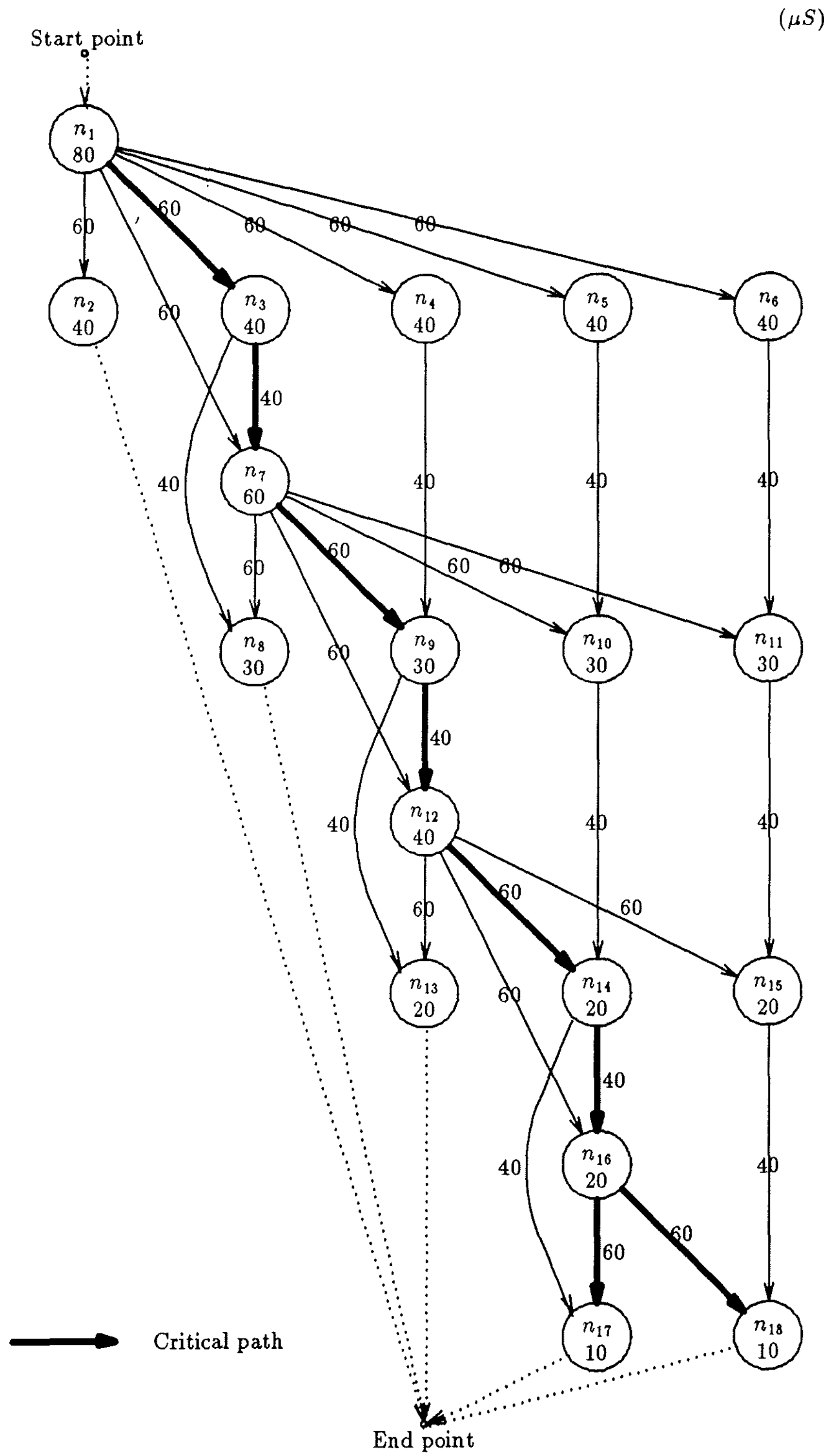
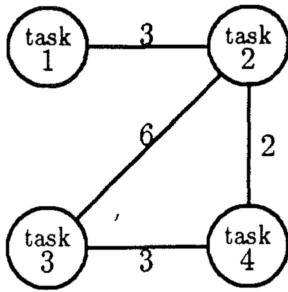
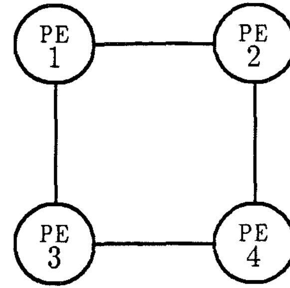


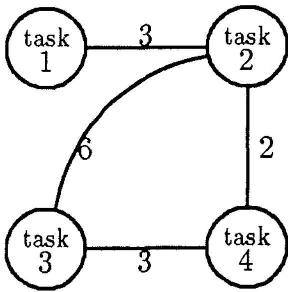
Fig. 4.4 The macro dataflow graph of Fig. 4.3 (N=4).



(a)

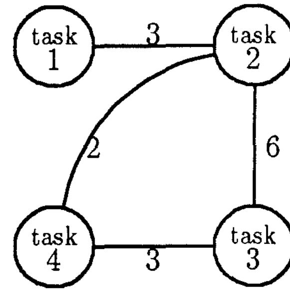


(b)



$F = 3 + 2 + 3 + 6*2 = 20$

(c)



$F = 3 + 6 + 3 + 2*2 = 16$

(d)

Fig.4.5 Mapping:
(a) task graph; (b) system graph;
(c) a mapping; (d) the optimal mapping.

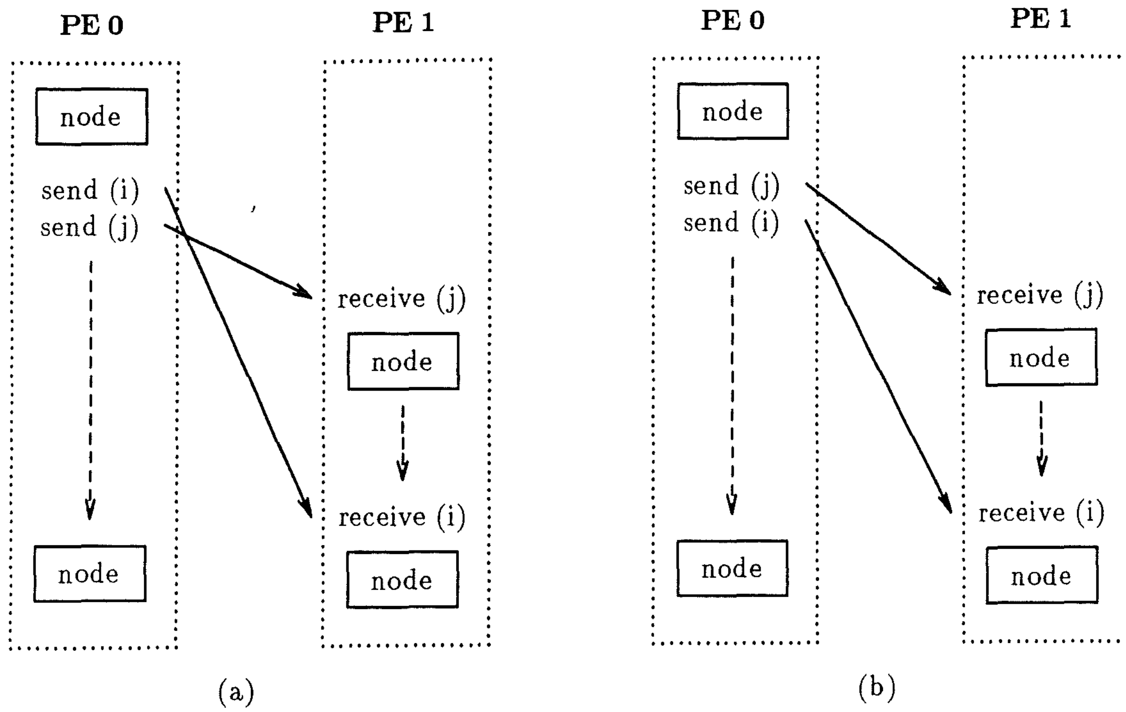


Fig. 4.6 Synchronization insertion (case 1).

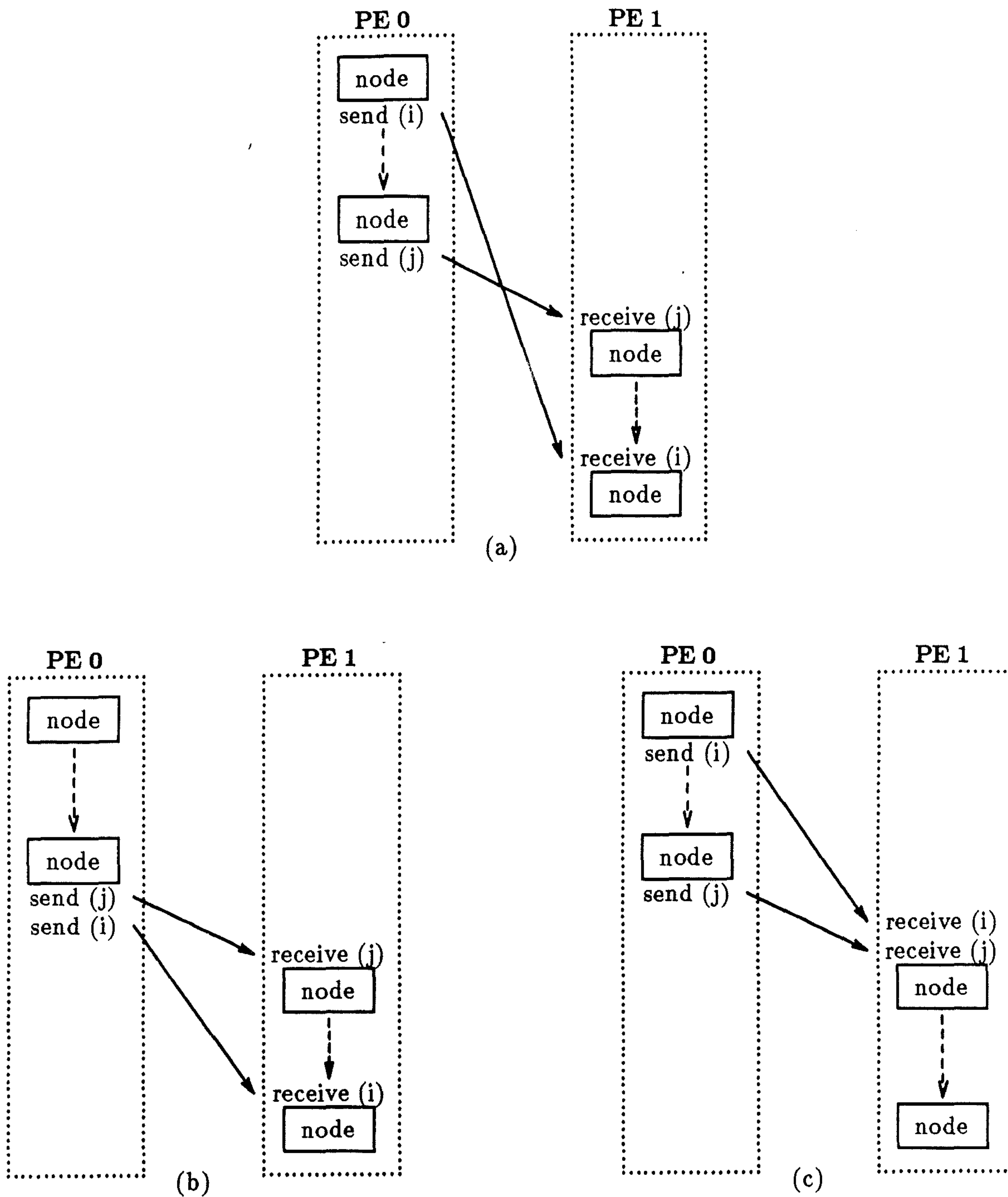


Fig. 4.7 Synchronization insertion (case 2).

```
/* for PE 0 */
    call FindMax(matrix[0][0], vector[0], vector[1], 0);
    send(PE1, vector[1]);
    call UpdateMtx(matrix[0][1], matrix[1][1], vector[1], 0);
    send(PE1, matrix[1][1]);
    call FindMax(matrix[1][1], vector[1], vector[2], 1);
    send(PE1, vector[2]);
    call UpdateMtx(matrix[0][2], matrix[1][2], vector[1], 0);
    call UpdateMtx(matrix[1][2], matrix[2][2], vector[2], 1);
    call FindMax(matrix[2][2], vector[2], vector[3], 2);
    call UpdateMtx(matrix[2][2], matrix[3][2], vector[3], 2);
    receive(PE1, matrix[2][3]);
    call UpdateMtx(matrix[2][3], matrix[3][3], vector[3], 2);
    call FindMax(matrix[3][3], vector[3], vector[4], 3);
    call UpdateMtx(matrix[3][3], matrix[4][3], vector[4], 3);
    receive(PE1, matrix[2][4]);
    call UpdateMtx(matrix[2][4], matrix[3][4], vector[3], 2);
    call UpdateMtx(matrix[3][4], matrix[4][4], vector[4], 3);

/* for PE 1 */
    receive(PE0, vector[1]);
    call UpdateMtx(matrix[0][3], matrix[1][3], vector[1], 0);
    call UpdateMtx(matrix[0][4], matrix[1][4], vector[1], 0);
    receive(PE0, matrix[1][1]);
    receive(PE0, vector[2]);
    call UpdateMtx(matrix[1][3], matrix[2][3], vector[2], 1);
    send(PE0, matrix[2][3]);
    call UpdateMtx(matrix[1][4], matrix[2][4], vector[2], 1);
    send(PE0, matrix[2][4]);
    call UpdateMtx(matrix[0][0], matrix[1][0], vector[1], 0);
    call UpdateMtx(matrix[1][1], matrix[2][1], vector[2], 1);
```

Fig. 4.8 The target machine code for each PE.

Table 5.1 Performance Comparison for Laplace Equation

Matrix size	# of PEs	Execution time (mS) Manual	Hypertool	Improvement in speed
8 * 8	4	5.6	4.1	37%
16 * 16	4	19.3	12.7	52%
32 * 32	4	72.5	44.8	62%
	16	45.2	18.3	147%
64 * 64	4	281.3	168.7	67%
	16	169.3	53.7	215%
128 * 128	4	1109.0	655.9	69%
	16	656.5	185.8	253%
256 * 256	4	4404.9	2587.1	70%
	16	2587.9	692.7	274%

Table 5.2 Performance Comparison for Gaussian Elimination

Matrix size	# of PEs	Execution time (mS)		Improvement in speed
		Manual*	Hypertool	
4 * 4	2	3.0	1.9	58%
8 * 8	2	14.5	9.4	54%
	4	10.1	6.2	63%
16 * 16	2	86.5	56.0	54%
	4	53.3	30.7	74%
	8	36.8	24.3	51%
32 * 32	2	594.2	374.1	59%
	4	334.3	193.5	73%
	8	205.3	106.0	94%
	16	142.6	94.6	51%

* UIUC code source

Table 5.3 Performance Comparison for Dynamic Programming

Problem size	# of PEs	Execution time (mS)		Improvement in speed
		Manual	Hypertool	
4	2	1.25	0.33	279%
6	2	3.43	0.89	285%
8	2	7.11	2.13	234%
	4	5.41	1.35	301%
12	2	20.91	6.97	200%
	4	15.29	4.42	246%

Table 5.4 Performance Comparison for Matrix Multiplication

Matrix size	# of PEs	Execution time (mS) Manual*	Automatic	Improvement in speed
8 * 8	4	9.7	7.9	23%
9 * 9	4	17.9	12.6	42%
16 * 16	4	68.0	61.0	11%
	16	19.4	15.3	27%
17 * 17	4	95.6	77.3	24%
	16	35.8	20.4	76%
32 * 32	4	515.7	481.3	7%
	16	136.1	128.4	6%
	64	39.5	30.2	31%
64 * 64	4	4027.0	3825.1	5%
	16	1033.7	956.4	8%
	64	273.1	239.2	14%
128 * 128	64	2067.3	1906.7	8%

* JPL code source

Table 5.5 Performance Comparison for Bitonic Sort

Problem size	# of PEs	Execution time (mS)		Improvement in speed
		Manual	Automatic	
64	4	10.7	10.2	5%
	8	10.1	9.1	11%
	16	10.9	9.9	10%
128	4	21.1	20.4	3%
	8	18.2	17.1	6%
	16	17.5	16.3	7%
	32	18.9	18.4	3%
256	4	44.0	43.1	2%
	8	35.2	33.3	6%
	16	31.3	29.5	6%
	32	30.6	29.9	2%
512	4	89.2	87.5	2%
	8	71.2	68.4	4%
	16	54.3	53.4	2%
	32	157.9	138.4	2%
1024	8	143.1	137.7	4%
	16	118.2	114.3	3%
	32	102.3	101.1	1%
2048	16	236.0	228.7	3%
	32	202.1	200.5	1%

* JPL code source

Table 5.6 The Effect of Estimation on Performance (Laplace Equation)

Problem size	Execution time (mS) Real	Estimated	Difference in speed
4*4	1.86	1.86	0%
8*8	4.23	4.26	-0.7%
16*16	12.78	12.82	-0.3%
32*32	44.84	44.88	-0.1%
64*64	168.8	168.8	0%
128*128	655.9	655.9	0%

Table 5.7 The Effect of Estimation on Performance (Bitonic Sort)

Problem size	Execution time (mS) Real	Estimated	Difference in speed
64	9.20	9.20	0%
128	17.08	17.10	-0.1%
256	33.17	33.30	-0.4%
512	68.03	68.31	-0.4%