Computer-Aided Programming for Message-Passing Systems: Problems and a Solution

MIN-YOU WU, MEMBER, IEEE, AND DANIEL D. GAJSKI, SENIOR MEMBER, IEEE

As the number of processors and the complexity of problems to be solved increase, programming multiprocessing systems becomes more difficult and error-prone. Program development tools are necessary since programmers are not able to develop complex parallel programs efficiently. Parallel models of computation, parallelization problems, and tools for computer-aided programming (CAP) are discussed. As an example, a CAP tool that performs scheduling and inserts communication primitives automatically is described. It also generates the performance estimates and other program quality measures to help programmers in improving their algorithms and programs.

I. 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. This 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 maintains that parallelization is a complex problem that can be performed only 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 maintains 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 the role of tools for developing programs on message-passing systems. In section II, we address some essential issues in parallelization and define several types of efficiency losses that determine the performance of a multiprocessing system. Two basic problems, namely, partitioning and communication, are discussed in detail. In section III, we describe program development methodologies and the partitioning and merging methods. A CAP tool, called Hypertool, is introduced in section IV as an example of programming tools for multiprocessing. Hypertool generates parallel code by automatic scheduling and synchronization insertion. Hypertool also generates performance estimates and quality measures for the developed parallel code.

Manuscript received June 30, 1988; revised March 30, 1989. This work was supported in part by the National Science Foundation under grant CCR-8700738.

M.-Y. Wu is with the Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520-2158.

D. D. Gajski is with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

IEEE Log Number 8933026.

II. Parallelization and Efficiency

A. Parallelization Issues for Multiprocessing Systems

A multiprocessing system may be modeled as in Fig. 1. Several identical PEs are connected by a communication network. Each PE has a main processor (MP) and a memory module (MM). Some systems may have one or more communication coprocessors (CCPs) in each PE. Two types of multiprocessing systems can be identified by two types of memory organization. In a shared memory system, communication between PEs is carried out via a shared memory.

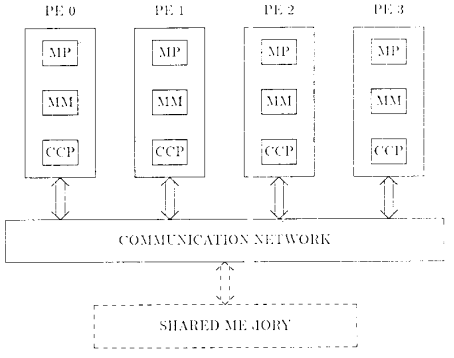


Fig. 1. Multiprocessor system model.

0018-9219/89/1200-1983$01.00 © 1989 IEEE

To exchange data, the producing processor stores data in 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.

The success of multiprocessing technology depends on successful parallelization of application problems [1], which in turn requires good partitioning of problems and minimization of communication between partitions. Parallelization problems include algorithm design, data and operation partitioning, granularity determination, load balancing, and minimization of network contention.

First the programmer must design an efficient parallel algorithm for the application problem. After algorithm development, the problem needs to be partitioned. Partitioning can be performed in two ways: "data partitioning" or "operation partitioning." The former partitions data and, consequently, assigns related operations to PEs. The later partitions operations and allocates the corresponding data to PEs. For many applications, the data partitioning is straightforward. It is the most suitable in cases where the computation amount associated with data is evenly distributed. On the other hand, although operation partitioning may require more complex analysis, it may be better suited for problems with irregular structures. With either approach, the partitioning style must be selected. For example, computation involving matrices may be partitioned along matrix rows, columns, or blocks. The basic rule suggests partitioning along data dependencies without cutting dependencies, which in turn reduces the amount of communication between partitions. With the selected partitioning style, the grain size of each partition must be determined. Usually, fine-grained partitions have more parallelism than crude-grained partitions. Unfortunately, they also have more dependencies.

Ideally, a problem should be partitioned into as many as possible independent tasks with no communication among these tasks. However, for most practical applications, there are dependencies between tasks. Data must be allocated to minimize breaking dependencies and fit communication network topologies. In the multiprocessing system of Fig. 1, the basic method is to allocate data to the local memory module of the PE that consumes the data. In many cases, a data item must be shared by several PEs. The data must be transferred through an interconnection network. To reduce communication, the data must be allocated to the PE that uses it most often. Improper data allocation leads to large communication overhead and PE suspension. More seriously, it may cause network contention when the amount of communication exceeds a certain percentage of the network capacity. To reduce network contention, mapping of tasks to PEs must minimize the network traffic in addition to the minimization of dependencies among tasks. Even when communication traffic is not heavy, "hot spots" may occur if many PEs try to access a set of data in the same memory module simultaneously [2]. In a shared memory system, hot spots may cause serious network contention, and in the worst case, saturate the entire network. One solution to hot spots is using expensive combining network. Another solution is to distribute the data to different memory modules. In a message-passing system, this problem is not so serious since such data can be broadcast to many PEs.

Improper partitioning and data allocation may cause large overhead, which leads to efficiency loss. Unbalanced load also leads to performance degradation when some PEs are suspended while waiting for data from other PEs. Another source of efficiency loss is parallel algorithm design.

B. 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 running on a single PE 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.

1) 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 [3], 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 [4]. The former converges much more slowly than the latter. Fig. 2 shows the number of iterations for convergency of the two algorithms. Note that the efficiency loss may be several orders of magnitude.

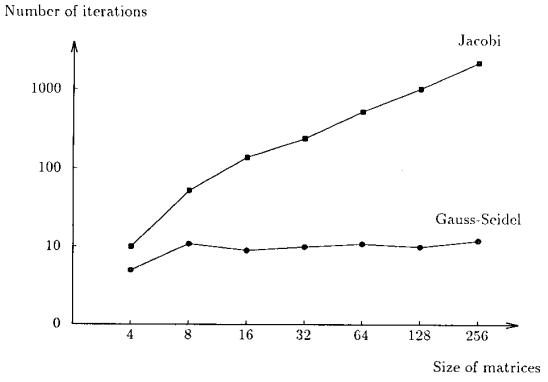


Fig. 2. Number of iterations for convergency of Jacobi and Gauss-Seidel algorithms.

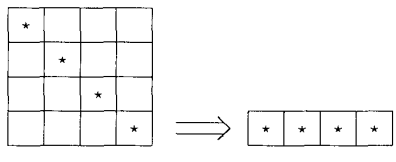
2) Efficiency Loss from Coding: When a parallel algorithm is coded into a parallel program, overhead is introduced, which causes efficiency loss. This overhead includes communication overhead, PE initializations, selection statements, duplication of operations, etc. The efficiency

loss from coding is defined as

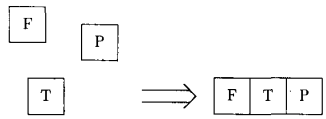
$$EL_C = \left(\sum_{i=1}^{N} T_{R_i}\right) \Big/ T_A$$

where T_{R_i} is the running time (not including the suspension time) of PE i accumulated during execution.

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, Fig. 3(a) shows the



(a)



F — Force
P — Position
T — Temperature

(b)

Fig. 3. Packing. (a) Same data type. (b) Different data types.

diagonal elements of a matrix being packaged and then sent. The other is to pack different types of elements. In Fig. 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. 4 may

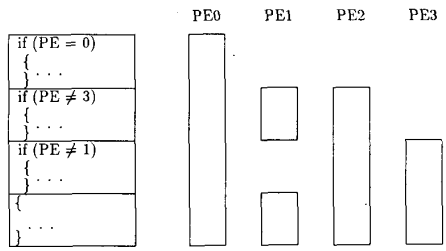


Fig. 4. Conditional statements to switch code segments for PEs.

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. 5(a), the statement $a = b * c$ is executed on one PE, and the result broadcasts to each PE. In Fig. 5(b), $a = b * c$ is executed on each PE, so that no communication is required.

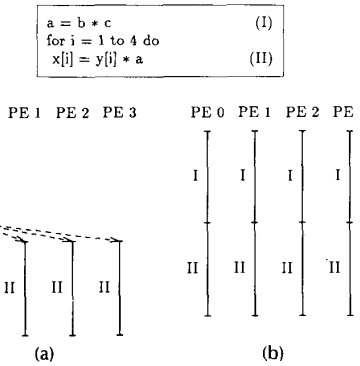


Fig. 5. Duplication operations. (a) PE 0 executes $a = b * c$ and broadcasts a. (b) All PEs execute $a = b * c$.

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.

3) Efficiency Loss from Processor Suspension: This efficiency loss is defined as

$$EL_P = NT_P \Big/ \sum_{i=1}^{N} T_{R_i}$$

where T_P is the execution time of the parallel program on N PEs. Since $T_{R_i} + T_{B_i} = T_P$ $(i = 1, 2, \cdots, N)$, where T_{Bi} is the total suspension times for PE i,

$$EL_P = 1 + \left(\sum_{i=1}^{N} T_{B_i} \Big/ \sum_{i=1}^{N} T_{R_i}\right).$$

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. 6(a), we divide a program into two equal parts without dependencies and load them onto two PEs, resulting in a balanced load. However, if one part depends on the other, even though the load is "balanced," one of the PEs becomes sus-

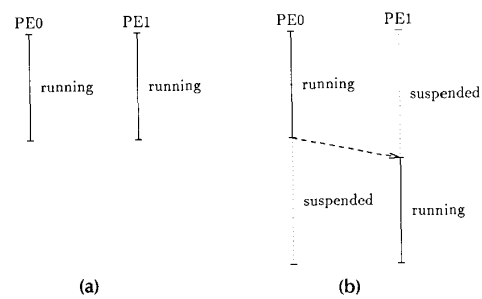


Fig. 6. Load balances. (a) Without suspension. (b) With suspension.

pended, as shown in Fig. 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 is the load 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 preceding efficiency losses. Overall, the efficiency can be expressed as $\mu = 1/EL_AEL_CEL_P$, and the speedup is $S = N/EL_AEL_CEL_P$. In fact, the term "speedup," frequently used by many people, is actually the speedup for the same algorithm—$S_a = N/EL_CEL_P$—or the speedup for the same code—$S_c = N/EL_P$.

C. 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 must 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 [5], 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 on each other, the alignment method and minimum-distance method [5] may be used to reduce or even eliminate these dependencies. If dependencies still exist after applying these methods, tasks must exchange data. Problem data must be partitioned to minimize dependencies among tasks and maximize parallelism. These two goals, however, may contradict each other. Fig. 7 shows an example of the Gauss-Seidel algorithm. When we partition the matrix into squares, we have less dependencies and less parallelism (Fig. 7(a)). On the other hand, if we partition it into strips, we have more parallelism and more dependencies (Fig. 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 subblock of matrix A and a subblock of B [3]. 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.

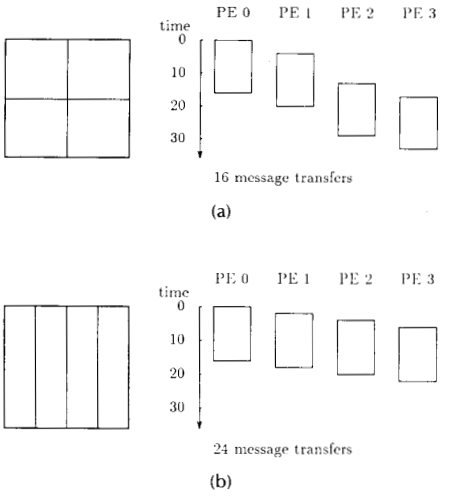


Fig. 7. Different partitioning for Gauss-Seidel algorithm.

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. If the computation overhead is dominant, we consider the "task granularity." The task granularity refers to the average amount of computation in a task. If the major overhead is from communication, the "communication granularity" is considered, which refers to the average amount of computation between two consecutive communications [6]. Fine granularity exploits more parallelism, but increases efficiency loss. Crude granularity, on the other hand, reduces efficiency loss but decreases parallelism [6]. Fig. 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

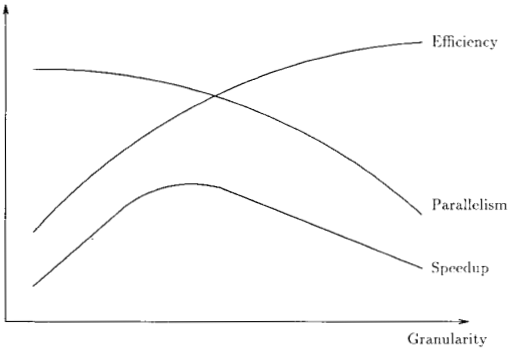


Fig. 8. Relationship between granularity, parallelism, efficiency, and speedup.

speedup. To determine proper granularity, two factors must be considered: dependency and overhead. When there are few dependencies, fine granularity may be used without increasing efficiency loss. When many dependencies exist, the overhead of communication becomes the dominant 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 [7]. The dataflow approach becomes practical when overhead is reduced to several operations. The spectrum of granularity is shown in Fig. 9.

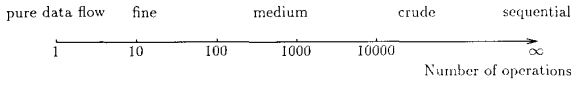


Fig. 9. Spectrum of granularity.

In general, granularity should be comparable to overhead to keep efficiency loss small [8]. Furthermore, granularity should be crude enough to tolerate the message trnasmission 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.

III. Program Development Aids

A. 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 [9]. 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 [3], [10], [11]. 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 highly dependent on programmers' experience and problem regularity [12]. For problems with irregular structures, the combined effect of dependencies and overhead is hard to estimate manually, causing poor load distribution. For example, load balancing may require some code segments to be moved from one PE to another, which would increase dependencies. These dependencies may lead to more processor suspension, resulting in even worse load imbalance. This approach could cause high efficiency losses from coding and processor suspension. Furthermore, since scheduling and communication is performed manually, debugging programs is difficult.

B. 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 [13] is a parallel programming environment for message-passing systems, which has been ported to the Cosmic Cube [14]. POKER provides a graphic representation of communication structure. DAPP [15] 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 [16] 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 [17]. PTOOL [18] 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. However, PTOOL only tests loops for independence and does not provide partitioning and synchronization mechanisms for nonparallel loops. CAMP [19] partitions both parallel and nonparallel 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 ech partitioning strategy. Brandes and Sommer [20] 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 the following section is based on the partitioning and 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.

C. The Partitioning and Merging Approach

Dependencies are defined by partitioning. There is no dependency when a program is treated as a single partition. This is one extreme. In the other extreme, a program can be partitioned at the operation level with the finest granularity and all dependencies. This partition is called the "intrinsic partition." Most partitioning styles fall in the middle, that is, one partition includes several operations. In this way, dependencies are reduced compared to the intrinsic partition. One of the partitioning approaches is to partition a problem into P tasks with the one-to-one mapping

between the tasks and the PEs, where P is the number of PEs. It is called "one-step partitioning" and is suitable for problems with regular structures [3]. However, it is difficult to balance load and minimize dependencies in one-step partitioning for the problems with irregular structures. The partitioning and merging (P&M) approach is a "two-step partitioning." The first step is to partition a problem into processes of small sizes. The partitioning strategy may be decided by a programmer or an automatic partitioner. A macro dataflow graph is generated automatically by the dependency analysis of these processes. In the second step, a scheduler is 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 [21]. Many scheduling algorithms may be used for this purpose. Most are based on the critical path algorithm [22]. 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 [23].

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

A 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 [24]–[26]. 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 $\{n_1, n_2, \cdots, n_n\}$ are connected by a set of directed edges $\{e_1, e_2, \cdots, e_e\}$, as shown in Fig. 10.

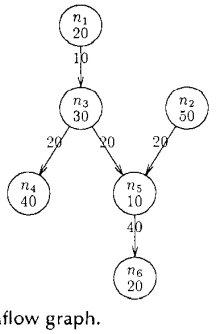


Fig. 10. Macro dataflow graph.

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. 11(a), when nodes 2 and 5, and 3 and 4, are merged, the number of dependencies 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. 11(b), the graph contains three dependencies without reduction of parallelism.

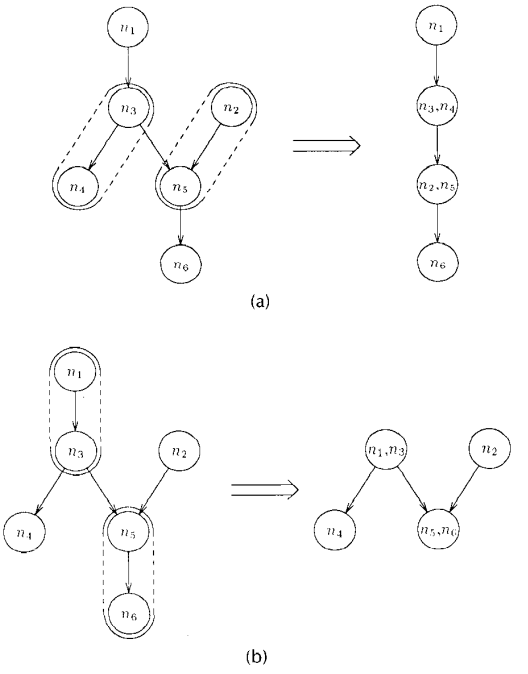


Fig. 11. Process merging. (a) With loss of parallelism. (b) Without loss of parallelism.

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 a 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.

IV. Hypertool

In this section, we present a version of Hypertool, which generates macro dataflow graphs, performs static scheduling and mapping, inserts communication primitives, and generates parallel codes. The performance estimation and quality measures also will be described.

The system diagram of our Hypertool is shown in Fig. 12. First, a user develops a proper algorithm, performs partitioning, and writes a program as a set of procedures. This program is automatically converted into the parallel program for a target machine by parallel code synthesis and optimization. Fig. 13 shows the organization of the program synthesis and optimization module. The lexer and the parser recognize data dependencies and user-defined parti-

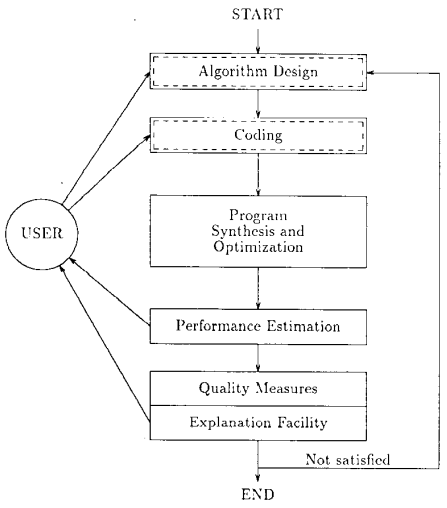


Fig. 12. Hypertool.

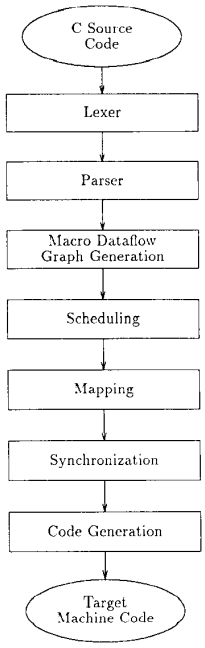


Fig. 13. Program synthesis and optimization.

tions. 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. There are many scheduling algorithms [22], [27], [28]; however, these algorithms did not model transmission time, since they assumed that the data transmission between PEs did not take any time. This is not true, especially for message-passing systems. The data transmission time is a significant factor that affects the overall performance of a system. A modified critical-path algorithm and a mobility-directed algorithm is used in Hypertool [29]. The mapping submodule maps each task to a physical PE in a given topology by minimizing network traffic. We 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 [30]. After scheduling and mapping are completed, the synchronization module inserts the communication primitives in a correct sequence. Finally, the code generator generates target machine code for each PE.

Hypertool then generates performance estimates. The performance estimator is implemented based on the SIMON [31]. SIMON is an event driven simulator which models the execution of a parallel application program on a message-passing system. Message transmissions between processors are simulated by a switch model. The switch model may be an Ethernet, a cross bar, or a hypercube network. The hypercube network is used in this performance estimator. SIMON generates the execution time and suspension time for each processor by counting the number of instructions executed. We have modified SIMON so that it can generate time for communication overhead in each processor. SIMON also generates statistics of the number of message transmissions, message lengths, the transmission time for each message, and the network delay for each communication channel. The transmission time and network delay is calculated from message length, channel bandwidth, and message starting time. When execution completes, the explanation facility displays graphic representation of data dependencies between PEs, parallelism, and load distribution [32]. The explanation facility also indicates which message causes a PE to suspend. The performance estimator and the explanation facility provides information so that if the performance is not satisfactory, the programmer can change the partitioning strategy and the size of the partitions for performance improvement.

Hypertool is currently running on a Sun workstation under UNIX. 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. Tables 1 and 2 show performance comparison of the Laplace equation and Gaussian elimination. 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. These problems have less regular structures so that good load balance is difficult to obtain man-

Table 1 Performance Comparison for Laplace Equation

Matrix Size	# of PEs	Execution Time (ms)		Improvement in Speed
		Manual	Hypertool	
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 2 Performance Comparison for Gaussian Elimination

Matrix Size	# of PEs	Execution time (ms)		Improvement in Speed
		Manual	Hypertool	
4 × 5	2	3.0	1.9	58%
8 × 9	2	14.5	9.4	54%
	4	10.1	6.2	63%
16 × 17	2	86.5	56.0	54%
	4	53.3	30.7	74%
	8	36.8	24.3	51%
32 × 33	2	594.2	374.1	59%
	4	334.3	193.5	73%
	8	205.3	106.0	94%
	16	142.6	94.6	51%

ually. 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. For more regular problems, such as the matrix multiplication, automatic scheduling gives performance similar to that of manual scheduling. However, developing programs on Hypertool is much easier.

V. 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 the parallelization tasks and back-annotating some of the quality measures has become a necessity to programmers.

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. Second, since communication primitive insertion is performed by Hypertool, many errors, such as incorrect computation sequence and deadlock, are eliminated. Finally, the program development tool generates better parallel codes since it uses good scheduling algorithms. 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.

Acknowledgment

The authors wish to acknowledge the contributions of Wei Shu, for her programming work, and Andrew Kwan, for carefully reading the manuscript.

References

[1] D. D. Gajski and J. Peir, "The essential issues in multiprocessor systems," *IEEE Computer*, vol. 18, pp. 9–27, June 1985.

[2] G. F. Pfister and V. A. Norton, "'Hot spot' contention and combining in multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-34, pp. 934–948, Oct. 1985.

[3] 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.* New York, NY: Prentice-Hall, 1988.

[4] A. Jennings, *Matrix Computation for Engineers and Scientists.* New York, NY: John Wiley, 1977.

[5] J. K. Peir, D. D. Gajski, and M. Y. Wu, "Programming environments for multiprocessors," in *Proc. of Int. Seminar on Scientific Supercomputers*, Feb. 1987, pp. 47–68.

[6] J. Mohan, A. Jones, E. Gehringer, and Z. Segall, "Granularity of parallel computation," in *Proc. of 8th Ann. Hawaii Int. Conf. Syst. Sciences*, 1985, pp. 249–256.

[7] J. D. Brock, A. R. Omondi, and D. A. Plaisted, "A multiprocessor architecture for medium-grain parallelism," in *Proc. 6th Int. Conf. Distributed Computing Syst.*, 1986, pp. 167–174.

[8] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Trans. Comput.*, vol. C-38, pp. 408–423, Mar. 1989.

[9] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graph and compiler optimizations," in *Proc. of 8th ACM Symp. Principles on Programming Lang.*, Jan. 1981.

[10] C. L. Seitz, "The COSMIC cube," *Commun. ACM*, vol. 28, pp. 22–33, Jan. 1985.

[11] J. L. Gustafson and G. R. Montry, "Programming and performance on a cube-connected architecture," in *IEEE COMPCON*, Mar. 1988, pp. 97–100.

[12] A. H. Karp and R. G. Babb II, "A comparison of 12 parallel Fortran dialects," *IEEE Software*, pp. 52–67, Sept. 1988.

[13] L. Snyder, "Parallel programming and the POKER programming environment," *IEEE Computer*, pp. 27–36, July 1984.

[14] L. Snyder and D. Socha, "POKER on the Cosmic cube: The first retargetable parallel programming language and environments," in *Proc. Int. Conf. on Parallel Processing*, Aug. 1986, pp. 628–635.

[15] W. F. Appelbe and C. McDowell, "Anomaly detection in parallel Fortran programs," in *Proc. Workshop on Parallel Processing Using the HEP*, May 1985.

[16] J. Purtilo, D. A. Reed, and D. C. Grunwald, "Environments for prototyping parallel algorithms," in *Proc. Int. Conf. on Parallel Processing*, Aug. 1987, pp. 431–438.

[17] D. A. Padua, D. J. Kuck, and D. L. Lawrie, "High speed multiprocessor and compilation techniques," *IEEE Trans. Comput.*, vol. C-29, pp. 763–776, Sept. 1980.

[18] R. Allan, D. Baumgartner, K. Kennedy, and A. Porterfield, "PTOOL: A semiautomatic parallel programming assistant," in *Proc. Int. Conf. on Parallel Processing*, Aug. 1986, pp. 164–170.

[19] J. K. Peir and D. D. Gajski, "CAMP: A programming aide for multiprocessors," in *Proc. Int. Conf. on Parallel Processing*, Aug. 1986, pp. 475–482.

[20] T. Brandes and M. Sommer, "A knowledge-based parallelization tool in a programming environment," in *Proc. Int. Conf. on Parallel Processing*, Aug. 1987, pp. 446–448.

[21] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Cambridge, MA: MIT Press, 1989.

[22] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Res.*, vol. 9, no. 6, pp. 841–848, 1961.

[23] B. Kruatrachue and T. Lewis, "Grain size determination for parallel processing," *IEEE Software*, pp. 23–32, Jan. 1988.

[24] E. Best and B. Randell, "A formal model of atomicity in asynchronous systems," *Acta Informat.*, vol. 16, pp. 93–124, 1981.

[25] P. Hudak and B. Goldberg, "Serial combinators: Optimal grains of parallelism," in *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science.* Berlin: Springer-Verlag, 1985, pp. 382–388.

[26] R. G. Babb, "Programming the HEP with large-grain data flow techniques," in *MIMD Computation: HEP Supercomputer and Its Applications*, J. S. Kowalik, Ed. Cambridge, MA: The MIT Press, 1985, pp. 203–227.

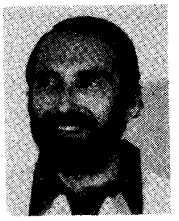
[27] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzales, "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. Comput.*, vol. C-21, pp. 137–146, Feb. 1972.

[28] B. Bussell, E. Fernandez, and O. Levy, "Optimal scheduling for homogeneous multiprocessors," in *Proc. IFIP Congress 74.* Amsterdam: North-Holland, 1974, pp. 286–290.

[29] M. Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," to appear in *IEEE Trans. Computers*.
[30] M. Hanan and J. Kurtzberg, "A review of the placement and quadratic assignment problems," *SIAM Rev.*, vol. 14, pp. 324–342, Apr. 1972.
[31] R. M. Fujimoto, "SIMON: A simulator of multicomputer networks," Tech. Rep. UCB/CSD/83/140, University of California at Berkeley, Sept. 1983.
[32] M. Y. Wu and D. D. Gajski, "A programming aid for message-passing systems," in *Proc. 3rd SIAM Conf. on Parallel Processing for Scientific Computing*, G. Rodrigue, Ed. Philadelphia, PA: Soc. for Indust. and Appl. Math., 1989, pp. 328–332.

Min-You Wu (Member, IEEE) received the M.S. degree in electrical engineering from the Graduate School of Academia Sinica, Beijing, China, in 1981, and the Ph.D. degree in electrical engineering from Santa Clara University, Santa Clara, CA, in 1984.

From 1986 to 1987 he was a Research Associate of the Coordinated Science Laboratory of University of Illinois at Urbana-Champaign, and from 1987 to 1988 he was a Visiting Researcher at the Department of Information and Computer Science, University of California, Irvine. Currently he is an Associate Research Scientist of the Department of Computer Science, Yale University, New Haven, CT. His research interests include parallel computation and silicon compilation.



Daniel D. Gajski (Senior Member, IEEE) received the Dipl. Ing. and M.S. degrees in electrical engineering from the University of Zagreb, Yugoslavia, and the Ph.D. degree in computer and information sciences from the University of Pennsylvania, Philadelphia.

After ten years of industrial experience in digital circuits, switching systems, supercomputer design, and VLSI structures, he spent ten years in academia, in the Department of Computer Science at the University of Illinois, Urbana-Champaign. Presently he is a Professor in the Department of Computer Science at the University of California at Irvine. His research interests are in multiprocessor architectures and programming tools, intelligent silicon compliers and expert systems for VLSI, and the science of design.