Hypertool: A Programming Aid for Message-Passing Systems

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

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 paper discusses programming assistance and automation concepts and their application to a program development tool for message-passing systems called Hypertool. It performs scheduling and handles the communication primitive insertion automatically. Two algorithms, based on the critical-path method, are presented for scheduling processes statically. Hypertool also generates the performance estimates and other program quality measures to help programmers in improving their algorithms and programs.

Index Terms—Automatic scheduling, message-passing systems, programming assistance, program automation, program development tool, synchronization insertion.

I. INTRODUCTION

MANY commercial message-passing systems have been introduced, such as Intel iPSC [1], Ametek System/14 [2], Ncube/ten [3], FPS T Series [4], and the Connection Machine [5]. The recent approach to program these machines is in the single program multiple data (SPMD) style. In this style, each processing element (PE) runs the same program but executes different code segments depending on its PE id and data in its local memory [6]. Programmers find this approach easy to develop parallel programs. However, to achieve the balanced load among PE's in the SPMD style is not a trivial task since the computation load for each PE may be different, especially for those problems with irregular structures. In these cases, computation loads on overloaded PE's must be moved to underloaded PE's.

There are two extreme approaches to develop programs in the SPMD style. One school of thought believes that these problems are complex and should be left to programmers [7]. However, programmers are error-prone to handle those tedious chores, such as communication primitive insertion. For example, system deadlock is the most common problem, and is difficult to detect once the program has been developed. Also, the developed programs are not portable since each machine has different communication primitives [8]. The other school of thought believes in restructuring compilers that will

M.-Y. Wu was with the Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92717. He is now with the Department of Computer Science, Yale University, New Haven, CT 06520.

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

IEEE Log Number 9036063.

extract parallelism and restructure sequential programs into parallel programs automatically [9]. However, the complete analysis for exploiting parallelism is very complex. Furthermore, the parallelism revealed in this way is restricted by the algorithms embodied in the sequential programs. We advocate an approach that falls between the above two extremes.

We believe the parallelization is a very complex problem which can only be fully solved by a human expert. Program development tools can assist users to solve creative chores, such as algorithm design. On the other hand, some of parallelization chores can be automated, especially scheduling and synchronization [10]. Program development tools can 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 a programming aid, named Hypertool, for automatic scheduling and synchronization on message-passing systems. Hypertool takes a partitioned program as input and generates parallel code for execution on message-passing machines. Hypertool also generates performance estimates and quality measures for the parallel code. We will discuss static scheduling problems and present new scheduling algorithms. The algorithms for mapping and synchronization will be discussed also. In particular, several issues for programming message-passing systems and computer-aided programming tools are discussed in Section II. Our Hypertool and its programming model are described in Section III. Following that, scheduling, mapping, and synchronization insertion are discussed in Sections IV, V, and VI, respectively. Several annotated examples are given in Section VII to illustrate the method and demonstrate the usefulness of the programming aid.

II. COMPUTER-AIDED PROGRAMMING

A programmer writing programs for a message-passing system must partition each problem into processes, group these processes into tasks, assign each task to a PE, and insert synchronization primitives for proper execution [11], [12].

A program segment that is not partitioned further is called a *process*, and is counted as one *unit of computation*. When two processes executing on different PE's must exchange the data, the data communication time becomes the overhead that slows down the computation. If this overhead is sizable, the maximum parallelism found in the program may not generate the optimum speedup. For this reason, several processes are merged together to form a *task*, which can be thought of as

one *unit of allocation*. The number of operations in each task defines the *operation granularity*, and the number of data items in each task defines the *data granularity*. The best speedup is obtained by proper granularity. A scheduler can be used in this purpose which modifies granularity by merging processes into tasks [13].

After merging processes into tasks, each task is assigned to a PE in a given topology. To reduce network traffic, two tasks that exchange data should be mapped to the neighboring PE's. Since this is not always possible, the assignment heuristic should attempt to minimize the total communication traffic in the network. For proper communication of tasks on different PE's, synchronization primitives must be inserted into a program. The tasks of partitioning, scheduling, mapping, and synchronization can be automated through an interactive programming aid.

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 [14] is a parallel programming environment for message-passing systems, which 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.

The other type of tool performs some program transformation. Most tools of this type are based on the theory of program restructuring [9]. PTOOL [16] 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. It also transforms control dependencies into data dependencies. However, PTOOL only tests loops for independence and does not provide partitioning and synchronization mechanisms for nonparallel loops. CAMP [17] partitions both parallel and nonparallel loops, and reduces dependencies 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.

III. HYPERTOOL

The programming aid presented here aims to increase programming productivity and take advantage of tedious tasks that computers perform better than humans. It takes user partitioned program as its input, automatically allocates these partitions to PE's, and inserts proper synchronization primitives where needed. The program development methodology to be used with Hypertool is shown in Fig. 1. First, a designer develops a proper algorithm, performs partitioning, and writes a program as a set of procedures. The program looks like a sequential program and it can run on a sequential machine for the purpose of debugging. This program is automatically converted into the parallel program for a message-passing target
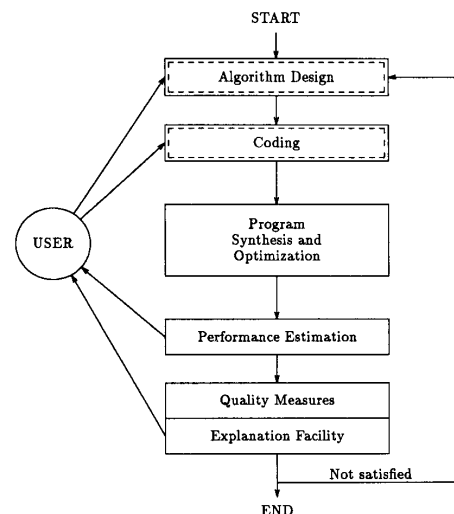

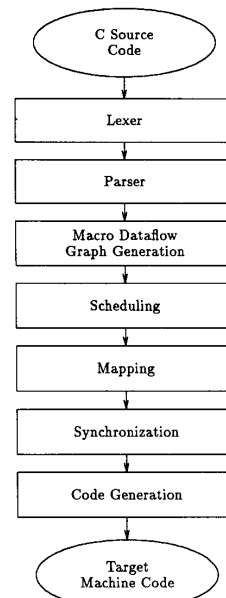Fig. 1.   The development tool.

Fig. 2.   Program synthesis and optimization.

machine by parallel code synthesis and optimization. Hypertool then generates performance estimates, including execution time, communication time, suspension time for each PE, and network delay for each communication channel. The explanation facility displays data dependencies between PE's, as well as parallelism and load distribution [18]. If the designer is not satisfied with the result, he/she will attempt to redefine the partitioning strategy and the size of partitions using the information provided by the performance estimator and the explanation facility.

Fig. 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 genera-

```
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 ***************************/

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

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

        for j = i to N do       /* do parallel operations on N-i+1 columns */
            call UpdateMtx(matrix[i][j],matrix[i+1][j],vector[i+1],i);
                                /* it 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                */

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

(a)

Fig. 3. A parallel Gaussian elimination algorithm.

tion 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. Mapping has little effect on the execution time of a dataflow graph. However, improper mapping does increase network traffic, which increases network contention. After scheduling and mapping are completed, the synchronization module inserts the communication primitives. Finally, the code generator generates target machine code for each PE.

To facilitate automation of program development, we use a programming style in which a program is composed of a set of procedures called from the main program. A procedure is an indivisible unit of computation to be scheduled on one processor. The grain sizes of procedures are determined by the programmer, and can be modified with Hypertool. Fig. 3 shows an example, a parallel Gaussian elimination algorithm, which partitions a given matrix by columns. The procedures FindMax and UpdateMtx are called several times. The control dependencies can be ignored, so that a procedure call can be executed whenever all input data of the procedure are available. 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. For a static program, the number of procedures are known before program execution. Data dependencies among the procedural parameters define a macro dataflow graph.

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 shows the macro dataflow graph of the program in Fig. 3. Note that only the parallel parts of Fig. 3 and the messages transferred among these procedures are shown in Fig. 4. A macro dataflow graph consists of a set of nodes $\{n_1, n_2, \cdots, n_n\}$ and a set of edges $\{e_1, e_2, \cdots, e_e\}$. 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 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 procedure, and the weight of the edge is equal to the transmission time of the message. In Fig. 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 to 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. In static scheduling, the number of nodes is known before program execution. The execution time of a node is obtained by running the corresponding procedure. The transmission time of a message is estimated by using the message startup time, message length, and communication channel bandwidth. We assume that the message transmission time is for neighbor communication. Nonneighbor communication takes a little more time. We also assume network traffic is not too heavy. Therefore, network contention is ignored in our model.

This programming style has good modularity, which is necessary for developing general application programs. Also, it

```
/********************** 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;

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

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

        for i = k+1 to N-1 do         /* calculate multiplying factors     */
            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;                */

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


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

(b)

Fig. 3. (Continued).

is system independent since communication primitives are not specified within the program. Such a program can be executed sequentially or in parallel. Furthermore, dynamic scheduling becomes simpler since it partitions a program into many procedures.

IV. SCHEDULING

In this section, we discuss methods for scheduling a macro dataflow graph. We will describe *static nonpreemptive* scheduling algorithms for homogeneous multiprocessors.

Critical-path scheduling has been addressed by Hu [19]. The critical-path algorithm has been proved to be near optimal [20], [21]. Basically, the critical-path algorithm assigns a label to each node according to the longest path from this node to the end point. It performs well for a bounded number of PE's. Ramamoorthy *et al.* developed algorithms to determine the minimum number of PE's required to process a program in the shortest possible time [22]. They used exhaustive search, which is not acceptable for large programs. Bussell *et al.* proposed an alternative method to reduce the number of PE's [23], but the efficiency of the algorithm is still dependent on an estimate of the number of processors. More importantly, these algorithms did not model transmission time, that is, they assumed that the data transmission between PE's did not take any time. This is not true, however, for most message-passing systems. The data transmission time is a significant factor which affects the overall performance of a system, which must be considered in modeling. Kruatrachue and
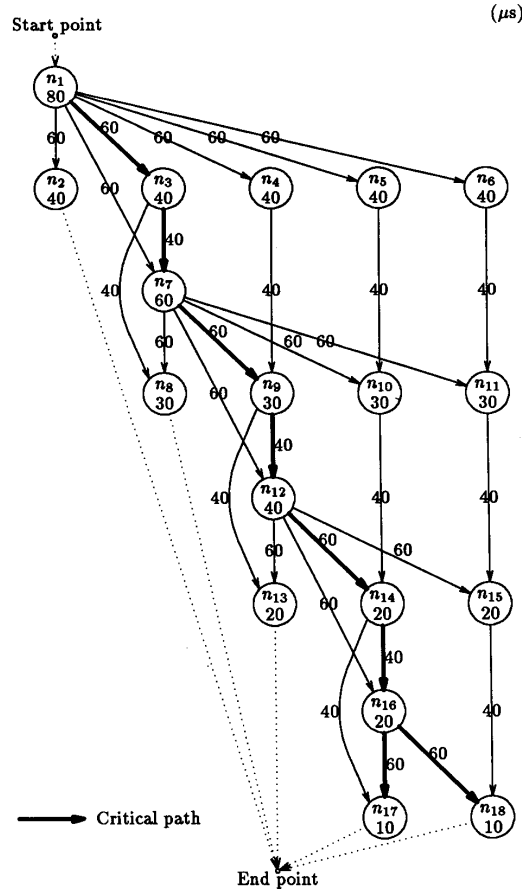
Fig. 4. The macro dataflow graph of Fig. 3 ($N = 4$).

Lewis have recently presented a model which assigned data communication time as weights of edges [24]. However, their scheduling algorithm is more suitable for fine granularity.

There are two objective functions to be considered for scheduling. The first function is the time taking to process a graph on bounded number of PE's. The second is the number of PE's required to process a graph in the shortest possible time. We will first modify the critical-path algorithm for a bounded number of PE's. But since this modified critical-path (MCP) algorithm is not able to efficiently determine the minimum number of PE's required to execute a program, a new algorithm, called mobility-directed (MD) scheduling, is presented. This algorithm generates solutions for the number of PE's required.

To define the scheduling algorithms succinctly we will define several concepts. First, we describe the *as-soon-as-possible* (*ASAP*) and the *as-late-as-possible* (*ALAP*) bindings [25] of a macro dataflow graph in order to determine mobilities. In an ASAP binding, the earliest possible execution is assigned to each node. The ASAP binding is created by moving forward through the macro dataflow graph. For example, the ASAP binding of the macro dataflow graph in Fig. 4 is shown in Fig. 5. The node n_1 is bound to the time

interval 0–80 μs. It then sends a message to nodes n_2, n_3, n_4, n_5, n_6, and n_7. When the messages arrive, at time 140 μs, these nodes are bound to the time interval starting at 140 μs. After performing the ASAP binding, each node is assigned an ASAP start time, denoted as $T_S(n_i)$. Conversely, the latest possible start time is assigned to each node for the ALAP binding, which is obtained by moving backward through the macro dataflow graph. The ALAP binding of Fig. 4 is shown in Fig. 6. Similarly, after performing the ALAP binding, each node is then assigned an ALAP time, denoted as $T_L(n_i)$.

The *moving range* of a node n_i is defined as the time interval from $T_S(n_i)$ to $T_L(n_i)$, in which node n_i may start its execution without delaying the execution of any other node on the critical path. The length of this range is defined as the *mobility* of node n_i, $M(n_i) = T_L(n_i) - T_S(n_i)$. The *relative mobility* of a node is defined by $M_r(n_i) = M(n_i)/w(n_i)$, where $w(n_i)$ is the weight of node n_i, that is, the execution time of the node. Moreover, we define $T_F(n_i) = T_L(n_i) + w(n_i)$ as the *latest finishing time* of node n_i. The *moving interval* of a node n_i is defined as the time interval from $T_S(n_i)$ to $T_F(n_i)$.

As an example, the ASAP time of node n_8 in Fig. 5 is 340, and its ALAP time in Fig. 6 is 630. Therefore, the moving
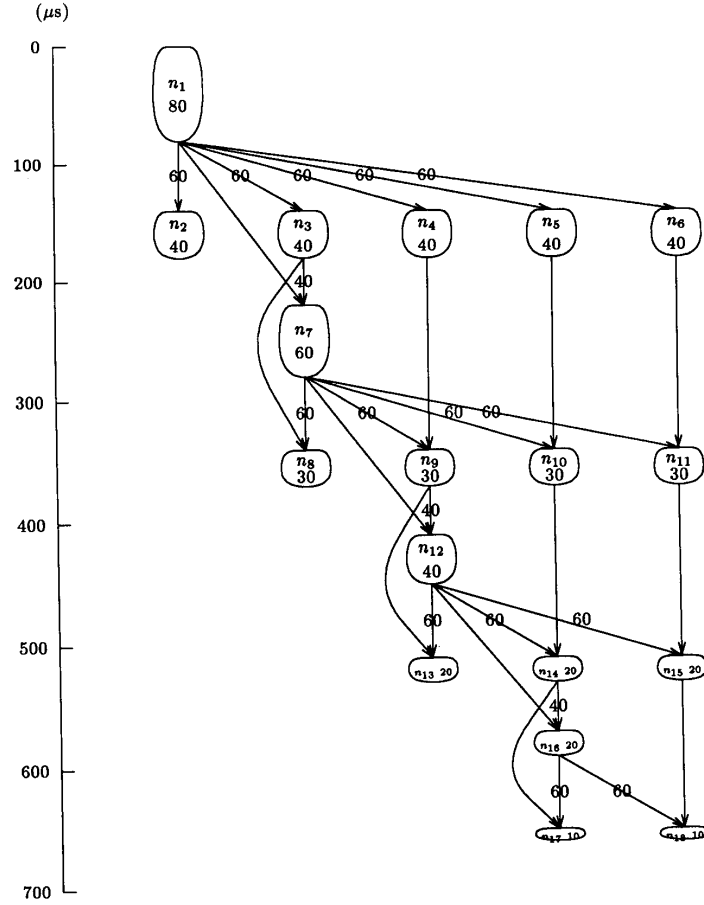
Fig. 5. The ASAP binding of Fig. 4.

range of n_8 is 340–630, and its mobility is 290. The relative mobility of the node is $M_r(n_8) = 290/30=9.7$. The moving ranges, mobilities, and relative mobilities of nodes in Fig. 4 are listed in Table I, sorted by relative mobilities.

All nodes with mobility equal to zero identify the *critical path* through the graph. The length of the critical path is the minimum time required for program execution. Obviously, there is at least one critical path in a macro dataflow graph, although several critical paths are possible. Moreover, if two nodes on the critical path are scheduled to a single PE, the weight of the edge connecting them becomes zero, and the critical path is shortened. Note that shortening a critical path may generate several new critical paths.

Now we describe a scheduling algorithm for a macro dataflow graph on a given number of PE's. This algorithm is based on the critical path algorithm presented by Sethi [26].

Algorithm 1: *Modified Critical-Path (MCP) Scheduling*:
Step 1: Perform the ALAP binding and assign the resulting ALAP time $T_L(n_i)$ ($i = 1, 2, \cdots, n$) to each node in the graph.
Step 2: For each node n_i create a list $l(n_i)$ which consists of T_L's of n_i and all its descendants in decreasing order. Sort

TABLE I
MOBILITIES OF THE NODES IN FIG. 4 (μs)

Node	Moving range	Mobility	Relative mobility
n_1	0 – 0	0	0
n_3	140 – 140	0	0
n_7	220 – 220	0	0
n_9	340 – 340	0	0
n_{12}	410 – 410	0	0
n_{14}	510 – 510	0	0
n_{16}	570 – 570	0	0
n_{17}	650 – 650	0	0
n_{18}	650 – 650	0	0
n_4	140 – 260	120	3
n_{10}	340 – 440	100	3.3
n_{15}	510 – 590	80	4
n_5	140 – 360	220	5.5
n_{11}	340 – 520	180	6
n_{13}	510 – 640	130	6.5
n_6	140 – 440	300	7.5
n_8	340 – 630	290	9.7
n_2	140 – 620	480	12

these lists in decreasing order lexicographically. According to this order, create a node list L.
Step 3: Schedule the first node in L to a PE that allows its earliest execution. Delete the node from L and repeat step 3 until L is empty. □

Fig. 6. The ALAP binding of Fig. 4.

The complexity of Step 1 is $O(n^2)$, since the ALAP binding is performed by the depth-first search. Step 2 requires $O(n \log n)$ operations to generate each list, and there are n lists, so it requires time of $O(n^2 \log n)$. Then sorting all lists requires $O(n^2 \log n)$ operations also. Therefore, the complexity of Step 2 is $O(n^2 \log n)$. The complexity of Step 3 and 4 is $O(n^2)$. Thus, the complexity of Algorithm 1 is $O(n^2 \log n)$.

Before discussing the scheduling method to minimize the number of PE's required to execute a graph, we give a fact for a node to be scheduled to a PE to which several other nodes have been already scheduled.

Fact 1: *Necessary and sufficient condition for scheduling a node to a PE*: Assume a node n_p is to be scheduled to PE m, to which l nodes, $n_{m_1}, n_{m_2}, \cdots, n_{m_l}$, have been scheduled. If the moving intervals of these nodes do not intersect with the moving interval of n_p, then n_p can be scheduled on PE m. Otherwise, assume the moving intervals of nodes $n_{m_i}, n_{m_{i+1}}, \cdots, n_{m_j}$ $(1 \le i \le j \le l)$ intersect the moving interval of n_p. n_p can be scheduled to PE m, if for each k $(i \le k \le j + 1)$,

$$w(n_p) \le \min(T_F(n_p), T_L(n_{m_k}))$$

$$-\max(T_S(n_p), T_S(n_{m_{k-1}}) + w(n_{m_{k-1}}))$$

where $w(n_i)$ is the weight of n_i; and
if $n_{m_{i-1}}$ does not exist, $T_S(n_{m_{i-1}}) = 0$ and $w(n_{m_{i-1}}) = 0$; and
if $n_{m_{j+1}}$ does not exist, $T_L(n_{m_{j+1}}) = \infty$.
Otherwise, the node cannot be scheduled to PE m. □

We now describe a mobility-directed algorithm to schedule a macro dataflow graph on an unbounded number of PE's.

Algorithm 2: *Mobility-Directed (MD) Scheduling*:

Step 1: Calculate relative mobilities for all nodes. Let L include all nodes initially.

Step 2: Let L' be the group of nodes in L with minimum relative mobility. Let n_i be a node in L' that does not have any predecessors in L'. Using Fact 1, schedule n_i on the first PE. If n_i cannot be scheduled on the first PE, schedule it on the second PE, and so on. When n_i is scheduled on PE m, all edges connecting n_i and other nodes already scheduled to PE m are changed to zero.

Step 3: If n_i is scheduled before node n_j on PE m, add an edge with weight zero from n_j to n_i in the graph. If n_i is scheduled after node n_j on the PE, add an edge with weight zero from n_j to n_i in the graph. This is to ensure there is no deadlock. Then check if the adding edges form a loop. If so, schedule n_i to the next available space.

Step 4: Recalculate relative mobilities for the modified graph. Delete n_i from L and repeat Steps 2, 3, and 4 until L is empty. □

The time for Step 1 includes $O(n^2)$ each for ASAP and ALAP bindings, and $O(n)$ for calculations. Thus, the complexity of Step 1 is $O(n^2)$. It takes $O(n)$ operations for scheduling a node on a PE and $O(n^2)$ operations for calculating relative mobilities. Therefore, the complexity of Algorithm 2 is $O(n^3)$.

V. MAPPING

The mapping algorithm presented in this section maps these virtual PE's used in the previous section to physical PE's connected in a predefined topology. A good mapping algorithm generates the minimum amount of communication traffic, reducing network contention. For best results, a traffic scheduling algorithm that balances the network traffic should be used [27]. However, traffic scheduling requires flexible-path routing which generates larger overhead. If network traffic is not too heavy, simpler algorithms which minimize total network traffic may be used. Since most problems being solved on message-passing systems are in this category, we will use an algorithm to minimize the total network traffic.

The mapping problem may be described as follows: given a virtual PE graph consisting of N_t nodes and E_t edges. A virtual PE graph is also called a task graph, which is generated by scheduling. Each node in this graph corresponds to a task, and each edge corresponds to messages transferred between two tasks. The weight of the edge, $w(e_i)$ $(i = 1, 2, \cdots, E_t)$, is the sum of transmission time of all messages between the two tasks. This task graph is to be mapped to a system graph. 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 PE's, with weight 1.

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 nonneighboring PE's. The corresponding message will be routed through the shortest path between the two PE's. The distance d between two PE's 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 between two PE's to which the two tasks connected by e_i are mapped. Therefore, F stands for the total communication traffic.

As an example, Fig. 7(a) and (b) shows a task graph and a system graph, respectively. When the task graph is mapped to the system graph as shown in Fig. 7(c), $F = 20$. A better mapping is shown in Fig. 7(d) with $F = 16$.

We may use the algorithms for the quadratic assignment problem to obtain an optimal mapping. The algorithm used in Hypertool is a heuristic algorithm, which was presented by Hanan and Kurtzberg to minimize the total communication

$F = 3 + 2 + 3 + 6*2 = 20$
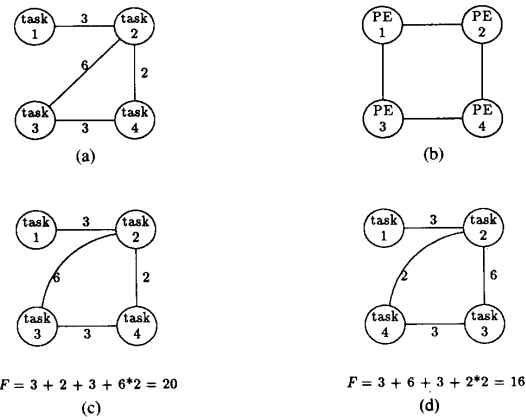(c)

$F = 3 + 6 + 3 + 2*2 = 16$
(d)

Fig. 7. Mapping: (a) task graph; (b) system graph; (c) a mapping; (d) the optimal mapping.

traffic [28]. 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 enough, even though it did not always guarantee an optimum solution [29].

VI. SYNCHRONIZATION

Synchronization of message-passing systems is carried out by communication primitives. The basic communication primitives are *send* and *receive*. Some message-passing systems supply synchronous communication primitives only, while some supply both synchronous and asynchronous primitives. Asynchronous primitives usually use communication coprocessors to handle network activities, which reduces the load on main processors.

The communication primitives are used to exchange messages between processors. They must be used properly to ensure the correct sequence of computation. In first-generation message-passing systems, communication primitives are inserted by programmers. It is possible to insert these primitives automatically, reducing programmer's load and eliminating insertion errors. Since our programming model partitions programs into procedures and message exchanges only take place before and after the procedures, the primitive insertion is relatively easy.

The communication primitive insertion may be described as follows. After scheduling and mapping, each node in a macro dataflow graph has been allocated to a PE. If an edge exits from this node to another node which 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. If a message is to be sent to many PE's, *broadcasting* is applied instead of sending the message to these PE's separately.

The insertion method described above does not ensure that the communication sequence is correct. For example, two possible cases are shown in Fig. 8(a) and Fig. 9(a). In Fig. 8(a),
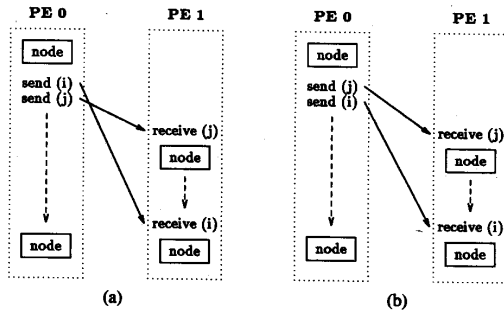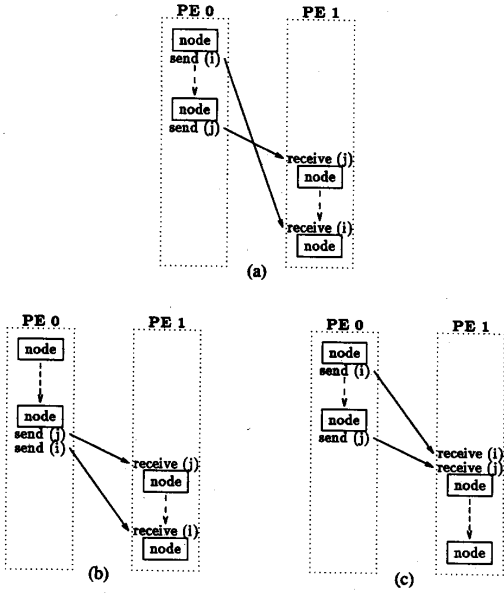
Fig. 8. Synchronization insertion (case 1).

Fig. 9. Synchronization insertion (case 2).

Fig. 10. The scheduling of Fig. 4 by MD algorithm.

the order of the *sends* is incorrect, and must be reordered as shown in Fig. 8(b). In Fig. 9(a), on the other hand, either the order of *sends* or the order of *receives* needs to be exchanged as shown in Fig. 9(b) or (c), respectively. We use a *send-first* strategy for this case. That is, we will reorder *receives* according to the order of *sends*, and obtain the solution shown in Fig. 9(c). The entire communication primitive insertion algorithm is described below.

Algorithm 3: *Communication Insertion:*

Assume after scheduling and mapping that each node n_i of the macro dataflow graph is assigned to PE $M(n_i)$, where M is a function mapping a node number to a PE number.

Step 1: For each edge e_k from node n_i to n_j for which $M(n_i) \neq M(n_j)$, insert a *send* primitive after node n_i in PE $M(n_i)$, denoted by $S(e_k, n_i, M(n_j))$; and insert a *receive* primitive before node n_j in PE $M(n_j)$, denoted by $R(e_k, n_j, M(n_i))$. Once a message has been sent to a PE, eliminate other *sends* and *receives* which transfer the same message to the same PE.

Now, for each PE, we have a sequence, $X(e_{m_1}, n_{m_1}, p_{m_1})$, $X(e_{m_2}, n_{m_2}, p_{m_2}), \cdots$, where X could be either S or R.

Step 2: For each pair of PE's, say p_1 and p_2, extract all $S(e_{m_i}, n_{m_i}, p_2)$ from PE p_1 to form a subsequence S_{p_1}, and extract all $R(e_{m_j}, n_{m_j}, p_1)$ from PE p_2 to form a subsequence R_{p_2}.

Step 2.1: Within each segment of the subsequence S_{p_1} with the same node number, exchange the order of *sends* according to the order of *receives* as defined by the subsequence R_{p_2}.

Step 2.2: If the two resultant subsequences are still not matched with each other, R_{p_2} is reordered according to the order of S_{p_1}. □

Reordering of *sends* and *receives* may not be necessary for a system supporting typed messages. However, even for this kind of system, message transmission reordering may reduce the message waiting time and the demand of system communication buffers.

VII. Examples and Comparison

We use the example of Fig. 4 to illustrate scheduling and synchronization. The MD algorithm is used to schedule the macro dataflow graph. Node n_1 is scheduled first, then node n_3. After n_3 is scheduled, the weight of the edge connecting n_1 and n_3 becomes 0, inducing modification of mobilities. Node n_7 is then scheduled, and so on. The macro dataflow graph is scheduled to two PE's, as shown in Fig. 10. Note that sending

```
/* For PE O */
    /* load matrix and vector from host */
        receive(HOST, vector[0]);
        receive(HOST, matrix[0][0]);
        receive(HOST, matrix[0][1]);
        receive(HOST, matrix[0][2]);

        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);

    /* unload matrix and vector to host */
        send(HOST, vector[4]);
        send(HOST, matrix[3][2]);
        send(HOST, matrix[4][3]);
        send(HOST, matrix[4][4]);

/* For PE 1 */
    /* load matrix from host */
        receive(HOST, matrix[0][3]);
        receive(HOST, matrix[0][4]);

        receive(PEO, 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(PEO, matrix[1][1]);
        receive(PEO, vector[2]);
        call UpdateMtx(matrix[1][3], matrix[2][3], vector[2], 1);
        send(PEO, matrix[2][3]);
        call UpdateMtx(matrix[1][4], matrix[2][4], vector[2], 1);
        send(PEO, 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);

    /* unload matrix to host */
        send(HOST, matrix[2][1]);
        send(HOST, matrix[1][0]);
```

Fig. 11.   The target machine code for each PE.

"vector" in the first iteration to the node n_5 in PE 1 means sending it to nodes n_6 and n_2 unnecessary. When communication primitives are inserted, the order of the second and the third *receives* in PE 1 is exchanged so that PE 1 receives the message from n_3 before that from n_7. Fig. 11 shows the generated code for two PE's according to Fig. 10. Note that only the main program for each PE is shown. The data structure is the same as in Fig. 3. The initial matrix is loaded to PE's such that each PE obtains a part of the matrix which is demanded for its computation. Consequently, the memory space can be compacted so that only what is demanded is allocated in each PE. In this example, the first, second, and the third columns of *matrix* are loaded to PE 0, and the fourth and the fifth columns to PE 1. Similarly, the third, fourth, and the fifth columns of the resulting *matrix* are unloaded from PE 0, and the first and the second columns from PE 1. Furthermore, to reduce the number of message transfers and consequently, the time to initiate messages, several messages can be packed and sent together. For example, the first three columns of the initial *matrix* may be packed into one message and sent to PE 0.

Fig. 12 shows another example. The Gauss-Seidel algorithm is used to solve Laplace equations since it converges faster than the Jacobi algorithm. In this algorithm, the update procedure for step k may be presented as

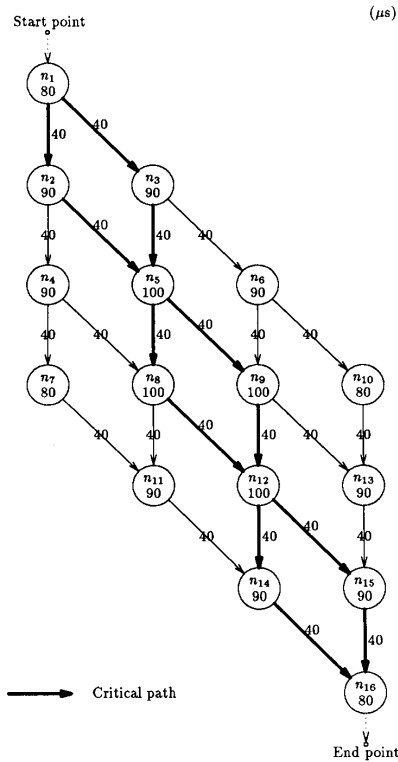$$A_{i,j}^{(k)} = \frac{1}{4}[A_{(i-1),j}^{(k)} + A_{i,(j-1)}^{(k)} + A_{(i+1),j}^{(k-1)} + A_{i,(j+1)}^{(k-1)}].$$

Fig. 12.   The macro dataflow graph of a parallel Laplace equation algorithm.

Fig. 13.   The scheduling of Fig. 12 by MCP algorithm.

Each value of $A_{i,j}$ at iteration k is obtained from two newly generated values for iteration k and two old values of iteration $k - 1$. This algorithm assumes a particular updating sequence. Since $A_{i,j}$ depends on the values of the same iteration, not all values of an iteration can be obtained simultaneously. A given matrix is partitioned by rows and columns. The macro dataflow graph in Fig. 12 may be scheduled to three PE's using the MD algorithm. However, if only two PE's are available, the MCP algorithm is applied. The resultant scheduling is shown in Fig. 13.

Our Hypertool is currently running on a Sun workstation under UNIX. It takes 21 s to schedule a program with 594 processes to 16 PE's. Several examples have been tested on Hypertool. By our experience, developing programs 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.

Next we compare the manually generated codes and Hypertool-generated codes. Both codes are generated from the same algorithms. The only difference is that automatic scheduling has applied to the Hypertool-generated codes but not to the manually generated codes. The scheduling method for the manually generated codes can be briefly described as follows: the data domain is partitioned equally in such a style that reduces dependencies among these partitions. Each PE is assigned a partition, and an SPMD program is coded for PE's. Although a programmer might use sophisticated scheduling, the working load is too heavy for human to produce good,
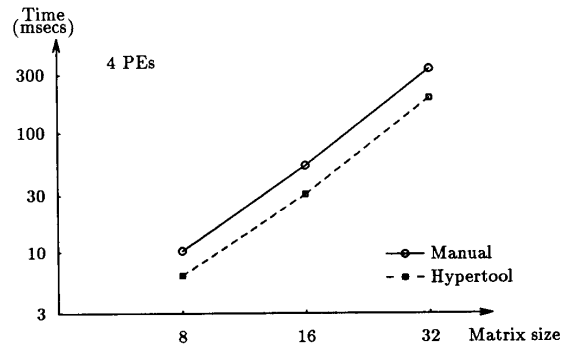

Fig. 14.   Performance comparison for Gaussian elimination.

error-free results. This sophisticated scheduling should be performed by an automatic tool.

By using Hypertool, some problems, such as the Gaussian elimination, Laplace equations, and Dynamic programming, resulted in up to 300% improvement in speed (see Figs. 14–16). These problems have less regular structures so that manual scheduling usually leads to an unbalanced load distribution among PE's because the computation amount in each data partition is different. On the other hand, automatic scheduling moves some nodes from overloaded PE's to underloaded PE's and achieves a better load balance. 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 Figs. 17 and 18. However,
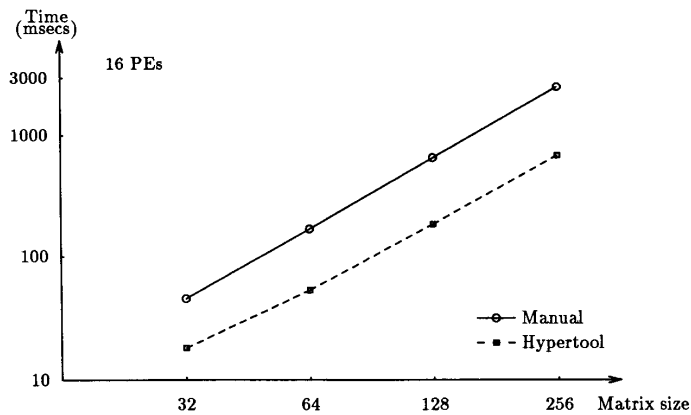
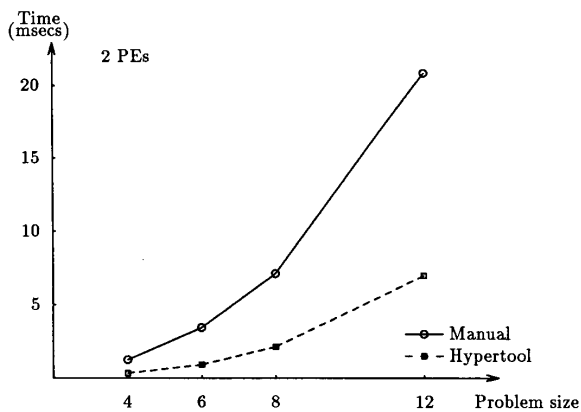Fig. 15. Performance comparison for Laplace equations.

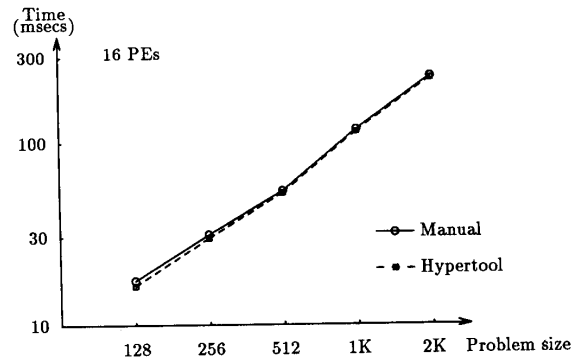Fig. 16. Performance comparison for Dynamic programming.

Fig. 17. Performance comparison for Matrix multiplication.
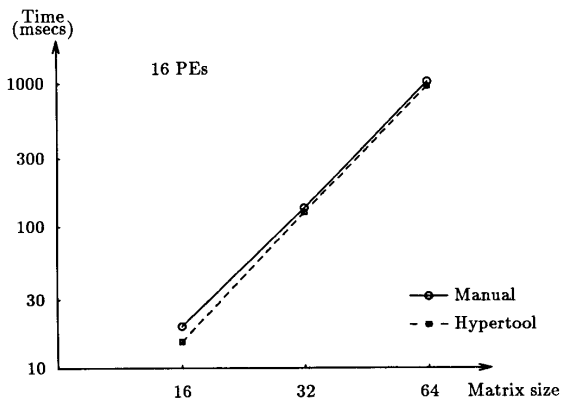

Fig. 18. Performance comparison for Bitonic sort.

TABLE II
PERFORMANCE COMPARISON FOR MCP AND MD ALGORITHMS

GAUSSIAN ELIMINATION				
Matrix size	Number of PEs	Execution time (msecs)		
		MCP	MD	Manual
4	4	1.8	1.8	2.4
8	5	6.3	6.3	9.5
16	7	24.2	24.0	40.0
32	12	103.6	102.6	165.6

even for these kinds of problems Hypertool still shows better performance when the size of the matrix cannot be evenly divided by the number of PE's.

Table II shows the comparison of Gaussian elimination for MCP and MD scheduling algorithms. These two algorithms deliver almost the same performance; however, the MD algorithm is useful for reducing the number of PE's required to execute a program. The execution times of manually generated codes are also listed for comparison.

Fig. 19 compares the execution times of random scheduling and MCP scheduling for Laplace equations. About 30% improvement in speed is obtained by the MCP scheduling algorithm. Surprisingly, in many cases, even the random scheduling algorithm generates better performance than manual scheduling.

The grain size of each procedure also affects the quality of generated parallel codes. Too large granularity reduces parallelism and too small granularity increases overhead. Only moderate grain size gives the best performance. Fig. 20 shows the performance for different grain sizes of procedures.

Since the execution times of nodes and the message transmission times are obtained by estimation, the performance affected by the difference between the estimated value and the
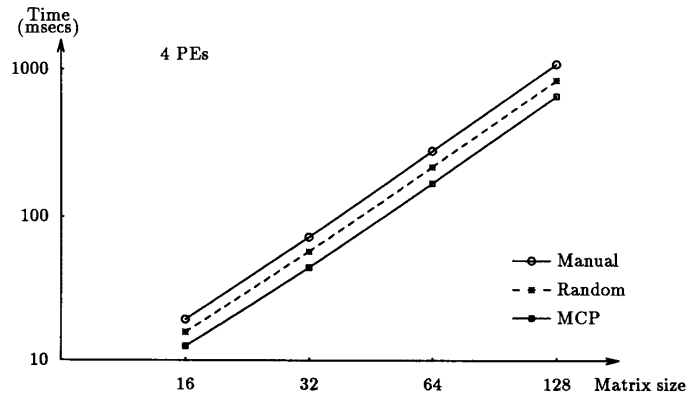
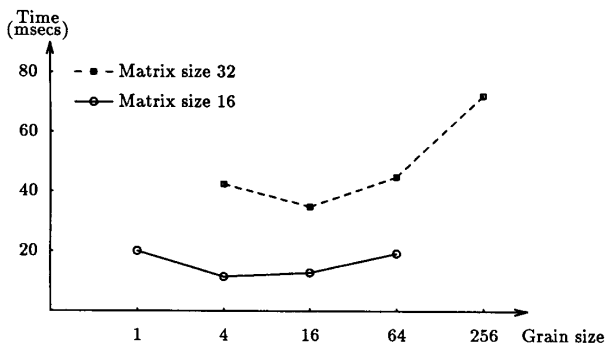Fig. 19. Comparison for random and MCP scheduling (Laplace equations).

Fig. 20. Comparison for different grain sizes (Laplace equations).

TABLE III
THE EFFECT OF ESTIMATION ON PERFORMANCE
(LAPLACE EQUATIONS)

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

TABLE IV
THE EFFECT OF ESTIMATION ON PERFORMANCE (BITONIC SORT)

Problem size	Execution time (msecs)		Difference
	Real	Estimated	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%

real value has been studied. Tables III and IV show the results for Laplace equations and Bitonic sort, respectively. For the Laplace equations, the node weight is estimated with matrix size of 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.

VIII. CONCLUSION

As both the number of PE's 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 have 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 show that the Hypertool approach is better than the manual methodology in many respects. First, it increases the programming productivity. Programmers only define partitions without indicating the task allocation or communication primitives. Second, since communication primi-

tive insertion is performed by Hypertool, synchronization errors 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 Hypertool generates target machine codes automatically, the programs developed on the tool are portable. The program 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.

The current version of Hypertool uses static scheduling and is applied to static problems. That is, all processes must be created before starting execution. Also, an estimate for the computation amount of each process must be known at compile time. One drawback of static scheduling is that a program must be rescheduled before it can run on another machine. A dynamic version of Hypertool is under investigation. Our Hypertool is able to schedule a macro dataflow graph with several thousands processes caused mainly by memory space limitation. A large graph may be partitioned into several strongly connected subgraphs for scheduling.

REFERENCES

[1] J. Rattner, "Concurrent processing: A new direction in scientific computing," in *Proc. National Comput. Conf.*, 1985, pp. 157–166.

[2] *Ametek System 14 User's Guide: C Edition*, Ametek Computer Research Division, Arcadia, CA, 1986.

[3] J. P. Hayes, T. Mudge, Q. F. Stout, S. Colley *et al.*, "A microprocessor-based hypercube supercomputer," *IEEE Micro*, vol. 6, pp. 6–17, Oct. 1986.

[4] H. L. Gustafson, S. Hawkinson, and K. Scott, "The architecture of a homogeneous vector supercomputer," in *Proc. Int. Conf. Parallel Processing*, Aug. 1986, pp. 649–652.

[5] L. W. Tucker and G. G. Robertson, "Architecture and applications of the Connection Machine," *IEEE Comput. Mag.*, pp. 26–38, Aug. 1988.

[6] A. H. Karp, "Programming for parallelism," *IEEE Comput. Mag.*, vol. 20, pp. 43–57, May 1987.

[7] C. L. Seitz, "The Cosmic Cube," *Commun. ACM*, vol. 28, pp. 22–23, Jan. 1985.

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

[9] 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.

[10] M. Y. Wu and D. D. Gajski, "Computer-aided programming for message-passing systems: Problems and a solution," *Proc. IEEE*, vol. 77, no. 12, pp. 1983–1991, Dec. 1989.

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

[12] J. K. Peir, D. D. Gajski, and M. Y. Wu, "Programming environments for multiprocessors," in *Supercomputing*. New York: North Holland, 1987, pp. 73–93.

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

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

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

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

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

[18] M. Y. Wu and D. D. Gajski, "A programming aid for message-passing systems," in *Parallel Processing for Scientific Computing*, G. Rodrigue, Ed. Philadelphia, PA: SIAM, pp. 328–332.

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

[20] T. L. Adam, K. Chandy, and J. Dickson, "A comparison of list scheduling for parallel processing systems," *Commun. ACM*, vol. 17, pp. 685–690, Dec. 1974.

[21] W. H. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Trans. Comput.*, vol. C-24, pp. 1235–1238, Dec. 1975.

[22] 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.

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

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

[25] D. Landskov, S. Davidson, B. Shriver, and P. Mallett, "Local microcode compaction techniques," *Comput. Surveys*, vol. 12, pp. 261–294, Sept. 1980.

[26] R. Sethi, "Algorithms for minimal-length schedules," in *Computer and Job-Shop Scheduling Theory*. New York: Wiley, 1976, ch. 2.

[27] R. P. Bianchini and J. P. Shen, "Interprocessor traffic scheduling algorithm for multiple-processor networks," *IEEE Trans. Comput.*, vol. C-36, pp. 396–409, Apr. 1987.

[28] M. Hanan and J. Kurtzberg, "A review of the placement and quadratic assignment problems," *SIAM Rev.*, vol. 14, pp. 324–342, Apr. 1972.

[29] S. Y. Lee and J. K. Aggarwal, "A mapping strategy for parallel processing," *IEEE Trans. Comput.*, vol. C-36, pp. 433–442, Apr. 1987.

Min-You Wu (S'84–M'87) 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, IL, 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 (M'77–SM'83) 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 in 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 compilers and expert systems for VLSI, and the science of design.