FASTEST: A Practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors

Yu-Kwong Kwok, *Member*, *IEEE*, and Ishfaq Ahmad, *Member*, *IEEE Computer Society*

Abstract—In the area of parallelizing compilers, considerable research has been carried out on data dependency analysis, parallelism extraction, as well as program and data partitioning. However, designing a practical, low complexity scheduling algorithm without sacrificing performance remains a challenging problem. A variety of heuristics have been proposed to generate efficient solutions but they take prohibitively long execution times for moderate size or large problems. In this paper, we propose an algorithm called FASTEST (*Fast Assignment and Scheduling of Tasks using an Efficient Search Technique*) that has $O(e)$ time complexity, where e is the number of edges in the task graph. The algorithm first generates an initial solution in a short time and then refines it by using a simple but robust random neighborhood search. We have also parallelized the search to further lower the time complexity. We are using the algorithm in a prototype automatic parallelization and scheduling tool which compiles sequential code and generates parallel code optimized with judicious scheduling. The proposed algorithm is evaluated with several application programs and outperforms a number of previous algorithms by generating parallelized code with shorter execution times, while taking dramatically shorter scheduling times. The FASTEST algorithm generates optimal solutions for a majority of the test cases and close-to-optimal solutions for the rest.

Index Terms—Automatic parallelization, compile-time scheduling, task graphs, multiprocessors, parallel processing, parallel programming tool, parallel algorithm, random neighborhood search.

1 Introduction

Scheduling the tasks of a parallel program to the processors is crucial for optimizing performance. Scheduling can be performed at compile-time if the characteristics of the parallel program, such as the execution times of the tasks, amount of communication data, and task dependencies, are known before program execution. For example, assuming that the loop bounds are known at compile-time, the parallel loop nest shown in Fig. 1a can be partitioned into six tasks connected as a *directed acyclic graph* (DAG), as shown in Fig. 1b. The DAG can be constructed by applying various static data dependency analysis [4], [8], [18] and program partitioning [3], [19], [22] techniques. Furthermore, the nodes and edges of the DAG are associated with weights, which are generated by using techniques such as execution profiling and analytical benchmarking [6], [9], [12] for representing amounts of execution time and communication time, respectively. The tasks can then be scheduled to the processors for execution by using a suitable scheduling algorithm. The objective of scheduling is to minimize the overall completion time or *schedule length* of the parallel program. As the scheduling algorithm is invoked *off-line*, this kind of multiprocessor scheduling problem is commonly referred to as *static scheduling* [5], [11], [14], [16], [17], [20], [23].

Static scheduling, except for a few highly simplified cases, is an NP-complete problem [5], [7]. Thus, heuristic approaches are generally sought to tackle the problem. Traditional static scheduling algorithms attempt to minimize the schedule length through iterative local minimization of the start times of individual tasks [5]. These algorithms differ primarily in their methods of selecting a task for start time minimization. For instance, the Modified Critical Path (MCP) algorithm [25] constructs a list of tasks before the scheduling process starts, while the Dynamic Level Scheduling (DLS) algorithm [24] dynamically selects tasks during the scheduling process. However, like most greedy algorithms, these scheduling approaches cannot avoid making a local decision which may lead to an unnecessarily long final schedule. Although static scheduling is done at compile-time and therefore can afford some extra time in generating a better solution, back-tracking techniques are not employed to avoid high complexity. Indeed, the time complexity of a scheduling algorithm is an important issue from a practical perspective because a slow scheduling algorithm is not desirable to be incorporated in a parallelizing compiler. In this regard, Yang and Gerasoulis [26] proposed some novel techniques for reducing the time complexity of scheduling algorithms. Our objective is to design a new static scheduling algorithm that has a lower time complexity and yet produces better solutions.

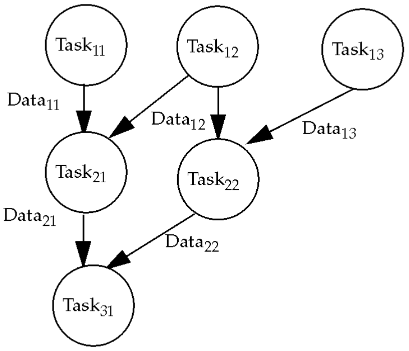
To meet the conflicting goals of good performance and low complexity, we employ an effective optimization technique

- *Y.-K. Kwok is with the Department of Electrical and Electronic Engineering, The University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: ykwok@eee.hku.hk.*
- *I. Ahmad is with the Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. E-mail: iahmad@cs.ust.hk.*

Manuscript received 15 Feb. 1998; revised 15 July 1998.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 108176.

```
Data_{0, j} := 0; for all j
for i = 1 to 3 do in parallel
    for j = 1 to 3-(i-1) do in parallel
        Task_{ij} (INPUT: Data_{i-1, j} , Data_{i-1, j+1};
        OUTPUT: Data_{ij})
```

(a)



(b)

Fig. 1. (a) A parallel program fragment; (b) a DAG representing the program fragment.

known as the random neighborhood search [21]. Simply put, in a neighborhood search-based algorithm, an initial solution with moderately good quality is quickly generated. Then, according to some predefined neighborhood in the search space, the algorithm probabilistically selects and tests whether a nearby solution in the search space is better or not. If the new solution is better, the algorithm adopts it and starts searching in a new neighborhood; otherwise, the algorithm tests another solution point. Usually, the algorithm stops after a specified number of search steps has elapsed or the solution does not improve after a fixed number of steps. The success of a neighborhood search technique relies heavily on the construction of the solution neighborhood. A judiciously constructed neighborhood, like the one defined in Kernighan and Lin's graph partitioning algorithm [13] (for undirected graphs) can potentially lead the search to attain a globally optimal solution.

The proposed scheduling algorithm, which is called FASTEST (Fast Assignment and Scheduling of Tasks using an Efficient Search Technique), is a linear time algorithm with $O(e)$ worst-case time complexity, where e is the number of edges in the task graph. The search is parallelized so as to further reduce the algorithm's complexity. In addition to using randomly generated task graphs, the FASTEST algorithm is evaluated by applying it in a prototype compile-time parallelization and scheduling tool called CASCH (Computer-Aided SCHeduling) [1] to several practical applications on an Intel Paragon. The algorithm outperforms numerous previous algorithms while the scheduling times required are dramatically shorter.

The paper is organized as follows: Section 2 describes the proposed FASTEST algorithm and its design principles. Section 3 contains the performance results. Section 4 provides some concluding remarks.

2 The Proposed Algorithm

In this section, we present the proposed algorithm and its design principles. To facilitate understanding the neighborhood search technique, we first restrict the discussion to the sequential version of the FASTEST algorithm, which is referred to as simply the FAST algorithm. We then describe the parallelization technique leading to the FASTEST algorithm.

2.1 A Solution Neighborhood Formulation

Neighborhood search is an old but effective optimization technique [13], [21]. The principle of neighborhood search is to refine an initial solution by searching through the neighborhood of the initial solution point in the solution space. To apply the neighborhood search technique to the DAG scheduling problem, a crucial step is to define a neighborhood of the initial solution point (i.e., the initial schedule). We can arrive at such a neighborhood definition by using the observation discussed below.

A neighboring point of a schedule in the solution space can be defined as another schedule which is obtained by transferring a node from a processor to another processor. In the DAG scheduling problem, one method of improving the schedule length is to transfer a *blocking-node* from one processor to another. The notion of blocking is simple: A node is called a *blocking-node* if removing it from its original processor can make the succeeding nodes start earlier. In particular, we are interested in transferring the nodes that block the *critical path*[1] nodes (CPNs) because the CPNs represent the more important tasks. However, a high complexity will result if we attempt to exhaustively locate the actual blocking-nodes on all the processors. Thus, in our approach, we only generate a list of *potential* blocking-nodes, which are the nodes that may block the CPNs. Again, to maintain a low complexity, the blocking-nodes list is static and is constructed before the search process starts. A natural choice of blocking-nodes list is the set including all the IBNs and OBNs[2] (with respect to an initial CP) [16] because these nodes have the potential to block the CPNs in the processors. In the schedule refinement phase, the blocking-nodes list defines the neighborhood that the random search process will explore. Assuming that the DAG, which has v nodes and e edges, is to be scheduled to p processors, the size of such a neighborhood is $O(vp)$ because there are $O(v)$ blocking-nodes.

2.2 Scheduling Serially

To generate an initial schedule, our technique is based on the traditional list scheduling approach in which we construct a

[1] A critical path (CP) is a path with the maximum sum of node and edge weights or, simply, the maximum length.

[2] An IBN (In-Branch Node) is an ancestor of a CPN but is not a CPN in itself. An OBN (Out-Branch Node) is a node which is neither a CPN nor an IBN.

list and schedule the nodes on the list one by one to the processors. The list is constructed by ordering the nodes according to the node priorities. The list is static so that the order of nodes on the list will not change during the scheduling process. The reason is that as the objective of our algorithm is to produce a good schedule in $O(e)$ time[3], we do not recompute the node priorities after each scheduling step while generating the initial schedule. Certainly, if the schedule length of the initial schedule is optimized, the subsequent random search process can start at a better solution point and thereby generate a better final schedule.

In the proposed algorithm, we use the CPN-Dominant list, which gives CPNs higher priorities, as the scheduling list. Given a DAG of v nodes $\{n_1, n_2, \ldots, n_v\}$ and e directed edges, each of which is denoted by (n_i, n_j), the CPN-Dominant List can be constructed using the following procedure in $O(e)$ time since each edge is visited only once.

Initial Task Ordering Procedure:

```
1) The CPN-Dominant list is initialized to be an empty list.
   Make the entry CPN (the one which does not have
   any predecessor) be the first node in the list.
   Set Position to 2.
   Let $n_x$ be the next CPN.
repeat
2) if $n_x$ has all its parent nodes in the list then
3)     Put $n_x$ at Position in the list and increment Position.
4) else
5)     Let $n_y$ be the parent node of $n_x$ which is not in the
       sequence and has the largest b-level.[4] Ties are broken
       by choosing the parent with a smaller t-level.
       repeat
5)         If $n_y$ has all its parent nodes in the sequence then
6)             put $n_y$ at Position in the sequence and increment
               Position.
7)         else
8)             Recursively include all the ancestor nodes of
               $n_y$ in the sequence so that the nodes with a
               larger value of b-level are considered first.
9)         endif
       until all the parent nodes of $n_x$ are in the list.
6)     Put $n_x$ at Position in the list.
7) endif
8) Make $n_x$ to be the next CPN.
until all the CPNs are in the list.
9) Append all the OBNs to the sequence in a decreasing
   order of b-level.
```

We make use of the CPN-Dominant list in a procedure called *InitialSchedule*, which has two steps: 1) constructing the CPN-Dominant list, 2) scheduling the nodes on the list one after another to the processors. In the *InitialSchedule* procedure, to avoid incurring high complexity, we do not search for the earliest slot on a processor but simply schedule a node to the ready time of a processor. Initially, the ready time of all available processors is zero. After a node is scheduled to a processor, the ready time of that processor is updated to the finish time of the last node. By doing so, a node is scheduled to a processor that allows the earliest start time, which is determined by checking the processor's ready time with the node's data arrival time (DAT). The DAT of a node can be computed by taking the maximum value among the message arrival times across the parent nodes. If the parent is scheduled to the same processor as the node, the message arrival time is simply the parent's finish time because the local communication cost is assumed to be zero; otherwise, it is equal to the parent's finish time (now on a remote processor) plus the communication cost of the edge. Not all processors need to be checked in this process. Instead, we can examine the processors accommodating the parent nodes together with an empty processor (if any). The time complexity of *InitialSchedule* is derived as follows: The first step takes $O(e)$ time. The cumulative time complexity of the second step is also $O(e)$ because each edge is visited once. Thus, the overall time complexity of *InitialSchedule* is $O(e)$.

To analyze the performance of *InitialSchedule*, we focus on a basic graph structure called the fork-set, as shown in Fig. 2a. This basic structure is the building block of more general graph structures and, thus, an algorithm's performance on such a structure can give some insight about its general performance. In the following, we use $w(n_i)$ to denote the weight of a node n_i and $c(n_x, n_y)$ to denote the weight of an edge (n_x, n_y).

We first analyze some properties of the fork set. Throughout the analysis, we assume that the target processor network consists of p fully connected processing elements (PEs). Without loss of generality, assume that for the fork structure, we have:

$$c(n_x, n_1) + w(n_1) \ge c(n_x, n_2) + w(n_2) \ge \ldots \ge c(n_x, n_v) + w(n_v).$$

The optimal schedule length is then equal to:

$$max\left\{ w(n_x) + \sum_{i=1}^{j} w(n_i),\ w(n_x) + c(n_x, n_{j+1}) + w(n_{j+1}) \right\},$$

where j is given by the following conditions:

$$\sum_{i=1}^{j} w(n_i) \le c(n_x, n_j) + w(n_j)$$

and

$$\sum_{i=1}^{j+1} w(n_i) > c(n_x, n_{j+1}) + w(n_{j+1}).$$

The optimal schedule for the fork set is shown in Fig. 2b. From the above expressions, it is clear that an algorithm has to be able to recognize the longest path in the graph in order to generate optimal schedules. Thus, algorithms which consider only *b-level* or only *t-level* may not guarantee optimal solutions. To make proper scheduling decisions, an algorithm has to properly examine both *b-level* and *t-level*.

[3] Throughout the paper, we assume that the DAG is connected and, hence, we have $e = O(v)$.

[4] The *b-level* of a node is the length (sum of the computation and communication costs) of the longest path from the node to an exit node. The *t-level* of a node is the length of the longest path from an entry node to this node (excluding the cost of this node) [16], [26].

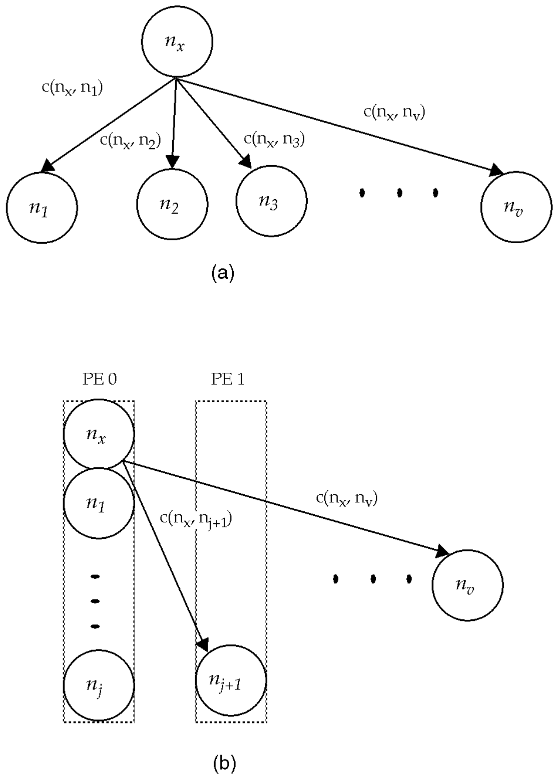


Fig. 2. (a) A fork set; and (b) its optimal schedule.

From the above analysis, it is easy to see that the initial schedule generated based on the CPN-Dominant list will be optimal if:

$$c(n_x, n_1) = c(n_x, n_2) = \ldots = c(n_x, n_v),$$

which in turn implies that:

$$w(n_1) \geq w(n_2) \geq \ldots \geq w(n_v).$$

Thus, the nodes will be arranged according to their ordinal order in the CPN-Dominant list because the nodes are considered in descending order of their *b-levels* which are just their node weights in the case of a fork set. The initial schedule will then be the optimal schedule as shown in Fig. 2b since the FASTEST algorithm minimizes the start time of every node in the CPN-Dominant list.

Based on the above analysis, we can expect that the FASTEST algorithm will be able to generate good initial schedules for graphs with relatively uniform edge weights. Such scenarios are found in many practical numerical applications in which the tasks communicate by forwarding partial results in fixed sized arrays.

Given the procedure *InitialSchedule,* we present the sequential version of our neighborhood search algorithm—the FAST algorithm. In order to avoid the search being trapped in a local optimal solution, we incorporate a probabilistic jump procedure in the algorithm. In the FAST algorithm outlined below, we use SL to denote the schedule length.

The FAST Algorithm:

```
NewSchedule = InitialSchedule
Construct the blocking-nodes list which contains all the IBNs and OBNs;
BestSL = infinity; searchcount = 0;
repeat
    searchstep = 0; counter = 0;
    do { /* neighborhood search */
        Pick a node $n_i$ randomly from the blocking-nodes list;
        Pick a processor $P$ randomly;
        Transfer $n_i$ to $P$;
        If schedule length does not improve, transfer $n_i$ back to its original processor and increment counter; otherwise, set counter to 0;
    } while (searchstep++ < MAXSTEP and counter < MARGIN);
    if BestSL > SL(NewSchedule) then
        BestSchedule = NewSchedule
        BestSL = SL(NewSchedule)
    endif
    NewSchedule = Randomly pick a node from the CP and transfer it to another processor; /* probabilistic jump */
until (searchcount++ > MAXCOUNT);
```

The total number of search-steps is *MAXSTEP* × *MAXCOUNT*. While the number of search steps in each iteration is bounded by MAXSTEP, the algorithm will also terminate searching and proceed to the step of probabilistic jump if the solution does not improve within a specified number of steps, denoted as MARGIN. This is done in order to further enhance the expected efficiency of the algorithm. The reason for making MAXSTEP, MARGIN, and MAXCOUNT as constants is two-fold. First, the prime objective in the design of the algorithm is to keep the time complexity low even when the size of the input graph is large. Second, the major strength of the FAST algorithm lies in its ability to generate a good initial solution by using the CPN-Dominant List. As such, the likelihood of improving the initial solution dramatically by using a large number of search steps is not high. Thus, we fix MARGIN to be 2, MAXSTEP to be 8, and MAXCOUNT to be 64 based on our experimental results which will be described in detail in Section 3.1.

The time complexity of the sequential FAST algorithm is derived as follows: As discussed earlier, the procedure *InitialSchedule* takes $O(e)$ time. The blocking-nodes list can be constructed in $O(v)$ time as the IBNs and OBNs are already identified in the procedure *InitialSchedule*. In the main loop, the node transferring step takes $O(e)$ time since we have to revisit all the edges once in the worst case after transferring the node to a processor. Thus, the overall time complexity of the sequential algorithm is $O(e)$.

To illustrate the scheduling mechanism of the FAST algorithm, consider the small example task graph shown in Fig. 3a. The CPN-Dominant List of the DAG is

$$\{n_1, n_3, n_2, n_7, n_6, n_5, n_4, n_8, n_9\}.$$

Note that n_8 is considered after n_6 because the latter has a smaller value of *t-level*. Using the CPN-Dominant List, the initial schedule produced by *InitialSchedule* is shown in Fig. 3b. The blocking-nodes list of the DAG is $\{n_2, n_3, n_4, n_5, n_6, n_8\}$.

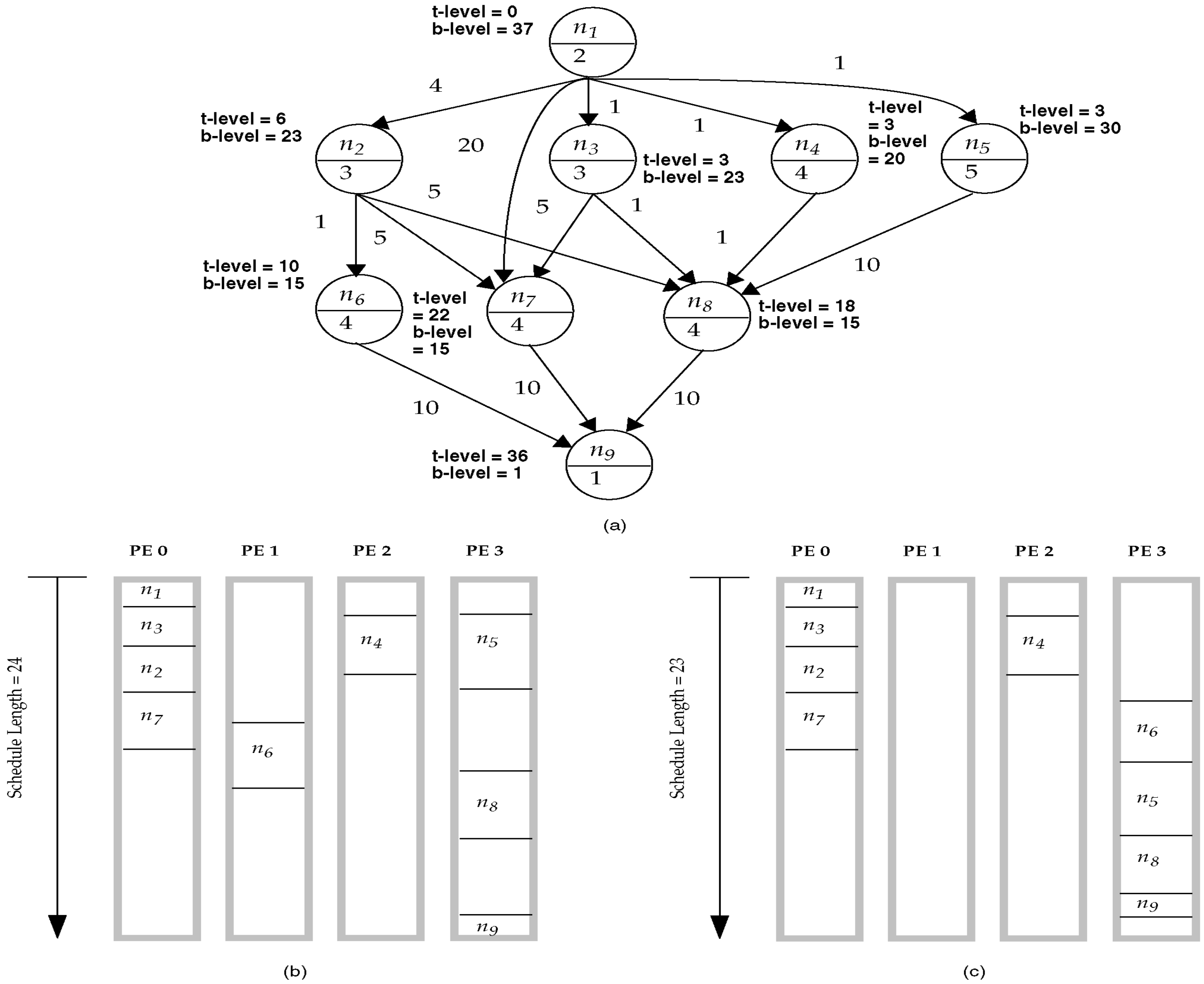


Fig. 3. (a) A task graph; (b) The schedule generated by *InitialSchedule*; (c) The schedule generated by the random neighborhood search.

We can see that the node n_6 blocks the CPN n_9. In the random search process, it is highly probable that n_6 is selected for transferring. Suppose it is transferred from PE 1 to PE 3. The resulting schedule is shown in Fig. 3c, where the final schedule length is shortened even though the start times of n_5 and n_8 are increased.

2.3 Parallel Probabilistic Search

The parallelization of the neighborhood search is based on partitioning of the blocking-nodes set into q subsets, where q is the number of available *physical processing elements* (PPEs), such as the processors in an Intel Paragon, on which the FASTEST algorithm is executed. Each PPE then performs a neighborhood search using its own blocking-nodes subset. The PPEs communicate periodically to exchange the best solution found thus far and start new search steps based on the best solution. Based on our experimental results (see Section 3.4), the period of communication for the PPEs is set to be T number of search-steps, which follows an exponentially decreasing sequence: initially $\lceil \frac{\tau}{2} \rceil$, then $\lceil \frac{\tau}{4} \rceil$, $\lceil \frac{\tau}{8} \rceil$, and so on, where $\tau = \lceil \frac{MAXCOUNT}{q} \rceil$. The rationale is that at early stages of the search, exploration is more important than exploitation. The PPEs should, therefore, work independently for a longer period of time. However, at final stages of the search, exploitation is more important for refining the solutions and, thus, the PPEs should communicate more frequently. The FASTEST algorithm is outlined below.

The FASTEST Algorithm:

```
if myPPE == master then
    Determine the initial schedule;
    Construct the blocking-nodes set;
    Partition the blocking-nodes set into $q$ subsets
    which are ordered topologically;
endif
Every PPE receives a blocking-nodes subset and the initial schedule;
repeat
    $i = 2$
    repeat /* search */
        Run FAST to search for a better schedule;
    until searchcount > $\lceil \frac{MAXCOUNT}{i \times q} \rceil$;
    Exchange the best solution;
until total searchcount = $\lceil \frac{MAXCOUNT}{q} \rceil$
```

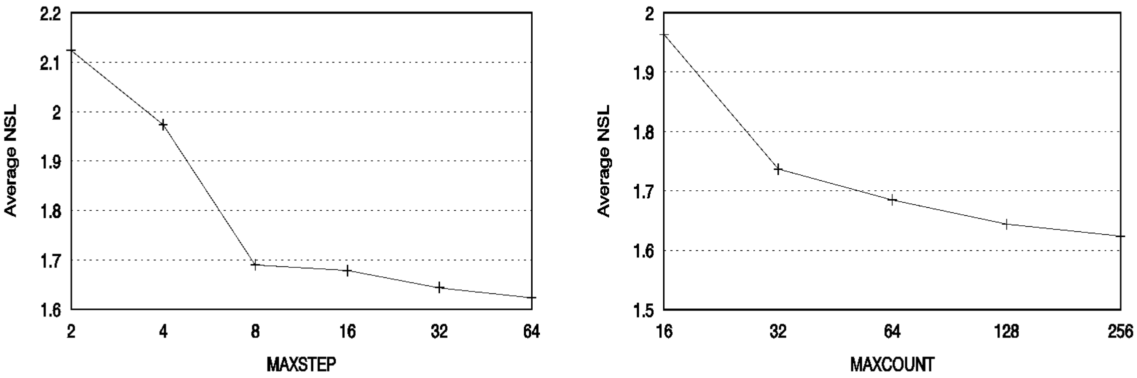


Fig. 4. Average normalized schedule lengths of the FAST algorithm for random task graphs with 1,000 nodes using various values of MAXSTEP and MAXCOUNT.

In the FASTEST algorithm, one PPE is designated as the master, which is responsible for preprocessing work including construction of an initial schedule, the blocking-nodes set, and the subsets.

Since the total number of search-steps is evenly distributed to the PPEs, the FASTEST algorithm should have linear speedup over the sequential FAST algorithm if communication takes negligible time. However, inter-PPE communication inevitably incurs an overhead and, thus, the ideal case of linear speedup is not achievable. But the solution quality of FASTEST can be better than that of the sequential FAST algorithm. This is because the PPEs explore different parts of the search space simultaneously through different neighborhoods induced by the partitions of the blocking-nodes set. The sequential FAST algorithm, on the other hand, has to handle a much larger neighborhood for the same problem size.

To perform a simple probabilistic analysis on the search mechanism of the FASTEST algorithm, suppose there is a CPN blocked by a certain blocking node in the list. That is, the CPN can be scheduled to start earlier on its processor if the blocking node is transferred to some other processors. The probability that the FASTEST algorithm will select this blocking node for transferal is $\frac{q}{\left(v-v_{cp}\right)}$ (the factor q is due to the partitioning of the blocking node list among q PPEs and v_{cp} is the number of CPNs) and the probability that a suitable processor (there may not be only one) is selected for accommodating the blocking node is at least $\frac{1}{p}$. Thus, the probability that the FASTEST algorithm will successfully improve the schedule is $\frac{q}{p\left(v-v_{cp}\right)}$ which increases as q. Thus, searching different neighborhoods in parallel by partitioning the blocking node list can potentially enhance the chance of improving the schedule.

3 Performance Results

The performance results of the proposed algorithm presented in this section serve a number of objectives. First, we examine the effects of the number of search steps. Second, we illustrate the practicality of the proposed algorithm in terms of its applicability in a real scheduling environment. To serve this purpose, we compared the FAST algorithm with the DSC (Dominant Sequence Clustering) [26], MD (Mobility Directed) [25], ETF (Earliest Task First) [10], and DLS (Dynamic Level Scheduling) [24] algorithms using a prototype software tool for program parallelization and scheduling. Third, we investigate the effects of different communication strategies used by the FASTEST algorithm. This is followed by a comparison of the schedules generated by the FASTEST algorithm with the optimal solutions. The results of using only the *InitialSchedule* procedure are also compared. Finally, to illustrate the scalability of the algorithms, we present the results of applying the algorithms to very large task graphs.

3.1 Number of Search Steps

We performed experiments to determine suitable values for the constants MAXSTEP and MAXCOUNT which govern the number of search steps and probabilistic jumps; for simplicity, we tested the FAST algorithm only. We generated 10 random task graphs with 1,000 nodes each (the method of generating these random graphs is elaborated in Section 3.5) and then tested the FAST algorithm with different values of MAXSTEP and MAXCOUNT. These experiments were performed without using MARGIN. In one set of experiments we varied MAXSTEP from 2 to 64 and fixed MAXCOUNT to be 256 which is large enough to isolate its effect. In the other set of experiments we varied MAXCOUNT from 16 to 256 and fixed MAXSTEP to be 64. The results of these experiments are shown in Fig. 4. Each point in the plots is the average normalized schedule lengths (NSLs) of 10 random graphs. The NSL of a graph is defined as the ratio of the schedule length to the sum of computation costs on the CP. As can be seen from the plots, the average NSLs did not improve considerably when we increased MAXSTEP beyond 8 and MAXCOUNT beyond 64. Thus, we fixed MAXSTEP to be 8 and MAXCOUNT to be 64 throughout the subsequent experiments. We set MAXCOUNT as 64 instead of 128 because the former value is a reasonable compromise between performance and time complexity.

Notice that the number of search steps for each neighborhood is bounded by MAXSTEP, for which a suitable value was found to be quite small. This indicates that the solution usually does not improve after several search

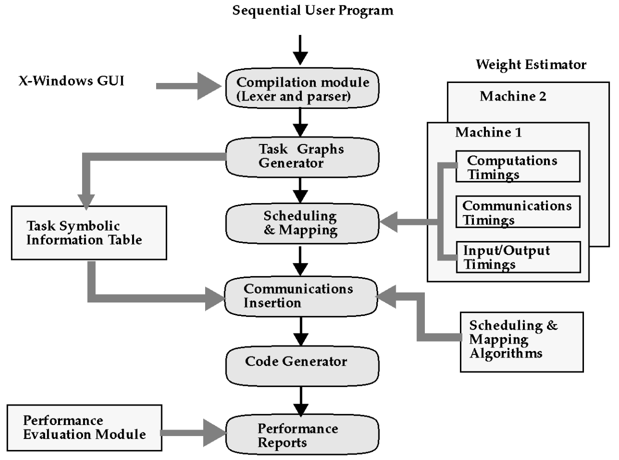


Fig. 5. The organization of the CASCH tool.

steps. Furthermore, from these experiments, we also observed that for most of the cases where the solution could not be improved within the first few search steps, additional searching steps within the current neighborhood yielded no further improvement. Thus, we use MARGIN, which is fixed to be 2, to detect whether a local optimum point is reached and to force the search to explore another neighborhood swiftly.

3.2 The CASCH Tool

We performed experiments using the CASCH (Computer-Aided SCHeduling) tool [1]. The system organization of the CASCH tool is shown in Fig. 5. It generates a task graph from a sequential program, uses a scheduling algorithm to perform scheduling, and then generates the parallel code in a scheduled form for an Intel Paragon. The timings for the nodes and edges on the DAG are assigned through a timing database which was obtained through profiling of the basic operations. CASCH also provides a graphical user interface to interactively run and test various algorithms, including the ones discussed in this paper. In addition to measuring the schedule length through a Gantt chart, we measure the running time of the scheduled code on the Paragon. Various scheduling algorithms, therefore, can be more accurately tested and compared through CASCH using real applications on an actual machine. It should be noted that after the task graph is generated from the input application on the host workstation, it can be uploaded to the Paragon for scheduling because the FASTEST algorithm can harness multiple processors to accomplish the scheduling process. The resulting symbolic schedule is then downloaded back to the host workstation for parallel code generation. The reader is referred to [1] for details about the CASCH tool.

3.3 Parallel Applications

In our first experiment, we tested the FAST algorithm with the DAGs generated from three practical applications: Gaussian elimination, Laplace equation solver, and Fast Fourier Transform (FFT) [1]. The Gaussian elimination and Laplace equation solver applications operate on matrices. Thus, the number of nodes in the DAGs generated from these applications is related to the matrix dimension N and is about $O\left(N^2\right)$. On the other hand, the FFT application accepts the number of points as input. We examined the performance in three aspects: application execution time, number of processors used, and the scheduling algorithm running time.

Using the CASCH tool on a SUN Sparc workstation, we compared the FAST algorithm with four sequential scheduling algorithms: the DSC (Dominant Sequence Clustering) [26], MD (Mobility Directed) [25], ETF (Earliest Task First) [10], and DLS (Dynamic Level Scheduling) [24] algorithms. The DSC algorithm works by dynamically tracking the critical path nodes for scheduling using a low complexity technique. The time complexity of the DSC algorithm is $O\left((e+v)\log v\right)$. The MD algorithm is also based on a dynamic strategy to schedule the critical path nodes as early as possible. The time complexity of the MD algorithms is $O\left(v^3\right)$. The ETF algorithm, at each scheduling step, schedules a task that can start at the earliest time among all ready tasks; the algorithm uses the static level to break ties. The time complexity of the ETF algorithm is $O\left(pv^2\right)$. The DLS algorithm schedules the task with the highest dynamic level, defined as the difference between a task's static level and the earliest start time. The time complexity of the DLS algorithm is also $O\left(pv^2\right)$. For a detailed discussion on the properties of these algorithms, the reader is referred to [16].

The results for Gaussian elimination are shown in Table 1. In Table 1a, we normalized the application execution times obtained through all the algorithms with respect to those obtained through the FAST algorithm. It was found that the programs scheduled by the FAST algorithm are up to 15 percent faster than the other algorithms. Note that the results of the DSC algorithm for matrix dimensions 16 and 32 were not available because the DSC used more than the available Paragon processors in scheduling the parallel program. This can be explicated by the fact that the DSC algorithm uses $O(v)$ processors. Concerning the number of processors used, the FAST, ETF, and DLS algorithms used about the same number of processors, as shown in Table 1b. The scheduling times of all the algorithms are shown in Table 1c, indicating that the DSC algorithm was the fastest algorithm with the proposed FAST algorithm very close to it. On the other hand, the ETF and DLS algorithms running times are relatively large, but are still shorter than those of the MD algorithm. This is because the MD algorithm has a higher time complexity than the other algorithms.

The results for the Laplace equation solver are shown in Table 2, from which we can see that the percentage improvements of the FAST algorithm over the other algorithms is up to 25 percent. Regarding the number of processors used, the FAST, MD, ETF, and DLS algorithms yield a similar performance, while the DSC algorithm again uses more processors than the other algorithms. For the scheduling times, the FAST algorithm is the fastest among all the algorithms. The MD algorithm is again slower than the other algorithms.

TABLE 1
NORMALIZED EXECUTION TIMES, NUMBER OF PROCESSORS USED, AND SCHEDULING ALGORITHM RUNNING TIMES FOR THE GAUSSIAN ELIMINATION FOR ALL THE SCHEDULING ALGORITHMS

	Matrix Dimension			
Algorithm	4	8	16	32
FAST	1.00	1.00	1.00	1.00
DSC	1.05	1.08	N.A.	N.A.
MD	1.00	1.03	1.08	1.10
ETF	1.00	1.07	1.10	1.15
DLS	1.00	1.08	1.10	1.14

(a)

	Matrix Dimension			
Algorithm	4	8	16	32
FAST	4	8	16	32
DSC	5	22	95	128
MD	2	3	4	7
ETF	3	7	16	32
DLS	3	7	16	32

(b)

	Matrix Dimension (Number of Tasks)			
Algorithm	4 (20)	8 (54)	16 (170)	32 (594)
FAST	0.06	0.09	0.15	0.52
DSC	0.04	0.06	0.09	0.21
MD	6.33	6.85	39.54	266.89
ETF	0.02	0.06	0.24	2.41
DLS	0.08	0.09	0.42	4.00

(c)

(a) Normalized execution times of Gaussian elimination on the Intel Paragon; (b) Number of processors used for the Gaussian elimination; (c) Scheduling times (sec) on a SPARC Station 2 for the Gaussian elimination.

TABLE 2
NORMALIZED EXECUTION TIMES, NUMBER OF PROCESSORS USED, AND SCHEDULING ALGORITHM RUNNING TIMES FOR THE LAPLACE EQUATION SOLVER FOR ALL THE SCHEDULING ALGORITHMS

	Matrix Dimension			
Algorithm	4	8	16	32
FAST	1.00	1.00	1.00	1.00
DSC	1.00	1.09	1.13	1.21
MD	1.00	1.12	1.15	1.25
ETF	1.00	1.11	1.14	1.24
DLS	1.00	1.10	1.13	1.23

(a)

	Matrix Dimension			
Algorithm	4	8	16	32
FAST	1	4	7	14
DSC	1	13	37	64
MD	1	5	8	13
ETF	1	5	8	16
DLS	1	5	8	15

(b)

	Matrix Dimension (Number of Tasks)			
Algorithm	4 (18)	8 (66)	16 (258)	32 (1026)
FAST	0.05	0.09	0.35	1.28
DSC	0.07	0.11	0.40	4.29
MD	6.23	7.64	111.46	768.90
ETF	0.04	0.05	0.28	3.06
DLS	0.06	0.11	0.55	5.33

(c)

(a) Normalized execution times of Laplace equation solver on the Intel Paragon; (b) Number of processors used for the Laplace equation solver; (c) Scheduling times (sec) on a SPARC Station 2 for the Laplace equation solver.

The results for the FFT are shown in Table 3. The FAST algorithm is again better than all the other four algorithms in terms of the application execution times and scheduling times. A plausible explanation for the consistent better performance of the FAST algorithm is that the numerical applications we used for the above experiments have highly regular graph structures [15]. For instance, most graphs have very regular edge weights, which is a major reason for superior performance of the initial schedule construction step in the FAST algorithm, as we have shown in Section 2.2.

3.4 Effect of Communication Strategy

In order to determine the most suitable periodic inter-PPE communication strategy for the FASTEST algorithm, we performed experiments using three different methods.

TABLE 3
NORMALIZED EXECUTION TIMES, NUMBER OF PROCESSORS USED, AND SCHEDULING ALGORITHM RUNNING TIMES FOR FFT FOR ALL THE SCHEDULING ALGORITHMS

	Number of Points			
Algorithm	16	64	128	512
FAST	1.00	1.00	1.00	1.00
DSC	1.03	1.08	1.10	1.15
MD	1.04	1.09	1.11	1.17
ETF	1.02	1.08	1.10	1.15
DLS	1.03	1.07	1.09	1.14

(a)

	Number of Points			
Algorithm	16	64	128	512
FAST	5	12	9	23
DSC	5	12	13	25
MD	5	10	6	21
ETF	3	10	11	11
DLS	7	10	11	11

(b)

	Number of Points (Number of Tasks)			
Algorithm	16 (14)	64 (34)	128 (82)	512 (194)
FAST	0.06	0.10	0.12	0.19
DSC	0.07	0.08	0.07	0.10
MD	6.38	9.09	9.87	75.17
ETF	0.05	0.08	0.09	0.16
DLS	0.05	0.18	0.20	0.67

(c)

(a) Normalized execution times of FFT on the Intel Paragon; (b) Number of processors used for the FFT; (c) Scheduling times (sec) on a SPARC Station 2 for FFT.

We used four PPEs to schedule 10 different 500-node random task graph for which optimal schedules are known (the method of generating such random graphs are described in detail in Section 3.5). In the first method, PPEs communicate with a constant period of one search step. That is, with MAXCOUNT equal to 64 and 4 PPEs, PPEs communicate 16 times throughout the search process. In the second method, PPEs communicate only twice: when half of the MAXCOUNT search steps have elapsed, and when the search process ends. In the third method, PPEs communicate periodically with exponentially decreasing periods. The results of the three methods are shown in Table 4. The average percentage deviation from optimal solutions is the best when frequent communication (method 1) is used, but the running time required is the longest. Infrequent communication is the worst in terms of solution quality even though it incurs less communication overhead. The exponentially decreasing schedule of communication is fast and its solution quality is slightly worse than the first method. Indeed, when we examined the variations in schedule lengths, we found that using the third method the PPEs quickly reached similar quality schedules during the last few search steps. But for first the method, there was some kind of premature convergence to some not so good schedules. Based on these results, we incorporated the exponentially decreasing communication schedule into the FASTEST algorithm.

TABLE 4
AVERAGE PERCENTAGE DEVIATIONS FROM OPTIMAL SCHEDULE LENGTHS AND THE RUNNING TIMES FOR SCHEDULING 10 500-NODE RANDOM TASK GRAPHS USING FOUR PPES

	Method1	Method 2	Method 3
Avg. % Dev.	29.98	50.34	30.71
Running Time (seconds)	7.2	2.5	3.4

3.5 Comparison Against Optimal Solutions

In this section, we present the performance results of the FASTEST algorithm. The objective is to investigate the solution quality of the algorithm by applying it to two different suites of random task graphs for which optimal solutions are known. The first suite of random task graphs consists of three sets of graphs with different values of *communication-to-computation ratio* (CCR), which is defined as the average edge weight divided by the average node weight in the DAG. In our experiments, three CCRs were used: 0.1, 1.0, and 10.0. Each set consists of graphs in which the number of nodes vary from 10 to 32 with increments of 2, thus totaling 12 graphs per set. The graphs within the same set have the same value of CCR. The graphs were randomly generated as follows: First, the computation cost of each node in the graph was randomly selected from a uniform distribution with mean equal to 40 (minimum = 2 and maximum = 78). Beginning with the first node, a random number indicating the number of children was chosen from a uniform distribution with mean equal to $\frac{v}{10}$, thus the connectivity of the graph increases with the size of the graph.

The communication cost of each edge was also randomly selected from a uniform distribution with mean equal to 40 times the specified value of CCR. Hereafter, this suite of graphs is designated type-1 random task graphs. To obtain optimal solutions for the task graphs, we applied a parallel A* algorithm [2] to the graphs. Since generating optimal solutions for arbitrarily structured task graphs takes exponential time, it is not feasible to obtain optimal solutions for larger graphs. However, to investigate the scalability of the FASTEST algorithm, it is desirable to test it with larger task graphs for which optimal solutions are known. To resolve this problem, we employ a method to generate task graphs with *given* optimal schedule lengths and number of processors used in the optimal schedules.

The method of generating task graphs with known optimal schedules is as follows: Suppose that the optimal schedule length of a graph and the number of processors used are specified as SL_{opt} and p, respectively. For each PE i, we randomly generate a number x_i from a uniform distribution with mean $\frac{v}{p}$. The time interval between 0 and SL_{opt} of PE i is then randomly partitioned into x_i sections. Each section represents the execution span of one task, thus, x_i tasks are "scheduled" to PE i with no idle time slot. In this manner, v tasks are generated so that every processor has the same schedule length. To generate an edge, two tasks n_a and n_b are randomly chosen such that $FT(n_a) < ST(n_b)$.[5] The edge is made to emerge from n_a to n_b. As to the edge weight, there are two cases to consider: 1) the two tasks are scheduled to different processors, and 2) the two tasks are scheduled to the same processor. In the first case, the edge weight is randomly chosen from a uniform distribution with maximum equal to $(ST(n_b) - FT(n_a))$ (the mean is adjusted according to the given CCR value). In the second case, the edge weight can be an arbitrary positive integer because the edge does not affect the start and finish times of the tasks which are scheduled to the same processor. We randomly chose the edge weight for this case according to the given CCR value. Using this method, we generated three sets of task graphs with three CCRs: 0.1, 1.0, and 10.0. Each set consists of graphs in which the number of nodes vary from 50 to 500 in increments of 50; thus, each set contains 10 graphs. The graphs within the same set have the same value of CCR. Hereafter, we call this suite of graphs the type-2 random task graphs.

The results of applying the FASTEST algorithm on these two suite of random task graphs using 1, 2, 4, 8, and 16 PPEs are included in Table 5. Using 1 PPE means that the algorithm used is the sequential FAST algorithm. As graph size does not show any significant impact, we computed the average percentage deviations (in schedule length) from the optimal solutions. Furthermore, the number of optimal solutions generated are also shown (in brackets following the percentage deviations). As can be seen from Table 5a, the FASTEST algorithm generated a significant number of optimal solutions and the percentage deviations were also small when optimal solutions were not produced. We also observe that the parallelized random neighborhood search process indeed improved the initial schedules considerably in most cases. One important observation is that the deviations from optimal did not vary much with increasing number of PPEs used. One explanation for this phenomenon is that the final solutions of such cases can be reached within a few transferal of blocking-nodes. Another observation is that when 16 PPEs were used, the deviations in general increased. This is presumably due to the small sizes of blocking-node subsets which restrict the diversity of the random search. It should be noted that for the small type-1 graphs, the blocking-nodes subsets of the PPEs were not disjoint so as to make each subset contain at least two nodes.

TABLE 5
RESULTS OF THE FASTEST ALGORITHM COMPARED AGAINST OPTIMAL SOLUTIONS

	CCR		
	0.1	1.0	10.0
InitialSchedule	16.24% (1)	17.51% (1)	29.39% (1)
1 PPE	11.71% (5)	9.24% (3)	20.00% (4)
2 PPE	11.71% (5)	9.24% (3)	20.00% (4)
4 PPE	11.71% (5)	9.24% (3)	20.00% (4)
8 PPE	11.71% (5)	9.24% (3)	20.00% (4)
16 PPE	13.41% (4)	10.00% (2)	21.94% (2)

(a)

	CCR		
	0.1	1.0	10.0
InitialSchedule	19.53% (0)	28.98% (0)	36.12% (0)
1 PPE	10.50% (1)	17.13% (0)	25.35% (0)
2 PPE	10.50% (1)	17.13% (0)	25.35% (0)
4 PPE	10.50% (1)	17.13% (0)	25.35% (0)
8 PPE	9.65% (1)	16.47% (0)	21.51% (0)
16 PPE	9.22% (1)	14.98% (0)	19.23% (0)

(b)

Percentage deviation and number of optimal solutions for (a) type-1 random task graphs (12 graphs); and (b) type-2 random task graphs (10 graphs).

Table 5b shows the results of the FASTEST algorithm for the type-2 random task graphs. For these much larger graphs, the FASTEST algorithm generated fewer optimal solutions. However, an encouraging observation is that the percentage deviations were still small. Indeed, the worst average deviation was only about 37 percent. One interesting observation is that in some cases, using more PPEs improved the schedule lengths. This observation is consistent with our analysis presented in Section 2.3, which shows that parallelization may potentially improve the solution quality. This is due to the partitioning of the search neighborhood which lets the random search to explore different regions of the search space simultaneously, thereby increasing the likelihood of getting better solutions.

The speedups of the FASTEST algorithm are shown in Fig. 6. We can see that the speedups are almost linear. Also, the speedups for type-2 task graphs are considerably higher. The linear speedup of the FASTEST algorithm is because of the following reasons. As noted in Section 2, the sequential execution time of the FASTEST algorithm (that is, the time required to execute the FASTEST algorithm using only one PPE), denoted by t_{seq}, is given by:

$$t_{seq} = t_{list} + kt_{sch},$$

where t_{list} is the time required to construct the initial schedule using the CPN-Dominant list, k is the total number of random neighborhood search steps (i.e., $k = MAXCOUNT \times MAXSTEP$), and t_{sch} is the time required to perform a list scheduling of the graph. Note that we only include the running time of the two dominant phases of the FASTEST algorithm. Moreover, it should be noted that both t_{list} and t_{sch} are $O(e)$.

[5] ST and FT denote start time and finish time, respectively.

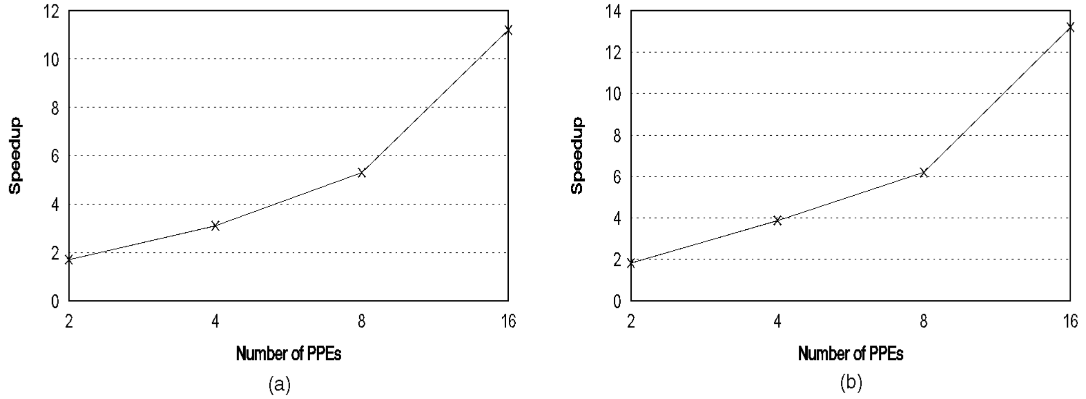


Fig. 6. Speedups of the FASTEST algorithm: (a) type-1 random graphs; (b) type-2 random graphs.

On the other hand, the running time of the FASTEST algorithm using q PPEs, denoted by t_{par}, is given by:

$$t_{par} = t_{list} + \frac{k}{q}t_{sch} + \left(\log\frac{k}{q}\right)t_{comm},$$

where t_{comm} is the time required for a inter-PPE communication phase which occurs $\log\frac{k}{q}$ times using the communication schedule with exponentially decreasing periods (notice that the logarithm is of base 2).

Given t_{seq} and t_{par}, the speedup of the FASTEST algorithm S_q is approximately given by:

$$S_q = \frac{t_{seq}}{t_{par}} = \frac{t_{list} + kt_{sch}}{t_{list} + \frac{k}{q}t_{sch} + \left(\log\frac{k}{q}\right)t_{comm}}.$$

We performed some experiments using the type-2 random graphs and found that t_{list} is approximately equal to t_{sch} (see Fig. 7). We then have:

$$S_q = \frac{k+1}{\frac{k}{q} + 1 + \left(\log\frac{k}{q}\right)\left(\frac{t_{comm}}{t_{sch}}\right)}.$$

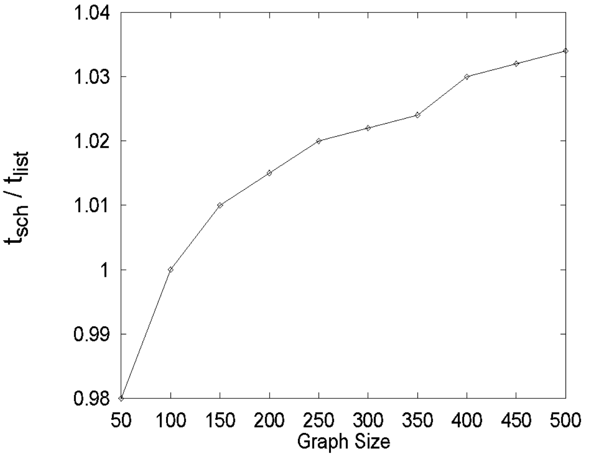


Fig. 7. The ratio of t_{sch} to t_{list} using the type-2 random graphs.

The speedup expression implies that the FASTEST algorithm can attain a linear speedup if k is large relative to q and t_{comm} is small relative to t_{sch}. However, as mentioned in Section 2.2, the value of k is fixed to be $8 \times 64 = 512$ (recall that $MAXSTEP = 8$ and $MAXCOUNT = 64$). In addition, we do not expect t_{comm} to be significantly smaller than t_{sch} on most coarse-grain parallel machines, including the Intel Paragon, IBM SP-2 or CM-5. It is because t_{comm} entails a broadcasting of a list of size v, which constitutes the best schedule. Thus, in practice, the speedup of the FASTEST algorithm will be less than linear. Nevertheless, it is interesting to note the approximate speedup values of the FASTEST algorithm with different values of $\frac{t_{comm}}{t_{sch}}$, as shown in Table 6. The speedups are computed using $k = 512$ and $q = 2, 4, 8, 16$.

From Table 6, we observe that the speedups show a larger deviation from linear as the number of PPEs used is larger. Moreover, the speedups are critically affected by the communication time to scheduling time ratio. Thus, as the inter-PPE communication for larger task graphs is not a significant overhead, the speedups for type-2 graphs are higher than that for type-1 graphs.

3.6 Results for Very Large DAGs

To test the scalability and robustness of the parallel FASTEST algorithm, we performed experiments with very large DAGs, which may be difficult to handle by the FAST algorithm. These DAGs include a 10,728-node Gaussian elimination graph, a 10,000-node Laplace equation solver graph, a 12,287-node FFT graph, and a 10,000-node random graph. For these graphs we simply measured the schedule length produced by the DLS, DSC, ETF, and FASTEST algorithms

TABLE 6
APPROXIMATE SPEEDUPS OF THE FASTEST ALGORITHM

Number of PPEs Used	$\frac{t_{comm}}{t_{sch}} = 0.5$	$\frac{t_{comm}}{t_{sch}} = 1$	$\frac{t_{comm}}{t_{sch}} = 2$
2	1.97	1.94	1.88
4	3.87	3.77	3.59
8	7.54	7.23	6.66
16	14.45	13.5	11.93

TABLE 7
NORMALIZED SCHEDULE LENGTHS AND SCHEDULING TIMES FOR THE LARGE DAGS FOR ALL THE SCHEDULING ALGORITHMS

	Graph types (Number of Nodes)			
Algorithm	Gauss (10728)	Laplace (10000)	FFT (12287)	Random (10000)
FASTEST	1.00	1.00	1.00	1.00
DSC	1.12	1.23	1.21	1.15
ETF	1.08	1.20	1.18	1.12
DLS	1.07	1.20	1.18	1.10

(a)

	Graph types (Number of Nodes)			
Algorithm	Gauss (10728)	Laplace (10000)	FFT (12287)	Random (10000)
FASTEST	30.24	31.68	48.88	40.68
DSC	298.34	228.23	600.23	463.42
ETF	6059.69	8235.23	10234.21	9324.82
DLS	16377.28	22877.40	29877.35	21908.43

(b)

(a) Normalized schedule lengths for large DAGs; the FASTEST algorithm used 16 PPEs on the Intel Paragon; (b) Scheduling times (sec) on the Intel Paragon; the FASTEST algorithm used 16 PPEs while other algorithms used 1 PPE.

using the Paragon. The FASTEST algorithm used 16 PPEs, while the other algorithms used one PPE.

The schedule lengths for the large DAGs, normalized with respect to that of the FASTEST algorithm, are shown in Table 7a. Note that the MD algorithm is not included in the comparison because it took more than 8 hours to produce a schedule for a 2,000-node DAG. The FASTEST algorithm outperformed all the algorithms in these test cases, with its percentage improvement ranging from 8 percent to 23 percent over the other algorithms. Concerning the scheduling times, Table 7b indicates that the ETF and DLS algorithms are considerably slower than the FASTEST and DSC algorithms, while the FASTEST algorithm outperforms the DSC algorithm both in terms of solution quality and complexity. These results with large DAGs indeed provide further evidence to the claim that the FASTEST algorithm is suitable for finding high quality schedules for large DAGs.

4 CONCLUDING REMARKS

In this paper, we have presented a low complexity algorithm to meet the conflicting goals of high performance and low time complexity for the compile-time multiprocessor scheduling problem. Based on a simple but robust technique, the FASTEST algorithm first generates an initial schedule and then refines it in parallel using probabilistic search.

We have compared the proposed algorithm with a number of well-known algorithms using both real applications and randomly generated task graphs. The results presented in this paper and the comparison of 14 algorithms in [1], [15] indicate that the FASTEST algorithm is better than these existing algorithms in terms of both solution quality and complexity. The algorithm generates good solutions within a short period of time for practical problem sizes which are too large to be efficiently handled by existing scheduling heuristics. The FASTEST algorithm has been incorporated as a core module in our automatic parallelization tool called CASCH for many numerical applications.

An interesting observation about the FASTEST algorithm is that parallelization can sometimes improve solution quality. This is due to the partitioning of the blocking-nodes set, which implies a partitioning of the search neighborhood that causes the algorithm to explore the search space simultaneously, thereby enhancing the likelihood of getting better solutions. Further research is needed on improved parallelization of search.

ACKNOWLEDGMENTS

We would like to thank the referees for their insightful comments. This research was supported by the Hong Kong Research Grants Council under contract numbers HKUST734/96E and HKUST6076/97E.

REFERENCES

[1] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu, "Automatic Parallelization and Scheduling of Programs on Multiprocessors Using CASCH," *Proc. 1997 Int'l Conf. Parallel Processing*, pp. 288-291, Aug. 1997.

[2] I. Ahmad and Y.-K. Kwok, "Optimal and Near-Optimal Allocation of Precedence-Constrained Tasks to Parallel Processors: Defying the High Complexity Using Effective Search Techniques," *Proc. 1998 Int'l Conf. Parallel Processing*, pp. 424-431, Aug. 1998.

[3] D.-K. Chen and P.-C. Yew, "On Effective Execution of Non-Uniform DOACROSS Loops," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 5, pp. 463-476, May 1996.

[4] M. Cosnard and M. Loi, "Automatic Task Graph Generation Techniques," *Parallel Processing Letters*, vol. 5, no. 4, pp. 527-538, 1995.

[5] H. El-Rewini, H.H. Ali, and T.G. Lewis, "Task Scheduling in Multiprocessing Systems," *Computer*, pp. 27-37, Dec. 1995.

[6] T. Fahringer, "Compile-Time Estimation of Communication Costs for Data Parallel Programs," *J. Parallel and Distributed Computing*, vol. 39, pp. 46-65, 1996.

[7] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[8] M. Girkar and C.D. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 2, pp. 166-178, 1992.

[9] M. Gupta and P. Banerjee, "Compile-Time Estimation of Communication Costs on Multicomputers," *Proc. Sixth Int'l Parallel Processing Symp.*, Mar. 1992.

[10] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM J. Computing*, vol. 18, no. 2, pp. 244-257, Apr. 1989.

[11] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. Computers*, vol. 33, no. 11, pp. 1,023-1,029, Nov. 1984.

[12] K. Kennedy, N. McIntosh, and K.S. Mckinley, "Static Performance Estimation in a Parallelizing Compiler," Technical Report TR 91-174, Dept. of Computer Science, Rice Univ., Dec. 1991.

[13] B.W. Kernighan and S. Lin, "An Effective Heuristic Procedure for Partitioning Graphs," *Bell Systems Technical J.*, vol. 49, pp. 291-308, Feb. 1970.

[14] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs onto Multi-Processors," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506-521, May 1996.

[15] Y.-K. Kwok and I. Ahmad, "Benchmarking the Task Graph Scheduling Algorithms," *Proc. 12th Int'l Parallel Processing Symp.*, pp. 531-537, Apr. 1998.

[16] Y.-K. Kwok and I. Ahmad, “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors,” *ACM Computing Surveys*, accepted for publication and to appear.

[17] B. Lee, A.R. Hurson, and T.Y. Feng, “A Vertically Layered Allocation Scheme for Data Flow Systems,” *J. Parallel and Distributed Computing*, vol. 11, pp. 175-187, 1991.

[18] Z. Li and P.-C. Yew, “Program Parallelization with Interprocedural Analysis,” *J. Supercomputing*, vol. 2, no. 2, pp. 225-244, Oct. 1988.

[19] Z. Li, P.-C. Yew, and C.-Q. Zhu, “An Efficient Data Dependency Analysis for Parallelizing Compilers,” *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, pp. 26-34, Jan. 1990.

[20] M.A. Palis, J.-C. Liou, and D.S.L. Wei, “Task Clustering and Scheduling for Distributed Memory Parallel Architectures,” *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 1, pp. 46-55, Jan. 1996.

[21] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, N.J.: Prentice Hall, 1982.

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

[23] B. Shirazi, M. Wang, and G. Pathak, “Analysis and Evaluation of Heuristic Methods for Static Scheduling,” *J. Parallel and Distributed Computing*, no. 10, pp. 222-232, 1990.

[24] G.C. Sih and E.A. Lee, “A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures,” *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 2, pp. 75-87, Feb. 1993.

[25] M.-Y. Wu and D.D. Gajski, “Hypertool: A Programming Aid for Message-Passing Systems,” *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330-343, July 1990.

[26] T. Yang and A. Gerasoulis, “DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors,” *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951-967, Sept. 1994.

Yu-Kwong Kwok received his BSc degree in computer engineering from the University of Hong Kong in 1991, and the MPhil and PhD degrees in computer science from the Hong Kong University of Science and Technology in 1994 and 1997, respectively. Currently, he is an assistant professor in the Department of Electrical and Electronic Engineering at the University of Hong Kong. Before joining the University of Hong Kong, he was a visiting scholar in the parallel processing laboratory of the School of Electrical and Computer Engineering at Purdue University for one year. His research interests include software support for parallel processing, heterogeneous cluster computing, and multimedia systems. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Ishfaq Ahmad received a BSc degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan, in 1985. He received his MS degree in computer engineering and PhD degree in computer science, both from Syracuse University in 1987 and 1992, respectively. Currently, he is an associate professor in the Department of Computer Science at the Hong Kong University of Science and Technology, where he is also the director of the Video Technology Center. His research interests are in the areas of parallel programming tools, scheduling and mapping algorithms for scalable architectures, video technology, and interactive multimedia systems. He has published more than 80 papers in the above areas. He has received numerous research and teaching awards, including the Best Student Paper Award at Supercomputing ’90 and Supercomputing ’91, and the Teaching Excellence Award from the School of Engineering at Hong Kong University of Science and Technology. He has served on the committees of various international conferences, has been a guest editor for two special issues of *Concurrency*: *Practice and Experience* related to resource management, and is co-guest-editing a forthcoming special issue of the *Journal of Parallel and Distributed Computing* on the topic of software support for distributed computing. He also serves on the editorial board of *Cluster Computing*. He is a member of the IEEE Computer Society.