Muhammad Kafil and Ishfaq Ahmad
The Hong Kong University of Science and Technology

A distributed system comprising networked heterogeneous processors requires efficient task-to-processor assignment to achieve fast turnaround time. The authors propose two algorithms based on the A technique, which are considerably faster, are more memory-efficient, and give optimal solutions. The first is a sequential algorithm that reduces the search space. The second proposes to lower time complexity, by running the assignment algorithm in parallel, and achieves significant speedup.*

Optimal Task Assignment in Heterogeneous Distributed Computing Systems

As computer networks and sequential computers advance, distributed computing systems—such as a network of heterogeneous workstations or PCs—become an attractive alternative to expensive, massively parallel machines. To exploit effective parallelism on a distributed system, tasks must be properly allocated to the processors. This problem, *task assignment*, is well-known to be NP-hard in most cases.[1] A task-assignment algorithm seeks an assignment that optimizes a certain cost function—for example, maximum throughput or minimum turnaround time. However, most reported algorithms yield suboptimal solutions. In general, optimal solutions can be found through an exhaustive search, but because there are n^m ways in which m tasks can be assigned to n processors, an exhaustive search is often not possible. Thus, optimal-solution algorithms exist only for restricted cases or very small problems. The other possibility is to use an *informed* search to reduce the state space.

The A* algorithm, an informed-search algorithm, guarantees an optimal solution, but doesn't work for large problems because of its high time and space complexity. Thus, we require a further-reduced state space, a faster search process, or both.

Problem definition

Like other NP-hard problems, there are three common ways to tackle task assignment:

- *Relaxation*. You can relax some of the requirements or restrict the problem.
- *Enumerative optimization*. If you can't compromise the solution's optimality, you can use enumerative methods, such as dynamic programming and branch-and-bound.
- *Approximate optimization*. You can use heuristics to solve the problem while aiming for a near-optimal or good solution.

Our goal is to assign a given task graph (see the "Related work" sidebar) to a network of processors to minimize the time

1092-3063/98/$10.00 © 1998 IEEE

required for program completion. We consider this problem, also known as the allocation or mapping problem,[2] using relaxed assumptions—such as arbitrary computation and task-graph communication requirements, and a network of heterogeneous processors connected by an arbitrary topology. We use the *task interacting graph* model, in which the parallel program is represented by an undirected graph: $G_T = (V_T, E_T)$, where V_T is the set of vertices, $\{t_1, t_2, ..., t_m\}$, and E_T is a set of edges labeled by the communication requirements among tasks. We can also represent the network of processors as an undirected graph, where vertices represent the processors, and the edges represent the processors' communication links. We represent the interconnection network of n processors, $\{p_1, p_2, ..., p_n\}$, by an $n \times n$ link matrix $\mathbf{L}$, where an entry $\mathbf{L}_{ij}$ is 1, if processors i and j are directly connected, and 0 otherwise. We do not consider the case where i and j are not directly connected.

We can execute a task t_i from the set V_T on any one of the system's n processors. Each task has an associated execution cost on a given processor. A matrix $\mathbf{X}$ gives task-execution costs, where $\mathbf{X}_{ip}$ is the execution cost of task i on processor p. Two tasks, t_i and t_j, executing on two different processors, incur a communications cost when they need to exchange data. Task mapping will assign two communicating tasks to the same processor or to two different, directly connected processors. A matrix $\mathbf{C}$ represents communication among tasks, where $\mathbf{C}_{ij}$ is the communication cost between tasks i and j, if they reside on two different processors.

A processor's load comprises all the execution and communication costs associated with its assigned tasks. The time needed by the heaviest-loaded processor will determine the entire program's completion time. The task-assignment problem must find a mapping of the set of m tasks to n processors that will minimize program completion time. Task mapping, or assignment to processors, is given by a matrix $\mathbf{A}$, where $\mathbf{A}_{ip}$ is 1, if task i is assigned to processor p, and 0 otherwise. The following equation then gives the load on p:

$$\sum_{i=1}^{m} \mathbf{X}_{ip} \bullet \mathbf{A}_{ip} + \sum_{\substack{q=1 \\ (q \neq p)}}^{n} \sum_{i=1}^{m} \sum_{j=1}^{m} \left(\mathbf{C}_{ij} \mathbf{A}_{ip} \mathbf{A}_{jq} \mathbf{L}_{pq} \right)$$

The first part of the equation is the total execution cost of the tasks assigned to p. The second part is the communication overhead on p. $\mathbf{A}_{ip}$ and $\mathbf{A}_{jq}$ indicate that task i and j are assigned to two different processors (p and q), and $\mathbf{L}_{pq}$ indicates that p and q are directly connected. To find the processor with the heaviest load, you need to compute the load on each of the n processors. The optimal assignment out of all possible assignments will allot the minimum load to the heaviest-loaded processor.

A* is a best-first search algorithm, which has been used extensively in artificial-intelligence problem solving.

Task assignment using the A* algorithm

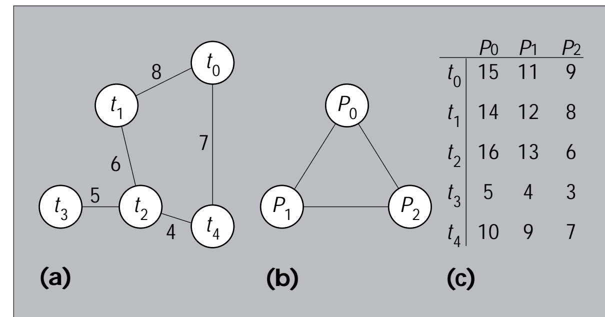
A* is a best-first search algorithm,[3] which has been used extensively in artificial-intelligence problem solving. Programmers can use the algorithm to search a tree or a graph. For a tree search, it starts from the root, called the start node (usually a null solution of the problem). Intermediate tree nodes represent the partial solutions, and leaf nodes represent the complete solutions or goals. A cost function f computes each node's associated cost. The value of f for a node n, which is the estimated cost of the cheapest solution through n, is computed as $f(n) = g(n) + h(n)$, where $g(n)$ is the search-path cost from the start node to the current node n and $h(n)$ is a lower-bound estimate of the path cost from n to the goal node (solution). To expand a node means to generate all of its successors or children and to compute the f value for each of them. The nodes are ordered for search according to cost; that is, the algorithm first selects the node with the minimum expansion cost. The algorithm maintains a sorted list, called `OPEN`, of nodes (according to their f values) and always selects a node with the best expansion cost. Because the algorithm always selects the best-cost node, it guarantees an optimal solution.

For the task-assignment problem under consideration,

- the search space is a tree;
- the initial node (the root) is a null-assignment node—that is, no task is assigned as yet;
- intermediate nodes are partial-assignment nodes—that is, only some tasks are assigned; and
- a solution (goal) node is a complete-assignment node—that is, all the tasks are assigned.

To compute the cost function, $g(n)$ is the cost of partial assignment (A) at node n—the load on the heaviest-loaded (p); this can be done using the equation from the previous section. For the computation of $h(n)$, two sets T_p (the set of tasks that are assigned to the heaviest-loaded p) and U (the set of tasks that are unassigned at this stage of the search and have one or more communication link with any task in set T_p) are defined. Each task t_i in U will be assigned either to p or any other processor q that has a direct communication link with p. So, you can associate two kinds of costs with each t_i's assignment: either X_{ip} (the execution cost of t_i on p) or the sum of the communication costs of all the tasks in set T_p that have a link with t_i. This implies that to consider t_i's assignment, we must decide whether t_i should go to p or not (by taking these two cases' minimum cost). Let cost(t_i) be the minimum of these two costs, then we compute $h(n)$ as

$$h(n) = \sum_{t_i \varepsilon U} \text{cost}(t_i).$$



	P_0	P_1	P_2
t_0	15	11	9
t_1	14	12	8
t_2	16	13	6
t_3	5	4	3
t_4	10	9	7

Figure 1. Examples of (a) a task graph and (b) a processor network, and the (c) execution-cost matrix of tasks on various processors.

Applying the algorithm

Shen and Tsai[4] first used the A* algorithm for the task-assignment problem. They ordered the tasks considered for assignment simply by starting with task 1 at the tree's first level, task 2 at the second, and so on. Ramakrishnan and colleagues showed that the order in which an algorithm considers tasks for allocation considerably impacts its performance.[5] Their study indicated significant performance improvement by considering, at shallow tree levels, tasks that have more weight in the optimal-cost computation. They proposed a number of heuristics to reorder the tasks, out of which the *minimax sequencing* heuristic performed the best. Minimax sequencing works as follows: Consider a matrix **H** of m rows and n columns where m is the number of tasks and n is the number of processors. The entry $\mathbf{H}(i, k)$ of the matrix is given by $\mathbf{H}(i, k) = \mathbf{X}_{ik} + h(v)$, where $h(v)$ is given by

$$h(v) = \sum_{j \in U} \min\left(\mathbf{X}_{jk}, \mathbf{C}_{ij}\right),$$

where U is the set of unassigned tasks that communicate with t_i. The minimax value, $mm(t_i)$ of task t_i, is defined as $mm(t_i) = \min\{\mathbf{H}(i, k), 1 \leq k \leq n\}$. The minimax sequence is then defined as $\Pi = \tau_1, \tau_2, \ldots, \tau_m\}$, $mm(\tau_I) \geq mm(\tau_{i+1}), \forall_i$.

An example

Let's illustrate the A* algorithm's operation for the assignment problem. Given a set of five tasks $\{t_0, t_1, t_2, t_3, t_4\}$ and a set of three processors $\{p_0, p_1, p_2\}$ (see Figure 1), we give the resulting search trees using the techniques proposed by Shen and Tsai (see Figure 2) and Ramakrishnan and his colleagues (see Figure 3). We will refer to these algorithms as A*O (A* Original) and A*R (A* with Reordering).

A search-tree node includes partial assignment of tasks to processors, and the value of f (the cost of partial assignment). The assignment of m tasks to n processors is indicated by an m-digit string, $a_0, a_1, \ldots, a_{m-1}$, where a_i $(0 \leq i \leq m-1)$ represents the processor (0 to $n-1$) to which the algorithm has assigned the ith task. A partial assignment means that some tasks are unassigned; the value of a_i equal to X indicates that ith task has not been assigned yet. Each level of the tree corresponds to a task; thus, assignment of this task to a processor replaces an X value in the assignment string with some processor number. Node expansion means adding a new task assignment to the partial assignment. Thus, the search tree's depth d equals the number of m tasks, and any node of the tree can have a maximum of n successors.

The root node includes the set of all unassigned tasks XXXXX. Next, for example, in Figure 2, we consider the allocations of t_0 to p_0 (0XXXX), t_0 to p_1 (1XXXX), and t_0 to p_2 (2XXXX), by determining the assignment costs at the tree's first level. Assigning t_0 to p_0 (0XXXX) results in the total cost $f(n)$ that is equal to 30. The $g(n)$, in this case, equals 15, which is the cost of executing t_0 on p_0. The $h(n)$ is also equal to 15, which is the sum of the minimum execution or communication costs of t_1 and t_4 (tasks communicating with t_0). We similarly calculate the costs of assigning t_0 to p_1 (26) and t_0 to p_2 (24). The algorithm inserts these three nodes into the list `OPEN`. Because 24 is the minimum cost, the algorithm selects the node 2XXXX for expansion.

The algorithm expands node 2XXXX in the following manner. At the tree's second level, the algorithm will consider t_1 for assignment, and 20XXX, 21XXX, and 22XXX are three possible assignments. The value of $f(n)$ for 20XXX is 28, and is computed as follows: first, the processor with the heaviest load is selected, which is p_0 in this case. $g(n)$ is equal to 22, which is the cost of executing t_1 on p_0 (14) plus the cost of communication between t_1 and t_0 (8), because they are assigned to two different processors. $h(n)$ is equal to 6, which is the minimum execution or communication cost of t_2 (the only unassigned task communicating with t_1). We similarly compute the values of $f(n)$ for 21XXX and 22XXX. At this point, nodes 0XXXX, 1XXXX, 20XXX, 21XXX, and 22XXX are in the `OPEN` list. Because node 1XXXX has the minimum node cost, the algorithm expands it next, resulting in nodes 10XXX, 11XXX, and 12XXX.

Attached to some of the nodes, the numbers in circles show the sequence in which nodes are selected for expansion. Bold lines show the edges connecting the nodes that lead to an optimal assignment. The search continues until the process selects the node with the complete assignment (20112) for expansion. At this point, because this node has a complete assignment and the minimum costs, it is the goal node. All assignment strings are unique.

In Figure 2, the order in which the algorithm considers tasks for assignment is $\{t_0, t_1, t_2, t_3, t_4\}$ and, during the search for an optimal solution, 42 nodes are generated and 14 are expanded. As Figure 3 shows, the A*R algorithm generates the minimax sequence $\{t_0, t_1, t_2, t_4, t_3\}$; therefore, t_4 is considered before t_3. You can similarly trace this example as demonstrated above, while considering the new task order. In this case, 39 nodes are generated, and 13 nodes are expanded. The optimal assignment is 20112, with the same optimal solution cost (28). In comparison, an exhaustive search will generate $n^m = 243$ nodes.

The proposed algorithms

We'll now describe our proposed algorithms. The first is a sequential algorithm that has considerably fewer memory requirements than the A*O and A*R algorithms. The second is a parallel algorithm that, compared with its serial counterpart, generates optimal solutions with good speedup.

Sequential search

The Optimal Assignment with Sequential Search (OASS) algorithm (see Figure

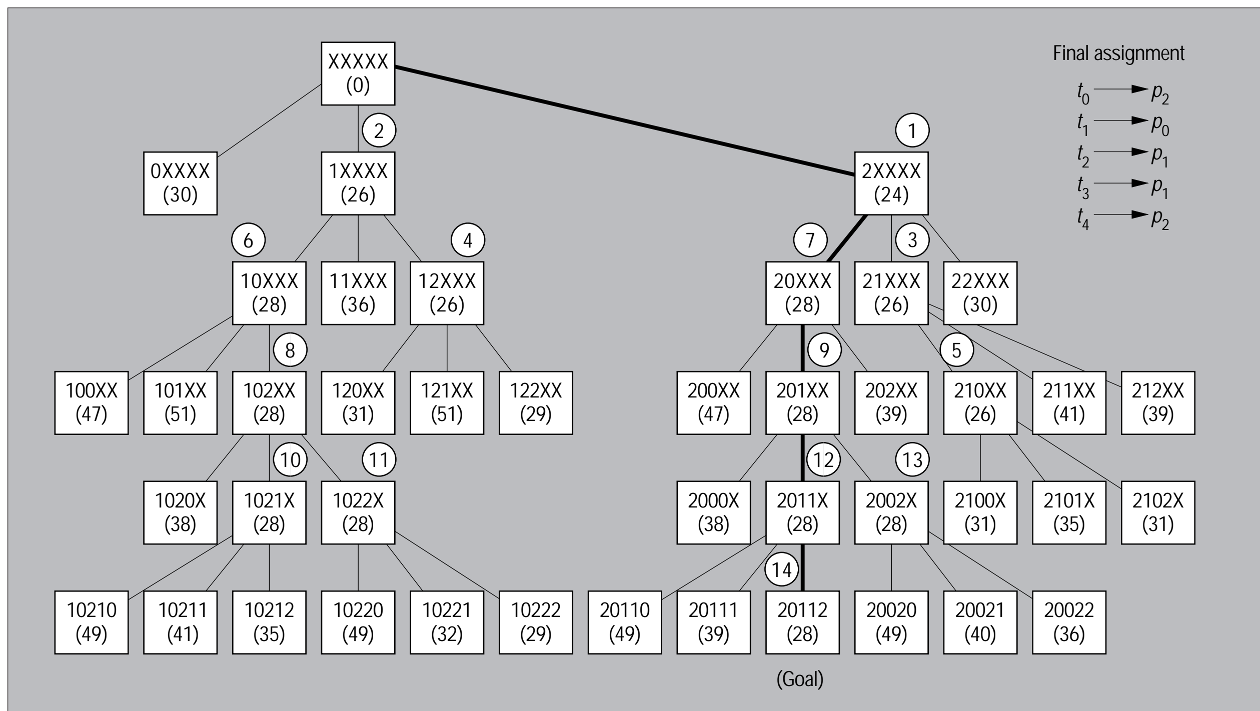


Figure 2. Search tree for the example problem using A*O (42 nodes generated, 14 nodes expanded).[4]

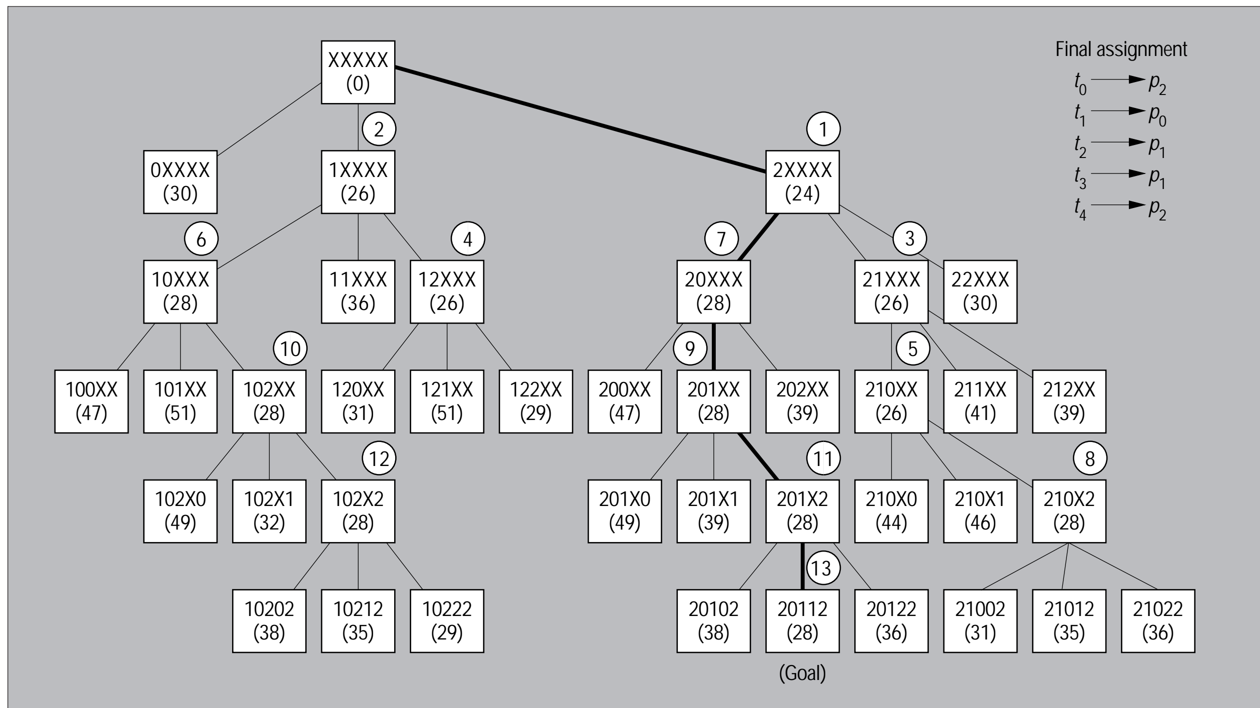


Figure 3. Search tree for the example problem using A*R (39 nodes generated, 13 nodes expanded).[5]

4) uses the A* search technique, but with two distinct features. First, it generates a random solution and prunes all the nodes with costs higher than this solution during the optimal-solution search. This is because the optimal solution cost will never be higher than this random-solution cost. Pruning unnecessary nodes not only saves memory, but also saves the time required to insert the nodes into `OPEN`. Second, the algorithm sets the value of $f(n)$ equal to $g(n)$ for all leaf nodes, because for a leaf node n, $h(n)$ is equal to 0. This avoids the unnecessary computation of $h(n)$ at the leaf nodes.

Figure 5 is search tree that results from using the OASS algorithm for our example problem. First, we used a faster and suboptimal version of A*[6] to generate a random solution. The cost of the random solution was 29. Therefore, we discard all nodes with a cost greater than 29. As a result, OASS generates only 14 nodes, while A*O produces 42 nodes

and A*R produces 39 nodes for the same optimal solution—20112. This algorithm's efficiency clearly depends on the initial solution's quality.

Parallel search

The parallel algorithm aims to speed up the search as much as possible using parallel processing. This is done by dividing the search tree among the *processing elements* (PEs) as evenly as possible and by avoiding the expansions of nonessential nodes—that is, nodes that the sequential algorithm does not expand. A. Grama and Vipin Kumar[7] and Vipin Kumar, K. Ramesh, and V.N. Rao[8] provide useful discussions of different issues in parallelizing the depth-first and best-first search algorithms. To distinguish the processors on which the parallel task-assignment algorithm is running from the processors in the problem domain, we will denote the former with the abbreviation PE—in our case, the Intel Paragon processor). We call our parallel algorithm Optimal Assignment with Parallel Search (OAPS).

Initially, we statically divide the search tree based on the number of PEs P in the system and the maximum number of successors S of a node in the search tree. There are three ways to achieve an initial partitioning:

- $P < S$. Each PE expands only the initial node, which results in S new nodes. Each PE gets one node, and the initial division distributes additional nodes by round-robin (RR).
- $P = S$. Each PE expands only the initial node, and each PE gets one node.
- $P > S$. Each PE keeps expanding nodes, starting from the initial node (the null assignment) until the list's number of nodes is greater than or equal to P. We sort the list in an increasing order of node-cost values. The first node in the list goes to PE_1, the second node to PE_p, the third node to PE_2, the fourth node to PE_{p-1}, and so on. Extra nodes are distributed by RR. Although this distribution does not guarantee that a best-cost node at the initial levels of the tree will lead to a good-cost node, the algorithm still tries to initially distribute the good nodes as evenly as possible among all the PEs.

If the search finds a solution, the algorithm terminates. There is no master PE responsible for first generating and then distributing the nodes to other PEs. Thus, compared to the host-node model, this static-node assignment's overhead is negligible. To illustrate this, try assigning 10 tasks to four processors using two PEs (PE_1 and PE_2). Here, S is 4 because a search-tree node can have a maximum of four successors; so each PE generates four nodes numbered 1 to 4 (as in Figure 6, where the boxed number is the node's f

```
(1)   Generate a random solution
(2)   Let S_Opt be the cost of this solution
(3)   Reorder the tasks
(4)   Build the initial node s and insert it into the
      list OPEN
(5)   Set f(s) = 0
(6)   Repeat
(7)      Select the node n with smallest f value.
(8)      if (n is not a Solution )
(9)          Generate successors of n
(10)         for each successor node n' do
(11)             if (n' is not at the last level in the
                      search tree)
(12)                 f(n') = g(n') + h(n')
(13)             else   f(n') = g(n')
(14)             if (f(n') <= S_Opt)
(15)                 Insert n' into OPEN
(16)         end for
(17)     end if
(18)     if (n is a Solution)
(19)         Report the Solution and stop
(20)  Until (n is a Solution) or (OPEN is empty)
```

Figure 4. The Optimal Assignment with Sequential Search algorithm (OASS).

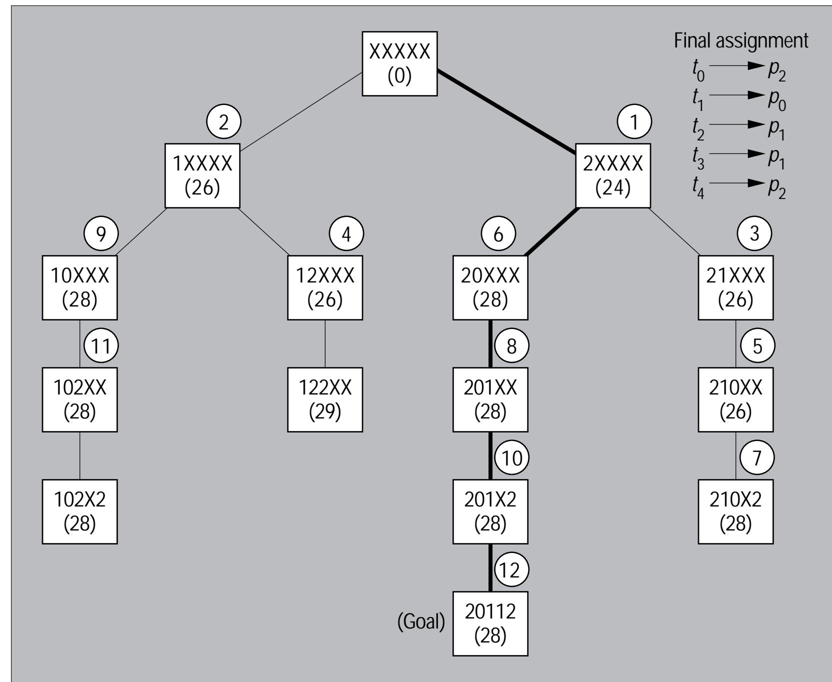


Figure 5. Search tree for the example problem using OASS (14 nodes generated, 12 nodes expanded).

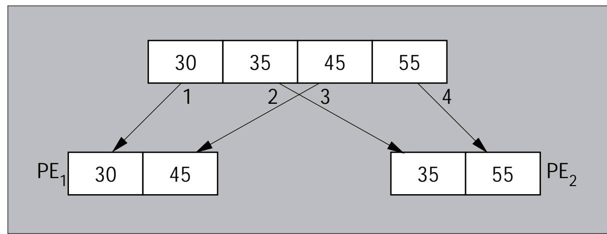


Figure 6. Initial static division for the Optimal Assignment with Parallel Search algorithm (OAPS).

value). PE_1 then gets assigned nodes 1 and 3, and PE_2 gets nodes 2 and 4.

After the initial static division, each PE will only search its own subtree. Some PEs might work on a good part of the search space, while others might expand unnecessary nodes that the serial algorithm does not expand. These actions will result in a poor speedup. To avoid this, the PEs need to communicate with each other to share the best part of the search space and to avoid unnecessary work. This communication can be global (that is, a PE broadcasts its nodes to all other PEs) or local (a PE communicates only with its neighbors). In our formulation, we used a RR communication strategy in the neighborhood.

Because the Paragon PEs are connected as a mesh topology, a PE can have a maximum of four neighbors. And because a PE communicates primarily with its neighbors, they incur a relatively small communication overhead, making this algorithm more scalable than the one using a global-communication strategy. Figure 7 describes the OAPS algorithm.

With an initial load division, every PE has one or more nodes in its `OPEN` list. All PEs then set up their neighborhood to find their neighboring PEs. A PE determines its neighborhood by using its own processor-mesh position and its x and y dimensions. A PE expands nodes starting with initial nodes. A PE periodically (when `OPEN` increases by a threshold u) selects a neighbor in a RR fashion and sends its best node to that neighbor. This achieves the sharing of the best part of the neighborhood's search space. Aside from this load balancing, a PE also broadcasts its solution (when it finds one) to all PEs. This helps avoid unnecessary work for a PE that is working on the bad part of the search space. Once a node receives a better cost solution than its current best node, it stops expanding the unnecessary nodes. The PE that finds the first solution broadcasts its cost to all other PEs. Then, a PE broadcasts a solution only if its cost is better than a solution received earlier. When a PE finds a solution, it records it and stops. The optimal solution will incur the minimum cost.

```
(1)  Init-Partition()
(2)  SetUp-Neighborhood()
(3)  Repeat
(4)      Expand the best cost node from OPEN
(5)      if (a Solution found)
(6)          if (it's better than any previously
                 received Solution)
(7)              BroadCast the Solution to all PEs
(8)          else
(9)              Inform neighbors that I am done
(10)         end if
(11)     Record the Solution and Stop
(12)     end if
(13)     If (OPEN's length increases by a threshold u)
(14)          Select a neighbor PE j using RR
(15)          Send the current best node from OPEN to j
(16)     end if
(17)     If (Received a node from a neighbor)
(18)         Insert it to OPEN
(19)     if (Received a Solution from a PE)
(20)         Insert it to OPEN
(21)         if (Sender is a neighbor)
(22)             Remove this from neighborhood list
(23)     end if
(24) Until (OPEN is empty) OR (OPEN is full)
```

Figure 7. The OAPS algorithm.

Experimental results

To test our two algorithms, we collected data for task graphs of 10 to 28 nodes with five different values of *communication-to-cost ratios* (CCR) and processor graphs of four nodes connected in three different topologies. For the parallel algorithm OAPS, we used two, four, eight, and 16 Paragon PEs.

Workload generation

A realistic workload is important to validate any assignment algorithm, but this area lacks standard benchmark test cases. Thus, to test the proposed sequential and parallel algorithms, we generated a library of task graphs and processor topologies.

In distributed systems, a number of process groups usually have heavy interaction in the group, but almost no interaction outside the group.[9] Using this intuition, we first generated a number of primitive task-graph structures such as pipeline, ring, server, and interference graphs of two to eight nodes. We generated complete task graphs by randomly selecting these primitives structures and combining them until reaching the desired number of tasks. To do this, we first selected a primitive graph and then combined it with a newly selected graph by a link labeled 1; the last node is connected back to the first node.

To generate the nodes' execution costs, the CCR's five values are 0.1, 0.2, 1.0, 5.0, and 10.0. The tasks' execution costs are generated in this manner: for example, if the total communication cost (the sum costs of all edges connected to the task) of task i is 16 and the CCR is 0.2, i's average

Related work

Task-to-processor allocation can be dynamic or static. In dynamic allocation, tasks are assigned on the fly, depending on the system's state. With static allocation, you assume that the programmer has information about the system's tasks and processors, and that the system makes task assignments before task execution. The main goal in static task-to-processor assignment is to assign an equal load to all processors and reduce their interaction overhead.

Given a parallel program represented by a task graph and a network of machines represented as a processors graph, assigning tasks to processors is also known as the *allocation* or *mapping problem*.[1] There are, in general, two models of task graphs: the *task precedence graph*, also called the *directed acyclic graph*, and the *task interacting graph*, with undirected edges.

The TPG model represents parallel programs by capturing the precedence relations between tasks.[2,3] The TIG model represents applications in which multiple tasks can run concurrently, regardless of their precedence. This model represents a wide range of iterative parallel programs, such as those that solve systems of equations for finite-element applications and power-system simulations, and VLSI simulation programs. Further discussion of TIG and TPG models appears elsewhere.[4]

A large number of task-assignment algorithms have been proposed, using various techniques such as network flow,[5] state-space search,[6–8] clustering,[9] bin-packing,[10] and probabilitic and randomized optimization.[11–14] Most of these algorithms can be classified using the taxonomy in Figure A. One can classify these algorithms at the hierarchy's first level into optimal and suboptimal categories. The optimal algorithms can be further classified to restricted or nonrestricted. Restricted algorithms give optimal solutions in polynomial time by restricting the program's structure, the processor network, or both. Nonrestricted solutions, on the other hand, consider the problem in general and give optimal solutions, but not in polynomial time.

Suboptimal algorithms can be classified as approximate or heuristic. Approximate algorithms for task assignment use the

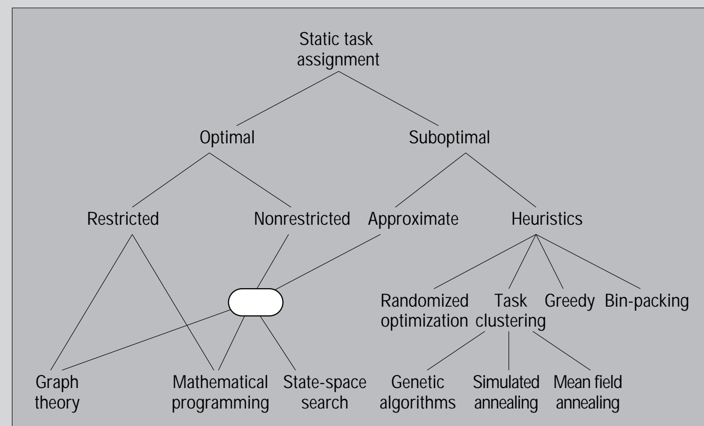


Figure A. A classification of task-assignment algorithms.

execution cost will be 80. Because we are assuming the processors to be heterogeneous (homogeneous processors are a special case of heterogeneous processors), the execution cost varies from processor to processor in the execution-cost matrix (**X**), but the average value remains the same across all processors.

Memory efficiency of OASS

First, we compare the results of the memory saved by using the OASS and A*R algorithms. Both A*R and OASS start by reordering the tasks, but OASS also generates a random solution to prune unnecessary nodes. Generating the random solutions first (see Table 1 and Figure 8) saves a significant amount of memory. In Table 1, task graphs of 10 to 20 nodes, using a fully connected topology of four processors with a CCR of 0.1 achieve, save an average of 63.84% memory. For all the cases, OASS expands fewer nodes than A*R. Figure 8 shows the number of tree nodes generated by A*R and OASS during the optimal-solution search for task graphs of 10 to 26 nodes using the ring topology. For this case, the average memory saving is 66.01%.

Table 2 presents the running times of A*O, A*R, and OASS for a fully connected topology of 4 processors. A*R shows a substantial improvement over A*O. Furthermore, A*O could not generate solutions with 16 or more tasks. An entry ** in a column means that the algorithm could not run using all available memory at a single Intel Paragon node, during an approximately four-hour run. For most of the cases, task graphs with lower CCRs (for example, 0.1 and 0.2) result in larger search trees; that is, they require more memory and take more time to run than graphs with high CCRs (1, 5, and 10). Graphs with a CCR of 10 have the lowest memory and runtime requirements, for two rea-

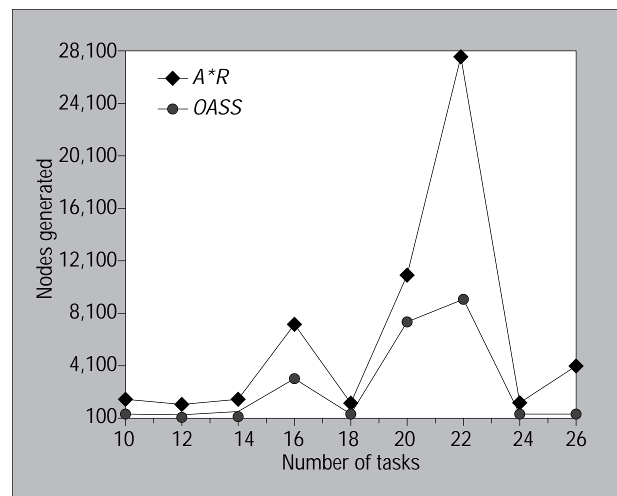


Figure 8. Nodes generated by A*R and OASS for a ring topology (CCR = 0.1).

same computational model as the optimal algorithm but aim for good solutions instead of searching the complete solution space for optimal solutions. Heuristic approaches use special parameters that affect the system indirectly—for example, clustering the groups of heavily communicating tasks together. A greedy heuristic starts from a partial assignment and, without any backtracking, assigns one task at each step until it obtains a complete assignment. A bin-packing technique uses a sizing policy, an ordering policy, and a placement policy for the tasks to be assigned. Randomized optimization methods start from a complete assignment and search for an improvement in the assignment by exchanging two tasks on different processors or moving a task to another processor.

Because of the assignment problem's intractable nature, most research focuses on developing heuristic algorithms. Some optimal algorithms are also available either for restricted problem cases or for very small problems.

References

1. S.H. Bokhari, "On the Mapping Problem," *IEEE Trans. Computers*, Vol. C-30, Mar. 1981, pp. 207–214.
2. Y. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs onto Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, Vol. 7, No. 5, May 1996, pp. 506–521.
3. L. Wang, H.J. Siegel, and V. Roychowdhury, "A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Environments," *HCW '96: Proc. Fifth Heterogeneous Computing Workshop*, IEEE Press, Piscataway, N.J.,1996, pp. 72–85.
4. P. Sadayappan, F. Ercal, and J. Ramanujam, "Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube," *Parallel Computing*, Vol. 13, No. 1, Jan. 1990, pp. 1–16.
5. H.J. Siegel et al., "Software Support for Heterogeneous Computing," *Parallel and Distributed Computing Handbook*, A.Y. Zomaya, ed., McGraw-Hill, New York, 1996, pp. 725–761.
6. S. Ramakrishnan, H. Chao, and L.A. Dunning, "A Close Look at Task Assignment in Distributed Systems," *Proc. IEEE Infocom '91*, IEEE Computer Society Press, Los Alamitos, Calif., 1991, pp. 806–812.
7. C.-C. Shen and W.-H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing System Using a Minimax Criterion," *IEEE Trans. Computers*, Vol. C-34, No. 3, Mar. 1985, pp. 197–203.
8. H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, Jan. 1977, pp. 85–93.
9. N.S. Bowen, C.N. Nikolaou, and A. Ghafoor, "On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems," *IEEE Trans. Computers*, Vol. 41, No. 3, Mar. 1992, pp. 197–203.
10. J.B. Sinclair, "Efficient Computation of Optimal Assignments for Distributed Tasks," *J. Parallel and Distributed Computing*, Vol. 4, No. 4, 1987, pp. 342–362.
11. I. Ahmad and M.K. Dhodhi, "Task Assignment Using the Problem-Space Genetic Algorithm," *Concurrency: Practice and Experience*, Vol. 7, No. 5, Aug. 1995, pp. 411–428.
12. T. Bultan and C. Aykanat, "A New Heuristic Based on Mean Field Annealing," *J. Parallel and Distributed Computing*, Vol. 16, No. 4, Dec. 1992, pp. 292–305.
13. S.M. Hart and C.-L.S. Chen, "Simulated Annealing and the Mapping Problem: A Computational Study," *Computers and Operations Research*, Vol. 21, No. 4, 1994, pp. 455–461.
14. L. Wang, H.J. Siegel, and V. Roychowdhury, "A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Environments," *HCW '96: Proc. Fifth Heterogeneous Computing Workshop*, 1996, pp. 72–85.

sons. First, for higher CCRs, the algorithm follows a straight path in the search tree, assigning a group of highly communicating tasks to the same processor, while for lower CCRs such as 0.1 and 0.2, it explores different paths (considers more options) and, hence, takes more time. Second, because the optimal solution's cost for higher CCRs is less than that of lower CCRs, the algorithm finds the optimal solution quickly, starting from an initial cost 0. For example, a task graph of 10 tasks using a CCR of 10 has a 7.36 solution cost. The same graph using a CCR of 0.1 has a 374.0 solution cost. Thus, a CCR of 10 takes only 0.4 seconds to find the optimal solution, while a CCR of 0.1 takes 4.3 seconds.

Processor topology also significantly impacts the search tree's size and running time. This is because the algorithm assigns two communicating tasks to two different processors only if the processors are directly connected. Based on this constraint, in a chain or ring topology, the algorithm prunes some search-tree nodes. On the other hand, no such pruning occurs for the fully connected case. Therefore, the task graphs with a CCR of 0.1 using the fully connected processor topology have the greatest running times and memory requirements among the graphs generated.

Table 1: Memory savings for a fully connected topology, CCR = 0.1. An entry ** in a column means that the algorithm could not run using all available memory at a single Intel Paragon node, during an approximately four-hour run.

	Nodes expanded			Nodes generated			
No. of Tasks	(A*O)	(A*R)	(OASS)	(A*O)	(A*R)	(OASS)	Memory savings (%)
10	6,046	2,296	2,265	24,180	9,188	2,456	73.27
12	52,554	3,135	2,953	210,212	12,544	4,976	60.33
14	113,407	4,057	1,743	453,624	16,232	2,791	82.81
16	**	73,904	71,953	**	295,620	106,572	63.95
18	**	85,294	82,917	**	341,180	121,306	64.45
20	**	58,367	57,833	**	233,468	144,149	38.26
Average		37,842	36,610		151,372	63,708	63.84

Table 2: Running times using a fully connected topology (CCR=0.1).

No. of Tasks	Time (seconds)		
	A*O	A*R	OASS
10	81.41	30.14	30.68
12	1,033.46	58.96	56.88
14	3,025.66	105.05	62.13
16	**	1,550.46	1,501.38
18	**	3,839.00	3,627.06
20	**	3,191.86	3,173.18

Table 3: Speedup with the parallel algorithm using a fully connected topology (CCR = 0.1).

No. of Tasks	Time(OASS) / Time(OPAS)			
	2 PEs	4 PEs	8 PEs	16 PEs
10	1.87	3.48	5.72	7.63
12	1.96	3.68	3.60	12.85
14	1.70	2.02	4.58	4.64
16	2.00	2.94	4.72	6.71
18	2.00	3.86	7.59	13.16
20	1.78	3.72	5.62	9.97
Average	1.89	3.28	5.30	9.13

Speedup using the parallel algorithm

We evaluated our parallel algorithm's performance by observing its speedup over its sequential counterpart on a varying number of processors. Table 3 presents the speedup for a fully connected topology of four processors with a CCR of 0.1. The second, third, fourth, and fifth columns include the speedups of the parallel algorithm over the serial algorithm for two, four, eight, and 16 Paragon PEs. The table's bottom row includes the average speedup of all the task graphs considered. The speedup is almost linear for most of the cases using two or four PEs, while for eight and 16 PEs speedup increases with the problem-size increase. Also, the problems with a CCR of 0.1 and 0.2 give good speedup in most of the cases, because the serial algorithm's running time is longer than the higher CCRs.

Figure 9 depicts the average speedup values for fully connected, ring, and chain topologies using different CCR values.

Because of the assignment problem's NP-completeness, its worst-case complexity remains exponential. However, our algorithms reduce the average-case complexity by a large margin and, therefore, can help generate optimal solutions for medium problems. The proposed algorithms apply to homogeneous or heterogeneous processors, although in this article we only considered the heterogeneous cases. The sequential algorithm provides considerable improvements in terms of memory and time over the previous studies (see the "Related work" sidebar). Using a fully connected processor topology will further improve the sequential algorithm's performance.[6] In addition, faster versions of the OASS algorithm are possible, without a guarantee of an optimal solution but with approximated close-to-optimal solutions.[6]

Parallelizing the assignment algorithm is an interesting problem that leads to several avenues of research. We have used a simple mapping scheme for the Paragon's parallel algorithm, but some fine-tuning of the search-tree partitioning can further improve performance. Also, to make additional improvements, further study is required to better understand the parallel algorithm's behavior. One possibility is to let a PE find more than one solution and implement a termination-detection policy. Also, more experimentation about what should be an ideal value for threshold u can be done. Researchers could implement and compare different load-balancing policies for the algorithm's speedup.

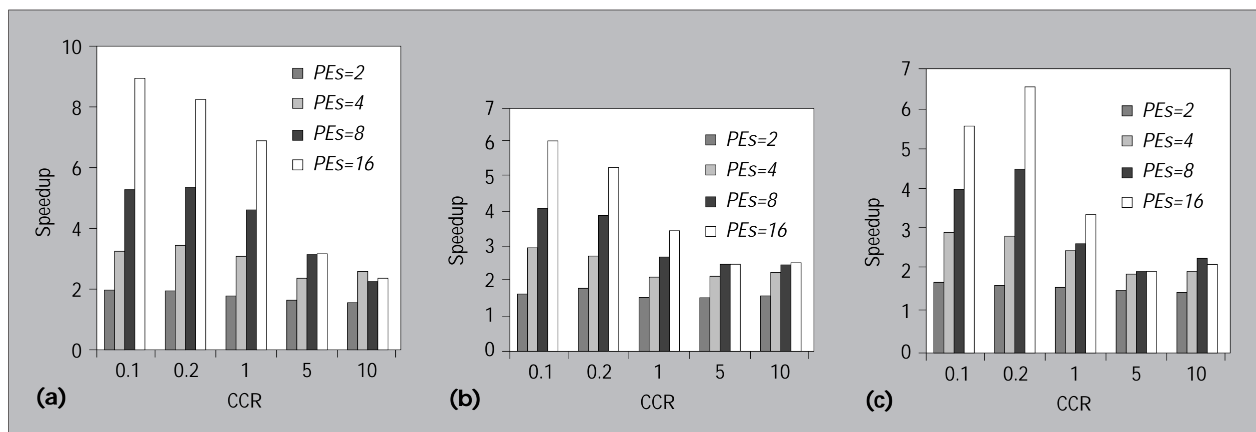


Figure 9. The parallel algorithm's average speedup for the (a) fully connected, (b) ring, and (c) chain processor topologies.

Call for Papers

Special Issue of *IEEE MultiMedia* on

Multimedia Telecommunications for National Information Infrastructures

for July–August 1999 issue

The era of building national information infrastructures (NII) has begun. This special issue will focus on topics and areas that document the experience provisioning multimedia telecommunications applications over the NII. Topics of interest comprise the following three main areas:

- **Case studies of the application of multimedia telecommunications in different sections of society (for example, government, healthcare, education, commerce, and entertainment), plus situational analyses, solution descriptions, and lessons from the field.**
- **Models and frameworks for the provisioning of NII services, including business models and regulatory and legal policies (multimedia content and transactions via telepresence are particularly encouraged).**
- **Review of technology assessment programs, international collaboration, and emerging standards that promote common understanding in multimedia telecommunications.**

We are also inviting related, but smaller works in progress, submissions to the Multimedia at Work and Project Reports departments as well as Standards and Media Reviews contributions.

Guest editors for this issue will be Ravi Sharma and Gurdeep Singh Hura. Submit eight hard copies or an electronic version of the article by **1 December 1998** to manuscript assistant Alkenia Winston, IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720. Complete author guidelines are available at http://computer.org/multimedia/edguide.htm.

Acknowledgments

We thank the anonymous referees whose comments helped improve this article. We also thank Vipin Kumar for useful discussions and suggestions. The Hong Kong Research Grants Council under contract numbers HKUST 734/96E and HKUST 6076/97E funded this research.

References

1. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Miller Freeman, San Francisco, 1979.
2. S.H. Bokhari, “On the Mapping Problem,” *IEEE Trans. Computers*, Vol. C-30, No. 3, Mar. 1981, pp. 207–214.
3. N.J. Nilson, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.
4. C.-C. Shen and W.-H. Tsai, “A Graph Matching Approach to Optimal Task Assignment in Distributed Computing System Using a Minimax Criterion,” *IEEE Trans. Computers*, Vol. C-34, No. 3, Mar. 1985, pp. 197–203.
5. S. Ramakrishnan, H. Chao, and L.A. Dunning, “A Close Look at Task Assignment in Distributed Systems,” *Proc. IEEE Infocom ’91*, IEEE Computer Society Press, Los Alamitos, Calif., 1991, pp. 806–812.
6. M. Kafil, *On Optimal Task Assignment Algorithms in Distributed Computing Systems*, master’s thesis, Dept. of Computer Science, Hong Kong Univ. of Science and Technology, 1996.
7. A. Grama and V. Kumar, “Parallel Search Algorithms for Discrete Optimization Problems,” *Operations Research Soc. America J. Computing*, Vol. 7, No. 4, Fall 1995, pp. 365–385.
8. V. Kumar, K. Ramesh, and V.N. Rao, “Parallel Best-First Search of State-Space Graphs: A Summary of Results,” *Proc. 1988 Nat’l Conf. Artificial Intelligence*, 1988, pp. 122–126.
9. N.S. Bowen, C.N. Nikolaou, and A. Ghafoor, “On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems,” *IEEE Trans. Computers*, Vol. 41, No. 3, Mar. 1992, pp. 197–203.

Muhammad Kafil’s research interests include parallel and distributed algorithms, operating systems, and video compression. He received his BSC in mathematics and physics from Bahauddin Zakariya University, Multan, Pakistan; his MS in computer science from De La Salle University, Manila; and a MPhil in computer science from the Hong Kong Univ. of Science and Technology. Contact him at the Dept. of Computer Science, Hong Kong Univ. of Science and Technology, Clear Water Bay, Kowloon, Hong Kong; kafeel@cs.ust.hk.

Ishfaq Ahmad is an associate professor in the Computer Science Department at the Hong Kong University of Science and Technology. His research interests include parallel-programming tools, scheduling and mapping algorithms for scalable architectures, multimedia systems, video compression, and distributed multimedia systems. He received his BSc in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan, and both his MS in computer engineering and his PhD in computer science from Syracuse University. He is a member of the IEEE Computer Society. Contact him at the Dept. of Computer Science, Hong Kong Univ. of Science and Technology, Clear Water Bay, Kowloon, Hong Kong; iahmad@cs.ust.hk; http://www.cs.ust.hk/faculty/iahmad/.

An earlier version of this work appeared as “A Parallel Algorithm for Optimal Task Assignment in Distributed Systems,” in *1997 Advances in Parallel and Distributed Computing Conf. (APDC ’97)* (IEEE Computer Society Press, Los Alamitos, Calif., 1997).