Optimal Task Assignment in Heterogeneous Computing Systems

Muhammad Kafil and Ishfaq Ahmad

Department of Computer Science
The Hong Kong University of Science and Technology, Hong Kong.

Abstract[1]

Distributed systems comprising networked heterogeneous workstations are now considered to be a viable choice for high-performance computing. For achieving a fast response time from such systems, an efficient assignment of the application tasks to the processors is imperative. The general assignment problem is known to be NP-hard, except in a few special cases with strict assumptions. While a large number of heuristic techniques have been suggested in the literature that can yield sub-optimal solutions in a reasonable amount of time, we aim to develop techniques for optimal solutions under relaxed assumptions. The basis of our research is a best-first search technique known as the A algorithm from the area of artificial intelligence. The original search technique guarantees an optimal solution but is not feasible for problems of practically large sizes due to its high time and space complexity. We propose a number of algorithms based around the A* technique. The proposed algorithms also yield optimal solutions but are considerably faster. The first algorithm solves the assignment problem by using parallel processing. Parallelizing the assignment algorithm is a natural way to lower the time complexity, and we believe our algorithm to be novel in this regard. The second algorithm is based on a clustering based pre-processing technique that merges the high affinity tasks. Clustering reduces the problem size, which in turn reduces the state-space for the assignment algorithm. We also propose three heuristics which do not guarantee optimal solutions but provide near-optimal solutions and are considerably faster. By using our parallel formulation, the proposed clustering technique and the heuristics can also be parallelized to further improve their time complexity.*

Keywords: Best-first search, parallel processing, task assignment. mapping, distributed systems.

1 Introduction

The fast progress of network technologies and sequential processors has made distributed computing systems, such as networks of heterogeneous workstations or PCs, an attractive alternative to massively parallel machines. To exploit the capabilities of these systems for an effective parallelism, the tasks of an application must be properly assigned to the processors.

Given a parallel program represented by a task graph and a network of processors also represented as a graph, the assignment problem is to find an allocation of the tasks to the processors that results in the minimum turnaround time. This is usually done by assigning an equal amount of load to all processors and by reducing the overhead of interaction among them. An assignment can be *static* or *dynamic*, depending upon on the time at which the allocation or assignment decisions are made. In a static assignment the information about the tasks and processors in the systems is assumed to be known in advance, and the tasks are allocated to the processors before starting the execution. The task assignment problem, also known as the *allocation problem* or the *mapping problem* [4], is well known to be NP-hard [6], but continues to be regarded as an interesting and important problem.

Most of the algorithms proposed in the past yield sub-optimal solutions while optimal algorithms exist only for restricted cases or small problem sizes. Optimal solutions, however, are required in many situations where performance is the primary goal. Also, once an optimal assignment of a program is determined, one can reuse this information for future mappings.

The simplest approach to finding an optimal solution is an exhaustive search. But since there are n^m ways for assigning m tasks to n processors, an exhaustive search is impractical. Another possibility is to reduce the size of the state-space using an *informed search*. The A* algorithm from the area of artificial intelligence is one such informed search algorithm. The algorithm, despite guaranteeing an optimal solution, is not feasible for problems of practically large sizes because of its high time and space complexity. Thus, we need ways to either further reduce the size of the state-space, or speedup the search process using parallel processing — or do both.

Since a parallel program is executed on multiple processors, it is natural to utilize the same processors to speedup the mapping of the program. Parallel processing can help in reducing the search time and allows to find optimal assignments for larger problem sizes, as compared to the serial algorithms. Even for a sub-optimal solution, parallel processing can help in solving a problem of larger size. However, very little work has been done on using parallel processing in solving the assignment problem; a few exceptions are the parallel heuristic for the scheduling problem proposed by Ahmad and Kwok [2] and the parallel heuristics for the assignment problem proposed by Bultan and Akyanat [5]. To the best of our knowledge, no prior work on finding an optimal assignment using parallel processing has been reported.

1. This research was supported by the Hong Kong Research Grants Council under contract number HKUST 619/94E.

We propose a parallel algorithm that generates an optimal solution for assigning an arbitrary task graph to an arbitrary network of heterogeneous processors. The algorithm, running on the Intel Paragon parallel machine, gives optimal assignments for small to medium size problems, with a reasonable speedup. We also propose a clustering based pre-processing algorithm that merges the high affinity tasks before starting the search. This reduces the problem size which in turn reduces the size of the state-space for the assignment algorithm. We also propose three heuristics which do not guarantee optimal solutions but yield near-optimal solutions and take considerably less execution time. The proposed heuristics and the clustering-based approach can also be parallelized using the proposed parallel formulation.

2 Problem Definition

A parallel program can be partitioned into a set of m communicating tasks 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 labelled by the communication costs between the vertices. The interconnection network of n processors, $\{p_1, p_2, .., p_n\}$, is represented by an $n*n$ matrix L, where an entry L_{ij} is 1 if the processors i and j are connected, and 0 otherwise.

A task t_i from the set V_T can be executed on any one of the n processors of the system. In a heterogeneous system [16], each task has an execution cost associated with it on a given processor. The execution costs of tasks are given by a matrix X, where the matrix entry X_{ip} is the execution cost of task i on processor p. When two tasks t_i and t_j executing on two different processors need to exchange data, a communication cost is incurred. Communication among the tasks is represented by a matrix C, where C_{ij} is the communication cost between task i and j if they reside on two different processors. The load on a processor is the combination of all the execution and communication costs associated with the tasks assigned to it. The total completion time of the entire program will be the time needed by the heaviest loaded processor.

Task assignment problem is to find a mapping of the set of m tasks to n processors such that the total completion time is minimized. The mapping or assignment of tasks to processors is given by a matrix A, where A_{ip} is 1 if task i is assigned to processor p and 0 otherwise. The load on a processor p is given by

$$\sum_{i=1}^{m} X_{ip} \bullet A_{ip} + \sum_{\substack{q=1 \\ (p \neq q)}}^{n} \sum_{i=1}^{m} \sum_{j=1}^{m} (C_{ij} A_{ip} A_{jq} L_{pq}) .$$

The first part of the equation represents the total execution cost of the tasks assigned to processor p, and the second part is the communication overhead on p. To find the processor with the heaviest load, the load on each of the n processors needs to be computed. The optimal assignment is the one that results in the minimum load on the heaviest loaded processor among all the assignments.

3 Related Work

A large number of task assignment algorithms have been proposed using various techniques such as network flow [17], integer programming [12], state-space search [14, 15, 18], clustering [3], bin-packing [19], randomized optimization [1, 5, 7, 8], etc. Most of these algorithms can be classified according to the taxonomy given in Figure 1. At the first level of the hierarchy these algorithms can be classified as *optimal* and *sub-optimal* categories, where the optimal algorithms can be further classified as *restricted* or *non-restricted* categories. Restricted algorithms yield optimal solutions in a polynomial time by restricting the structure of the program and/or the multicomputer system. Non-restricted algorithms, on the other hand, consider the problem in a more general context; they give optimal solutions but not necessarily in a polynomial time.

Sub-optimal algorithms can be divided into *approximate* or *heuristics* classes. Approximate algorithms [9] assume the same computational model used by the optimal algorithm. But instead of searching the complete solution space for optimal solution, approximate algorithms guarantee a solution that is within a certain range from the optimal solution. Heuristic algorithms make use of special parameters which affect the system in indirect ways, for example, clustering the groups of heavily communicating tasks together. A greedy heuristic starts from a partial assignment and assigns one task at each step until a complete assignment is obtained; in general, backtracking is not allowed. Bin-packing techniques use a sizing policy, an ordering policy, and a placement policy for the tasks to be assigned. Randomize optimization methods start from a complete assignment and search for an improvement in the assignment by exchanging and moving tasks among different processors.

Because of the intractable nature of the problem most of the research is focused on the development of heuristic algorithms. There are also some optimal algorithms available either for restricted cases of the problem or for very small problem sizes.

4 Overview of the A* Technique

The A* algorithm is a *best first* search algorithm [13]. It has been extensively used for problem solving in artificial intelligence. The algorithm is used to search efficiently in a search-space (which is a tree in our case but can be some other type of graph). It searches the nodes of the tree starting from the root called the *start node* (usually a null solution of the problem). Intermediate nodes represent the partial solutions while the leaf nodes represent the complete solutions or *goals*.

Associated with each node is a cost which is computed by a cost function f. The nodes are ordered for search according to this cost, that is, the node with the minimum cost is searched first. The value of f for a node n is computed as:

$f(n) = g(n) + h(n)$

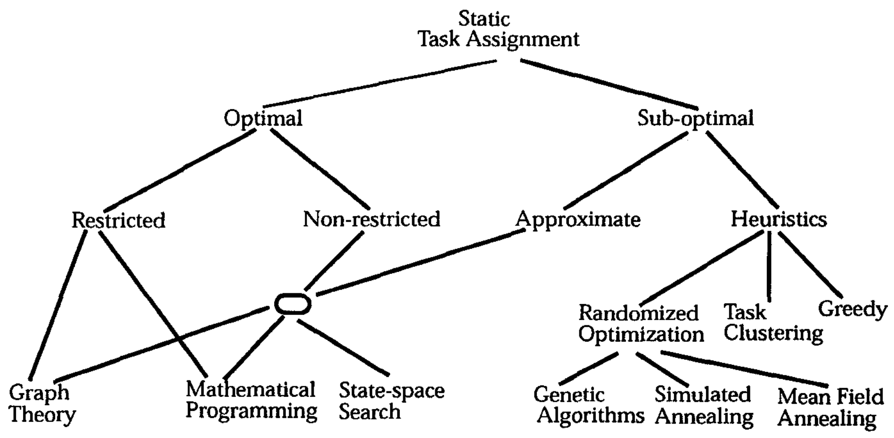


Figure 6: A classification of task assignment algorithms.

where *g(n)* is the cost of the search path from the start node to the current node *n*; *h(n)* is a lower bound estimate of the path cost from node *n* to the goal node (solution). Expansion of a node is to generate all of its successors or children and compute the *f* value for each of them. The algorithm maintains a sorted list, called *OPEN*, of nodes (according to their *f* values) and always selects a node with the best cost for expansion. Since the algorithm always selects the best cost node, it guarantees an optimal solution. Since for a leaf node *n, h(n)* is 0, we will set the value of *f(n)* equal to *g(n)* for all leaf nodes.

4.1 Application to Task Assignment

For the task assignment problem under consideration, the search space is a tree. The initial node (the root) is a node with null assignment, i.e., no task is assigned; intermediate nodes are nodes with partial assignments, i.e., some tasks are assigned while others are still unassigned at this stage. A solution (goal) node is a node with a complete assignment (all task are assigned). For the computation of the cost function, g(*n*) is the cost of partial assignment (A) at node *n*, that is, the load on the heaviest loaded processor. For the computation of *h(n)*, two sets T_p (the set of tasks which are assigned to the heaviest loaded processor *p*) and *U* (the set of tasks which are unassigned at this stage of the search and have a communication link with any task in set T_p) are defined. Now each task t_i in *U* will be assigned to either processor *p* or any other processor *q* which has a direct communication link with *p*. Thus, there can be two kinds of costs associated with the assignment of each t_i: $X_{i,p}$ (the execution cost of t_i on processor *p*) and the sum of communication cost with all the tasks in set T_p. Let cost (t_i) be the minimum of these two costs, then *h(n)* is computed as;

$$h(n) = \sum_{t_i \in U} cost(t_i)$$

The algorithm A* is described as follows:

The A* Algorithm

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

A study by Ramakrishnan *et al.* [14] showed that the order in which the tasks are considered for allocation has a great impact on the performance of the algorithm (for the same cost function used). Their study indicated that a significant performance improvement could be achieved by first considering the tasks with larger weights in the computation of the optimal cost at the shallow levels of the tree. They proposed a number of heuristics for ordering the tasks. Out of these heuristics the so called *minimax sequencing* heuristic has been shown to perform the best. The 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 *H* (*i*, *k*) of the matrix is given by

$$H(i,k) = X_{ik} + h(v),$$

where *h*(*v*) is given by

$$h(v) = \sum_{j \in U} min(X_{jk}, C_{ij}),$$

where *U* is the set of unassigned tasks which communicate with t_i. The minimax value, $mm\,(t_i)$ of task t_i is defined as

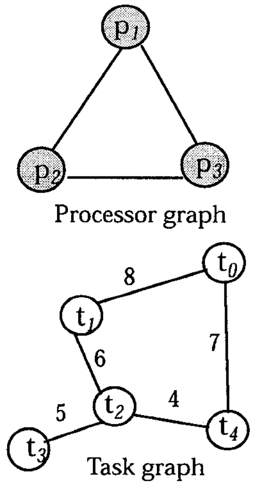
$$mm\,(t_i) = min\,\{H\,(i,k)\,,\,1 \le k \le n\}.$$

The minimax sequence is then defined as:

$$\Pi = \{\tau_1, \tau_2, \ldots, \tau_m\}\,,\,mm\,(\tau_i) \ge mm\,(\tau_{i+1})\,,\,\forall i.$$

4.2 An Illustrative Example

Given a set of 5 tasks, $\{t_0, t_1, t_2, t_3, t_4\}$ and a set of 3 processors $\{p_0, p_1, p_2\}$ as shown in Figure 2, the algorithm first generates the minimax sequence $\{t_0, t_1, t_2, t_4, t_3\}$.



	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

Execution cost matrix

Figure 2: An example task graph and a processor and the network, execution costs of the tasks on various processors.

Figure 2 illustrates the search tree for finding the assignment for this example.

A node in the search tree includes the partial assignment of tasks to processors as well as the value of f (the cost of partial assignment). The assignment of m tasks to n processors is indicated by an m digit string '$a_0a_1...a_{m-1}$', where a_i ($0 \le i \le m-1$) represents the processor (0 to $n-1$) to which ith task has been assigned. 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 replacing an 'X' value in the assignment string with some processor number. Node expansion is to add the assignment of a new task to the partial assignment. Thus the depth (d) of the search tree is equal to the number of tasks m, and any node of the tree can have a maximum of n (no of processors) successors.

The root node includes the set of all unassigned tasks 'XXXXX'. For example in Figure 2, the allocations of t_0 to p_0 ('0XXXX'), t_0 to p_1 ('1XXXX'), and t_0 to p_2 ('2XXXX') are considered by determining the costs of assignments at the first level of the tree. The assignment of t_0 to p_0 ('0XXXX') results in the total cost $f(n)$ being equal to 30. The $g(n)$ in this case equals 15 which is the cost of executing t_0 on p_0. The $h(n)$ in this case also equals 15 which is the sum of minimum of the execution or communication costs of t_1 and t_4 (tasks communicating with t_0). The costs of assigning t_0 to p_1 (26) and t_0 to p_2 (24) are calculated in a similar fashion. These three nodes are inserted to the list *OPEN*. Since 24 is the minimum cost, the node '2XXXX' is selected for expansion. The search continues until the node with the complete assignment ('20112') is selected for expansion

At this point since this is the node with a complete assignment and the minimum cost, it is the goal node. Notice that all assignment strings are unique. A total of 39 nodes are generated and 13 nodes are expanded. In comparison, an exhaustive search will generate $n^m = 243$ nodes in order to find the optimal solution.

5 The Proposed Algorithms

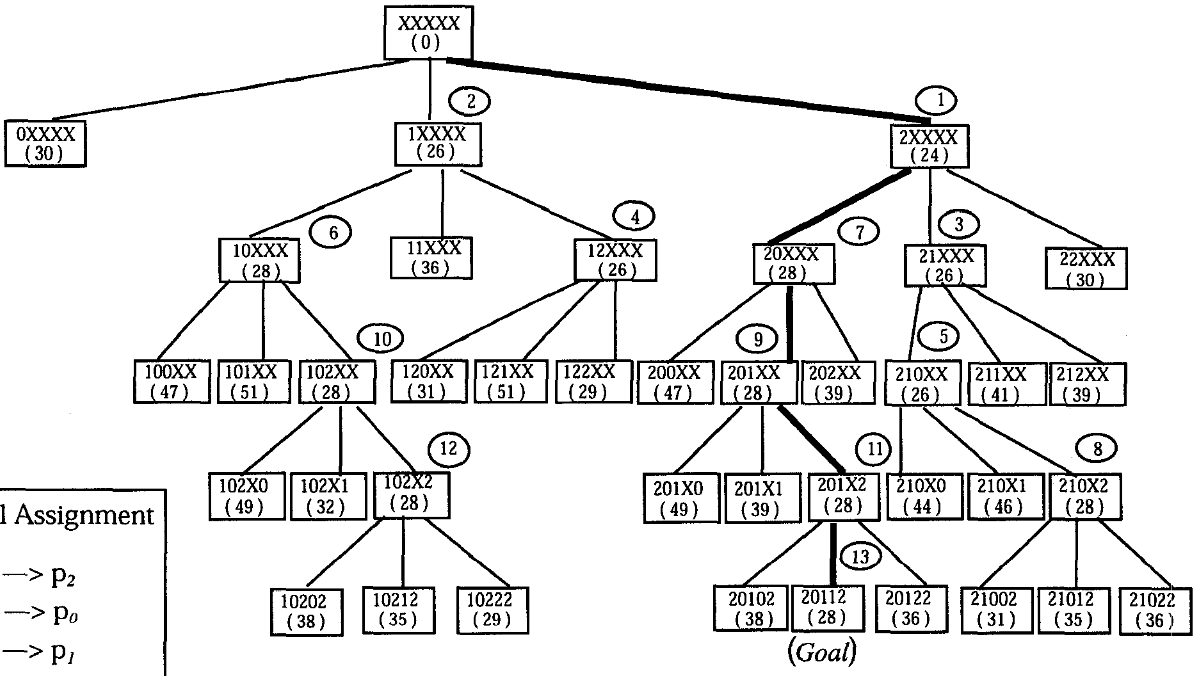
In this section, we describe our proposed parallel and clustering algorithms for optimal solutions. The sub-optimal algorithms are also explained in this section.

5.1 The Parallel Algorithm

The objective of the parallel algorithm is to divide the search tree among the processing elements (PEs) as evenly as possible and to avoid the expansions of non-essential nodes, that is, the nodes which are not expanded by the sequential algorithm. A good overview of parallel depth-first and best-first search algorithms are given in [10][11]. 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 (processing element which in our case is the Intel Paragon processor). We call this parallel algorithm the *Optimal Assignment with Parallel Search* (OAPS) algorithm.

The OAPS Algorithm:

```
(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 previously received Solutions)
(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)
```



Final Assignment

$t_0 \longrightarrow p_2$

$t_1 \longrightarrow p_0$

$t_2 \longrightarrow p_1$

$t_3 \longrightarrow p_1$

$t_4 \longrightarrow p_2$

Figure 3: The search tree for the example problem (nodes generated = 39, nodes expanded = 13).

Initially the search tree is divided statically based on the number of processing elements (PEs) P in the system and the maximum number of successors, S, of a node in the search tree. There could be three situations:

Case 1) $P < S$: Each PE will expand only the initial node which results in S new nodes. Each PE will get one node and additional nodes are distributed in a round robin (*RR*) fashion.

Case 2) $P = S$: Only the initial node will be expanded and each PE will get one node.

Case 3) $P > S$: Each PE will keep expanding nodes starting from the initial node (the null assignment) until the number of nodes in the list is greater than or equal to P. List is sorted in an increasing order of cost values of the nodes. The first node in the list will go to PE_1, the second node will go to PE_p, the third node goes to PE_2, the fourth node goes to PE_{p-1}, and so on. Extra nodes will be distributed in RR fashion. Although there is no guarantee that a best cost node at the initial levels of the tree will lead to a good cost node after some expansions, the algorithm still tries to distribute the good nodes as evenly as possible among all the PEs.

If a solution is found during the search, the algorithm terminates. Note that there is no master PE which is responsible for generating and distributing nodes among the PEs. Therefore, the overhead of the static node assignment is negligible as compared to the host-node style because the whole process is done in parallel. To illustrate this, we consider the example of the task assignment problem of assigning 10 tasks to 4 processors using 2 PEs (PE1 and PE2). Here S is 4 since a node in the search tree can have a maximum of 4 successors. Each PE, therefore, generates 4 nodes numbered from 1 to 4 (as shown in Figure 4 where the number in a box is the f value of the node). PE1 will then get the first and third node 3, while PE2 will get the second and fourth node.

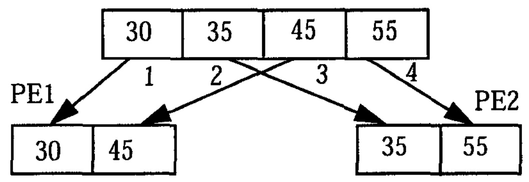


Figure 4: An initial static assignment.

If there is no communication among the PEs after the initial static assignment (i.e., every PE just searches its own tree), some of them may work on a good part of the search space, while others may expand unnecessary nodes (i.e., the nodes which the serial algorithm will not expand). This can result in a poor speedup. To avoid this, PEs need to communicate to share the best part of the search space and to avoid unnecessary work. This communication can be global (a PE broadcast its nodes to all other PEs) or local (a PE communicates only with its neighbors).

In our formulation we have used a round robin (RR) within neighborhood communication strategy. With this communication strategy a PE can share the best part of the

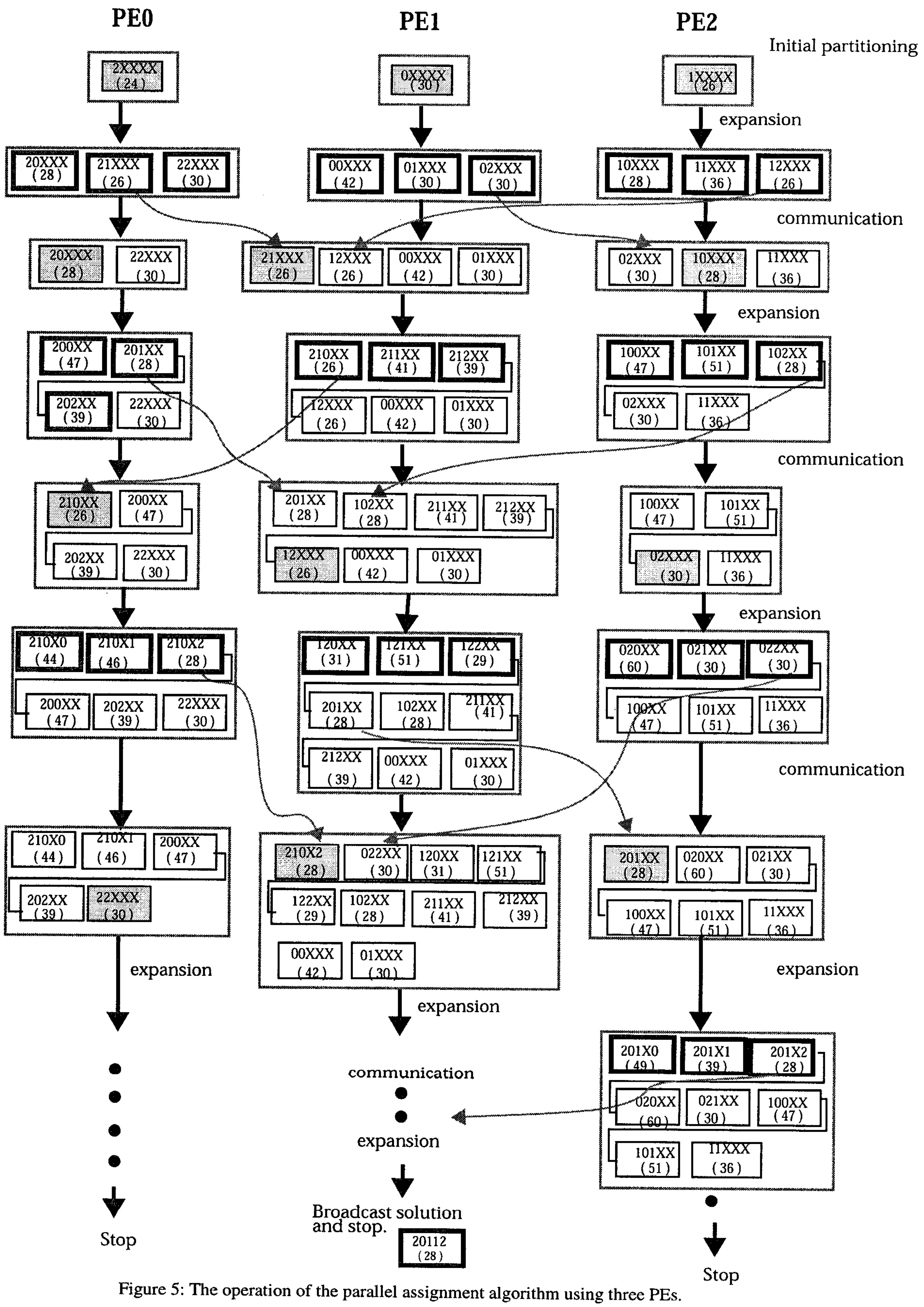


Figure 5: The operation of the parallel assignment algorithm using three PEs.

search space. Further, a PE can avoid unnecessary work explicitly by communicating with its neighbors and implicitly by broadcasting its solution to all other PEs. Since the Paragon PEs are connected together with a mesh topology, a PE can have a maximum of 4 neighbors. Since most of the time a PE communicates only with its neighbor, a low communication overhead is incurred making the algorithm more scalable as compared to a global communication strategy.

A PE periodically (when *OPEN* increases beyond a threshold *u*) selects a neighbor in a RR fashion and then sends its best node to that neighbor. As a result, the load is balanced and the best part of the search space is shared within the neighborhood of a PE. At finding a solution, a PE broadcasts it to all the PEs, thus helping in avoiding the 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 result to all other PEs, and from that point each PE broadcasts its solution only if its cost is better than a previously received solution.

With an initial partitioning, every PE has one or more nodes in its list *OPEN.* Each PE then determines the PEs in its neighbor by using its own position in the mesh (topology of the Intel Paragon). A PE starts expanding new nodes starting from the initial nodes. PEs then interact with each other for exchanging their best nodes and to broadcast their solutions. When a PE finds a solution, it records it in a common file (opened by all PEs) and stops. The optimal solution is the solution with the minimum costs among all PEs.

To illustrate the operation (see Figure 5) of the OAPS algorithm, we consider the example used earlier for the sequential assignment algorithm. Here we assume that the parallel algorithm runs on three PEs connected together as a linear chain. Initially three nodes are generated as in the sequential case. Then, through the initial partitioning, these nodes are assigned to the three PEs. Each PE then goes through a number of steps. In each step, there are two phases: the expansion phase and the communication phase[1]. In the expansion phase, a PE sequentially expands its nodes (the newly created nodes are shown with thick borders). It will keep on expanding until it reaches the threshold (*u*) (which is set to be 3 in this example). In the communication phase, a PE selects a neighbor and then sends its best cost node to it. The selection of the neighbors is done in a RR fashion. In Figure 5, the exchange of the best cost nodes among the neighbors is shown by dashed arrows. In the 5th step, PE1 finds its solution, broadcasts it to other PEs, and then stops. In the final step, PE0 also broadcasts (not shown here for the sake of simplicity) its solution to PE2 which finally records its solution and stops.

1. The synchronous operation of PEs shown here is just to illustrate the concept; the actual algorithm is fully asynchronous and thus may follow a different sequence — the final result will of course be the same.

5.2 The Preprocessing Clustering Algorithm

The algorithm starts by clustering (or merging) the tasks in the task graph. Two tasks are merged if the communication cost among them is so high that they will never be assigned to two different processors in the optimal assignment; Equations 5.1 and 5.2 given below ensure that the two tasks under consideration are never assigned to two different processors. Clustering reduces the size of the task graph and hence the depth (*d*) of the resulting search tree.

The algorithm first sorts the edges of the task graph, and then selects the largest edge (*i, j*), where task *i* and *j* are the tasks connected with the edge. The cost of an edge when mapped onto an edge of the processor graph is defined as the sum of the edge cost and the minimum execution cost of task *i* or *j* on the processors of the processor edge. The cost is computed using the following equation:

$$\min_{p,\,q = 1 \text{ to } n} \begin{pmatrix} \min\{ (X_{ip} + C_{ij}) \bullet L_{pq}, (X_{jq} + C_{ij}) \bullet L_{pq} \} \\ \min\{ (X_{jp} + C_{ij}) \bullet L_{pq}, (X_{iq} + C_{ij}) \bullet L_{pq} \} \end{pmatrix} \quad (5.1)$$

The cost of assigning tasks *i* and *j* to the same processor is the minimum execution cost of two tasks on either of the two processors of the processor edge. This cost is given by the following equation.

$$\min_{p,\,q = 1 \text{ to } n} \{ (X_{ip} + X_{jq}), (X_{iq} + X_{jq}) \} \quad (5.2)$$

A selected edge is merged if the cost of mapping it onto all of the processor edges is higher than the cost of assigning the two tasks on the same processor. The clustering process is repeated for all the edges of the processor graph.

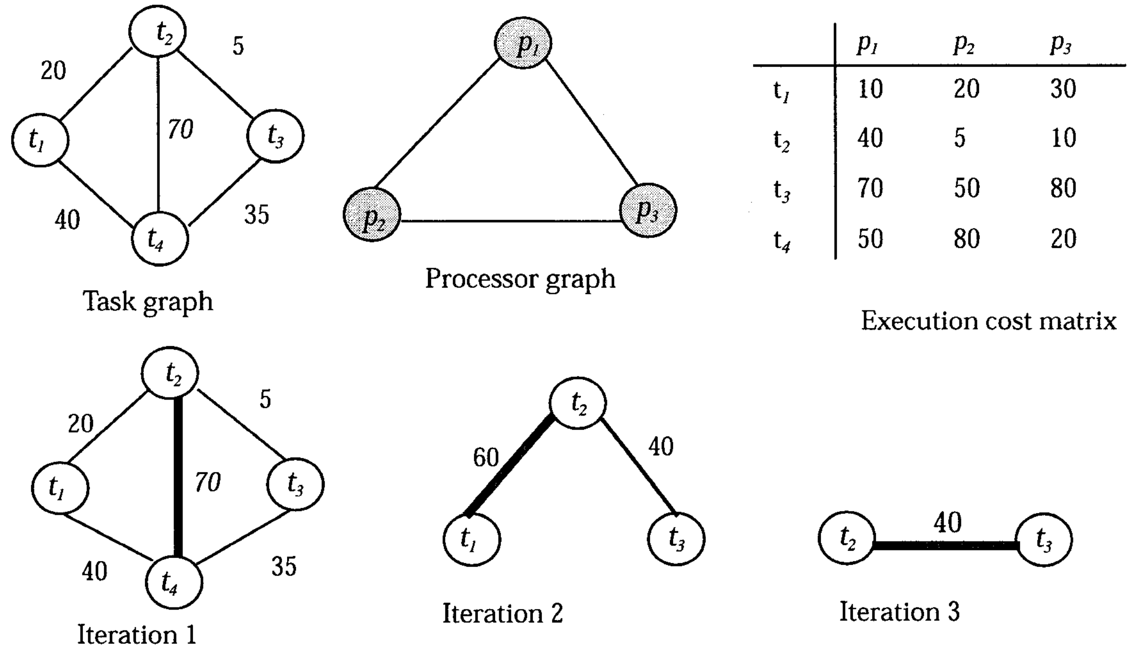
The clustering process is illustrated by an example, given in Figure 6, where the largest edges selected are shown as thick edges. In the first iteration the edge (t_2, t_4) is selected and task t_4 is merged with t_2 and its communication links with other tasks are added to t_2. In the second iteration t_1 is merged. In the third iteration, the selected edge is not merged, and the algorithm stops.

After clustering, the tasks are reordered using the minimax sequencing as discussed in Section 4.1. Now the tasks are selected for the assignment using this sequence.

The clustering procedure guarantees an optimal assignment only when the processors are fully-connected since the searching algorithm assigns two communicating tasks only to the directly connected processors.

5.3 Sub-optimal Algorithm

The sub-optimal algorithm, henceforth referred to as the *Sub-Optimal Assignments* (SA) algorithm, is designed to obtain the solution faster and to overcome the high memory requirements of A*. The basic idea in this algorithm is that when the search process reaches a certain level deep in the search tree, some search can be avoided (some tree nodes can be discarded) without moving far from the optimal solution. Based on this reasoning, we



	p_1	p_2	p_3
t_1	10	20	30
t_2	40	5	10
t_3	70	50	80
t_4	50	80	20

Figure 6: Illustration of the clustering procedure.

propose three heuristics, SA1, SA2 and SA3. The first heuristic (SA1) is explained as follows. When the algorithm selects a node for expansion and that node belongs to a level *R* or deeper than that in the search tree, it generates only its best successor instead of generating all the successors (i.e., it discards all successors except the best one). The second heuristic (SA2) is similar to the first: when the search reaches at level *R* for the first time, the algorithm starts discarding all successors except the best node among all the nodes selected for expansion. The third heuristic (SA3) is similar to the second heuristic except the nodes are discarded from the global list (*OPEN*). For example, if *n* nodes are generated, then all of them are inserted to *OPEN* and $n - 1$ high cost nodes are discarded.

There is a little chance of running out of memory for the above mentioned heuristics. This is because when a node at level *R* is selected, the algorithm inserts only one node to *OPEN* for expansion and takes one node from it. Thus, no extra memory is required. Moreover, the running time of the algorithm is reduced by a large factor since the algorithm explores fewer nodes once it reaches the level *R*.

6 Experimental Results

We first discuss the workload used in our study and then present the experimental results obtained by the proposed algorithms.

6.1 Workload Generation

A realistic workload is important to validate an assignment algorithm but very little information is available about process communication patterns encountered in distributed systems. In distributed systems, there is usually a number of process groups with heavy interaction within the group, and almost no interaction with the processes outside the group [3]. With this intuition, we first generated a number of primitive task graph structures such as the pipeline, the ring, the server, and the interference graphs, all consisting of 2 to 8 nodes. The complete task graphs, consisting of 10-28 nodes, were generated by randomly selecting these primitives structures and combining them until the desired number of tasks was reached. This was done by first selecting a primitive graph and then combining it with a newly selected graph through a link labelled with cost 1; the last node was connected back to the first node.

Since we assume the processors to be heterogeneous (a homogeneous processor system is a special case of a heterogeneous processor system), the execution cost varies from processor to processor in the execution cost matrix (X); the average value, however, remains the same. To generate the execution costs for the nodes and the communication costs for the edges, we used a parameter called the *communication-to-cost ratio* (CCR) which is the value of the average computation cost divide by the average communication cost per node. For example, if the total communication cost (sum of the cost of all of the edges connected to this task) of task *i* is equal to 16.0 and the CCR is equal to 0.2, then the average execution cost of *i* will be given by: 16.0 /0.2 = 80. We used the following values of CCR: 0.1, 0.2, 1.0, 5.0, and 10.0.

For the processor graphs, we used 3 topologies each comprising 4 nodes. For the parallel algorithm OAPS, we used 2, 4, 8, and 16 Paragon PEs.

6.2 Running Times of the Serial Algorithm

In this section we present the running times of various versions of the serial assignment algorithm. Table 1 and 2 include the running times for different variations of the serial algorithm for the fully-connected topology comprising 4 processors. The running times of the serial algorithm without any task ordering or clustering are given in column 2; we will refer to it as A* in these tables. An entry '**' in a column means the algorithm could not generate the solution for this case using 50 MB of memory, i.e., it ran out of memory after a few hours (usually 5 to 6 hours). The third column shows the running times of the algorithm with the task ordering; we will refer to this technique as A*R. The fourth column shows the running times of the algorithm with clustering and then ordering; we will refer to this technique as A*C. The fifth column is the ratio of the running times of the two algorithms.

For the fully-connected topology of 4 processors and with CCR equal to 1.0 (see Table 1), the clustering algorithm is on the average 3.95 times faster than A*R. Table 2 presents the running times for the same topology but with CCR equal to 5.0. The clustering algorithm is on the average 281 times faster. The clustering algorithm performs well when the value of CCR is high because for these cases the optimal algorithm also assigns highly communicating tasks to the same processor. For lower values of CCR the algorithm does no merging for most of the cases.

It is observed that for most of the cases, task graphs with CCR equal to 0.1 and 0.2 result in larger search trees as compared to the graphs with CCR equal to 1.0, 5.0, and 10. The task graphs with CCR equal to 10.0 take the lowest running times. This is because the cost of the optimal solution for a higher CCR is less than a lower CCR and thus the algorithm finds the optimal solution quickly starting from an initial cost 0. For example, a task graph consisting of 10 tasks with the CCR equal to 10.0 has the solution cost equal to 7.36, while the same graph with the CCR equal to 0.1 has the solution cost equal to 374.00. Thus, the former takes only 0.40 seconds to find the solution while the latter takes 4.30 seconds.

The processor topology also has a great impact on the size of the search tree as well as on the running time. This is because the algorithm assigns two communicating task to two different processors only if the processors are directly connected. So, in case of the line or ring topology, the algorithm prunes some of the nodes in the search tree based on this constraint. On the other hand, no such pruning is done for the fully-connected case.

6.3 Speedup Using the Parallel Algorithm

In this section, we present the speedup of the parallel algorithm using various number of processors. The speedup is defined as the running time of the serial algorithm over the running time of the parallel algorithm.

Table 3 presents the speedup data for the fully-connected topology comprising 4 processors and the task graphs with CCR equal to 0.1. The second column includes the running time of the serial algorithm while the third, fourth, fifth, and sixth columns include the speedup of the parallel algorithm over the serial algorithm using 2, 4, 8 and 16 Paragon PEs, respectively. The bottom row of the table indicates the average speedup of all the task graphs.

We can observe that the speedup increases with an increase in the problem size. Also the problems with a lower value of CCR yield a better speedup in most of the cases, since the running times of the serial algorithm in those cases are much longer compared to the parallel algorithm.

Table 1: The running times using the fully-connected topology (CCR = 1.0)

No. of Tasks	T(A*) (sec)	T(A*R) (sec)	T(A*C) (sec)	$\frac{T(A^*R)}{T(A^*C)}$
10	3.35	0.87	0.17	5.12
12	139.54	0.73	0.77	0.95
14	270.70	4.82	3.77	1.28
16	822.08	36.08	1.67	21.60
18	**	31.62	30.76	1.03
20	**	55.78	22.19	2.51
22	**	67.70	67.78	1.00
24	**	191.27	55.21	3.46
26	**	206.63	143.06	1.44
28	**	2451.56	2124.08	1.15
Avg				3.95

Table 2: The running times using the fully-connected topology (CCR=5.0).

No. of Tasks	T(A*) (sec)	T(A*R) (sec)	T(A*C) (sec)	$\frac{T(A^*R)}{T(A^*C)}$
10	0.24	0.27	0.08	3.37
12	1.53	0.49	0.12	4.08
14	35.98	1.73	0.25	6.92
16	10.29	1.67	0.27	6.19
18	6195.63	29.79	0.55	54.16
20	**	21.96	1.14	19.26
22	**	3.98	3.18	1.25
24	**	3387.58	4.15	816.28
26	**	4134.28	2.19	1887.80
28	**	52.86	3.87	13.66
Avg				281.30

The values of the average speedup for the fully-connected, ring, and line topologies are shown graphically in Figure 7.

6.4 Results of the Heuristics

In this section, we present the result of comparing the three proposed heuristics (SA1, SA2, SA3) with the optimal algorithm. We make two kinds of comparisons. First, we compare the percentage deviation of the solution produced by SA1, SA2 and SA3 from that of OASS. This deviation is defined as follows:

%D = (Cost(SA) - Cost(OASS) * 100) / Cost(OASS)

Second, we compare ratios of the running times of SA1, SA2 and SA3 to those of OASS. Optimal solutions are first obtained for the five task sets discussed in Section 5.2 and then sub-optimal solution are obtained using SA1, SA2 and SA3 for the same task sets. Heuristic tree level used is:

$$R = \left\lfloor \frac{d}{3} \right\rfloor,$$

where d is the maximum depth of the search tree. Table 4 presents the results for the ring topology with 4 processors and the task graphs with CCR equal to 0.2. Each entry in the table is the average of five runs of each algorithm for 5 task graphs generated using various permutations of the pipeline, the ring, the server and the interference sub-graphs. The average values of the percentage deviation in the solution and the ratios of the running times are indicated in the bottom row.

The results indicate that SA3 always gives good solutions in terms of the percentage cost deviation from the optimal. This is because SA3 discards high cost nodes from the global list *OPEN,* so good nodes are always prevented from deletion. SA2 deviates more than SA3 but is faster.

The average cost deviation and the ratio of time improvement for the fully-connected topology (with different values of CCR) is shown in Figure 8. It can be noted that the average percentage cost deviation for the cases with CCR equal to 5.0 and 10.0 is quite high as compared to the cases with lower values of CCR. This is because when the task graph has a larger value of CCR the optimal algorithm assigns more tasks to a single processor (for some cases all the tasks goes to one processor). Therefore, the optimal algorithm follows a rather straight path in the search tree considering less options. If the sub-optimal algorithm discards a node on this path, it will deviate far from the optimal.

The availability of the optimal algorithm, sub-optimal heuristics, and the parallel algorithm gives a choice to the user to select a suitable algorithm depending upon the objective. If the objective is to find a solution in a short time, then SA2 can be used. To obtain a near-optimal assignments for a task graphs with higher values of CCR, SA3 can be used. If finding the optimal solution is the main objective without any regard to the algorithm running time, then the sequential A* can be used. If the resources, such as a parallel machine, are available, then OAPS can be used to speedup the running time of the optimal algorithm.

7 Conclusions and Future Work

We proposed algorithms for optimal and sub-optimal assignments of tasks to processors. We considered the problem under relaxed assumptions such as an arbitrary task graph with arbitrary costs on the nodes and edges of the graph, and processors connected through an interconnection network. Our algorithms can be used for homogeneous as well as heterogeneous processors, although in this paper we considered only the heterogeneous cases. We believe that to the best of our knowledge, ours is the first attempt in proposing a parallel algorithm for the optimal task-to-processor assignment problem. Although we kept the mapping of the algorithm on the Paragon PEs simple, some fine refinements are possible to further improve the performance.

A further study is required to understand the behavior of the parallel algorithm. One possibility is to implement quantitative load balancing of the tree nodes after a processor finds its solution, i.e., let the processor find more than one solution. Also, additional experimentation is required to find the ideal value of the threshold u. The clustering algorithm and the sub-optimal heuristic SA3 may be combined in order to obtain faster and close-to-optimal assignments for task graphs with high values of CCR. Our future plans also include a parallelization and analysis of the heuristic algorithms (for an ideal tree level R) to start applying the heuristics would also require more future works.

References

[1] I. Ahmad and M. K. Dhodhi, "Task Assignment using Problem-Space Genetic Algorithm," *Concurrency: Practice and Experience*, vol. 7, no. 5. pp. 411-428, August 1995.

[2] I. Ahmad and Yu-Kwong Kwok, "A Parallel Approach to Multiprocessor Scheduling," *International Parallel Processing Symposium*, Santa Barbra, CA, April 1995, pp. 289-293.

[3] N. S. Bowen, C. N. Nikolaou, and A. Ghafoor, "On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems," *IEEE Trans on Computers*, vol. 41, no. 3, pp. 197-203, March 1992.

[4] S. H. Bokhari, ''On the Mapping Problem,'' *IEEE Trans. on Computers* vol. c–30, March 1981, pp. 207–214.

[5] T. Bultan and C. Aykanat, "A New Heuristic Based on Mean Field Annealing," *Journal of Parallel and Distributed Computing,* vol. 16, no. 4, pp. 292-305, Dec 1992.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, (Freeman, San Francisco, CA, 1979).

[7] D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning," (Addison, Wesely, Reading, MA 1989)

[8] S. M. Hart and Chuen-Lung S. Chen, "Simulated Annealing and the Mapping Problem: A

Computational Study," *Computers and Operations Research,* vol. 21, no. 4, pp 455-461, 1994.

[9] M. A. Iqbal and S. H. Bokhari, "Efficient Algorithms for a Class of Partitioning Problems," *IEEE Trans. on Parallel and Distributed Systems,* vol. 6, no. 2, Feb. 1995.

[10] A. Grama and Vipin Kumar, "Parallel Search Algorithms for Discrete Optimization Problems," *ORSA Journal on Computing,* vol.7, no.4 (Fall 1995) pp 365-385.

[11] V. Kumar, K. Ramesh, and V. Nageshwara Rao. "Parallel best-first search of state-space graphs: A summary of results," *Proceedings of the 1988 National Conference on Artificial Intelligence*, pp. 122-126, Aug. 1988.

[12] P.-Yio R. Ma, E. Y. S Lee, "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. on Computers*, vol. c-31, no. 1, Jan. 1982.

[13] N. J. Nilson, *Problem Solving Methods in Artificial Intelligence.* New York: McGraw-Hill, 1971.

[14] S. Ramakrishnan, H. Chao, and L.A. Dunning, "A Close Look at Task Assignment in Distributed Systems," *IEEE INFOCOM '91*, pp. 806-812, 1991.

[15] C.-Ch. Shen and W.-H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing System Using a Minimax Criterion," *IEEE Trans. on Computers*, vol. c-34, no. 3, pp. 197-203, March 1985.

[16] H. J Siegel, J. K. Antonio, R. C. Metzger, Min Tan, and Yan A Li, "Heterogeneous Computing", Parallel and Distributed Computing Handbook, pp. 725-761, McGraw-Hill, New York.

[17] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering*, SE-3, vol. 1, pp. 85-93, Jan. 1977.

[18] J. B. Sinclair, "Efficient Computation of Optimal Assignments for Distributed Tasks," *Journal of Parallel and Distributed Computing*, vol. 4, 1987, pp. 342-362.

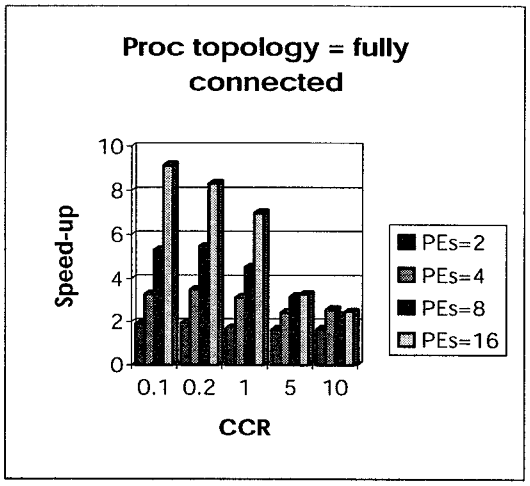
[19] C. Woodside and G. Monforton, "Fast Allocation of Processes in Distributed and Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 4, no. 2, Feb. 1993.

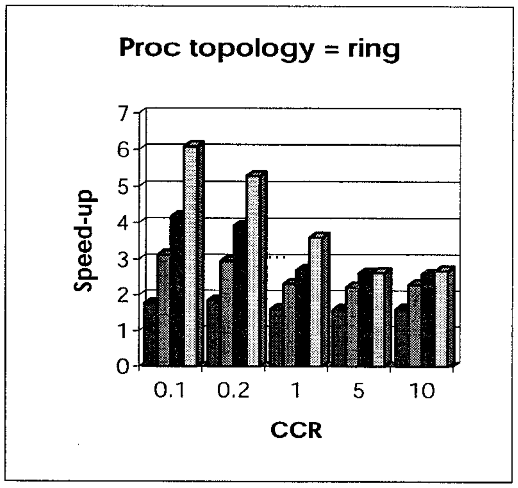
Table 3: The speedup using the fully-connected topology (CCR=0.1).

No. of Tasks	T(A*R) (sec)	$\frac{T(A*R)}{T(OPAS)}$			
		PEs=2	PEs=4	PEs=8	PEs=16
10	30.14	1.87	3.48	5.72	7.63
12	58.96	1.96	3.68	3.60	12.85
14	105.05	1.70	2.02	4.58	4.64
16	1550.46	2.00	2.94	4.72	6.71
18	3839.00	2.00	3.86	7.59	13.16
20	3191.86	1.78	3.72	5.62	9.97
Avg		1.89	3.28	5.30	9.13

Table 4: The time and cost comparison using the ring topology (CCR=0.2).

No. of Tasks	$\frac{C(SA1) - C(OASS) *100}{C(OASS)}$	$\frac{C(SA3) - C(OASS) *100}{C(OASS}$	$\frac{C(SA3) - C(OASS) *100}{C(OASS}$	$\frac{T(O)}{T(SA1)}$	$\frac{T(O)}{T(SA2)}$	$\frac{T(O)}{T(SA3)}$
10	7.20	8.82	0.00	1.72	2.68	1.95
12	1.96	2.24	1.49	1.54	3.18	1.90
14	4.08	4.21	0.55	1.75	3.94	2.48
16	2.68	3.41	0.55	4.17	8.65	4.24
18	1.86	2.07	1.15	3.53	7.22	2.81
20	6.04	6.31	2.30	2.35	5.11	2.91
22	2.81	4.15	3.27	7.13	37.97	22.59
24	1.53	2.51	0.94	6.19	25.89	10.19
26	3.52	4.39	3.88	15.72	108.75	40.81
Avg	3.52	3.53	1.67	4.90	22.60	9.99





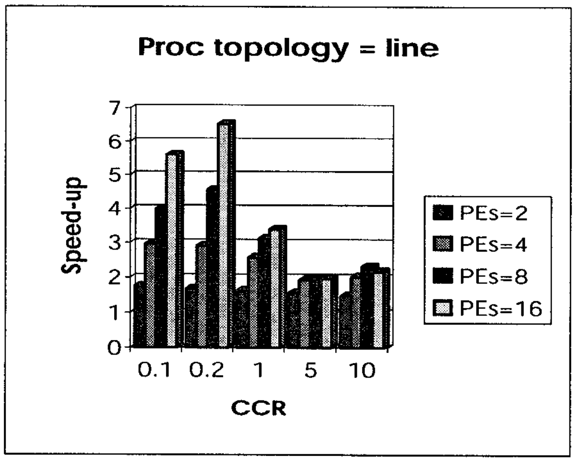
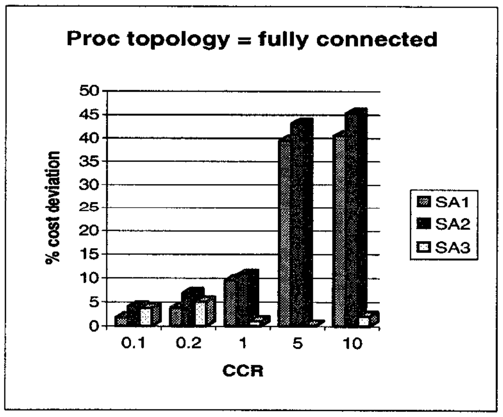


Figure 7: The average speedup of the parallel algorithm.



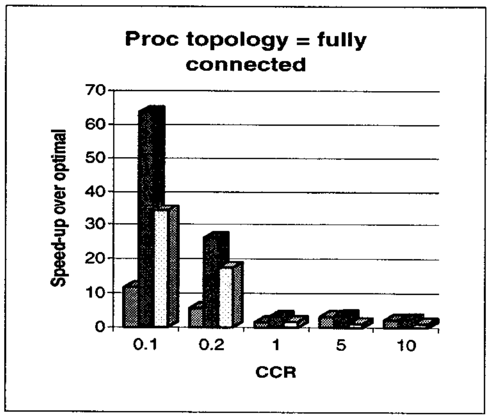


Figure 8: The percentage cost deviation and speedup of the sub-optimal algorithms over the optimal algorithm.