A Parallel Algorithm for Optimal Task Assignment in Distributed Systems

Ishfaq Ahmad and Muhammad Kafil

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

Abstract[1]

An efficient assignment of tasks to the processors is imperative for achieving a fast job turnaround time in a parallel or distributed enviornment. The assignment problem is well known to be NP-complete, except in a few special cases. Thus heuristics are used to obtain suboptimal solutions in reasonable amount of time. While a plethora of such heuristics have been documented in the literature, in this paper we aim to develop techniques for finding optimal solutions under the most relaxed assumptions. We propose a best-first search based parallel algorithm that generates optimal solution for assigning an arbitrary task graph to an arbitrary network of homogeneous or heterogeneous processors. The parallel algorithm running on the Intel Paragon gives optimal assignments for problems of medium to large sizes. We believe our algorithms to be novel in solving an indispensable problem in parallel and distributed computing.

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

1 Introduction

Given a parallel program represented by a task graph and a network of processors also represented as a graph, the problem of assigning tasks to processors is known as the *allocation problem* or *mapping problem* [1]. In this problem, the assignment of tasks to processors is done in a static fashion, and the main goal is to assign equal amount of load to all the processors and reduce the overhead of interaction among them. The problem, however, is a well known to be NP-hard [5], and has been a focus of attention due to its importance and complexity. A considerable amount of research effort using a variety of techniques has been reported in the literature.

The assignment problem is an optimization problem, and the cost function to be optimized depends on the assumptions made about the application model and hardware details (system topology, communication bandwidth, and the possibility of overlapping different functions). The system may be a parallel machine or a network of workstations connected as a virtual parallel machine (such as in the computing model of PVM or MPI). A heterogeneous system (i.e., the processors are of different speeds) adds more constraints to the assignment problem. In general, without making strict assumptions, the assignment algorithm can be computationally very extensive.

The approaches used to solve the task assignment problem for optimal or suboptimal solutions can be classified as graph theoretic [1] [14], simulated annealing [8], genetic techniques [6], [7], solution space enumeration and search [13] [15], mathematical programming [11], and heuristics [3], [9]. Most of the reported solutions are based on heurtistics, and optimal solutions exist only for restricted cases or small problem sizes.

The contribution of this paper is a best-first search-based parallel algorithm that generates optimal solution for assigning an arbitrary task graph to an arbitrary network of homogeneous or heterogeneous processors. The algorithm running on the Intel Paragon gives optimal mappings with a good speed-up.

The rest of this paper is organized as follows. In the next section, we define the assignment problem. In Section 3, we give an overview of the A* search technique which is the basis of our proposed algorithm. In Section 4, we present the proposed algorithm. Section 5 includes the experimental results, and the last section concludes the paper.

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. Each task has an execution cost associated with it on a given processor. The execution costs of the tasks are given by a matrix X; where 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 communications cost will be incurred.

The communication among tthe asks 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 a combination of all the execution

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

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. 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

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

The first part of the equation is the total execution cost of the tasks assigned to processor p, and second part is the communication overhead on p.

In order to find the processor with the heaviest load, the load on each of the n processors needs to be computed. The optimal assignment now will be the one which results in the minimum load on the heaviest loaded processor among all the assignments. There are n^m possible assignments, and finding the optimal assignment is known to be an NP-hard problem [5].

3 Overview of the A* Technique

A* is a *best first* search algorithm [10] which has been used to solve optimization problems in artificial intelligence as well as other areas. The algorithm constructs the problem as a search tree. It then searches the nodes of the tree starting from the root called the *start node* (usually a null solution). Intermediate nodes represent the partial solutions while the complete solutions or goals are represented by the leaf nodes. 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, a node with the minimum cost is searched first. The value of f for a node n is computed as $f(n) = g(n) + h(n)$ 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).

The algorithm maintains a sorted list of nodes (according to the f values of the nodes) and always selects a node with the best cost for expansion. Expansion of a node is to generate all of its successors or children. Since the algorithm always selects the best cost node, it guarantees an optimal solution.

3.1 The Sequential Approach

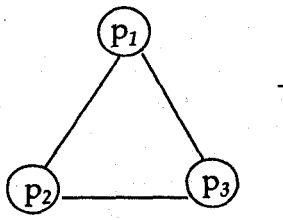
In this study we will use the A* technique for assignment problem and call it the *Optimal Assignment with Sequential Search* (OASS) algorithm. The OASS algorithm is described as follows:

(1) Build initial node s and insert it into the list *OPEN*
(2) Set $f(s) = 0$
(3) **Repeat**
(4) Select the node n with smallest f value.
(5) **if** (n <> 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 = Solution)
(15) Report the solution and Stop
(16)**Until** (n is a Solution) **or** (OPEN is empty)

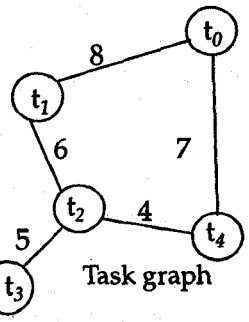
Shen and Tsai [13] formulated a state-space search algorithm for optimal assignments. In this formulation each node in the search tree represents an partial assignment and a goal node represents a complete assignment. The algorithm A* is then applied to traverse the search space.

A subsequent study by Ramakrishnan [13] showed that the order in which tasks are considered for allocation has a great impact on the performance of the algorithm (for the cost function used)Their study indicated that a significant performance improvement can be achieved by a careful ordering of tasks. They proposed a number of heuristics out of which the so called *minimax sequencing* heuristic has been shown to perform the best.

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 1, the resulting



Processor graph



Task graph

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

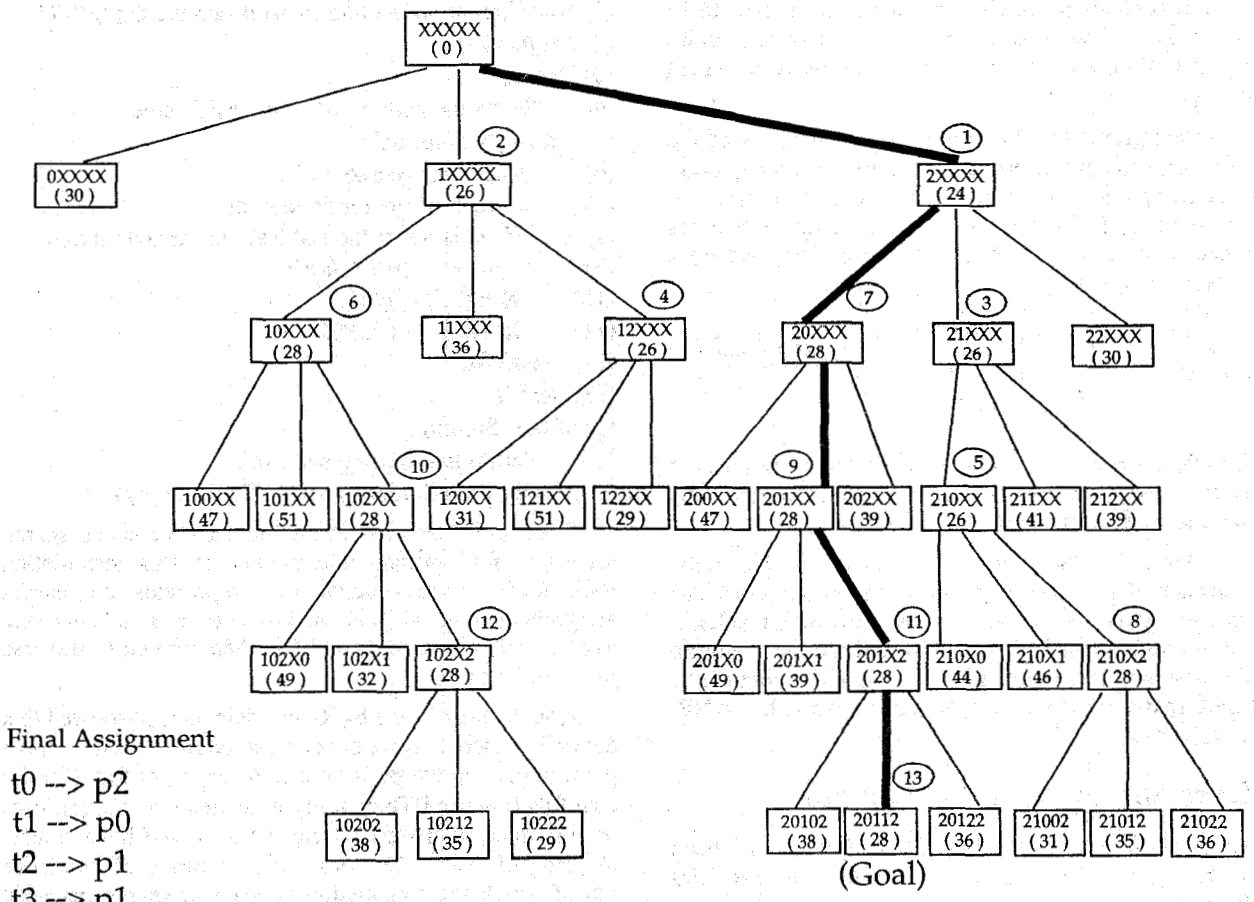


Figure 2: Search tree for the example problem.
(nodes generated = 39, nodes expanded = 13).

search trees is given in Figure 2. A node in the search tree includes the partial assignment of tasks to processors, and the value of f (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. Theses 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 costs, 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, exhaustive search will generate $n^m = 243$ nodes in order to find the optimal solution. The minimax sequence generated is {t_0, t_1, t_2, t_4, t_3}. therefore, t_4 has been considered before t_3.

4 The Parallelizing Approach

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 our parallel algorithm the *Optimal Assignment with Parallel Search* (OAPS) algorithm. First we describe the initial partitioning and dynamic load balancing strategies.

4.1 Initial Partitioning

Initially the search space 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* get additional nodes in Round Robin (*RR*) fashion. Case 2) P = S: Only 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 until the number of nodes in the list are greater than or equal to P. List is sorted in increasing order of cost values of the nodes. First node in the list will go to PE_1, second node will go to PE_p, third node goes to PE2, 4th node goes to PE_{p-1} and so on. Extra nodes will be distributed using RR. (Although there is no guarantee that a best cost node will lead to good cost node after some expansions but still algorithm tries to initially distribute the good nodes as evenly as possible among the PEs). If a solution is found during this process, the algorithm will terminate. Note that there is no master *PE* which generates first the nodes and then distribute among other *PEs*.

4.2 Dynamic Load Balancing

If there is no communication between the *PEs* after the initial static assignment, some of them may work on good part of the search space, while others may expand unnecessary nodes (the nodes which the serial algorithm will not expand). This will result in a poor speed up. To avoid this, *PEs* need to communicate to share the best part of the search space and to avoid unneccessary work. In our formulation, a *PE* achieves this explicitly using a round robin (RR) communication strategy within its neighbourhood, and implicitly by broadcasting its solution to all *PEs*.

In steps 13-16 of the algorithm, a *PE* periodically (when OPEN increases by a threshold *u*) selects a neighbour in *RR* fashion and then sends its best node to that neighbour. This will achieve the sharing of best part of the search space within neighbourhood. Aside from this load balancing, a *PE* also broadcast its solution (when it finds one) to all *PEs*. This will help in avoiding the unnecessary work for a *PE* which is working on the bad part of the search space. Since once a node receives a better cost solution than its current best node, it will stop expanding the unnecessary nodes. A solution is broadcasted only if its cost is better than an earlier solution received from any other PE. The OAPS algorithm is described below:

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)
```

Given an initial partitioning, every PE first sets up its neighbourhood to find out which PEs are in its neighbor. Some initial nodes are generated for each PE and then starting from initial nodes every PE will run some iterations of sequential A*. 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 into a common file opened by all PEs. A PE which finds the solution does not expand nodes any further and waits to receive the solution from other PEs. Finally, the best solution is the solution with the minimum costs among all PEs.

To illustrate the operation of the OAPS algorithm, we use the same example that was used earlier for the sequential assignment algorithm. This operation is shown in Figure 3. Here we assume that the parallel algorithm runs on three PEs connected together as a linear chain, i.e, PE_0 and PE_2 have one neighbour PE_1, while PE_1 has two neighbours since it is in the middle. First, three nodes are generated as in the sequential case. Then through the initial partitioning, these nodes are assigned to 3 PEs. Each PE then goes through a number of steps. In each step, there are two phases: the expansion phase and the communication phase. 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*) — this is set to be 3 in this example. In the communication phase, a PE selects a neighbour and then sends its best cost node to it. The selection of neighbours is in a RR fashion. In the example, the exchange of the best cost nodes among the neighbours is shown by the 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 stop.

5 Experimental Results

For experiments we used task graphs with 10-28 nodes with 5 different values of (communication-to-computation ratio) CCR and processor graphs of 4 nodes connected in 3 different topologies. For the OAPS algorithm, we used 2, 4, 8, and 16 Paragon PEs.

5.1 Workload Generation

To test the proposed sequential and parallel algorithms, we generated a library of task graphs as well as 3 processor topologies. In distributed systems, there is usually a number of process groups with heavy interaction within the group, and almost no interaction with processes outside the group [2]. So, using this intuition, first we generated a number of primitive task graph structures such as pipeline, ring, server, and interference graphs of 2 to 8 nodes.

Complete task graphs were generated by randomly selecting these primitives structures and combining them until the desired number of tasks are reached. This is done by first selecting a primitive graph and then combining it with a newly selected graph by a link labelled 1; the last node is connected back to the first node. To generate the execution costs for the nodes, 0.1, 0.2, 1.0, 5.0 and 10.0 are used as communication to cost ratios (*CCR*). Since 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 remains the same. These costs are generated in the following manner. For example, if the total communication cost (sum of the cost of all of the edges connected to this task) of task i is 16 and *CCR* used is 0.2 then, average execution cost of i will be, $16/0.2 = 80$.

5.2 Speed-up Using Parallel Algorithm

This sub-section discusses 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. It is observed that the speedup increases with an increase in problem size. Also the problems with CCR equal to 0.1 and 0.2 give good speedup in most of the cases, since the running time of serial algorithm is longer compared to larger CCRs. Table 1 presents the speedup for fully-connected topology of 4 processors with CCR equal to 0.1. Second column is the running time of the serial algorithm while third, fourth and fifth columns are the speed up of the parallel algorithm over serial one for 2, 4, 8 and 16 Paragon PEs, respectively. Bottom row of the table is the average speedup of all the task graphs considered. The values of average speedup for fully-connected, ring, and line topologies are shown graphically in Figure 4.

6 Conclusions

In this paper, we proposed a aparallel algorithm for optimal assignments of tasks to processors. We considered the problem under most relaxed assumption such as arbitrary task graph including arbitrary costs on the nodes and edges of the graph, and processors connected through an interconnection network. Our algorithm can be used for homogeneous or heterogeneous processors although in this paper we only considered the heterogeneous cases. We believe that to the best of our knowledge, ours is the first attempt in designing a parallel algorithm for the optimal task-to-processor assignment problem. There are a number of further studies that are required to understand the behavior of the parallel algorithm and some possible fine improvements. We are currently exploring these possibilities.

References

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

[2] 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.

[3] V. Chaudhary and J. K. Aggarwal, "A generalized scheme for mapping parallel algorithms," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 3, Mar 1993.

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

[5] D. E. Goldberg, "Genetic algorithms in search, optimization, and Machine learning," (Addison, Wesely, Reading, MA 1989)

[6] S. Hurley, "Taskgraph Mapping Using a Genetic Algorithm: A comparision of Fitness Functions," *Parallel Computing*, vol. 19, pp. 1313-1317, Nov 1993.

[7] V. Mary Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans on Computers.* vol 37, no 11, Nov1988.

[8] 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.

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

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

[11] 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.

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

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

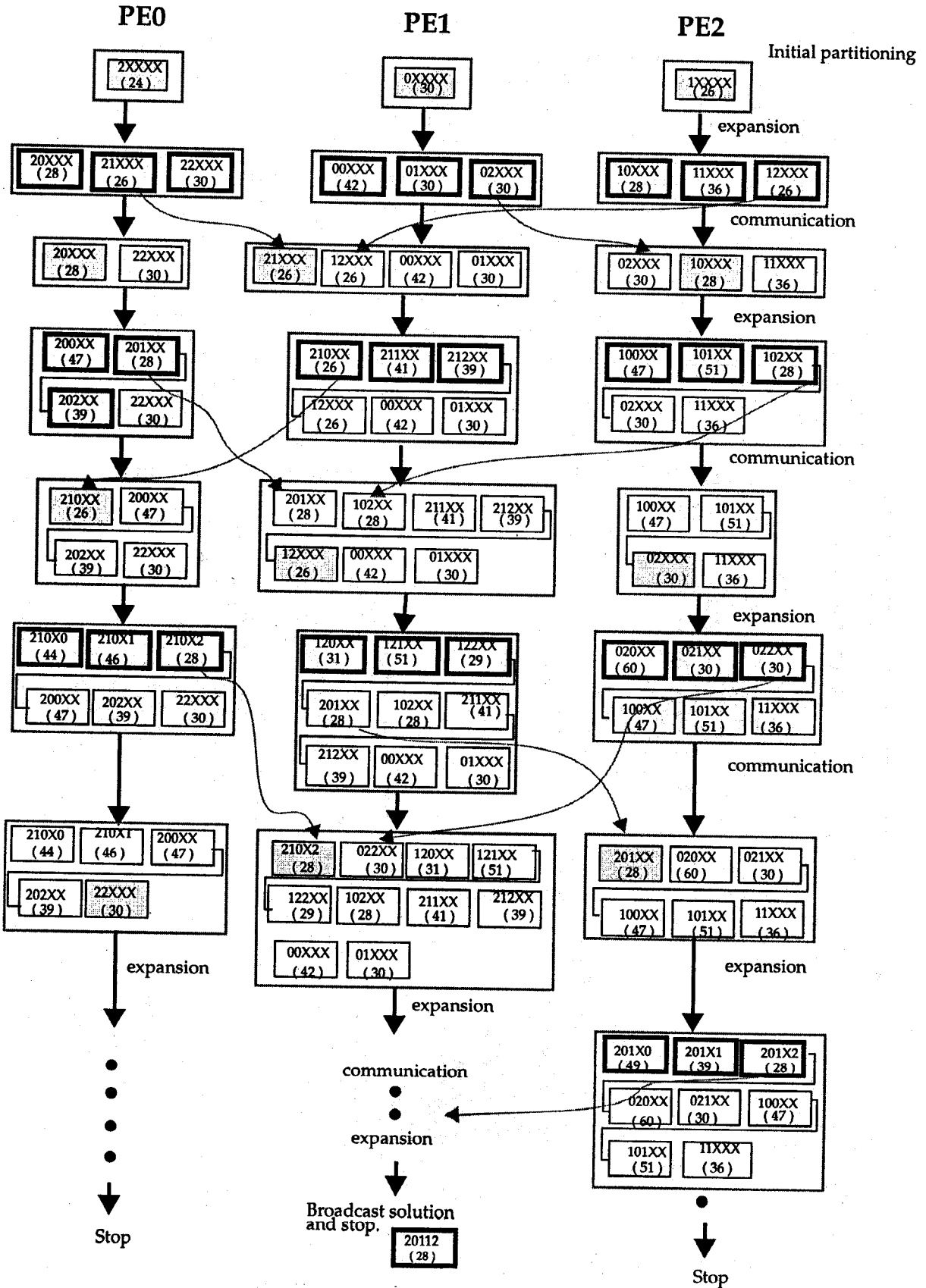


Figure 3: The operation of the parallel assignment algorithm using three PEs.

Table 1: Speedup with the parallel algorithm using fully-connected topology (CCR=0.1).

No. of Tasks	Time(OASS) / Time(OPAS)			
	PEs=2	PEs=4	PEs=8	PEs=16
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
Avg	1.89	3.28	5.30	9.13

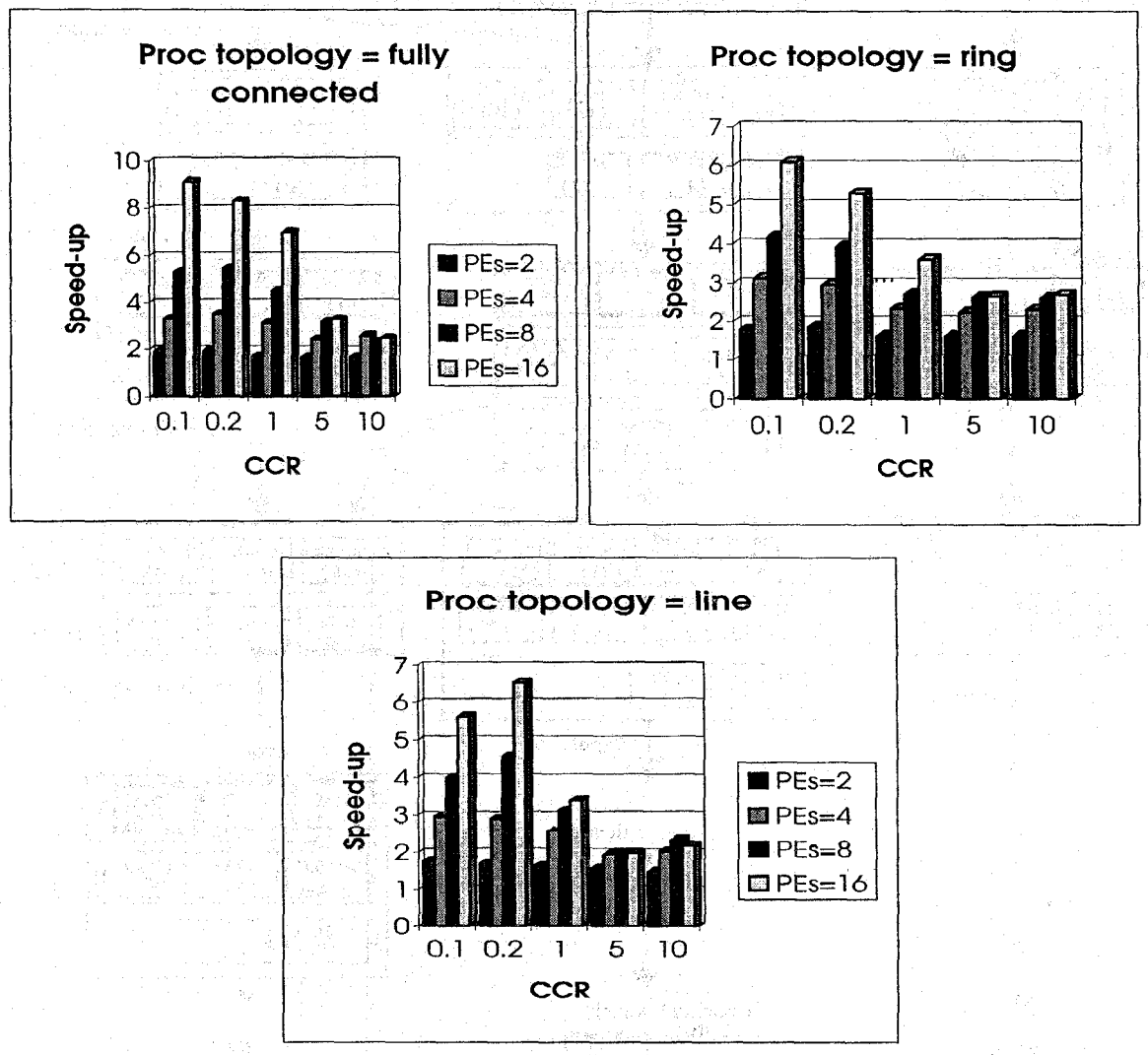


Figure 4: Average speedup of the parallel algorithm.