ALLOCATING CONCURRENT PARALLEL TASKS IN PARALLEL ENVIRONMENTS

SILVIA M. FIGUEIRA

Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566, USA
sfigueira@scu.edu
http://www.cse.scu.edu/~sfigueira

Abstract

In this paper, we focus on the execution of high-performance distributed applications on parallel platforms, such as MPPs or homogeneous clusters. These applications are usually formed by tasks, which are mapped to the nodes available in the parallel platform. When these tasks can execute concurrently, it may be necessary to space-share the nodes available within the application. In this case, partitioning the nodes among the concurrent tasks is a key part of the mapping process. This paper presents a polynomial algorithm that provides an optimal partitioning of nodes to concurrent parallel tasks.

Key Words: Task Allocation, Task Scheduling, Clusters of Workstations, Parallel Processors

1 Introduction

This paper focuses on the execution of distributed applications formed by two or more data-parallel tasks (for example, [17, 19, 20, 23]), on parallel platforms, such as MPPs or homogeneous clusters (e.g., the IBM SP/2 or Beowulf Clusters [25]). These parallel platforms are space-shared and, whenever these applications execute on such a platform, the tasks need to be mapped to the nodes available. This mapping process is key in obtaining good performance, and mapping strategies have been discussed extensively in the literature. These strategies assume a distributed model in which execution sites are time- (e.g. [1, 6, 7, 16]) or space-shared [11] by different applications. However, when executing distributed applications formed by data-parallel tasks, if tasks can execute concurrently, it may be necessary to space-share the nodes available within the application.

To illustrate, consider the application shown in Figure 1. The application is comprised of three tasks: ID (which initializes data), SOR (which implements a red-black SOR algorithm to solve Laplace's Equation), and GE (which implements a Gaussian Elimination algorithm). Both the SOR and the GE tasks were implemented using PVM [22], and require data generated by the initial task. Communication is required when either SOR or GE, or both, does not execute on the same machine as ID.

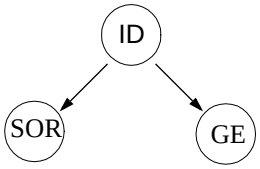

Figure 1: The graph represents an application formed by three tasks, ID, SOR, and GE. Tasks SOR and GE can execute concurrently.

Depending on the number of nodes available in the parallel platform, the application can execute in one of the following ways:

- execute ID, then execute GE, then execute SOR, or
- execute ID, then execute SOR, then execute GE, or
- execute ID, then execute GE and SOR concurrently, by having GE and SOR space-share the nodes.

In order to efficiently execute GE and SOR concurrently in a parallel environment, an optimal partitioning of the nodes to the tasks must be found.

For an example of optimal partitioning, consider executing tasks GE and SOR on an 8-node DEC Alpha-Farm. The DEC Alpha-Farm used is a cluster formed by eight DEC Alpha 3000/400 workstations located at the San Diego Supercomputer Center. The Alphas are connected via a dedicated GIGAswitch. SOR executes on a matrix of size 500x500, whereas GE executes on a matrix of size 500x501. Execution times are available for each task from individual runs. Table 1 shows the execution times for both tasks, SOR and GE, with different numbers of nodes. Both tasks execute in minimum time with 6 nodes. However, only 8 nodes are available, and the tasks cannot execute concurrently in minimum time. The time to execute both tasks on the DEC Alpha-Farm with 6 nodes, one after the other, would be 107.905 seconds. However, with an optimal partitioning, 3 nodes are assigned to SOR and 5 nodes to GE. For this partitioning, since no slowdown is imposed by contention for the GIGAswitch, the overall execution

time for tasks SOR and GE is 62.084 seconds.

Table 1: Execution Time (in seconds) on the Alpha-Farm

Number of Nodes	SOR	Gaussian Elimination
1	133.227	153.773
2	82.447	104.329
3	61.655	81.157
4	51.537	67.943
5	49.229	62.084
6	46.465	61.440
7	47.400	61.474
8	49.279	63.145

Note that, in this case, the overall execution time for the application depends on the partitioning of the nodes among the tasks. For this reason, finding an optimal partitioning is an essential part of the mapping process. In fact, optimal partitioning is extremely important in metacomputing [13] and grid environments [14], where applications are scheduled automatically to different platforms, which may consist of space-shared parallel platforms, such as MPPs or Beowulf clusters.

The partitioning, or mapping, problem, which consists of assigning tasks to nodes in order to minimize the completion time, is known to be an NP-hard problem [15]. In fact, a lot of effort has been made towards both solving tractable instances of the problem [3, 21] and developing heuristic approaches [2, 4, 5, 9, 10, 18, 24]. In this paper, we present a polynomial algorithm to determine an optimal partitioning of nodes to concurrent data-parallel tasks. This is a tractable instance of the problem, since data-parallel tasks have a peculiar behavior that allows for a greedy algorithm to provide an optimal partitioning in polynomial time.

This paper is organized as follows. Section 2 addresses the problem of partitioning nodes in parallel platforms. Section 3 presents the partitioning algorithm and discusses its correctness and complexity. Section 4 summarizes and discusses possible extensions for the algorithm.

2 Partitioning Nodes

The partitioning problem consists of partitioning a number of homogeneous nodes among data-parallel tasks in an optimal way, i.e., so that the tasks can execute concurrently in the minimum amount of time. When multiple tasks are executing concurrently, the *overall time* to execute them is the time to execute the longest one. An *optimal partitioning* of nodes to tasks is a partitioning for which the overall execution time is minimum.

As discussed in the previous section, the mapping problem is known to be NP-hard. However, since data-parallel tasks present a peculiar behavior, a greedy algorithm provides an optimal partitioning in polynomial time. The algorithm is based on the fact that the function that gives the execution time for different numbers of nodes is typically decreasing for values which do not exceed S_K, where S_K is the smallest number of nodes for which the execution of task *K* is the fastest. Figure 2 shows an example of a curve representing the execution time of a typical data-parallel task *K* parameterized by the number of nodes. The thicker part of the curve represents execution time with $s \leq S_K$ nodes. It is clear that, on this section of the curve, execution time correlates inversely with the number of nodes. Note that the 'local' minimum in this sector of the curve corresponds to the global minimum.

Figure 3 and Figure 4 show actual curves obtained by executing two tasks - a Gaussian Elimination algorithm and an SOR algorithm to solve Laplace's equation - on the SDSC DEC Alpha-Farm with different numbers of nodes. The graphs show that $S_{GE} = 4$ whereas $S_{SOR} = 6$. It is clear that, in both cases, the function is decreasing for values smaller than or equal to the smallest number of nodes which provides the minimum execution time.

Given this behavior, the goal is to decrease the time to execute the longest task by increasing its number of nodes, while increasing the other tasks' execution time by decreasing their number of nodes. Suppose there are two tasks *A* and *B*, and task *A* takes longer than task *B*. Also, task *A* would execute faster with one more node. If transferring a node from task *B* to task *A* does not make task *B* execute for a longer time than task *A*, then this transference leads to a better partitioning (in spite of task *B* taking longer to execute). This is explained by the fact that the overall time decreases when the longest task (in this case, task *A*) is able to execute faster. Intuitively, transferring all possible nodes to the longest task leads to an optimal partitioning, and a greedy approach, which transfers nodes from each task to the longest one, will solve the problem.

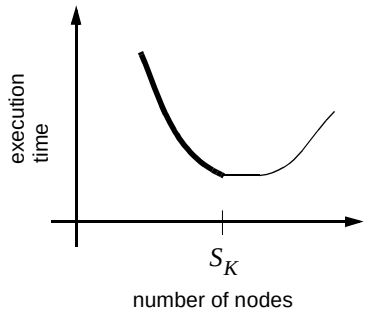

Figure 2: The graph shows a typical curve for execution time of task *K* parameterized by the number of nodes used. The thicker part of the curve represents the region of the curve for which the task uses a number of nodes smaller than or equal to S_K, the smallest number of nodes for which the execution is the fastest.

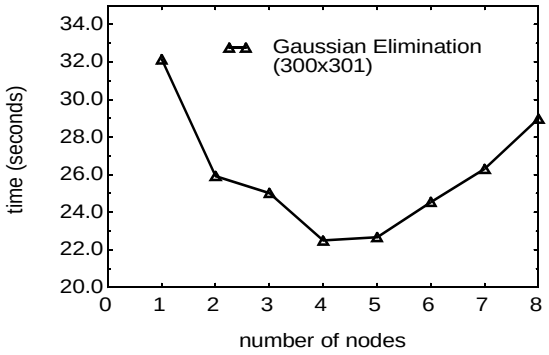

Figure 3: Execution time for Gaussian Elimination with different number of nodes.

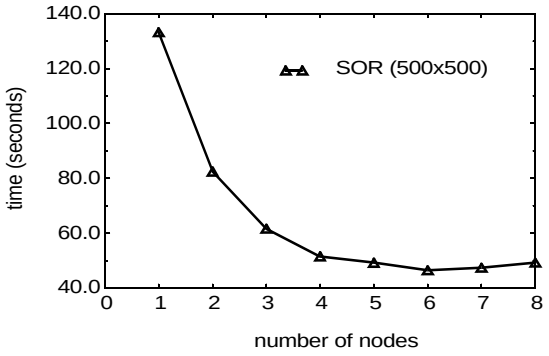

Figure 4: Execution time for SOR with different number of nodes

3 The Partitioning Algorithm

Given a set of tasks and a set of nodes, the partitioning algorithm divides the nodes among the tasks in an optimal way, so that the total time to execute the tasks concurrently is optimal. This is a greedy algorithm, which works based on the assumption that the tasks are data-parallel and behave according to the curve shown in the previous section.

Note that the algorithm assumes that the environment has the following characteristics:

- The nodes in the cluster are homogeneous.
- The nodes are not time-shared. Consequently, execution times for each task are deterministic and known.
- Contention for the network is not significant.

We assume that the following quantities are known or have been estimated:

- the time to execute a task K on n nodes of a parallel machine (or homogeneous cluster) M, and
- the smallest number of nodes S_K for which task K executes on machine M in the minimum amount of time.

These values can be provided by the user or obtained from a model such as Driscoll and Daasch's [8], who have developed a strategy to estimate these amounts for MPPs. If exact costs are not available for the tasks on variable-sized sets of nodes, estimates can be used. For resource-intensive problems which require the use of parallel platforms, users generally have a good estimate of execution times. As with any scheduling strategy, the more accurately these parameters can be predicted, the better the derived mapping will be.

3.1 The Algorithm

The pseudo-code for the algorithm is shown in Figure 5. The algorithm works as follows:

- Assign S_K nodes to each task K.
- If the total nodes assigned is smaller than or equal to N (the total number of nodes available), then each task is already executing in minimum time, and no repartitioning is necessary.
- If the total nodes assigned is greater than N, repartitioning is necessary. In this case, extract nodes from the tasks uniformly until the sum of all the nodes assigned to each task is equal to N, and each task has at least one node.
- Place each task K in an array, ordered by execution time (according to the node assignment).
- Traverse the array, from left to right, transferring nodes from each task to the longest one (always on the extreme right of the array) whenever the transference will cause the overall execution time to decrease, i.e., the time to execute the longest task decreases with one more node, and the time to execute the yielding thread with one less node is not greater than the time to execute the longest thread.
- For each position in the array, the algorithm transfers as many nodes as possible from the corresponding task to the longest one.
- After each transference, the position of both the yielding and longest tasks are re-evaluated and the array is updated. The algorithm continues to transfer nodes from whichever task is in the current position until no node can be transferred.
- The algorithm terminates when the overall time cannot be decreased, i.e., no node transference is possible. This happens when the longest task's execution time does not decrease with extra nodes or when the traversal finishes. In this case, all tasks had a chance to yield as many nodes as possible to the longest one.

3.2 Correctness

We show that, after traversing the array, the partitioning obtained is an optimal partitioning, in which no transference of nodes decreases the overall time. For a formal proof, see [12].

The algorithm terminates

1. when there is enough nodes to execute all the tasks with the optimal number of nodes,
2. when transferring a node from any task to the longest one just increases the longest task's execution time, or
3. when the array has been totally traversed.

```
Partitioning Algorithm

assign each task its optimum number of nodes
if the total number of nodes assigned is smaller or equal to
the total number of nodes available
        return <repartitioning is not needed>
else <repartition nodes>
        extract nodes from each task until only the nodes
        available are assigned
        sort tasks according to the time to execute with
        the nodes assigned and place sorted tasks in array A
        for j = 1 to T - 1 do
        <traverse the array, T is the number of tasks>
            while task in the jth position has
            more than one node do
                J = task in the jth position
                P = longest task
                if giving one more node to task P does not
                decrease its time then
                    return <overall time cannot be improved>
                endif
                if task J with one less node is still faster than
                task P then
                    transfer a node from task J to task P
                    update the position of task J in array A,
                    according to its new time
                    update the position of task P in array A,
                    according to its new time
                else
                    break <go to next position,
                    task in the jth position cannot
                    yield any more nodes>
                end if
            end while
        end for
    end if
return <array traversal is over>
```

Figure 5: Pseudo-code for the Partitioning Algorithm.

In case 1, the partitioning is obviously optimum because, since every task is executing with its optimal number of nodes, no other partitioning will be better than that. In case 2, the partitioning is optimal because transferring any node to the longest task would not decrease its time. That means, the longest task is already executing at full speed, and it is impossible to obtain a partitioning in which the overall time would be any shorter.

In case 3, the array has been traversed, and all the tasks have yielded as many nodes as possible to decrease the overall time. Traversing the array once is enough because after a task yields all the nodes it can in a first traversal, it will not be able to yield any nodes in a second traversal, since the longest task's execution time will have only decreased. Even though tasks are reordered, every task has a chance to yield as many nodes as it can, since the only possible changes in task position are:

1.Change in the position of the task which has just yielded a node to the longest task.

Since the time to execute the yielding task will increase (it only increases when yielding), the task will move to the right in the array, and therefore will be rechecked later. In this case, another task will move to the left and occupy the position being checked. Since the algorithm will recheck the current position until no node can be yielded, the task just moved will be checked. This case is illustrated in Figure 6. In Figure 6.(a), the algorithm is checking the third position on the array, which contains task *B*. Figure 6.(b) shows the updated array after task *B* yields a node to task *P* (which is the longest task): Task *B* has moved to the right making task *C* move to the left. The algorithm will now go on checking the task in the third position, which is task *C*. Task *B* will be rechecked again, when the algorithm gets to its position. Therefore, in spite of the change, each task will be checked until no node transference is possible.

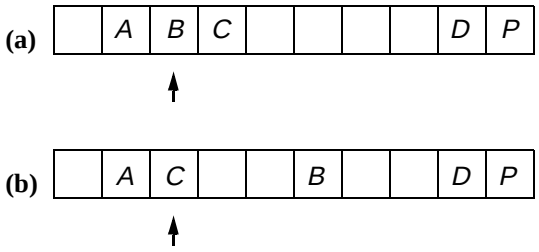

Figure 6: Possible changes of positions in the array of tasks, considering the yielding task.

2.Change in the position of the longest task.

When the time to execute the longest task decreases, it moves left in the array. Figure 7 illustrates this case. In Figure 7.(a), the algorithm is checking the third position on the array, which contains task *B*. Suppose task *B* yields a node to task *P*, which is the longest task. Task *P* could execute faster than task *D*, but slower than task *B* (Figure 7.(b)). In this case, the algorithm goes on checking task *B*, and each task which is faster than the new longest one - *D*. Task *P* could also execute faster than task *B*. In this case, if it is still slower than task *A* (Figure 7.(c)), the algorithm will check task *P*, which cannot yield nodes (during the traversal, no task is able to yield nodes after having been the longest task, because the overall time has only decreased since it was the longest task, and its time with one less node would be greater than the new overall time) and go on. If it executes faster than task *A* (Figure 7.(d)), the algorithm will recheck task *A*, which cannot yield nodes (tasks that cannot yield a node at some point during the traversal will not be able to yield a node because the longest task's execution time always decreases) and go on. Since task

P cannot yield a node (during the traversal, no task is able to yield nodes after having been the longest task), it does not need to be checked.

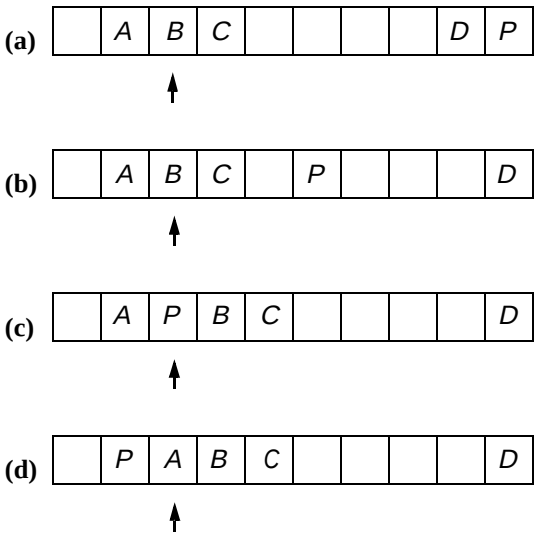

Figure 7: Possible changes of positions in the array of tasks, considering the longest task.

3.3 Complexity

Assigning nodes to the tasks take $O(T)$, and extracting nodes from the tasks take $O(N)$. The time-consuming parts of the algorithm are: sorting the tasks, which takes $O(TlogT)$, and traversing the array, which takes $O(TN)$. Therefore, the algorithm executes in time $O(T + N + TlogT + NT)$. Note that $N \gg T$, since the tasks are data-parallel and each task generally executes in several nodes. Therefore, since $N > 0$, $T > 0$, and $N \gg T$, we can consider the algorithm to execute in $O(NT)$ steps.

4 Summary

Distributed applications may contain tasks that can execute concurrently. When executing these applications in parallel platforms, it may be necessary to partition the nodes among the concurrent tasks. Minimizing the overall execution time of these tasks is key to the efficient utilization of these platforms, but depends on determining an optimal partitioning of nodes. We have presented a polynomial algorithm to solve this problem.

The partitioning algorithm executes in time $O(NT)$, where N is the number of nodes whereas T is the number of tasks. It determines an optimal partitioning of nodes to concurrent tasks by transferring nodes from each task to the longest one until no transference is possible. The algorithm works based on the fact that a partitioning in which no transference of nodes decreases the overall execution time is an optimal partitioning.

The partitioning algorithm can be easily extended to take into account constraints such as the amount of memory required and/or thresholds on the size of the partition available. Note that the partitioning algorithm can also be used to improve the utilization of multicomputers by applications which have relaxed constraints on execution time, e.g., applications that do not require minimum execution time but need to finish in a fixed amount of time. Transferring nodes between these applications may allow more applications to share the multicomputer, decreasing the average response time. In this case, there is a straightforward extension of the partitioning algorithm, which allows transference of nodes only when the yielding application does not increase its time above a pre-determined threshold.

The algorithm can also be extended to accommodate for heterogeneous clusters of workstations. In this case, machines may be different (i.e., have different computational capacities) and also time-shared, and these aspects need to be taken into account when partitioning the nodes. Partitioning heterogeneous nodes will have the same time complexity. However, due to the heterogeneity, the algorithm will partition the nodes by partitioning the capacity of the cluster.

Acknowledgement

We would like to thank Prof. Francine Berman for helping revise the paper, and we would like to thank our colleagues in the San Diego Supercomputer Center for their support, specially Mike Vildibill, Ken Steube, Cindy Zheng, and Victor Hazlewood.

References

[1] T. Anderson, D. Culler, D. Patterson, and the NOW team, “A Case for NOW (Networks of Workstations),”, in IEEE Micro, vol. 15, no. 1, pp. 54-64, February 1995.

[2] S. H. Bokhari, “On the Mapping Problem”, IEEE Transactions on Computers, vol. C-30, pp. 207-214, March 1981.

[3] S. H. Bokhari, “Partitioning Problems in Parallel, Pipelined, and Distributed Computing, IEEE Transactions on Computers, vol. 37, no. 1, pp. 48-57, January 1988.

[4] S. W. Bollinger and S. F. Midkiff, “Heuristic Technique for Processor and Link Assignment in Multicomputers,” IEEE Transactions on Computers, vol. 40, no. 3, pp. 325-333, March 1991.

[5] N. S. Bowen, C. N. Nikolaou, A. Ghafoor, “On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems,” IEEE Transactions on Computers, vol. 41, no. 3, pp. 257-273., March 1992.

[6] A. Bricker, M. Litzkow, and M. Livny, “Condor Technical Summary”, Technical Report #1069, University of Wisconsin, Computer Science Department, May 1992.

[7] H. G. Dietz, W. E. Cohen, and B. K. Grant, “Would you run it here... Or there? (AHS: Automatic Heterogeneous Supercomputing”, Proceedings of the 1993 International Conference on Parallel Processing, vol. II, pp. 217-221, 1993.

[8] M. A. Driscoll and W. R. Daasch, “Accurate Predictions of Parallel Program Execution Time,” Journal of Parallel and Distributed Computing, vol. 25, pp. 16-30, 1995.

[9] K. Efe, “Heuristic Models of Task Assignment Scheduling in Distributed Systems,” IEEE Computer, vol. 15, no. 6, pp. 50-56, 1982.

[10] F. Ercal, J. Ramanujam, and P. Sadayappan, “Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning,” Journal of Parallel and Distributed Computing, vol. 10, pp. 35-44, 1990.

[11] D. G. Feitelson, “A Survey of Scheduling in Multiprogrammed Parallel Systems,” Technical Report RC 19790 (87657), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, October 1994.

[12] S. M. Figueira, “Modeling the Effects of Contention on Application Performance in Multi-User Environments,” Ph.D. Dissertation, CSE Department, UCSD, December 1996.

[13] I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” International Journal on Supercomputing Applications, vol. 11, no. 2, pp. 115-128, 1997.

[14] I. Foster and C. Kesselman, editors, “The Grid: Blueprint for a New Computing Infrastructure,” Morgan Kaufmann Publishers, 1999.

[15] M. R. Garey and D. S. Johnson, “Computers and Intractability: A Guide to the Theory of NP-Completeness,” W. H. Freeman, San Francisco, 1979.

[16] A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Reynolds Jr., “Legion: The Next Logical Step Toward a Nationwide Virtual Computer”, University of Virginia, CS Technical Report CS-94-21, June, 1994.

[17] A. Kuppermann and M. Wu, “Quantum Reaction Dynamics on a Gigabit/Sec Network,” Proceedings of the Gigabit Testbed Maxijam, November 1994.

[18] V. M. Lo, “Heuristic Algorithms for Task Assignment in Distributed Systems,” IEEE Transactions on Computers, vol. 37, no. 11, pp. 1384-1397, November 1988.

[19] C. R. Mechoso, J. D. Farrara, J. A. Spahr, “Running a Climate Model in a Heterogeneous, Distributed Environment,” Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing, pp. 79-84, August 1994.

[20] C. R. Mechoso, C. Ma, J. D. Farrara, J. A. Spahr, and R. W. Moore, “Distribution of a Climate Model across High-Speed Networks,” Proceedings of the Supercomputing’91, pp. 253-260, 1991.

[21] D. M. Nicol and D. R. O’Hallaron, “Improved Algorithms for Mapping Parallel and Pipelined Computations,” IEEE Transactions on Computers, vol. 40, no. 3, pp. 295-306, 1991.

[22] V. Sunderam, “PVM: A Framework for Parallel Distributed Computing”, Concurrency: Practice and Experience, vol. 2, n. 4, pp. 315-339, December 1990.

[23] Virtual Environments and Distributed Computing at SC’95. GII Testbed and HPC Challenge Applications on the I-WAY. Edited by Holly Korab and Maxine D. Brown. A publication of ACM/IEEE Supercomputing’95.

[24] L. Tao, B. Narahari, and Y. C. Zhao, “Heuristics for Mapping Parallel Computations to Heterogeneous Parallel Architectures,” Proceedings of the Heterogeneous Computing Workshop, pp. 36-41, April 1993.

[25] The Beowulf Project at NASA: http:// www.beowulf.org/