Mapping Parallel Applications to Distributed Heterogeneous Systems

Silvia M. Figueira[1] and Francine Berman[2]

Department of Computer Science and Engineering
University of California, San Diego

{silvia,berman}@cs.ucsd.edu

Abstract

Fast networks have made it possible to coordinate distributed heterogeneous CPU, memory and storage resources to provide a powerful platform for executing high-performance applications. However, the performance of parallel applications on such systems is highly dependent on the mapping of application tasks to machines. In this paper, we propose a mapping strategy for applications formed by multiple tasks targeted to heterogeneous platforms. We first define a mapping model, the **match-tree**, which reflects the data movement and conversion costs of distributed algorithms and allows for alternative implementations of individual tasks on different machines. We then define the **find-mapping** and **split-partition** algorithms, based on the match-tree model, to determine the best allocation of tasks to resources in heterogeneous systems. We illustrate the use of these algorithms with a sample distributed application.

1 Introduction

In the last decade, distributed heterogeneous systems have emerged as a powerful platform for executing high-performance applications. Current experience shows that many high-performance distributed applications are composed of a few coarse-grained tasks and execute on heterogeneous systems formed by a few machines [1, 15, 17]. These machines can be MIMD MPPs, visualization engines, SIMD MPPs, workstations, and/or vector computers. For many of these applications, both heterogeneity and parallelism in the code are exploited [4, 10, 13, 14]. For instance, in a molecular structure determination application which computes the global minimum of a potential energy function [14],

1. Supported by a scholarship from CAPES and UFRJ (Brazil).
2. Supported in part by NSF contract number ASC-9301788.

an MPP is used to compute distinct local minima whereas a vector computer is used to compute a convex quadratic underestimator function by solving a linear program.

The mapping of tasks to machines in distributed systems is known to be NP-hard [3, 8]. One would assume that for heterogeneous applications, typically composed of a few tasks and targeted to 2 or 3 machines, the mapping problem would be easy - in particular, exhaustive search would solve the problem.

In fact, determining the best match between tasks and machines, even when the number of both machines and tasks is small, is hard for reasons other than scalability. Estimating the performance of task/machine matches on production systems depends on both application characteristics (execution flow, data dependences between tasks, implementation specifics, etc.) and platform features (computation capacity, number of processors, communication bandwidth, memory size, libraries available, etc.). For example, communication between tasks which do not execute on the same machine may happen in different ways, depending on the application implementation and on the communication mechanism provided by the platform. Communication in opposite directions may occur concurrently or serially. Also, computation and communication may be overlapped in some systems. These are enhancements which may decrease the effect of the communication costs on the overall time and must be reflected in the performance estimation of a candidate mapping.

Most mapping strategies in the literature assume a distributed model in which execution sites are time-shared (e.g. [2, 5, 9, 11, 18]). In practice, an execution site in a heterogeneous system may be an MPP (such as the Intel Paragon or the Cray T3D) or a dedicated cluster[3] (such as the IBM SP-2 or the DEC Alpha Farm). For these sites, concurrent application tasks may execute **concurrently** in space-shared rather than time-shared mode. Hence applications in such heterogeneous systems can be distributed both **between** execution sites and **within** execution sites. This is an especially useful strategy for distributed applications with data-parallel tasks. Note that the overall execution time for the application depends on the partitioning of nodes (within the heterogeneous network and within an individual multicomputer) among tasks, and determining an optimal assignment of nodes to tasks is an essential part of the mapping process.

In this paper, we focus on the mapping of parallel applications to distributed heterogeneous resources. The paper describes two mechanisms which promote efficient mappings: First, we define a **match-tree** model which represents the assignment options for mapping application tasks to heterogeneous resources. Second, we discuss the **find-mapping** and **split-partition** algorithms which determine the best allocation of tasks to resources in heterogeneous systems. In particular, the split-partition algorithm determines an optimal partitioning of nodes in an MPP or cluster site to concurrent tasks. Together, the match-tree model and the algorithms provide a strategy which may be used for mapping parallel applications to distributed heterogeneous systems in an optimal fashion. We conclude with an example which shows how these results can be used in practice.

This paper is organized as follows: Section 2 addresses the mapping of applications to platforms. Section 3 presents the find-mapping and split-partition algorithms. Section 4 provides an example to show how the algorithms can be used in practice, and Section 5 summarizes our results. The proofs of the Theorems, used to show that the partitioning

---

[3]. We define a dedicated cluster as a platform comprised of nodes interconnected by a network dedicated to them (e.g., a GIGAswitch).

is optimal, can be found in the Appendix.

2 Mapping Parallel Applications

2.1 Representing Applications

We represent the execution of an application as a **flow-graph**. A flow-graph is a directed graph (V, E) such that

V = {*u* | *u* is an atomic task}, and

E = {(*u*, v) | data is communicated from task *u* to task *v*}.

Figure 1 shows an example of a flow-graph.

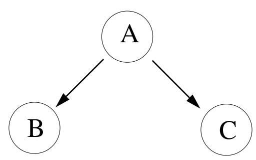

Figure 1: Flow-graph representing an application formed by three tasks, A, B, and C. Tasks B and C can execute concurrently, but both require data generated by task A.

2.2 Representing Mappings

The possible mappings between machines and tasks are represented by a **match-tree**. Each edge on the match-tree is associated with a cost, and each path from the match-tree's root to a leaf represents a possible mapping between tasks and machines. The least-cost path (determined by the sum of the costs associated with the edges along the path) from the root to a leaf in the match-tree corresponds to an optimal mapping.

We define an **execution activity** to be the execution of a task or the communication of data between tasks. A match-tree is a tree (V, E) such that

V = {*u* | *u* is a boundary between execution activities}, and

E = {(u, v) | there is an execution activity with initial boundary u and terminal boundary v}.

Note that each path in the match-tree represents a possible implementation of the application on the target platform, and the aggregation of execution activities on a path reflects the cost of the actual implementation.

Each edge in E is associated with a computation or communication cost as appropriate. Edges labeled X_m represent the cost to execute task X on machine M_m. Edges labeled $X_m \rightarrow Y_n$ represent the cost to transfer data between task X and task Y, when task X executes on machine M_m and task Y executes on machine M_n. If there are tasks which can execute in parallel, edges labeled $X_m/.../Y_n$ represent the cost to execute the tasks concurrently (X_m indicates the allocation of task X to machine M_m and Y_n indicates the allocation of task Y to machine M_n).

As an example, consider the application formed by three tasks (A, B, and C) shown by Figure 1. This is the flow-graph for one iteration of a General Circulation Model (GCM) [1, 12, 13], which can be used to study the nonlinear interaction and feedback between components of a climate system. Task A (an Atmospheric/Physics task) generates data to be used by tasks B (an Atmospheric/Dynamics task) and C (an Ocean Simulation), which can execute concurrently. Communication is required when either task B or C, or both, does not execute on the same machine as task A.

Assume the GCM application is implemented on a heterogeneous system comprised of machines M_1 and M_2[4]. For now, assume that both M_1 and M_2 have a single processor. Suppose further that task A can only execute on machine M_1, but both tasks B and C can

4. This is indeed how the heterogeneous implementation is performed in practice [1, 12, 13]. For instance, in the UCLA implementation of GCM, machine M_1 is a Cray C90 and machine M_2 is an Intel Paragon.

run on M_1 or M_2. The best mapping between tasks and machines will be the one with the least-cost path on the match-tree shown on Figure 2. The possible choices for implementations are:

1. execute all the tasks on machine M_1, (one after the other),
2. execute A on M_1, send data for B to M_2, execute B on M_2, send data for C to M_2, execute C on M_2,
3. execute A on M_1, send data for C to M_2, execute C on M_2, send data for B to M_2, execute B on M_2,
4. execute A on M_1, send data for C to M_2, execute task B on M_1 and task C on M_2 (concurrently), and
5. execute A on M_1, send data for B to M_2, execute task B on M_2 and task C on M_1 (concurrently).

Now assume that machine M_2 is an MPP or dedicated cluster. In this case, tasks can be executed concurrently on M_2, and the match-tree must accommodate the additional implementation option (shown in bold in Figure 3) which is:

6. execute A on M_1, send data for B and C to M_2, execute both B and C on M_2 (concurrently).

Determining the cost of executing B and C concurrently on M_2 involves calculating the cost of all partitionings of the available nodes between the two tasks. In particular, an optimal partitioning of nodes to tasks must be found. We have developed the split-partition algorithm to determine an optimal partitioning of nodes to tasks for this case. We present this algorithm in Section 3.

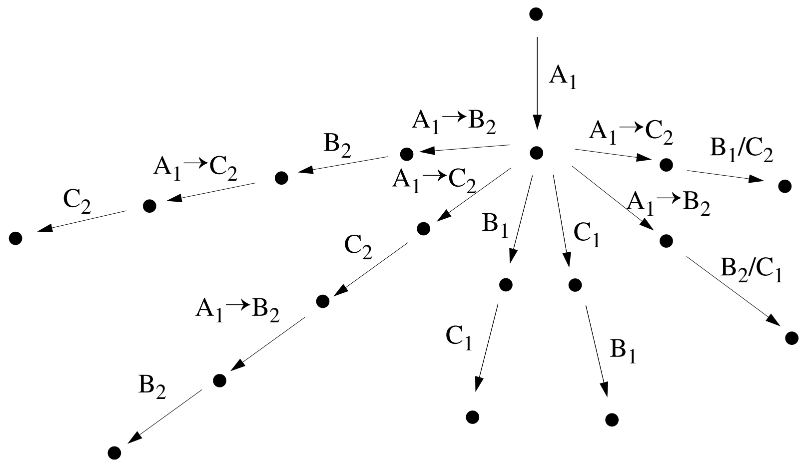

Figure 2: Match-tree representing the possible implementations of the application shown in Figure 1 on a two-machine platform formed by machines M_1 and M_2.

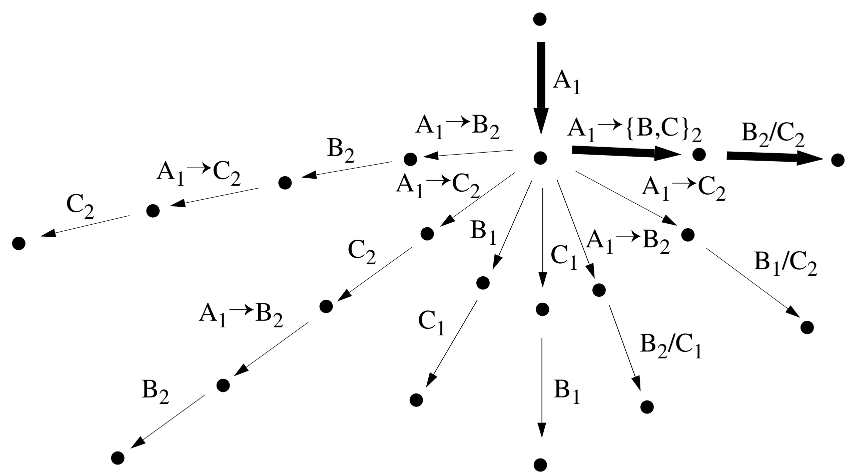

Figure 3: Match-tree representing the possible implementations of the application showed in Figure 1 on a two-machine platform formed by machines M_1 and M_2, where M_2 is an MPP or a dedicated cluster. The label $A_1 \rightarrow \{B, C\}_2$ represents data communication from task A on machine M_1 to machine M_2 for both tasks B and C in some order.

In production systems, contention effects (synthesized into a slowdown factor [7]) must be included in the communication and computation costs (or estimates) associated with each edge in the match-tree. When concurrent tasks share an MPP or dedicated cluster, the effects of contention between multiple non-communicating tasks are typically negligible. In the unusual case where these effects are non-negligible, a slowdown factor should be applied to the tasks/machine estimates as well.

3 Splitting a Partition

In order to determine the optimal mapping given by a match-tree, we use the following **find-mapping** algorithm:

```
find-mapping()
    for all paths from root to leaf
        assess cost
return path with the lowest cost
```

This algorithm executes in time $O(CL)$, where C is the number of leaves and L is the length of the longest path in the match-tree. Since in practice the number of tasks for heterogeneous applications in practice is typically ≤ 4 and the number of machines used is typically ≤ 3 [17], $L \leq 12$ and $C \leq 639$ (although, in practice, $C \ll 639$).

Note that we cannot execute the algorithm without a value for all paths. For paths with an edge labeled $X_m/.../Y_n$, i.e., for paths with an assignment of concurrent tasks to an MPP or dedicated cluster, we must determine the optimal partitioning of nodes to tasks. The split-partition algorithm presented below determines this partitioning.

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

- the time to execute a task K on n nodes of a machine M, and
- the smallest number of nodes $\hat{n}_K$ for which task K executes on machine M in the minimum amount of time.

These can be provided by the user or obtained from a model such as Driscoll and Daasch's [6], who have developed a strategy to estimate these amounts for MPPs.

Our split-partition algorithm utilizes a function *time*(K, n_K), which provides the time to execute task K with n_K nodes. The algorithm is based on the fact that this function is typically decreasing for values which do not exceed $\hat{n}_K$. Figure 4 shows an example of a curve representing the execution time of a typical task K parameterized by the number of nodes. The thicker part of the curve represents execution time with $n_K \leq \hat{n}_K$ 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.

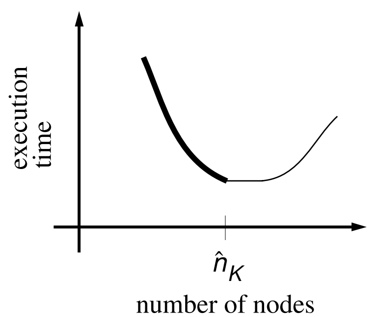

Figure 4: The graph shows a typical curve for execution time of task K parameterized by the number of nodes used. The thicker part of the curve represents the region of the curve for which the task uses a number of nodes smaller than or equal to $\hat{n}_K$, the smallest number of nodes for which the execution is the fastest.

Figure 5 and Figure 6 show actual curves obtained by executing two tasks - a Gaussian Elimination algorithm and an SOR algorithm to solve Laplace's equation - on an Alpha Farm[5] with different numbers of nodes. The graphs show that $\hat{n}_{GE} = 4$ whereas $\hat{n}_{SOR} = 6$. 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.

5. The Alpha Farm used is a cluster of eight DEC Alpha 3000/400 workstations located at the San Diego Supercomputer Center. The Alphas are connected via a dedicated GIGAswitch and communicate using PVM [16].

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. The Split-Partition algorithm finds an optimal partitioning of nodes among tasks by transferring nodes from each task to the longest one, until the transference of nodes does not decrease the overall time.

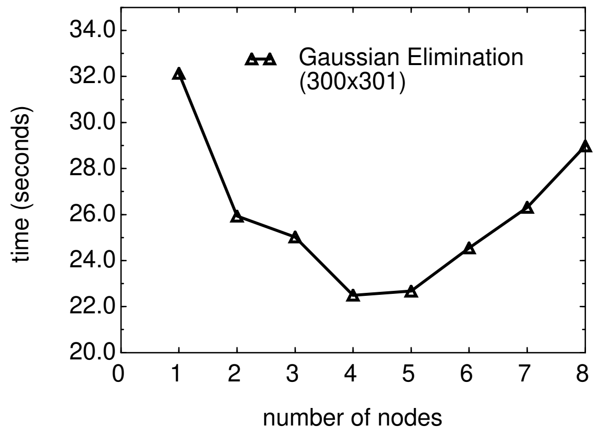

Figure 5: Execution time for Gaussian Elimination with different number of nodes.

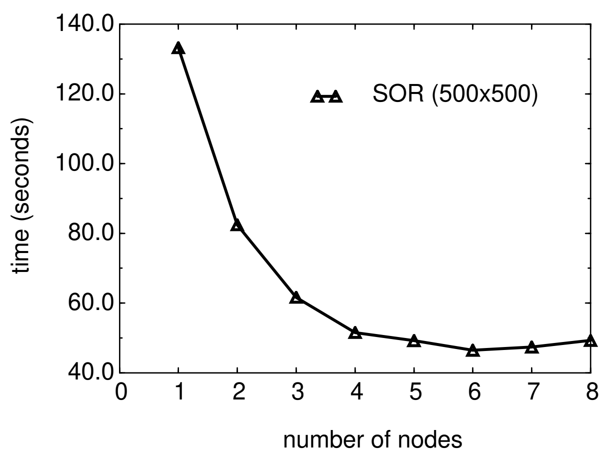

Figure 6: Execution time for SOR with different number of nodes

Split-partition algorithm:

<T = number of tasks>

<N = number of nodes>

<*time(i, n_i)* = function which returns the time to execute task *i* with n_i nodes>

<*time(i, 0)* = ∞, for any task *i*>

<$\hat{n}_i$ = smallest number of nodes *n* for which *time*(*i*, *n*) is minimum>

<*nodes(i)* = number of nodes assigned to task *i* in the given partition>

<*A*[*j*].*task* = task in the *j*th position of array *A*>

split-partition ()

```
nodes(i) = n̂_i, for each task i
if Σ_{i=1}^{T} n̂_i ≤ N
then
    return <repartitioning is not needed>
else <repartition nodes>
    extract-nodes()
    sort-tasks()
    for j = 1 to T - 1 do <traverse the array>
        while nodes(A[j].task) > 1 do
            J = A[j].task
            P = A[T].task
            if time(P, nodes(P) + 1) ≥ time(P, nodes(P))
            then
                return <overall time cannot be improved>
            endif
            if time(J, nodes(J) − 1) < time(P, nodes(P))
            then
                <transfer a node from task J to task P>
                update nodes(J) and nodes(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, because task in position j cannot yield any more nodes>
            endif
        enddo
    enddo
endif
return <array traversal is over>
```

The algorithm produces an optimal node partitioning in which no transference of nodes decreases the overall time (Theorem 1 and Theorem 2, see Appendix). The algorithm assigns each task i the smallest number of nodes which provides the minimum execution time, given by $\hat{n}_i$. If $\sum_{i=1}^{T} \hat{n}_i \leq N$ (where N is the total number of nodes available), the number of nodes available is enough for each task to execute in minimum time. In this case, repartitioning the nodes is not needed.

However, if $\sum_{i=1}^{T} \hat{n}_i > N$, repartitioning is needed. In this case, nodes are extracted from each task i, where $1 \leq i \leq T$, until $\sum_{i=1}^{T} nodes(i) = N$ (i.e., the sum of all the nodes assigned to each task is equal to N) and each task has at least one node. This is done by procedure **extract-nodes**, which loops over the set of tasks decreasing *nodes*(*i*) by 1 for each task *i*, which has more than one node, until $\sum_{i=1}^{T} nodes(i) = N$.

The tasks are then sorted and placed in array *A* in a non-decreasing order according to $time(i, nodes(i))$, the time to execute task *i* using *nodes*(*i*) nodes. This is done by procedure **sort-tasks**. After sorting the tasks, the algorithm traverses array *A*, 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. 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 overall execution time is minimized.

Correctness of the split-partition algorithm:

We show that the algorithm always terminates with an optimal node partitioning for multiple concurrent tasks. Termination occurs in the algorithm at 3 points:

- **Line 4:** In this case, the sum of the $\{\hat{n}_i\}$ does not exceed the total number of nodes available. Since each $\hat{n}_i$ is the smallest number of nodes for which task *i* executes the fastest, the partitioning is optimal.
- **Line 14:** The algorithm terminates at line 14 when increasing the number of nodes of the longest task does not decrease its execution time. According to Theorem 1 (Appendix), in this case, the partitioning is optimal.
- **Line 28:** In this case, the array was traversed once. According to Theorem 3 (Appendix), after traversing the entire array, additional transferences of nodes are not possible. By Theorem 2 (Appendix), when no transference is possible, the partitioning is optimal.

Complexity:

Lines 1 and 2 execute in time $O(T)$, where T is the number of tasks. Procedure extract-nodes executes in time $O(N)$, where N is the number of nodes, while procedure sort-tasks executes in time $O(T log T)$. The loop in lines 8-26 executes in time $O(TN)$, because each line from 10 to 24 executes $O(N)$ times, and lines 20 and 21 each execute in time $O(T)$. Therefore, the algorithm executes in time $O(T + N + T log T + NT)$. Since $N > 0$, $T > 0$ and, generally, $N \gg T$, we can consider the algorithm to execute in $O(NT)$ steps. Note that, in practice, T is generally no bigger than 4.

4 Using the Algorithm in Practice

We intend for the algorithm to be used as part of a scheduling strategy for parallel applications on heterogeneous systems, particularly those containing MPPs and/or dedicated clusters. We provide an example of how the match-tree and algorithms can be used in this section.

Consider the application shown in Figure 7 comprised of three tasks: ID (which initializes data), SOR (which implements an SOR algorithm), and GE (which implements a Gaussian Elimination algorithm). Both the SOR and the GE tasks require data generated by the initial task. The application executes on a two-machine platform at the San Diego Supercomputer Center. The platform is formed by a Sun Sparcstation (M_1) and a DEC Alpha Farm (M_2) in which there are 8 nodes available. Assuming ID has been implemented for M_1, and both SOR and GE have been implemented for M_2, Figure 8 shows the match-tree with the possible mappings between the application and the two-machine platform. Note that the data for SOR and GE can be sent in any order, when they execute concurrently on M_2.

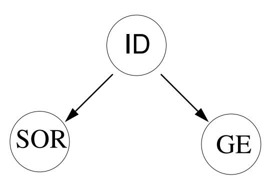

Figure 7: Flow-graph representing an application formed by three tasks, ID, SOR, and GE. Tasks SOR and GE can execute concurrently.

In this example, 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, and communication costs can be based on typical interprocessor communication costs for the appropriate sized messages. 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.

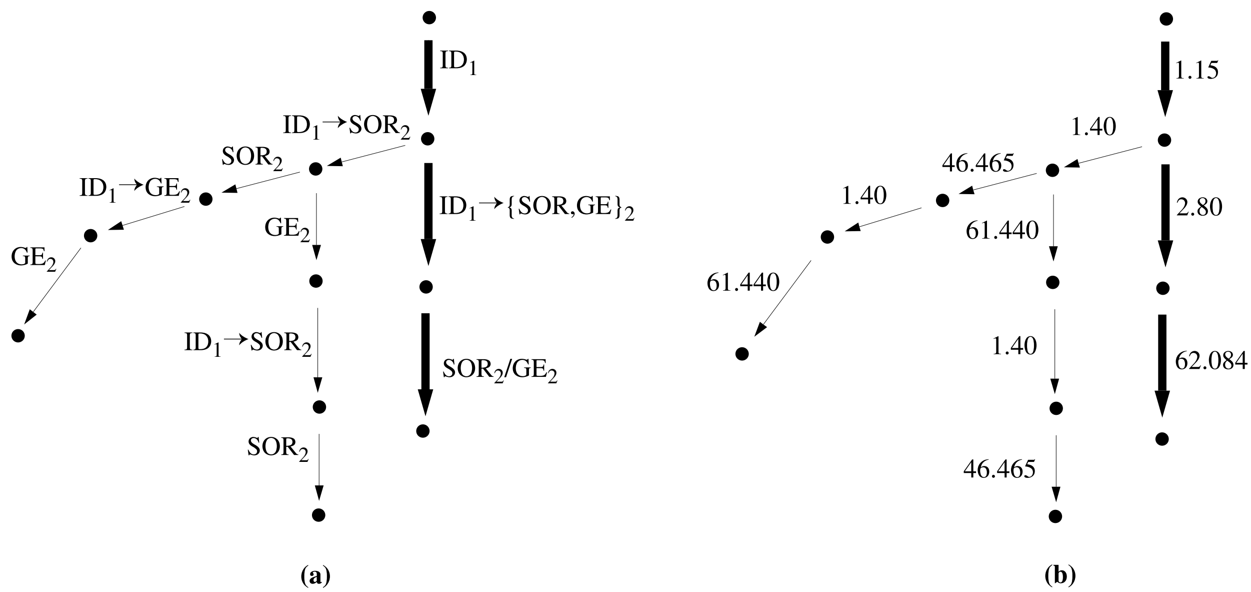

Figure 8: Match-trees representing the mappings between the application showed in Figure 7 and a two-machine platform formed by machines M_1 and M_2, where M_2 is a DEC Alpha Farm. Match-tree (a) shows the computation and communication activities whereas match-tree (b) shows the corresponding costs in terms of seconds. Note that the order in which the activities appear in a path reflects the application implementation. Therefore, each different implementation must have its own path represented in the match-tree.

By the split-partition algorithm, an optimal partitioning assigns 3 nodes 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. Note that the time to execute both tasks on the Alpha Farm with 6 nodes, one after the other, would be 107.905 seconds.

We use the split-partition decomposition and the costs shown in Figure 8 in the find-mapping algorithm. The best mapping consists of executing both tasks concurrently on M_2, which takes 66.034 seconds. This was determined by evaluating the cost for each path. The least-cost path (shown in bold in Figure 8) corresponds to the best mapping.

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 heterogeneous and clustered platforms, users generally have a good estimate of $time(i, n_i)$, as well as of $\hat{n}_i$, for a task i. As with any scheduling strategy, the more accurately these parameters can be predicted, the better the derived mapping will be.

Table 1: Execution Time 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

5 Summary

In this paper, we have discussed the mapping of parallel applications with multiple tasks to heterogeneous platforms. We have proposed a mapping strategy based on the match-tree, which reflects the data movement and conversion costs of distributed algorithms and allows for alternative implementations of individual tasks on different machines.

Performance-efficient mappings depend on application characteristics and platform features. Mappings between tasks that can execute concurrently and platforms containing at least one multicomputer (an MPP or cluster of nodes) may involve concurrent execution of tasks on the multicomputer. Minimizing the overall execution time is key to the efficient utilization of these platforms, but depends on determining an optimal assignment of nodes to tasks. We have presented the split-partition algorithm to solve this problem.

The split-partition 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 tasks by transferring nodes from each task to the longest one until no transference is possible. In order to show the correctness of the algorithm, we have proven that a partitioning in which no transference of nodes decreases the overall execution time is an optimal partitioning.

The split-partition 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 split-partition algorithm can also be used to improve the utilization of multicomputers by applications which have relaxed constraints on execution time, e.g., applications which 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 split-partition algorithm which allows transference of nodes only when the yielding application would not increase its time above a pre-determined threshold.

Acknowledgment

We are grateful to Jennifer Schopf for her useful suggestions. We are also grateful for the support of Ken Steube and Cindy Zheng at the San Diego Supercomputer Center, as

well as Elizabeth Simon.

References

[1] C. Anglano, J. Schopf, R. Wolski, and F. Berman, “Zoom: A Hierarchical Representation for Heterogeneous Applications”, University of California, San Diego - Technical Report CS95-451.

[2] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam, “Graphical Development Tools for Network-Based Concurrent Supercomputing”, Proceedings of Supercomputing’91, pp. 435-444.

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

[4] J. Demmel and S. Smith, “Parallelizing a Global Atmospheric Chemical Tracer Model,” Proceedings of the Scalable High-Performance Computing Conference, pp. 718-725, August 1994.

[5] 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.

[6] 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.

[7] S. M. Figueira and F. Berman, “Modeling the Effects of Contention on the Performance of Heterogeneous Applications,” To appear in Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, August 1996.

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

[9] D. A. Hensgen, T. Kidd, and E. Keith, “Adding Rescheduling to and Integrating Condor with SmartNet,” Proceedings of the Heterogeneous Computing Workshop, pp. 4-11, April 1995.

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

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

[12] 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.

[13] 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.

[14] A. Phillips, J. Rosen, and V. Walke, “Molecular Structure Determination by Convex Global Underestimation of Local Energy Minima,” University of Minnesota Supercomputer Institute Research Report UMSI 94/126, July 1994.

[15] J. Schopf. Behavior of Heterogeneous Applications. Parallel Computation Lab, University of California, San Diego. Manuscript in preparation.

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

[17] 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.

[18] 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.

Appendix

In this section, we present and prove Theorem 1, Theorem 2, and Theorem 3, which are used to show the correctness of the split-partition algorithm. We also present and prove Lemma 1, Lemma 2, and Lemma 3, which are used to prove the theorems.

Theorem 1: Let $\hat{n}_i$ be the smallest number of nodes which minimizes *time*(i, n_i), and let S be a partition which assigns, to each task i, at most $\hat{n}_i$ nodes. Assume the execution time function for each task is decreasing to a global minimum. Let P be the longest task. Then, for each $i \neq P$, if transferring nodes from task i to task P does not decrease the execution time of task P, the partitioning is optimal.

Proof: By the premise, the execution time function is decreasing to a global minimum. We assume that task P has at most the smallest number of nodes ($\hat{n}_P$) with which it executes the fastest, and increasing its number of nodes does not decrease its execution time. Hence task P has exactly $\hat{n}_P$ nodes. Since, by definition, the overall time is equal to the time to execute the longest task (the tasks in the split partition are executing concurrently), the overall time cannot be decreased and the partitioning is optimal. ❏

Lemma 1: Assume that, for each task i, *time*(i, *nodes*(i)) is decreasing to the minimum for $nodes(i) \leq \hat{n}_i$. Let $x, y \leq \hat{n}_K$. Then,

$$x = y \Leftrightarrow time(K, x) = time(K, y)$$

and

$$x < y \Leftrightarrow time(K, x) > time(K, y) .$$

Proof: Since the function is decreasing for values smaller or equal to $\hat{n}_K$, this is obvious. ❑

Theorem 2: Let $\hat{n}_i$ be the smallest number of nodes which minimizes *time*(i, n_i). Suppose that S is a partitioning of nodes to tasks, such that each task i has $nodes_S(i) \leq \hat{n}_i$. Let P be the longest task. For each task $i \neq P$, if transferring 1 node from task i to task P increases the time to execute task i to *time*(i, $nodes_S(i)$ - 1), with

$$time(i, nodes_S(i) - 1) \geq time(P, nodes_S(P)), \qquad (1)$$

then the partitioning is optimal.

Proof: We will prove the theorem by contradiction.

Since P is the longest task in partitioning S,

$$time(P, nodes_S(P)) \geq timei, nodes_S(i)) \text{, for any } i \neq P. \qquad (2)$$

Suppose there is another partitioning R, with at most the same number of nodes as partitioning S, such that

$$time(P, nodes_S(P)) > time(Q, nodes_R(Q)), \qquad (3)$$

where Q is longest task in partitioning R, $nodes_R(Q)$ is the number of nodes assigned to task Q in partitioning R, and *time*(Q, $nodes_R(Q)$) is the time to execute task Q with $nodes_R(Q)$ nodes.

Since Q is the longest task with respect to partitioning R,

$$time(Q, nodes_R(Q)) \geq time(i, nodes_R(i)) \text{, for any } i \neq Q. \qquad (4)$$

Since P and Q can be either the same or different tasks, we consider each case:

- **Case 1:** $P = Q$ (i.e., the same task is the longest in both partitionings).

Since each task i has at most $\hat{n}_i$ nodes, by Lemma 1, decreasing its time means increasing its number of nodes. Therefore, since $P = Q$, from Lemma 1 and (3)

$$nodes_R(P) > nodes_S(P) \ . \tag{5}$$

Since partitioning R has at most the same number of nodes as partitioning S, there is at least one task $K \neq P$, such that

$$nodes_R(K) \leq nodes_S(K) - 1 \ , \tag{6}$$

where $nodes_R(K)$ is the number of nodes assigned to task K in partitioning R, and $nodes_S(K)$ is the number of nodes assigned to task K in partitioning S.

Since $P = Q$ and P is the longest task in partitioning R, from (4),

$$time(K, nodes_R(K)) \leq time(Q, nodes_R(Q)) \ . \tag{7}$$

Since no task in partitioning S can yield a node and decrease the overall time, we get from (1)

$$time(P, nodes_S(P)) \leq time(K, nodes_S(K) - 1) \ . \tag{8}$$

Since each task i has at most $\hat{n}_i$ nodes, from Lemma 1 and (6) we have

$$time(K, nodes_S(K) - 1) \leq time(K, nodes_R(K)) \ . \tag{9}$$

By transitivity, from (8) and (9),

$$time(P, nodes_S(P)) \leq time(K, nodes_R(K)) \ . \tag{10}$$

By transitivity, from (7) and (10),

$$time(P, nodes_S(P)) \leq time(Q, nodes_R(Q)) \ , \tag{11}$$

but this contradicts (3). Hence there is no optimal alternative partitioning R satisfying the premises such that the longest task in R is identical to the longest task in S.

• **Case 2:** $P \neq Q$ (i.e., each partitioning has a different longest task).

Recall that P is the longest task for S and Q is the longest task for R.

From (4), for $i = P$,

$$time(Q, nodes_R(Q)) \geq time(P, nodes_R(P)) . \tag{12}$$

By transitivity, from (3) and (12),

$$time(P, nodes_S(P)) > time(P, nodes_R(P)) . \tag{13}$$

Since each task i has at most $\hat{n}_i$ nodes, by Lemma 1 and (13),

$$nodes_S(P) < nodes_R(P) . \tag{14}$$

Since partitioning R has at most the same number of nodes as partitioning S, there is at least one task $K \neq P$, such that

$$nodes_R(K) \leq nodes_S(K) - 1 . \tag{15}$$

Since Q is the longest task in partitioning R, from (4),

$$time(K, nodes_R(K)) \leq time(Q, nodes_R(Q)) . \tag{16}$$

According to partitioning S, no task can yield a node and decrease the overall time. Hence, from (1),

$$time(P, nodes_S(P)) \leq time(K, nodes_S(K) - 1) . \tag{17}$$

Since each task i has at most $\hat{n}_i$ nodes, from Lemma 1 and (15),

$$time(K, nodes_S(K) - 1) \leq time(K, nodes_R(K)) . \tag{18}$$

By transitivity, from (17) and (18),

$$time(P, nodes_S(P)) \leq time(K, nodes_R(K)) . \tag{19}$$

By transitivity, from (16) and (19),

$$time(P, nodes_S(P)) \le time(Q, nodes_R(Q)) \ , \tag{20}$$

but this contradicts (3). Hence the partitioning S is optimal. [End of Theorem 2] ❏

Lemma 2: Tasks that cannot yield a node at some point during the traversal of the array (loop 8-26) will not be able to yield a node until the end of the traversal.

Proof: During the traversal of the array, the overall time for the partitioning never increases. If transferring a node from task K to the longest task does not decrease the current overall time, task K will never be able to yield a node because its time with one less node will always be greater than or equal to the overall time during the remainder of the traversal. ❏

Lemma 3: During the traversal of the array (loop 8-26), no task yields nodes after having been the longest task.

Proof: During the traversal of the array, the overall time for the partitioning never increases. Suppose task P is the longest task. If it receives a node, the overall time decreases and, after that, task P will never be able to yield a node because its time with one less node will always be greater than the overall time during the remainder of the traversal. ❏

Theorem 3: Traversing the array of tasks once is sufficient.

Proof: Lemma 2 shows that a task which cannot yield nodes to the longest task at some point during the traversal of the array, will not be able to yield nodes until the end of the traversal. It is clear that since each task needs to be checked exactly once, traversing the array once is enough if the position of each task in the array does not change. However, the position of the tasks in the array may change during the traversal, so we also need to show that all the tasks are checked for possible transferences until no additional transference is possible. We show below that changes in position due to transference of nodes between tasks do not affect the traversal. Two types of changes are possible:

- **Case 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 (Lemma 1), 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 9. In Figure 9.(a), the algorithm is checking the third position on the array, which contains task *B*. Figure 9.(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.

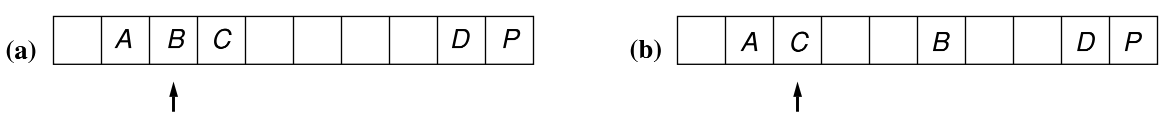

Figure 9: Possible changes of positions in the array of tasks.

- **Case 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 10 illustrates this case. In Figure 10.(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 10.(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 10.(c)), the algorithm will check task *P* (which cannot yield nodes according to Lemma 3) and go on. If it executes faster than task *A* (Figure 10.(d)), the algorithm will recheck task *A* (which cannot yield nodes according to Lemma 2) and go on. Since task *P* cannot yield a node (Lemma 3), it does not need to be checked.

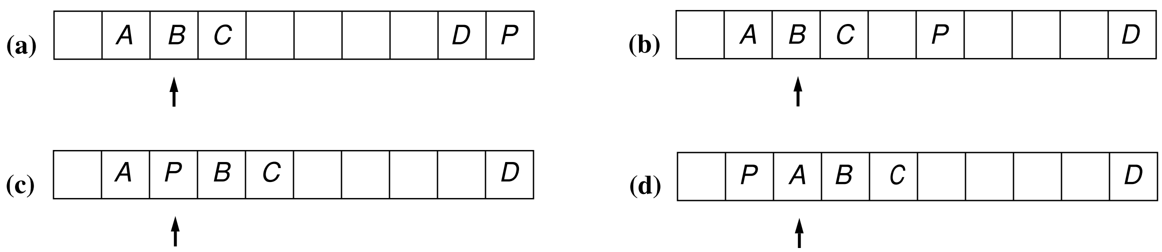

Figure 10: Possible changes of positions in the array of tasks.

In summary, tasks on the left of the current position being checked have already been checked and do not need to be rechecked. Tasks in the current position or on the right will still be checked until no transference is possible. Therefore, every task is checked until no node transference is possible and, according to Lemma 2, traversing the array once leaves no node transference to be done. ❑