A COMPARISON STUDY OF HEURISTICS FOR MAPPING PARALLEL ALGORITHMS TO MESSAGE-PASSING MULTIPROCESSORS

Marios D. Dikaiakos Anne Rogers Ken Steiglitz
Department of Computer Science
Princeton University
email: mdd@princeton.edu

Abstract

This paper presents a comparison study of popular clustering and mapping heuristics which are used to map task-flow graphs to message-passing multiprocessors. To this end, we use task-graphs which are representative of important scientific algorithms running on data-sets of practical interest. The annotation which assigns weights to nodes and edges of the task-graphs is realistic. It reflects current trends in processor, communication channel, and message-passing interface technology and takes into consideration hardware characteristics of state-of-the-art multiprocessors. Our experiments show that applying realistic models for task-graph annotation affects the effectiveness and functionality of clustering and mapping techniques. Therefore, new heuristics are necessary that will take into account more practical models of communication costs. We present modifications to existing clustering and mapping algorithms which improve their efficiency and running-time for the practical models adopted.

1 Introduction

In this paper we present a comparison study of popular clustering and mapping heuristics which are used to map task-flow graphs to message-passing multiprocessors. To this end, we use task-graphs which are representative of two important algorithms for the *N-Body* problem, running on data-sets of practical interest. The annotation which assigns weights to the nodes and the edges of the task-graphs is realistic. It reflects current trends in processor, communication channel, and message-passing interface technology and takes into consideration hardware characteristics of state-of-the-art multiprocessors. Our experiments show that applying realistic models for task-graph annotation affects the effectiveness and functionality of clustering and mapping techniques. Therefore, new heuristics are necessary that will take into account more practical models of communication costs. We present modifications to existing clustering and mapping algorithms which improve their efficiency and running-time with the practical models adopted.

Task-graphs are derived with FAST, a software system that we built to evaluate the execution of parallel scientific algorithms on message-passing systems [DRS93a, DRS94]. These graphs are a special case of the *data dependence graphs* (DDG's) that are used frequently as abstract representations of parallel programs [GVY90, Pol88, SH86, YG92, Sar89, RSL93]. The nodes of DDG's correspond to single program instructions or sets of instructions, depending on the DDG-granularity desired. Their arcs correspond to dependences, which enforce a partial order of execution on program statements.

A key issue that arises in systems employing data dependence graphs is the execution of these graphs on the processors of a parallel computer. There are many approaches for addressing this problem, most of which can be classified as *static* or *dynamic*. Static schemes apply in systems where the DDG's can be constructed before program execution. In that case, the user-program or the compiler can take advantage of information pertinent to the DDG for making decisions that will guide the assignment of graph-nodes to different processors, and the scheduling of tasks within each processor [aDA91, YG92]. It is not always possible, however, to create the DDG's before the program execution. In that case, execution of DDG's is accomplished with dynamic schemes that are enforced through the operating system or the hardware.

In this paper we examine algorithms used in static schemes. Such algorithms assume for simplicity that the processors of a parallel system form a clique interconnection topology (fully-connected network). Mapping is usually accomplished in two phases [GVY90, SH86, Sar89]:

1. The *clustering* or *internalization* phase, seeks to minimize communication overhead and improve parallel time by deciding that certain tasks must go together on the same processor, even if other processors are available.

2. The *mapping* or *processor assignment* phase, maps the groups of tasks formed by the clustering phase to the processors of the parallel architecture at hand. At the same time, it seeks to preserve a small parallel time.

In conjunction with clustering and mapping, it is necessary to perform *scheduling* of tasks that are assigned to the same cluster.

The organization of this paper is as follows: in the next section we give the graph-theoretical framework that we use to implement and evaluate clustering and mapping heuristics. In Sections 3 and 4 we describe the clustering and mapping heuristics studied and suggest modifications that will improve their effectiveness and performance, given the practical annotation model adopted. Section 5 presents simulation results and comparison-studies, and Section 6 gives our conclusions.

2 Modeling parallel executions

The task-graphs used in our study are called *parallel-execution graphs* and follow the *Macro-Dataflow* model of computation [Sar89]. In this model, each task starts executing upon receipt of all incoming messages and continues to completion without interruption. Upon completion, it forwards its results to adjacent tasks. Each node in a parallel-execution graph is assigned the computation time of the corresponding task and each edge is assigned the latency of the respective message. A parallel-execution graph is an abstraction of the parallel execution, which enables us to estimate parallel time and available parallelism easily, and study the mapping of the parallel computation onto some realistic message-passing multiprocessor. Parallel-execution graphs are formally defined as follows:

$$G_{pe} \;=\; G(V,\ E_{pe},\ proc,\ T,\ D)$$

where:

1. V is the set of tasks.

2. $E_{pe} \;=\; E \cup E_{sch}$ is the set of edges. Edges in E correspond to explicit messages, and represent program-determined dependences between tasks. E_{sch} is a set of edges introduced in the graph to define the order of execution among tasks mapped on the same processor and with no program-determined dependences between them.

3. *proc*: A mapping from the set of task nodes V to the set of processors P: $\forall\ v \in V$, $proc(v)$ gives the processor in P that executes task v.

4. $T(v),\ v \in V$ is the time it takes processor $proc(v)$ to perform v's computations.

5. $D(e),\ e = (u, v) \in E$ is the weight assigned to edge e. $D(e)$ denotes the time-interval between the time that task u finishes its execution and the time that task v gains access to the data carried by edge e. If u and v are mapped onto different processors, $D(e)$ is equivalent to the interval between the time when $proc(u)$ has finished executing task u, and the time when message e has been loaded into the buffers of the destination processor's $proc(v)$ network interface. For a single-hop message, it is:

$$D(e) \;=\; t_{delay}(e) \;+\; S_{ov}(e) + W(e)/B + t_{congestion}(e) + R_{ov}(e)$$

where: $t_{delay}(e)$ is the delay between the time the sending processor issues the *Send* instruction initiating message e, and the time that this processor starts loading the message body to the buffers of its network interface. $S_{ov}(e)$ is the time it takes the sending processor to load its network interface's output buffers with the contents of message e and with control information (setup cost). $W(e)$ is the number of bytes carried by message e, B is the bandwidth of the communication links (in bytes per second), $t_{congestion}$(e) is the time e spends waiting in busy queues of the interconnection network, and $R_{ov}(e)$ is the time it takes a message to be loaded in the input buffers of the receiving processor's network interface. Additionally, we use $\delta(e)$ to denote the time it takes the message to propagate through the communication channels and then to be loaded into the input buffers of its destination's network interface. For one-hop messages this is equal to : $W(e)/B + t_{congestion}(e) + R_{ov}(e)$.

On the parallel-execution graph we can now define the *Parallel Time* as the weight of its critical path, i.e, of the path with the largest sum of node and edge weights.

Message-Passing Interface Primitives

Point-to-point communications in parallel systems are implemented with *Send* and *Receive* primitives issued by parallel tasks. These primitives can be characterized as *blocking* or *non-blocking*, and as *synchronous* or *asynchronous* [Cyp93]. Such characterizations determine the point in time when a communication primitive returns control to the task that called it. Also, they define the semantics of communication, and affect its performance. In the Macro-Dataflow model of computation edges in E represent pairs of *Send* and *Receive* primitives. *Send*'s can be either blocking or non-blocking, and synchronous or asynchronous. *Receive*'s must be blocking because of the definition of Macro-Dataflow. According to the non-blocking communication paradigm, messages are dispatched *simultaneously* at the end of the

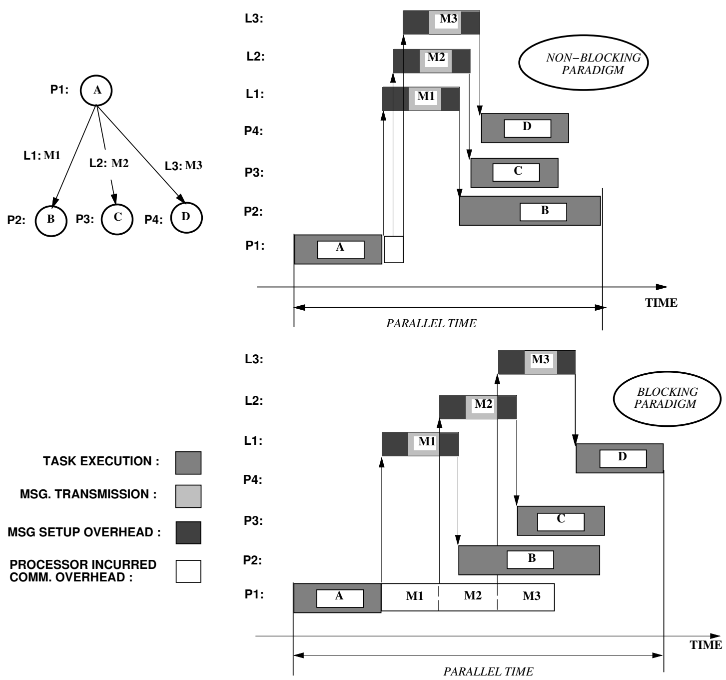


Figure 1: Blocking vs. Non-blocking *Send*'s.

execution of a task. In contrast, according to the blocking paradigm, messages are transmitted *serially* from tasks with no overlap between the loading of a buffer and the subsequent message-dispatches or computation (see Figure 1). Therefore, the choice of message-passing interface primitive affects the annotation of task-graph edges and, hence, the clustering and mapping steps taken.

3 Clustering

Clustering specifies the sequential units of computation in a parallel program by mapping tasks to clusters. A cluster is a set of tasks that execute sequentially on the same processor. The principle goal of clustering is to achieve the minimum parallel time for a given task graph on a clique architecture, with as many processors as tasks ("abundant" clique). If communication overhead were zero, the trivial solution to clustering would assign each task to a different processor of an "abundant" clique. In the realistic case, however, a parallel execution that assigns every node of a task-graph to a different processor of an "abundant" clique might not achieve minimum completion time because of communication delays and overhead. For example, in Figure 2 (left), each task is assigned to a different processor and the parallel time of the execution graph is 24. In contrast, in Figure 2 (right), nodes A, B, and D are clustered within the same processor and the parallel time is only 16. This is because clustering eliminates the communication cost of messages (A, B) and (B, D).

Formally, *clustering* is the problem of partitioning the nodes of a parallel-execution graph G_{pe} into clusters, and deriving the clustered parallel-execution graph with the *shortest* parallel time among all possible clustered graphs G^c_{pe} mapped on "abundant" cliques. It has been proven that finding the optimal clustering of a directed acyclic graph that follows the Macro-Dataflow model of computation is

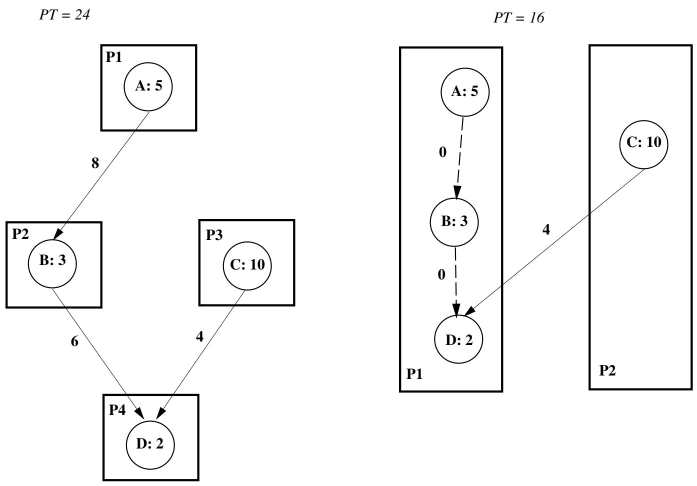


Figure 2: Different clustering choices.

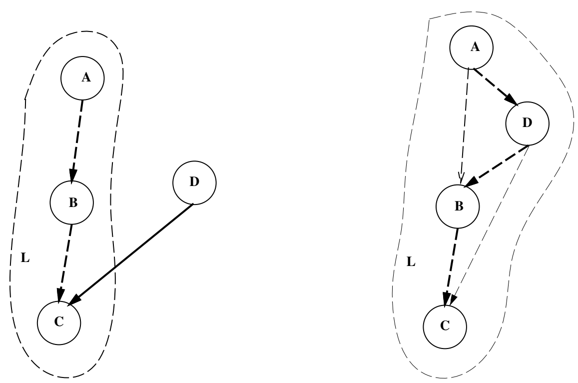


Figure 3: Scheduling edges. The bold arrows denote the sequential paths of execution in the clusters.

NP-hard in the strong sense, if the cost function is the minimization of parallel time of the graph on an "abundant" clique architecture [Sar89]. A number of heuristics have been developed to cope with the clustering problem [ERL90, KB88, Sar89, YG91].

Clustering heuristics applied to G_{pe} will update its *proc* information to reflect the formation of clusters. If, for instance, nodes u and v are clustered within the same cluster L, then $proc(u) = proc(v) = L$. Furthermore, clustering alters E, the set of edges of G_{pe}, by introducing new "scheduling" edges that express the scheduling priorities among nodes belonging to the same cluster. For example, in Figure 3, cluster L merges with node D. If task D is scheduled to run after task A and before task B, the edges (A, D) and (D, B) are inserted in the clustered graph to determine the new schedule.

Finally, clustering heuristics change the weights assigned to the edges of G_{pe}. For example, we consider a node $u \in V$ that sends $n + 1$ messages to nodes w_1, w_2,..., w_k, v, w_{k+1},..., w_n, in that order (see Figure 4). Assume that $proc(w_i) \neq proc(w_j) \neq proc(u) \neq proc(v)$, $\forall i \neq j$. If the clustering heuristic assigns nodes u and v to the same cluster, the weights of the outgoing edges (u, w_1),..., (u, w_k) of u will

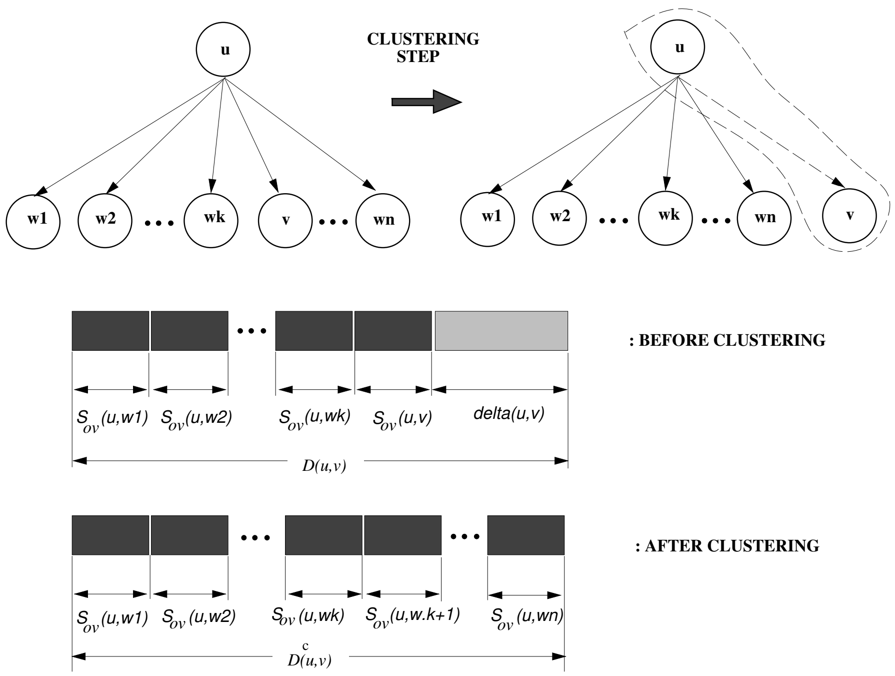


Figure 4: Edge weights (blocking *Send*'s).

remain the same. The weight of (u,v) will be changed from:

$$D(u,v) \;=\; (\sum_{i=1}^{k} S_{ov}(u,w_i)) \;+\; S_{ov}(u,v) \;+\; \delta(u,v).$$

to:

$$D^c(u,v) \;=\; \sum_{i=1}^{n} S_{ov}(u,w_i)$$

(see Figure 4). The weights of edges (u,w_{k+1}),...,(u,w_n) will be reduced to

$$D^c(u,w_i) \;=\; D(u,w_i) \;-\; S_{ov}(u,v), \quad i = k+1,\ldots,n.$$

These formulas correspond to the case where the message-passing interface of the "abundant" clique provides a *blocking Send* communication primitive. Most clustering heuristics, however, have been designed with the assumption that, after clustering, $D^c(u,v)$ will be equal to zero and $D^c(u,w_i)$ will be the same as $D(u,w_i)$.

4 Clustering Heuristics

The clustering heuristics examined here perform a number of refinement steps on the input parallel-execution graph. Each step performs a refinement on the output of the previous clustering step by merging two clusters, and scheduling their tasks within the newly formed clusters. In the initial parallel-execution graph, each task-node is a cluster by itself. The heuristics complete and report a final clustering when an end-condition is satisfied.

We focus on *edge-zeroing* heuristics with *no backtracking*. These algorithms proceed by merging *connected* nodes of the parallel-execution graph. Assigning two connected nodes to the same cluster eliminates the message that corresponds to the edge connecting them. After clustering, the message will be carried out through local memory *Write*'s and *Read*'s at the memory of the processor that executes the cluster. There is no backtracking, that is, once a cluster has been formed at one step of the heuristic, it cannot be split at a later step.

Various algorithms belonging to this class of clustering methods can be characterized with respect to:

1. The method for choosing which edge to eliminate.
2. The end-condition of the heuristic.
3. The scheduling heuristic employed when merging two clusters into a sequential thread of execution.

The choice of scheduling heuristic is orthogonal to the method for zeroing edges and to the end-condition. Here we give concise presentations of a number of clustering heuristics. A comprehensive discussion on clustering can be found in [GY92].

4.1 Sarkar's Clustering Algorithm

Sarkar's heuristic clusters a parallel-execution graph in a number of steps described below [Sar89]:

1. Sort the edges $e \in E$ of the graph in descending order of their weights $D(e)$.
2. Merge the two clusters that include the head and tail node of the edge with the greatest weight, if this change does not increase parallel time.
3. Repeat step 2 until all edges are scanned.

It is not difficult to see that the complexity of Sarkar's heuristic is $O(|E| \cdot (|V| + |E|))$. This results in very high execution times for large graphs. Therefore, we also implemented a variation of Sarkar's method that sorts the edges in descending order of their weights and examines only a percentage of them, starting from the one with the largest weight.

4.2 Kim and Browne's Algorithm

Kim and Browne's method takes a different approach to clustering [KB88]:

1. Mark all edges in the parallel-execution graph as *unexamined*.
2. Find the *critical path* in the graph composed of unexamined edges only. This is the path with the longest cumulative weight in the graph. The cumulative weight of a path $(u_1, u_2), (u_2, u_3), \ldots, (u_{n-1}, u_n)$ is equal to $\sum_{i=1}^{n-1} \Big(T(u_i) + D(u_i, u_{i+1}) \Big) + T(u_n)$.
3. Merge in the same cluster the nodes belonging to the critical path and mark all edges incident to nodes of the critical path as *examined*.

4. Apply steps 2 and 3 to the subgraphs formed by nodes and unexamined edges, until all edges are examined.

The complexity of Kim and Browne's heuristic is $O(|V| \cdot (|V| + |E|))$, since there are at most $|V|$ connected components in a graph and it takes $O(|V| + |E|)$ time to find the critical path in each component.

4.3 Greedy Dominant Sequence Algorithm

The clustering algorithm by Yang and Gerasoulis [YG91], identifies at every step the critical path of the graph, named the *Dominant Sequence* (*DS*). The heuristic chooses one edge belonging to the DS and merges the clusters of its adjacent nodes, if this decision leads to a shorter parallel time. After the clustering, the algorithm computes the new DS. The complexity of Yang and Gerasoulis' heuristic is $O((|E| + |V|) \cdot log|V|)$. We implemented a simpler, greedy version of this algorithm, which we call *Greedy Dominant Sequence* (*GDS*) algorithm:

1. Identify the Dominant Sequence of the graph.
2. Choose the edge of the Dominant Sequence whose elimination results in the largest decrease of parallel time. Merge the clusters of the nodes adjacent to the selected edge.
3. Repeat Steps 1 and 2 until there is no edge in the DS whose elimination can decrease parallel time.

Identifying the Dominant Sequence requires a depth-first search of the graph which takes $O(|E| + |V|)$ time. Choosing which edge of the Dominant Sequence to eliminate takes time proportional to the number of edges in the Dominant Sequence, that is, $O(|V|)$. The algorithm will perform $O(|V|)$ clusterings and, therefore, the total complexity of the Greedy Dominant Sequence is $O(|V| \cdot (|E| + |V|))$.

4.4 Greedy-Linear Algorithm

Kim and Browne's heuristic improves parallel time in the case where a simple scheme is used to assign weights to edges, and "elimination" of an edge results in zeroing its weight. Under the more realistic scheme employed in our study, however, Kim and Browne's heuristic may result in clustered graphs with larger parallel times than the unclustered ones. With this consideration in mind, we modified this heuristic and introduced a version that we call *Greedy-Linear*. This algorithm is called "linear" because, as in Kim and Browne's method, it outputs clusters that are linear chains of task-nodes. The heuristic works as follows:

1. Mark all edges in the parallel-execution graph as *unexamined*.
2. Find the *critical path* in the graph composed of unexamined edges only.
3. For every edge of the critical path, cluster its adjacent nodes only if this does not result in a larger parallel time. Mark all the edges incident to nodes of the critical path as *examined*.
4. Apply steps 2 and 3 to the subgraphs formed by nodes and unexamined edges, until all edges are examined.

Testing whether the clustering of an edge results in a larger parallel time can be accomplished in constant time, without having to recompute the parallel time of the graph. Therefore, the complexity of this algorithm is $O(|V| \cdot (|V| + |E|))$ as well.

5 Mapping

Clustering produces a clustered parallel-execution graph with a number of clusters usually much larger than the number of available processors of the target architecture. *Optimal Mapping* is the problem of finding an assignment of clusters to processors, leading to a parallel time shorter than the times derived by all other assignments, for the given number of processors [NT93]. The Optimal Mapping problem of a clustered directed acyclic graph has been proven to be NP-Complete [Sar89]. In this section, we present a number of heuristics used to map parallel-execution graphs following the Macro-Dataflow model, to a given set of processors.

5.1 Sarkar's Algorithm

Sarkar's heuristic is a modified version of the Priority List Scheduling algorithm [SH86]. It uses a list, *pblock*, of size P, where P is the number of available processors. *pblock* entries are initially empty. When the algorithm completes, $pblock[i]$ contains the tasks assigned to processor i, for $i = 1, \ldots, P$. The algorithm creates a priority list of task-nodes, according to a topological-sort ordering of the graph. Then, at each step, it processes the next node T in the priority list. If T has not already been assigned to a processor, the algorithm performs the following tasks:

1. Choose a processor i, such that, the merging of clusters $proc_c(T)$ and $pblock[i]$ will result in a parallel time shorter than the one derived from the merging of $proc(T)$ with any other cluster $pblock[j]$.
2. Merge clusters $proc_c(T)$ and $pblock[i]$, and assign the result to $pblock[i]$.
3. Assign all task-nodes of cluster $proc[T]$ to processor i.
4. Reduce the number of clusters by one.

The algorithm completes when the total number of clusters in the graph becomes equal to P. It is not difficult to see that its computational complexity is $O\big(P \cdot |proc| \cdot (|V| + |E|)\big)$, where $|proc|$ is the initial number of clusters.

5.2 SNC Heuristic

Sarkar's mapping algorithm is slow because of the large constants involved in its complexity. We implemented a modified version to improve its running time, although without achieving a better asymptotic complexity. This version follows exactly the same steps as the original heuristic. It does not, however, take into consideration communication costs when calculating parallel time. We call it *SNC*, that is, *Sarkar's algorithm with No Communication Costs.*

5.3 Yang and Gerasoulis' Algorithm

In [YG92], Yang and Gerasoulis introduced a fast heuristic for mapping a clustered graph to the processors of a parallel system. This algorithm seeks to optimize the load-balancing of the available processors. It is comprised of four steps:

1. Estimate the average processing time, A, of the processors, as the sum of the processing times of all clusters, over the number P of processors.

2. Sort the clusters in an increasing order of their loads.

3. Assign each cluster with a processing time higher than the average A to a different processor.

4. Use a wrap mapping for the remaining clusters, that is, number these clusters from 1 to their total number; then, assign each of them on the processor whose number is equal to the number of the cluster modulo P.

The complexity of this method is $O(|V| \cdot \log |V| + |E|)$.

5.4 Priority List Scheduling Heuristics

We also implemented two versions of *Priority List Scheduling* [Cof76], which apply directly to *non-clustered* graphs. In Priority List Scheduling, each task is assigned a priority. The tasks are inserted in a priority list according to the descending order of their priorities. Subsequently, they are assigned to processors following the order defined by the priority list. Before presenting the scheduling heuristics implemented, we introduce some useful notation. Given a directed-acyclic graph $G = G(V, E)$, we denote by V_i the set of "input" nodes, that is, nodes in V with no incoming edges. With V_o, we denote the set of "exit" nodes, that is, nodes in V with no outgoing edges. We define *ptime(u)* as the total weight of the longest path from node u to the nodes of V_o. Similarly, we define *stime(u)* as the total weight of the longest path, among all possible paths going from the nodes of V_i to u, not including $T(u)$. Finally, we define the *level* of a node in the graph as follows: $level(u) = \max_{\pi \in \Pi(V_i,u)} ||\pi||$, where $\Pi(V_i, u)$ represents the set of all possible paths in G from the nodes in V_i to node u, and $||\pi||$ represents the number of edges in path π, that is, the length of π.

The first heuristic orders nodes of the graph according to the *Topological Sort/Earliest Task First* (*TS/ETF*) approach [MT91]. It performs a topological sort of the parallel-execution graph and assigns *level* values to its nodes. If node u precedes node v in the topological-sort order, that is, $level(u) < level(v)$, then u will be assigned a higher priority than v. If, however, *level(u)* equals *level(v)*, then *TS/ETF* assigns a higher priority to the node with the smaller *stime* value. The relative priorities of nodes with equal *level* and *stime* values, is assigned by *TS/ETF* randomly.

The second heuristic implements the *CP/MISF* principle [KN84]. It uses topological sort and critical-path information to construct a priority list of nodes. Nodes are introduced in the priority list according to the descending order of their *ptime* values. For nodes with the same *ptime* value, *CP/MISF* assigns a higher priority to the ones with the larger number of immediate successors, that is, with the larger number of outgoing edges.

After constructing the priority lists, the heuristics traverse them and map each task to the processor that will start executing it at the earliest possible time.

6 Scheduling

The scheduling problem arises during the merging of two clusters, when their tasks have to be ordered according to some sequential order of execution. A scheduling algorithm should specify an ordering of tasks that achieves the shortest parallel time and, at the same time, complies to existing precedence constraints. For general directed-acyclic parallel-execution graphs, the problem of finding the optimal task sequences that minimize overall parallel time is NP-Complete [SH86]. Consequently, for our experiments, we implemented the *CP/MISF* scheduling heuristic, which is based on the principles of Priority List Scheduling. The results do not change when using *TS/ETF*.

7 Experimental Results

In this section we present experimental results using the clustering, scheduling, and mapping presented in the previous sections. We give data derived when using these heuristics on parallel-execution graphs representative of two algorithms solving the *N-Body* problem. The first graph has 1445 task-nodes and $12,860$ message-edges. It corresponds to the parallel computation of one time-step of the *Fast Multipole Method* (FMM) on 1000 bodies [GG87, DRS93b]. The second graph has 2532 task-nodes and 12,918 message-edges. It represents the parallel computation of one time-step of the *Barnes-Hut* (BH) algorithm on 1000 bodies [BH86, DRS93b]. Further experiments, performed on task-graphs representing other instances of the two algorithms, corroborate the results presented in the following sections.

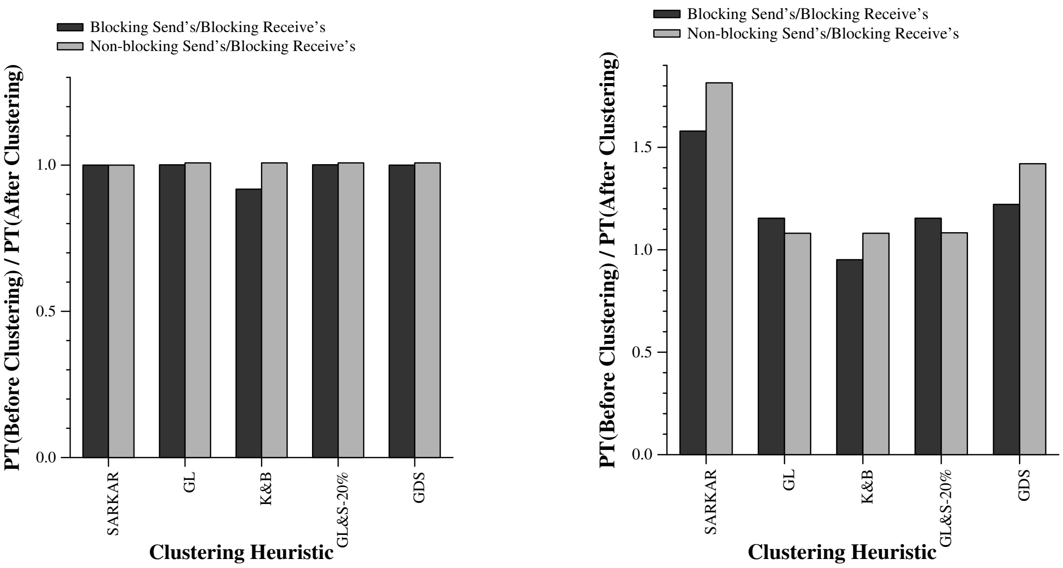
The clustering algorithms used the *CP/MISF* heuristic for scheduling. For the annotation of the task-graphs we used values representative of Intel's $iPSC/860$ multiprocessor, which has very high S_{ov} and R_{ov} values (see Table 1).

Machine	S_{ov} (*in* μsec)	R_{ov} (*in* μsec)	Bandwidth
iPSC/860			(*in Bytes per* μsec)
msg $<$ 100bytes	37.25	37.25	2.3
msg $>$ 100bytes	85.99	85.99	2.52

Table 1: Hardware parameters used in our experiments.

7.1 Clustering

Figure 5 shows the ratio of the parallel time of the clustered parallel-execution graphs over the parallel time of the unclustered graphs, for a number of different clustering techniques and for two message-passing interface paradigms (blocking or non-blocking *Send*'s): Sarkar's method; Greedy-Linear algorithm (*GL*); Kim and Browne's method (*K&B*); running Greedy-Linear on the graph and then applying



Fast Multipole Method task-graphs.

Modified Barnes-Hut task-graphs.

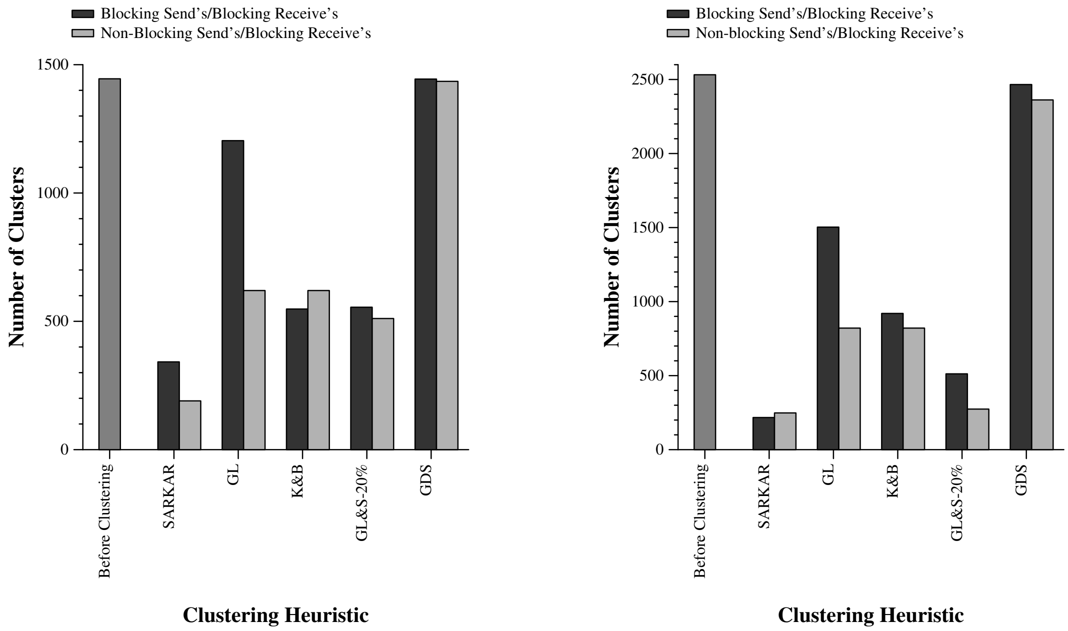
Figure 5: Effects of clustering to parallel time.

Sarkar's heuristic for only the 20% of the edges (*GL&S-20%*), and Greedy Dominant Sequence approach. In most cases, the clustering heuristics do not improve the parallel time of the clustered graph with respect to the parallel time of the unclustered graph. Only when applying *GDS* and Sarkar's heuristics to the task-graph of the Barnes-Hut algorithm, do we get improvements larger than 20% and 50% (respectively). This remark holds for both message-passing interface paradigms adopted (blocking or non-blocking *Send*'s).

The diagrams in Figure 6 present the numbers of the clusters produced by the different clustering heuristics. This is an interesting metric, since the performance of mapping algorithms depends on the number of clusters generated by the clustering heuristics which precede mapping; clearly, mapping is faster for clustered graphs with fewer clusters. As expected, applying Sarkar's heuristic results in the smallest number of tasks. The reason is that the algorithm considers all the edges in the graph for "zeroing."

In contrast, the Greedy Dominant Sequence method results in a number of clusters almost identical to the initial number of tasks. *GDS* eliminates only edges belonging to the Dominant Sequence (that is, the critical path of the parallel-execution graph) and, thus, clusters few of the tasks belonging to the DS. Under the realistic model used here, however, clustering these tasks does not necessarily alter the DS. Hence, the algorithm can complete without further clustering.

The Greedy-Linear (*GL*) and Kim&Browne's heuristics do not check the Dominant Sequence only. Instead, after performing clustering on the DS, they proceed by clustering tasks belonging to the critical paths of the subgraphs formed when deleting edges adjacent to the initial DS. The *GL* method results in a larger number of clusters in the case of blocking *Send*'s than in the case of non-blocking *Send*'s. This is due to the fact that "zeroing" an edge on the critical path of a parallel-execution graph, will always result in a smaller cumulative weight for this path, if the message-passing interface paradigm provides for non-blocking *Send*'s. This is not always the case with blocking *Send*'s and, thus, there are fewer



Fast Multipole Method task-graphs. Modified Barnes-Hut task-graphs.

Figure 6: Number of clusters.

opportunities for the clustering heuristic to perform effective clusterings.

Kim and Browne's method performs the clustering of linear chains of tasks, even if such an alteration results in a larger parallel time. Therefore, the cluster-numbers reported for this algorithm are relatively small, both for the blocking and the non-blocking paradigms.

Finally, *GL&S-20%* reports cluster numbers which are proportional to the numbers reported by Sarkar's algorithm. This is expected since, in its first pass, the method applies *GL* to the graph. This does not decrease the number of clusters substantially. The second pass applies Sarkar's heuristic, but only for the 20% heaviest edges.

In Figure 7, we present a plot of execution-time measurements for the various clustering heuristics examined. The execution times represent measurements on FAST simulations of the FMM running on a DEC-Alpha workstation. As expected, Sarkar's algorithm is substantially slower than the other heuristics.

7.2 Mapping

To compare the mapping algorithms implemented in FAST, we applied them to the clustered parallel-execution graphs derived from the examples of the previous section, and mapped the clusters to 16 processors connected in a clique topology. Experiments with different numbers of processors resulted in similar conclusions. In Figure 8, we present speedups for twelve different combinations of clustering and mapping algorithms. The speedup is defined as the ratio of the sequential time of the task-graph, that is, the sum of the weights of all the tasks, over its parallel-time. It represents a measure of the efficiency of the parallel computation described by the parallel-execution graph. Therefore, it can be used as a metric for the effectiveness of the mapping techniques applied.

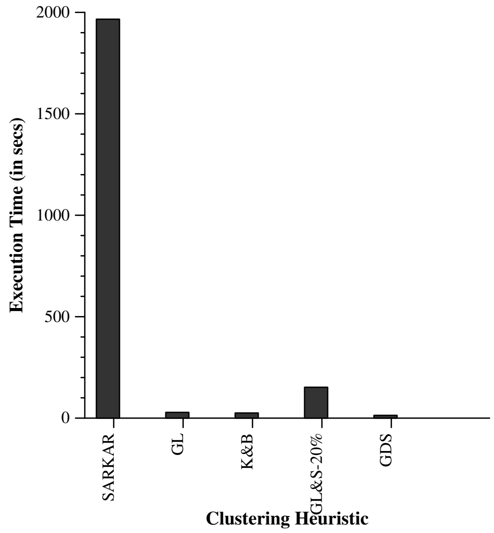


Figure 7: Execution times of clustering heuristics.

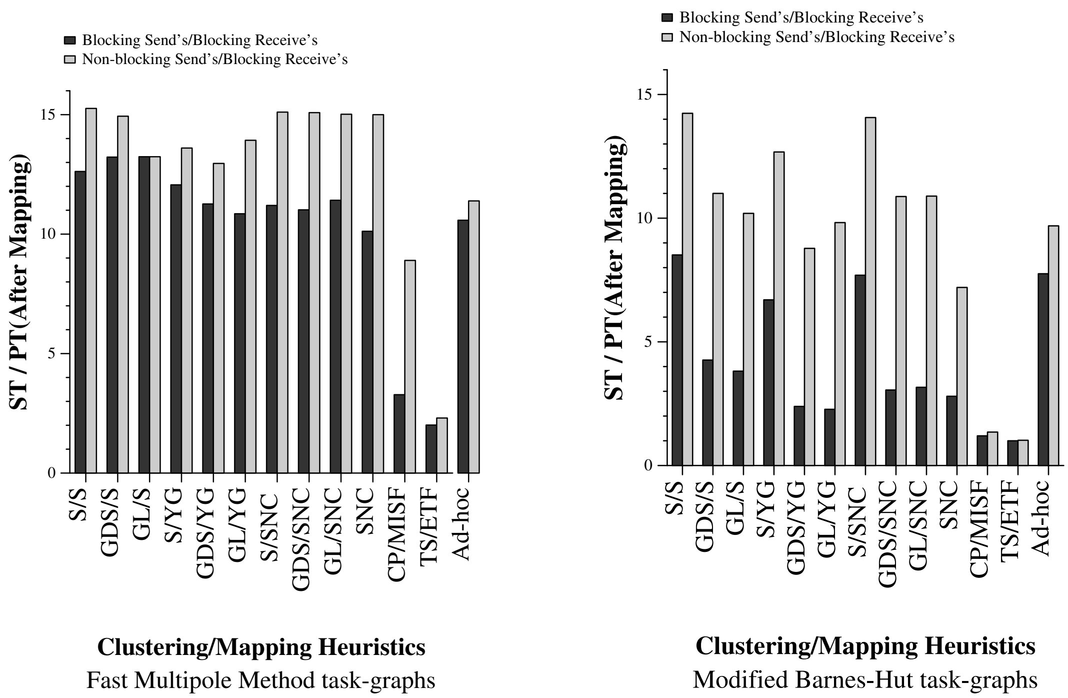


Figure 8: Speedups for different clustering-mapping strategies (16 processors).

Notation	Clustering Algorithm	Mapping Algorithm
S/S	Sarkar's	Sarkar's
GDS/S	Greedy Dominant Sequence	Sarkar's
GL/S	Greedy-Linear	Sarkar's
S/YG	Sarkar's	Yang&Gerasoulis
GDS/YG	Greedy Dominant Sequence	Yang&Gerasoulis
GL/YG	Greedy-Linear	Yang&Gerasoulis
S/SNC	Sarkar's	Sarkar's - No Communication Cost
GDS/SNC	Greedy Dominant Sequence	Sarkar's - No Communication Cost
GL/SNC	Greedy-Linear	Sarkar's - No Communication Cost
SNC	none	SNC
CP/MISF	none	Priority List Scheduling, CP/MISF
TS/ETF	none	Priority List Scheduling, TS/ETF
Ad-hoc	none	none

Table 2: Clustering and Mapping algorithms' notation.

Table 2 explains the notation used in the plot of Figure 8. In addition to the results corresponding to combinations of clustering and mapping heuristics, we present speedups obtained with an ad-hoc approach for partitioning and parallelizing the problems under consideration. From Figure 8, we can see that the various combinations of heuristics perform differently for the two task-graphs examined. This difference is due to the different characteristics of the task-graphs: the computation-to-communication ratio (average task execution time over the average message delay) is much higher in the task-graph that corresponds to the Fast Multipole Method than in the task-graph corresponding to the Barnes-Hut algorithm.

In the case of the task-graph representing a parallel execution of an instance of the Fast Multipole Method (Figure 8, left) the measured speedup depends primarily on the choice of the mapping heuristic. More specifically, Sarkar's mapping method achieves the best results regardless of the clustering heuristic adopted. The *SNC* approach performs almost as well as Sarkar's method, although it disregards communication costs in the parallel-execution graph. Therefore, the reason that the measured speedups are lower than the ideal linear speedups is not communication overhead but lack of load-balancing and the data-dependences in the task-graph that result in almost sequential portions of the execution. The Yang and Gerasoulis mapping algorithm reports smaller speedups, which are on the average less than 15% below the speedups reported by Sarkar's method. Using *SNC* on the parallel-execution graph of our example, with no prior clustering, gives speedups comparable to those derived when running *SNC* on the clustered graph. The Priority List Scheduling algorithms with no clustering (*CP/MISF* and *TS/ETF*) report the lowest speedups.

In the case of the task-graph representing an instance of the modified Barnes-Hut algorithm, our experiments show that the speedups depend more on the choice of the clustering than on the mapping heuristic. More specifically, speedups derived from mapping the graph clustered with Sarkar's heuristic, are higher than speedups derived from mapping graphs clustered with other heuristics. (see Figure 8, right). Moreover, Priority List Scheduling with no clustering performs poorly.

Another observation that can be drawn from Figure 8, is that the speedups reported from simulations of the non-blocking *Send*/blocking *Receive* primitives are 20% to 50% higher than the speedups reported

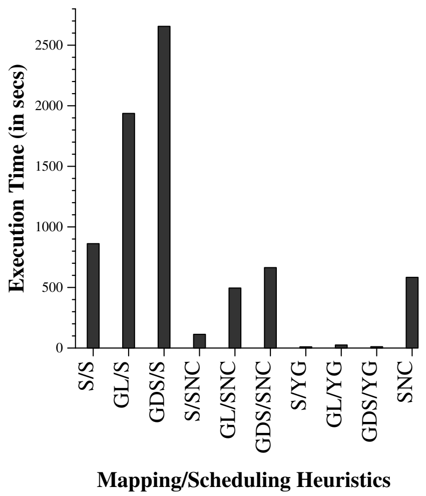


Figure 9: Execution times for the Mapping heuristics.

for blocking *Send*/blocking *Receive* primitives. This is expected since the non-blocking *Send*'s incur smaller communication overhead to the processors of a parallel system. The speedup improvement is higher in the case of Barnes-Hut task-graphs than in the case of Fast Multipole Method graphs, since the former have a lower computation-to-communication ratio.

In Figure 9, we present a diagram of execution time measurements for the various mapping algorithms. These measurements were extracted from FAST simulations of the Fast Multipole Method. Sarkar's algorithm is the slowest. *SNC*, which performs the mapping without taking into consideration communication delays and overhead, has a moderate execution time. Therefore, the high running-time of Sarkar's approach is partly a result of the overhead for estimating communication costs while testing the different mapping choices at each step of the method. For the cases where Sarkar's clustering algorithm was used before the mapping, the running-time of the mapping was smaller. This is because Sarkar's clustering heuristic results in low numbers of clusters.

8 Conclusions

In this paper we examined popular clustering and mapping heuristics used for assigning task-graphs to message-passing multiprocessors. We used task-graphs representative of the two most popular algorithms for the N-Body problem and employed a realistic scheme for annotating these graphs and accurately modeling task-processing time and communication delay.

Our experiments reveal interesting aspects of the effectiveness of clustering heuristics. We conclude that for graphs of coarse-granularity (Fast Multipole Method graphs), with a high ratio of average task execution time to average message delay, clustering does not improve the parallel time of the graph substantially.

In contrast, for graphs with average task execution time comparable to the average message delay (Barnes-Hut graphs), clustering does improve the parallel time of the task-graph. The greatest improvement is achieved with Sarkar's algorithm (more than 50% for both blocking and non-blocking *Send*'s).

The next biggest improvement is achieved with the *GDS* algorithm (more than 20%). In contrast, Kim and Browne's method results in an increase of parallel time after clustering, in the case of blocking *Send*'s; this is a side-effect of the realistic scheme we employed to annotate the task-graph. The *GL* heuristic introduced in this paper, which is based on a principle similar to that of Kim and Browne's, takes into consideration the realistic modeling of computation and communication costs and improves parallel time by approximately 10%.

All the clustering heuristics examined, except *GDS*, result in numbers of clusters which are significantly smaller than the number of tasks; partitioning a task-graph into a small number of clusters expedites the mapping process that follows clustering.

Data from mapping experiments show that, in the case of coarse-grain task-graphs, all mapping heuristics that are used in conjunction with some clustering heuristic have similar effectiveness, regardless of the clustering heuristic used. For fine-grain task-graphs, however, the mapping heuristics examined report very low speedups, except in the case where the task-graphs were previously clustered with Sarkar's clustering method. Finally, it is clear that combining clustering and mapping heuristics gives consistently better results than one-phase mapping algorithms, such as *Priority List Scheduling*.

Our experiments reveal a critical tradeoff between the effectiveness and the running time of clustering and mapping heuristics. Best results, in terms of number of clusters and speedup, are achieved when using Sarkar's clustering and mapping heuristics. Their running time, however, is prohibitively high for task-graphs of medium to large size. Another remark is that communication overhead does not play an important role in the mapping of clustered task-graphs to processors - the mapping heuristic *SNC* does not take into consideration communication costs. Nevertheless, it reports speedup figures which are close to the ones reported by *Sarkar*'s heuristic, which does account for communication overhead.

We conclude that, mapping task-graphs to message-passing multiprocessors effectively and efficiently requires a clustering heuristic that will minimize communication overhead and decrease parallel time under the practical communication-cost model presented earlier, and for task-graphs of various granularities. Such a clustering heuristic can then be combined with a fast, "communication-cost insensitive" method, such as *SNC*, for mapping the clustered task-graphs to the limited number of available processors, and achieving load-balancing of the processors.

9 Acknowledgements

This work was supported by NSF Grants MIP-8912100 and MIP-9201484.

References

[aDA91] S.B. Shukla and D.P. Agrawal. Scheduling Pipelined Communication in Distributed Memory Multiprocessors for Real-time Applications. In *The 18th Annual International Symposium on Computer Architecture*, pages 222–231, May 1991.

[BH86] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[Cof76] E.G. Coffman. *Computer and Job-Shop Scheduling Theory*. Wiley, 1976.

[Cyp93] Robert Cypher. Message-Passing Models for Blocking and Nonblocking Communication. In *DIMACS Workshop on Models, Architectures, and Technologies for Parallel Computation, Technical Report 93-87*. DIMACS Center, Rutgers University, September 1993.

[DRS93a] M. Dikaiakos, A. Rogers, and K. Steiglitz. Functional algorithm simulation: A new approach for modeling the parallel execution of scientific applications. In *DIMACS Workshop on Models, Architectures, and Technologies for Parallel Computation, Technical Report 93-87*. DIMACS Center, Rutgers University, September 1993.

[DRS93b] M. Dikaiakos, A. Rogers, and K. Steiglitz. Functional algorithm simulation: Implementation and experiments. Technical Report TR-429-93, Department of Computer Science, Princeton University, June 1993.

[DRS94] M.D. Dikaiakos, A. Rogers, and K. Steiglitz. FAST: A Functional Algorithm Simulation Testbed. In *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems - MASCOTS'94*. IEEE-Press, 1994.

[ERL90] H. El-Rewini and T.G. Lewis. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *Journal of Parallel and Distributed Computing*, (9):138–153, 1990.

[GG87] L. Greengard and W. Gropp. A Parallel Version of the Fast Multipole Method. In Garry Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 213–222. SIAM, 1987.

[GVY90] A. Gerasoulis, S. Venugopal, and T. Yang. Clustering Task Graphs for Message Passing Architectures. In *1990 International Conference on Supercomputing*, pages 447–457, 1990.

[GY92] A. Gerasoulis and T. Yang. A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[KB88] S.J. Kim and J.C. Browne. A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures. In *International Conference on Parallel Processing*, volume 3, pages 1–8, 1988.

[KN84] H. Kasahara and S. Narita. Practical Multi-processor Scheduling. *IEEE Transactions on Computers*, C-33:1023–1029, 1984.

[MT91] S. Manoharan and P. Tranish. Assigning Dependency Graphs onto Processor Networks. *Parallel Computing*, 17:63–73, 1991.

[NT93] M.G. Norman and P. Thanisch. Mapping in Multicomputers. *ACM Computing Surveys*, 25(3):264–302, September 1993.

[Pol88] C. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[RSL93] M. Rinard, D. Scales, and M. Lam. Jade: A High-Level Machine-Independent Language for Parallel Processing. *Computer*, 26(6):28–38, June 1993.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.

[SH86] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. *SIGPLAN Notices*, 21(7):17–26, July 1986.

[YG91] T. Yang and A. Gerasoulis. A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors. In *Supercomputing 91*, 1991.

[YG92] T. Yang and A. Gerasoulis. PYRROS: Static scheduling and code generation for message passing multiprocessors. In *6th ACM International Conference on Supercomputing*, pages 428–437, July 1992.