# On Partitioning Grid-Structured Parallel Computations

BHAGIRATH NARAHARI

*Department of Electrical Engineering and Computer Science*
*The George Washington University*
*Washington, DC 20052*
`narahari@seas.gwu.edu`

RAHUL SIMHA

*Department of Computer Science*
*College of William and Mary*
*Williamsburg, VA 23185*
`simha@cs.wm.edu`

**Abstract**

This paper studies a special instance of the graph partitioning problem motivated by an application in parallel processing. When a parallel computation is represented by a weighted task graph, we consider the problem of mapping each node in the graph to a processor in a linear array. We focus on a particular type of computation, a grid structured computation (GSC), where the task graph is a grid of nodes. The general task graph mapping problem is known to be intractable and thus past research efforts have either proposed heuristics for the general problem or optimally solved a constrained version of the general problem. Our contributions in this paper fall into both categories. We weaken past constraints and optimally solve a less constrained problem than has been solved optimally before and also present and analyze a simple greedy heuristic.

Optimal solutions have been given in the past when one places the *contiguity constraint* that each partition must consist of entire columns (or rows) of the GSC. We show that a more efficient solution can be found by relaxing the constraints on the partitions to allow parts of consecutive columns to be mapped to a processor; we call this weaker contiguity constraint the *part-column* constraint. Our first result is to show that the problem of finding an optimal mapping satisfying the contiguity constraint remains NP-complete, where the contiguity constraint simply requires adjacent nodes to be mapped to the same or adjacent processors. We then design an $O(M^2p)$ algorithm (that uses $O(Mp)$ space) which finds the optimal part-column partitioning of a grid of $M$ modules to a linear array of $p$ processors. A simple greedy $O(M)$ heuristic part-column partitioning algorithm is also presented which performs within a constant factor (two) of the optimal algorithm. Our loosening of past constraints is shown to lead to a forty percent improvement in some cases. Other experimental results compare the proposed heuristic with the optimal algorithm.

**Keywords:** task graphs, partitioning, mapping, parallel architectures.

# 1 Introduction

A parallel computation is often only as efficient as the evenness with which its workload is distributed over the processors in a parallel machine. The time to execute the parallel program will depend on the processor that completes last – the processor with the largest load. Therefore, the general problem of efficiently partitioning a large computation over several processors is an important problem at the core of parallel processing and has received considerable attention in the literature. A common approach to the mapping problem is graph-theoretic in nature: a parallel program is modeled by a graph, the *task graph* $G = (V, E)$, whose weighted nodes $V$ represent program modules with varying execution times and whose weighted edges $E$ represent communication requirements (data) between program modules. Similarly, the target parallel architecture is represented by a graph $G_a = (V_a, E_a)$ whose nodes $V_a$ are the processors and edges $E_a$ are communication links between processors. A mapping $\pi : V \mapsto V_a$ is a function that maps nodes in the task graph to nodes in the architecture; it partitions the task graph into a set of partitions and assigns each partition to a processor, resulting in a workload for each processor. The number of processors $|V_a|$ is typically much smaller than the number of program modules $|V|$. Based on this model, an execution time for the whole computation can be estimated for any mapping of subtasks to processors by adding the weights of nodes assigned to each processor and taking the maximum among all processors. The central question addressed in this paper thus arises: how to find a mapping that minimizes the execution time for the parallel computation?

We focus on a particular type of parallel computation, a *grid structured computation* (GSC), whose importance can be explained by the large number of applications that fall under the GSC label, most of them design problems in engineering. Briefly, a GSC is a weighted task graph in which the edges form a grid or mesh. The graph itself is often constructed from the underlying computation in a preprocessing stage [15]. Grid structured task graphs arise in applications exhibiting data parallelism in two dimensional data domains, such as image processing, finite element methods, signal processing, and scientific computing [4, 16, 18]. For example, many image processing applications involve operations between neighboring pixels (data points) in the image. The features in the image can be distributed unevenly [8] and thus the module weights can be non-uniform.

Since mapping GSC's is NP-complete in general, past work typically consists of optimally solving more constrained versions of the problem as well as developing heuristics for the general case. One constraint of practical interest is a *contiguity constraint* – adjacent (communicating) task nodes must be placed onto the same processor or onto adjacent processors – a condition justified by the fact that communication between adjacent (directly connected) processors is significantly faster than between processors two or more links apart.

In this paper, we study the specific problem of mapping a GSC of $M = m \times m$ nodes onto a linear array of $p \leq m$ processors, such that communicating modules are mapped to adjacent processors. To satisfy the contiguity constraint, past work has typically required that each partition consist of entire columns (or rows) of the GSC [4, 7, 11, 16, 18]. We develop a partitioning scheme that is more efficient in its ability to evenly distribute load than past methods by weakening the entire-column constraint to a *part-column* constraint. By solving this less-constrained version optimally, we produce more efficient mappings since the solution space is now larger and properly includes that explored by previous algorithms [7, 16]. Unfortunately, the increased efficiency comes at a cost: our optimal part-column algorithm has complexity $O(M^2p)$ while optimal algorithms for the entire-column version are $O(M+mp)$ for homogeneous processors [7] and $O(Mp)$ for the hetergeneous processors [16]. Nonetheless, this extra overhead is reasonable since the mapping process usually takes much less time than the parallel computation. Additional results include the following: the problem remains NP-complete even with the contiguity constraint; a simple heuristic that performs within a factor of two of the optimal algorithm; and experimental results showing the improvement in mapping quality using our part-column algorithm.

The paper is organized as follows. The next section formulates the problem and establishes the NP-completeness of the problem of finding arbitrary partitions that satisfy the contiguity constraint. A review of relevant prior work is given at the end of this section. Section 3 develops and presents our algorithm to map a GSC to a linear array. An $O(M)$ heuristic part-column partitioning algorithm and its performance bound is also presented. Section 4 discusses the results of experiments, on partitioning some randomly generated grids, that compare the performance of the part-column partitioning algorithms (both the optimal and the heuristic) with that of optimally partitioning the grid into full columns. Section 5 describes how the optimal part-column algorithm may be modified to incorporate communication times, and concluding remarks are provided in Section 6.

## 2 Problem Formulation and Preliminaries

A grid structured computation is represented by a weighted graph $G=(V,E)$ of $M=m_1\times m_2$ nodes, where $V=\{v_{i,j}|1\le i\le m_1, 1\le j\le m_2\}$, and each non-boundary node $v_{i,j}$ has four edges connecting it to the four neighboring nodes in the grid; the set of edges incident at each node $v_{i,j}$ is $\{(v_{i,j},v_{i+1,j}),(v_{i,j},v_{i-1,j}),(v_{i,j},v_{i,j+1}),(v_{i,j},v_{i,j-1})\}$. A node in $V$ represents a subtask of the grid structured parallel program, where node $v_{i,j}$ denotes the subtask in the $i$-th row and $j$-th column. The edges in $E$ represent communication between modules. For simplicity, in the remainder of this paper we assume that $m_1=m_2=m$, although our algorithms are easily adjusted to handle the general case where $m_1\ne m_2$. We will also add 'dummy' nodes of zero weights around the entire border of the grid task graph so that definitions do not have to be separately written for the border nodes in $V$. For any node $v_{i,j}$ the adjacent nodes $v_{i-1,j}, v_{i+1,j}, v_{i,j+1}$ and $v_{i,j-1}$, are called the *north, south, east* and *west* neighbors, respectively, and we refer to the corresponding edges as the north, south, east and west edges.

Each node $v_{i,j}$ in $G$ is has a weight $w_{i,j}\in\mathbb{Z}^+\cup 0$ which denotes the execution time (*i.e.,* load or amount of computation) of the program module $v_{i,j}$ on a single processor. In general, each edge can also be assigned a weight which indicates the amount of communication between the two neighboring nodes. For ease of understanding, in the next two sections we will assume that there are no communication weights. In Section 5 we discuss how some of the algorithms developed in this paper can be generalized to consider communication weights.

The linear array of $p$ processors is represented by the graph $G_p = (V_p, E_p)$ where $V_p = \{P_1, P_2, \ldots, P_p\}$ and $E_p = \{(P_i, P_{i+1}) | 1 \leq i \leq p-1\}$. A mapping $\pi$, $\pi : V \mapsto V_p$, is a function that assigns the $m^2$ modules to the $p$ processors. The mapping partitions the task graph nodes into $p$ partitions, and each partition is mapped to a distinct processor; the set of nodes mapped to processor $P_k$ is given by $\pi^{-1}(k) = \{x \in V \mid \pi(x) = k\}$. These set of nodes form the "k-th partition".

For any mapping $\pi$, the computation *load* on processor $k$, is the time $T_e(\pi^{-1}(k))$ to execute all nodes in partition $k$; *i.e.,* the load or *cost* of partition $k$. This is defined as:

$$T_e(\pi^{-1}(k)) = \sum_{v_{i,j} \in \pi^{-1}(k)} w_{i,j}.$$

The natural objective of a mapping is to minimize the completion time of the parallel computation by evenly balancing the load across all $p$ processors. This time is determined by the processor that is assigned the largest load, *i.e.,* the partition with the largest cost. We call this quantity the *cost* of the mapping:

$$C(\pi) = \max_{1 \leq \ell \leq p} T_e\left(\pi^{-1}(\ell)\right)$$

An optimal mapping is any mapping that achieves the minimum cost, i.e., any mapping $\pi^*$ such that $C(\pi^*) \leq C(\pi)$ for any other mapping $\pi$.

### 2.1 The Contiguous Mapping Problem

Since the problem of finding an optimal mapping is known to be NP-complete we focus on adding a suitable constraint to make the problem polynomially solvable. Such optimal solutions are often reasonable solutions for the unconstrained problem. The constraint most often studied in the parallel processing literature is geometric in nature: nodes adjacent to each other in the task graph must be assigned to either the same processor or to adjacent processors. In practice, such a condition is justified by the fact that cost of communication between neighboring processors is significantly lower than communication with nonadjacent processors. Therefore, to keep communication costs small, adjacent task nodes (i.e., communicating program modules) are mapped to the same or neighboring processors.

A mapping $\pi$ from a GSC to a linear array, is said to be a *contiguous mapping* if it satisfies the property that for any $v_{i,j}$ and $v_{k,l}$ with $(v_{i,j}, v_{k,l}) \in E$, we have $\pi(v_{i,j}) = \pi(v_{k,l})$ or $(\pi(v_{i,j}), \pi(v_{k,l})) \in E_p$; *i.e.,* neighboring nodes are mapped to the same processor or to adjacent processors. The *contiguous mapping problem* is to find a contiguous mapping with minimum cost. Unfortunately, as we now show, this constraint is alone not sufficient to make the problem tractable. The contiguous mapping problem is easily shown to be NP-complete by reducing the *multiprocessor scheduling* problem [10].

**Definition 2.1 (Contiguous mapping problem)** *The decision version of the contiguous mapping problem may be defined as follows:*

> **Instance:** *A Grid structured task graph, or GSC, of $m_1 \times m_2$ nodes, linear array of $p$ processors, and an integer $K$.*
> **Question:** *Is there a contiguous mapping $\pi$ of the GSC to the linear array such that the cost of the mapping is at most $K$.*

**Theorem 2.1** *The contiguous mapping problem is NP-complete even with two processors.*

**Proof:** It is easy to see that the problem is in NP since given any mapping we can compute the cost in polynomial time. The multiprocessor scheduling problem for two processors is defined as:

> **Instance:** A finite set $A$ of $m$ tasks with lengths $\{a_1, ..., a_m\}$ and a deadline $K$.
> **Question:** Does there exist a partition of $A$ into $A'$ and $A - A'$ such that $\max\{\sum_{i \in A} a_i, \sum_{i \in A-A'} a_i\} \leq K$.

Given an instance of multiprocessor scheduling we can construct the following instance of the contiguous mapping problem for a $m \times 3$ grid and $p = 2$, as also shown in Figure 1. For all $i$, where $1 \leq i \leq m$, we have $w_{i,1} = w_{i,3} = 0$, and $w_{i,2} = a_i$; the weights of nodes in the first two columns are zero and the nodes in the second column have non-zero weights. Note that any partition of this grid will satisfy the contiguity constraint, and any partition of the GSC corresponds to a partition of the set $A$. ∎

## 2.2 The Part-Column Constraint

Since Theorem 2.1 demonstrates that the contiguity constraint alone does not make the mapping problem tractable, we now examine a somewhat stronger contiguity constraint, the *part-column* constraint. Under this constraint, each partition consists of modules from successive columns with the flexibility that modules from parts of two successive columns can be assigned to a partition. More formally,

**Definition 2.2 (Part-column constraint)** *The part-column constraint placed on a mapping $\pi$, from a GSC to a linear array, can be stated as:*

1. *if $\pi(v_{i,j}) = P_x$ then $\pi(v_{i,j+1}) = P_x$ or $\pi(v_{i,j+1}) = P_{x+1}$, for any $1 \leq i \leq m$, $1 \leq j < m$;*
2. *if $\pi(v_{i,j}) = P_x$ then $\pi(v_{i+1,j}) = P_x$ or $\pi(v_{i+1,j}) = P_{x+1}$, for any $1 \leq i < m$, $1 \leq j \leq m$;*
3. *if $\pi(v_{i,j}) = \pi(v_{k,l}) = P_x$, for some $i, j, k, l$ where $j < l$, then*
   - *(a) for all $i' > i$, $\pi(v_{i',j}) = P_x$;*

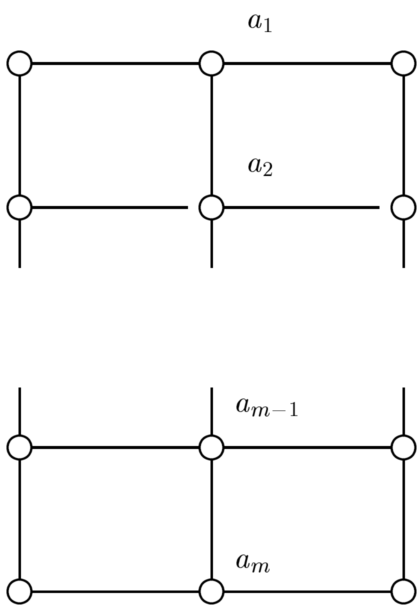


Figure 1: Problem instance

*(b) for all $j'$ where $j < j' < l$ and all $i'$, $1 \leq i' \leq m$, $\pi(v_{i',j'}) = P_x$;*

*(c) for all $i' < k$, $\pi(v_{i',l}) = P_x$.*

The first two conditions merely state the contiguity constraint and the third condition defines the part-column condition in the following sense. If one were to number the task graph nodes such that the nodes in the first column are numbered $1, \ldots, m$, the nodes in the second column are numbered $m+1, \ldots, 2m$, and so on, until the nodes in the last column are numbered $m(m-1)+1, \ldots, m^2$. Then, if any two nodes are assigned to the same processor, so are all nodes in-between with the above numbering (which we will call the *unidimensional* labelling). From the definitions above, it can be seen that the mapping implicitly binds the assignment of a partition to a processor in the architecture. Consequently, we use the terms partitioning, mapping and assignment interchangeably in this paper. We note that the *full-column* constraint, where a partition is a collection of entire columns, can be defined by restating condition (iii) above as: if $\pi(v_{i,j}) = P_x$ for some $x$ and $1 \leq i, j \leq m$, then $\pi(v_{k,j}) = P_x$ for all $1 \leq k \leq m$.

The problem of finding optimal full-column partitions has been studied implicitly as a problem of mapping a linear array (chain) of nodes to a chain of processors [4, 7, 16] An entire column of nodes in the GSC is aggregated to form a node of a linear array, as in [18]. Figure 2(a) illustrates a part-column partitioning, and Figure 2(b) shows a full-column partitioning of a GSC with $p = 4$. From the figure, it can be observed that part-column partitioning provides greater flexibility in the type of partitions and can thus lead to more efficient mappings. Note that an immediate

Figure 2: Part-Column and Full-Column Partitioning

consequence of the part-column and full-column constraints is the condition that $p \leq m$. This is a justifiable assumption since in practice the number of processors is typically much smaller than the granularity of the problem.

A mapping that satisfies the part-column constraint will be called *feasible*. Next, if the modules $v_{i,j}$ and $v_{k,l}$ are the first and last modules in the unidimensional numbering assigned to a processor, we will use the notation $< v_{i,j}, v_{k,l} >$ to represent the set of nodes in the partition and compute its load (or cost) as:

$$T_e(v_{i,j}, v_{k,l}) = \sum_{i \leq x \leq m} w_{x,j} + \sum_{\substack{1 \leq x \leq m \\ j < y < l}} w_{x,y} + \sum_{1 \leq x \leq k} w_{x,l} \tag{1}$$

We now make some observations about prior work on our problem of interest. As mentioned earlier, the full-column constraint has been studied [4, 7, 16, 18]. In addition, it also been addressed under the topic of embedding a task graph to an architecture graph [6, 12], and mapping finite element grids [2, 18] While efficient embeddings have been constructed, the underlying model of a task graph is an unweighted graph thereby implying that all modules and edges have equal weights. The algorithm in [2] does not satisfy the contiguity constraint or consider edge weights. We next study the problem of finding the best part-column mapping.

# 3 Partitioning Algorithms

To design an algorithm to find the optimal part-column partitioning, we first define an assignment graph $G^*$ for the problem. We shall then show that paths in $G^*$ correspond to feasible mappings; an optimal mapping is an optimal bottleneck path. Based on this concept, we present an $O(M^2p)$ algorithm using a dynamic programming approach.

As described earlier, in the unidimensional labelling of task graph nodes, a node $v_{i,j}$ is relabelled $u_k$ where $k = i + (j-1)m$; thus, a relabelled node $u_k$ denotes module $v_{i,j}$ where $i = ((k-1) \bmod m) + 1$ and $j = \lceil k/m \rceil$. With this relabelling scheme, a partition referred to as $< i, j >$ is really the partition $< v_{i_1,j_1}, v_{i_2,j_2} >$; $T_e(v_{i_1,j_1}, v_{i_2,j_2})$ is now referred to as $T_e(i,j)$ where $i = i_1 + (j_1 - 1)m, j = i_2 + (j_2 - 1)m$. As noted earlier, to satisfy the part-column constraint each processor $P_k$, $2 \le k \le p-1$, must be assigned at least $m$ task nodes. The end processors (partitions) $P_1$ and $P_p$ can be assigned any number of nodes.

## 3.1 The Assignment Graph

The assignment graph $G^* = (V^*, E^*)$, for mapping GSCs to linear arrays, is an edge weighted graph where:

$$V^* = \left\{u_0, u_1, \ldots, u_{m^2}\right\}$$

$$E^* = E_1 \bigcup E_2 \bigcup E_3$$

and the edge sets $E_1$, $E_2$ and $E_3$ are defined as:

- $E_1 = \left\{(u_0, u_i) | \, 1 \le i \le m^2\right\}$
- $E_2 = \left\{(u_i, u_{m^2}) | \, 1 \le i \le m^2 - 1\right\}$
- $E_3 = \left\{(u_i, u_j) | \, 1 \le i \le m^2 - m, \; i + m \le j \le m^2\right\}$

Note that $V^*$ consists of all nodes in the GSC along with one new vertex $u_0$. The weight $w(u_i, u_j)$ of an edge $(u_i, u_j)$ is defined to be the time (load) $T_e(i+1, j)$ of partition $< i+1, j >$:

- $w(u_i, u_j) = T(i+1, j)$.

The assignment graph is constructed such that an edge $(u_i, u_j)$ represents a partition $< i+1, j >$ and the weight of the edge denotes the load (or cost) $T(i+1, j)$ of that partition. For example, consider an edge $(v_i, v_j)$ in $E_3$; this edge denotes a partition of the form $< i+1, j >$ where $j - (i+1) \ge m$ thereby satisfying the part-column constraint. Edges in $E_1$ ($E_2$) denote the possible partitions we could assign to the first (last) processor. Figure 3 illustrates part of an assignment graph for a $3 \times 3$ GSC, and the relabelling of nodes is also shown in the figure. For clarity, only all edges originating from $u_1$ are shown in the figure while some edges from other nodes are not shown.

Figure 3: Assignment Graph $G^*$

The following theorem proves that a path of length $p$, *i.e.*, using exactly $p$ edges, from $u_0$ to $u_{m^2}$ in $G^*$ corresponds to a feasible mapping. Also, any feasible mapping $\pi$ defines a path of $p$ edges from $u_0$ to $u_{m^2}$.

**Theorem 3.1** *$\pi$ is a feasible mapping if and only if there exists a path in $G^*$ from $u_0$ to $u_{m^2}$ using exactly $p$ edges.*

**Proof:** We first prove the if part, *i.e.*, if there exists a path $P = (e_1, e_2, \ldots, e_p)$ using exactly $p$ edges from $u_0$ to $u_{m^2}$ then there exists a feasible mapping $\pi$. Note that $e_1 = (u_0, u_l)$ for some $l \geq 1$ and $e_p = (u_k, u_{m^2})$ for some $1 \leq k < m^2$. Given a path $P$ we define the corresponding mapping $\pi$ as follows:

- If $(u_i, u_j)$ is the $k$-th edge $e_k$, for some $1 \leq k \leq p$, in the path then the nodes in partition $< i+1, j >$ are mapped to processor $P_k$.

It follows that $\pi$ is a mapping since it maps all nodes in $V^*$. We now show that it satisfies the part-column constraint. Consider a partition of the form $< i+1, j >$ corresponding to edge $e_k$, for any $1 < k < p$. The edge goes from vertex $u_i$ to vertex $u_j$ and is in edge set $E_3$ (in $E^*$). Therefore $j - i \geq m$, from the definition of $E_3$, and at least $m$ nodes are mapped to the processor $P_k$. From the unidimensional relabelling procedure it follows that nodes in partition $< i+1, j >$ comprise a partition satisfying the part-column constraint. The partitions mapped to the first (last) processors comprise any number of nodes from successive columns and thus satisfy the contiguity constraint. Therefore $\pi$ is a feasible mapping.

Now consider any feasible mapping $\pi$, *i.e.*, one that satisfies the contiguity constraint. We can define a corresponding path in $G^*$. The set of edges in the path are:

- If partition $< i, j >$ is mapped to processor $P_k$ then edge $(u_{i-1}, u_j)$ is the $k$-th edge on the path $P$.

From the above construction it can be seen that the path $P$ has $p$ edges. Also, since the mapping satisfies the part-column constraint, for every partition of the form $< i, j >$, where $i > 1$, $j - i \geq m$ is an edge $(u_{i-1}, u_j)$ in $E^*$. Since a mapping must assign all nodes to processors, node $u_1$ is in the partition $< 1, u_i >$, for some $i$, and therefore $u_0$ is the first vertex in the path. Similarly, it follows that $u_{m^2}$ is the last vertex in the path. Therefore $P$ is a path from $u_0$ to $u_{m^2}$ using exactly $p$ edges. ■

Having seen the equivalence between paths in $G^*$ and mappings, we now consider the cost of a mapping. The cost of the mapping is the maximum among all partition loads. Therefore, it follows that the cost of the mapping is the bottleneck cost (*i.e.,* cost of the heaviest edge in the path). More precisely, if $C(P)$ is the cost of the path $P = (e_1, \ldots, e_p)$ then:

$$C(P) = \max_{e_i \in P} w(e_i).$$

Finding a mapping which minimizes the maximum load can be accomplished by finding a path with the minimum bottleneck cost, called the bottleneck path. This path can be determined using dynamic programming.

Let $D^k(0, i)$ denote the cost of the optimal bottleneck path in $G^*$ from node $u_0$ to node $u_i$ using exactly $k$ edges. It is useful to note that $D^1(j, i) = w(j, i)$, the cost of edge $(u_j, u_i)$, which is infinite if there is no edge. Any path $P$ using exactly $k$ edges, from $u_0$ to $u_i$, consists of a path $P'$ using exactly $k - 1$ edges from $u_0$ to some intermediate vertex $u_j$, and an edge from $u_j$ to $u_i$. If $P$ is the optimal path, then the path $P'$ must be the optimal path using $k - 1$ edges from $u_0$ to $u_j$. Therefore, from the principal of optimality we have the following relation for $D^k(0, i)$:

$$D^k(0, i) = \min_{j<i} \Big\{ \max\{D^{k-1}(0, j), D^1(j, i)\} \Big\}. \tag{2}$$

The cost of the optimal mapping is $D^p(0, m^2)$. By storing the value of $j$ at which the minimum occured, for each $D^k(0, i)$, the optimal assignment can be recovered.

The relation in equation 2 shows that an optimal mapping, satisfying the part-column constraint, can be determined in polynomial time using a simple dynamic programming approach. Assuming that the graph $G^*$ has been previously constructed, a direct implementation of equation 2 results in a $O(M^2 p)$ algorithm (where $M = m^2$ is the total number of task graph nodes and hence the input size). However, computing the weight of an edge in a direct manner may take $O(M)$ time (for example, we may have to compute the sum of $O(M)$ node weights) and thus construction of the graph itself may take $O(M^3)$ time. As we shall show in the ensuing discussion, by exploiting the relationship between edge weights and providing an $O(M)$ startup procedure, we can develop an algorithm that takes $O(M^2 p)$ time.

The $O(M)$ startup procedure allows edge weights, *i.e.*, the variable $D^1(j,i)$ in equation 2, to be computed on the fly, inside each iteration of $j$, in constant time and itself takes $O(M)$ time and $O(M)$ space. Recall that $D^1(j,i) = w(u_j, u_i)$, the cost of edge $(u_j, , u_i)$, for $0 \leq j < i \leq m^2$, and that the edge cost is defined to be the load $T_e(j+1,i)$ of the partition $< j+1, i >$. Thus, $D^1(j,i)$ can be expressed as:

$$D^1(j,i) = T_e(j+1,i) = \sum_{j+1 \leq l \leq i} w_l = \sum_{1 \leq k \leq i} w_k - \sum_{1 \leq l \leq j} w_l = T_e(1,i) - T_e(1,j) = D^1(0,i) - D^1(0,j)$$

If we first compute $D^1(0,M)$ then successive (decreasing $j$) values of $D^1(0,j)$, for $1 \leq j \leq M-1$, can be computed in one step using the relation: $D^1(0,j) = D^1(0,j+1) - w_{j+1}$. Precomputing $D^1(0,M)$ takes $O(M)$ and, since there are $O(M)$ values of $D^1(0,j)$, for $1 \leq j \leq M-1$, precomputing all $D^1(0,j)$ takes an additional $O(M)$ steps. Note that these precomputed quantities can be stored in some $M$-element array, and thus $D^1(j,i)$, when needed, can be computed in constant time. With this concept we outline an algorithm based on equation 2.

**Algorithm Part-Column Mapping**
**Input:** a GSC $G(V,E)$ with unidimensionally relabelled nodes
**Output:** the optimal assignment $D^p(0,M)$

Compute array of execution weights $D^1(0,i)$ for $1 \leq i \leq M$
**for** $k := 2$ **to** $p-1$ **do**
  **for** $i := (k-1)m+1$ **to** $M$ **do**
  { iterate over all feasible values of $j$ }
    **for** $j := (k-2)m+1$ **to** $i-m$ **do**
      $D^1(j,i) = D^1(0,i) - D^1(0,j)$
      $D^k(0,i) := \min\{D^k(0,i), \max[D^{k-1}(0,j), D^1(j,i)]\}$
    **endfor**
  **endfor**
**endfor**
**for** $j := (p-2)m+1$ **to** $M-1$ **do**
  $D^1(j,M) = D^1(0,M) - D^1(0,j)$
  $D^p(0,M) := \min\{D^p(0,M), \max[D^{p-1}(0,j), D^1(j,M)]\}$
**endfor**

Now consider the complexity of the algorithm. The precomputation takes $O(M)$ time and requires $O(M)$ space. To conclude the bounds on the number of iterations for $i$, observe that except for the first (and last) processor all other processors must be assigned at least $m$ modules. Therefore when using $k$, for $k < p$, processors at least $1 + (k-1)m$ modules must be mapped to the $k$ processors. The bound $j \leq i - m$ follows from the condition that at least $m$ modules must be mapped to each processor; implying that at most $i - m$ modules are mapped to the first $k-1$ processors. The last processor can be assigned any number. Finally, each iteration of $j$ takes constant time. Since there are a total of $O(M(M-m)p)$ iterations, the algorithm has complexity $O(M^2p)$. The only values to be stored after each iteration of $k$ are $D^{k-1}(0,i)$ for all $M$ values of $i$, thus requiring $O(M)$ space to determine the optimal cost, although $O(Mp)$ space is required to obtain the solution. The correctness of the algorithm is established by Theorem 3.1.

### 3.2 A Fast Heuristic

We now discuss a fast and simple $O(M)$ greedy heuristic, and derive a worst-case performance bound on the heuristic. Experiments presented later indicate that the heuristic performs significantly better than this worst-case bound.

**Algorithm Heuristic Part-Column Mapping**
**Input:** a GSC $G(V, E)$ with relabelled nodes
**Output:** an assignment $H$ of nodes to processors

Compute Average cost $C_{avg} = (1/p)\sum_{i=1}^{m^2} w_i$
$load := 0; \quad i := 1$
**while** $i \leq m^2$
  **if** $load < C_{avg}$ **then**
    Assign $i$ to Processor $k$
    $load := load + w_i$
    $i := i + 1$ { place next node into processor }
  **else** {if current load if more than $C_{avg}$}
    $k := k + 1$ { start assigning to next processor }
    Assign $m$ nodes from $i$ to $i + m$ to proc. $k$
    Update $load := \sum_{l=i}^{i+m} w_i$
    $i := i + m$
  **endif**
**endwhile**
end

For any unconstrained mapping, *i.e.,* regardless of whether it satisfies the part-column constraint, a lower bound on its cost can be defined as $\Omega = \max\{C_{avg}, C_{max}\}$, where $C_{avg} = (1/p)\sum_{i=1}^{m^2} w_i$ and $C_{max} = \max_{1 \leq i \leq m^2}\{w_i\}$. ($C_{avg}$ denotes the average load/weight across $p$ processors, and $C_{max}$ denotes the maximum among all node weights.) The greedy heuristic works by attempting to assign to each processor a load as close to average as possible. It uses the unidimensional labelling for the nodes in the GSC. Except for the first (and last) processor, starting with some node $u_i$ it first assigns the next $m$ nodes to the partition $k$ (to satisfy the part-column constraint) with a resulting *load* for the partition (i.e., load on processor $k$). If this *load* is less than the average $C_{avg}$,

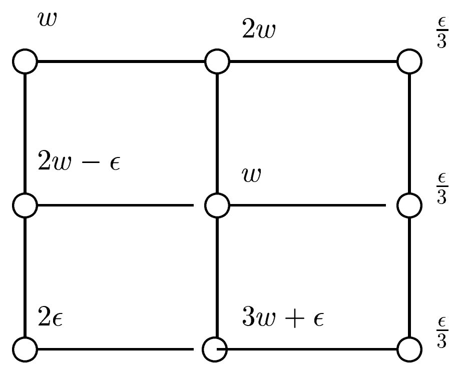


Figure 4: Worst-case example for heuristic

it keeps assigning consecutive nodes to this partition (i.e., processor) until the *load* exceeds the average after assigning some node $u_i$. It then starts assigning to the next processor $k + 1$, starting from node $u_i$. The algorithm terminates when all $M$ modules have been assigned. This process is formally shown below. The complexity is $O(m^2) = O(M)$ which is the number of iterations of the "while" statement. We now derive a worst-case performance bound for the mapping $H$ generated by the above heuristic.

**Theorem 3.2** *Let $C(H)$ denote the cost of the mapping generated by the heuristic, and let $C_*$ denote the cost of the optimal mapping. Then $C(H)/C_* \leq 2$.*

**Proof:** For the mapping produced by the heuristic, let $(u_i, u_j)$ (in unidimensional labelling) denote any partition with the cost $C(H)$. In other words, nodes $i$ through $j$ are placed into one partition and their sum is $C(H)$. We consider two cases for $i - j$: (1) $j - i > m$ or (2) $j - i = m$. In case (1), it follows that placing nodes $i$ to $j - 1$ will have cost less than $C_{avg}$ and after placing node $j$, with weight $w_j \leq C_{max}$, the current value of *load* exceeds $C_{avg}$. Therefore we have $C(H) \leq C_{avg} + C_{max}$. But since $C_* \geq \Omega$, and $\max\{C_{avg}, C_{max}\} = \Omega$, we have $C(H)/C_* \leq 2$.

Now consider case (2). This case implies that exactly $m$ nodes have been placed in the partition to satisfy the part-column constraint. From the conditions of the part-column constraint, in the optimal part-column mapping these $m$ successive nodes can be divided across at most two processors. Thus, the worst of these processors cannot get a load less than $C(H)/2$ and therefore $C_* \geq C(H)/2$. ∎

To see that the performance bound is asymptotically tight, consider the instance in Figure 4 for a $3 \times 3$ grid and for 3 processors, and let $w >> \epsilon$. The average cost is $3w + \epsilon$. The optimal part-column mapping has a cost of $3w + 2\epsilon$. The heuristic assigns the entire second column to a processor with resulting cost $6w + \epsilon$, and thus the ratio is seen to be $C(H)/C_* = 6w + \epsilon/(3w + 2\epsilon)$. For $\epsilon << w$ we have $C(H)/C_* \to 2$.

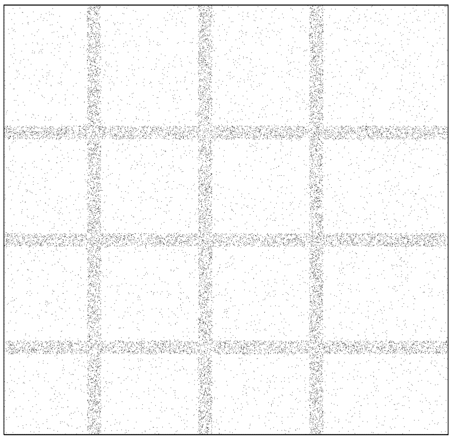

Figure 5: Six-stripe

# 4 An Experimental Comparison

The last two sections presented algorithms that allow part-column partitioning of a GSC. As discussed earlier, the part-column method attempts to find an efficient mapping by searching a larger solution space than that searched by full-column partitioning [4, 7, 16, 18]. While theoretically better, it is useful to experimentally evaluate the part-column algorithm.

We tested the three partitioning schemes, full-column, part-column and heuristic using randomly generated workloads designed to resemble typical grid computations. Four such workloads are shown in Figures 5 to 8. Each workload was obtained as follows. First, points were generated non-uniformly in a box according to each specified shape (for example, the L-shape figure had most of its points inside the L-shaped region shown in Figure 8). Following this, the box was divided evenly with grid lines, creating small square regions each containing different numbers of points. The corresponding weighted task graph was then obtained by associated a vertex with each small square region and taking the number of points in the square region as the weight of the vertex. This node weight is directly proportional to the workload since, in the applications of interest, the computation usually consists of processing each of these grid points in an identical manner. Experiments were performed for $M = 32 \times 32$ task graphs and different numbers of processors in the range 2 to 16. The experiments determined the cost of the optimal part-column mapping, the heuristic part-column mapping, and the optimal full-column mapping for each of the GSC's shown in the figures.

Table 1 shows the cost of the mapping produced by each algorithm for a $32 \times 32$ grid using 16 processors. $C_P$, $C_F$ and $C_H$ respectively denote the costs of the part-column, full-column and heuristic mappings. Ratios of costs and percentage improvements are also provided; $\alpha_{PF}$ is the

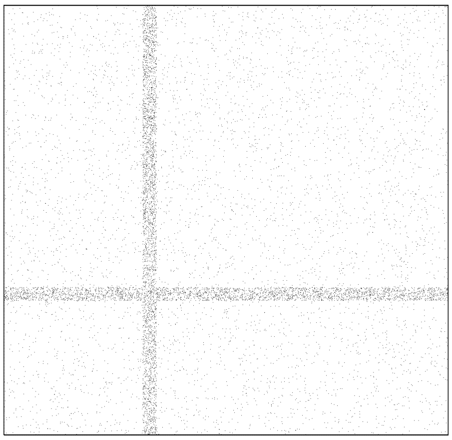

Figure 6: Two-stripe

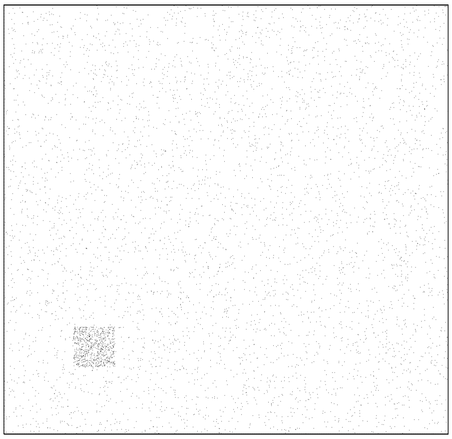

Figure 7: Small box

| Figure | $C_P$ | $C_F$ | $C_H$ | $C_P/C_H$ | $\alpha_{P/F}$ | $C_H/C_F$ | $C_H/C_P$ | $\alpha_{H/F}$ |
|---|---|---|---|---|---|---|---|---|
| 6 Stripes | 1739 | 2996 | 2615 | 0.58 | 42% | 0.87 | 1.50 | 13% |
| 2 Stripes | 1644 | 2948 | 2565 | 0.55 | 44% | 0.87 | 1.79 | 13% |
| Box | 397 | 443 | 418 | 0.89 | 11% | 0.94 | 1.05 | 6% |
| L-shape | 2948 | 2998 | 3060 | 0.98 | 2% | 1.02 | 1.04 | -2% |

Table 1: Mapping costs of the three methods

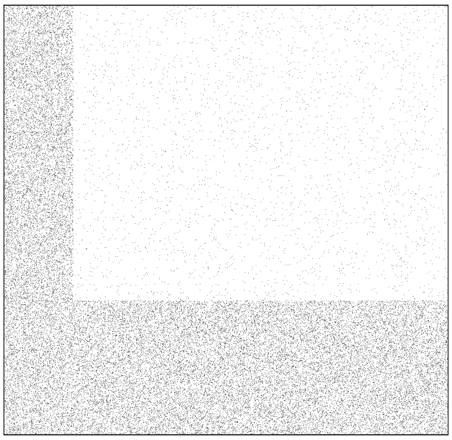

Figure 8: L-shape

| Figure | Grid size | $C_P$ | $C_F$ | $C_H$ | $\alpha_{P/F}$ | $\alpha_{H/F}$ |
|---|---|---|---|---|---|---|
| 6 stripes | 64 × 64 | 579 | 776 | 611 | 25% | 21% |
| | 128 × 128 | 371 | 399 | 372 | 7% | 6.8% |
| 2 stripes | 64 × 64 | 427 | 756 | 573 | 43% | 24% |
| | 128 × 128 | 332 | 351 | 333 | 5.4% | 5.1% |
| Box | 64 × 64 | 321 | 367 | 323 | 12% | 12% |
| | 128 × 128 | 313 | 328 | 313 | 4.5% | 4.5% |
| L-shape | 64 × 64 | 909 | 1109 | 914 | 18% | 17% |
| | 128 × 128 | 456 | 478 | 459 | 4.6% | 3.9% |

Table 2: Costs with larger grid sizes

| Method | Measured execution time |
|---|---|
| Part-column | 100 |
| Full-column | 14 |
| Heuristic | 0.449 |

Table 3: Typical execution times of the three methods

| Number of processors | $C_P$ | $C_F$ | $C_H$ | $\alpha_{PF}$ | $\alpha_{HF}$ |
|---|---|---|---|---|---|
| 2 | 10611 | 11469 | 10604 | 7% | 7% |
| 4 | 5340 | 5933 | 5344 | 10% | 10% |
| 6 | 3544 | 4625 | 3592 | 23% | 22% |
| 8 | 2687 | 2996 | 2718 | 10% | 9.2% |
| 10 | 2281 | 2996 | 2526 | 24% | 16% |
| 12 | 1909 | 2996 | 2234 | 36% | 25% |
| 14 | 1795 | 2996 | 2770 | 40% | 7.5% |
| 16 | 1739 | 2996 | 2615 | 42% | 13% |
| 32 | 1739 | 2996 | 2709 | 42% | 9.5% |

Table 4: Performance for increasing numbers of processors

percentage improvement of part-column over full-column; similarly $\alpha_{HF}$ is the percentage improvement obtained in using the heuristic over the full-column algorithm. The results demonstrate that significant improvement (about forty percent) can result from using the optimal part-column method. They also show that much improvement is not always the case, as the results for the L-shape make clear. Next, the heuristic results in costs usually between those produced by the the part- and full-column methods. However, as Table 3 shows, the heuristic runs much faster than the other algorithms (the measured execution times for the six-stripe figure are tabulated as a percentage of the part-column algorithm, the slowest of the three) and, since it performs at least as well as the full-column method, may prove to be very useful in applications. Results for larger grid sizes are shown in Table 2.

The superior performance of the part-column method in some instances can be explained by the contours of grid weights in those cases. The L-shape figure cannot be exploited by the part-column method since load balancing will ultimately cause the part-column algorithm to select partitions with approximately the same weight as in one column of the full-column method. Table 4 shows costs produced for various numbers of processors when the six-stripe shape is used. As we might expect, the difference between the part- and full-column methods first increases as the number of processors increases and later decreases as the marginal benefit of adding additional processors decreases.

## 5 Communication Costs

In this section we briefly explain how the optimal algorithm may be generalized to partition edge weighted task graphs. The problem formulation in Section 2 defined the weight of each node to reflect the amount of computation required by that program module. In a similar manner, each edge can be assigned a weight denoting the amount of communication on the edge if the two modules are mapped to adjacent processors. We assume that the communication cost between modules mapped to the same processor is negligible. Thus, the weights of the four edges from node $v_{i,j}$ can be denoted as: $c^n_{i,j}$ – the weight of the north edge $(v_{i,j}, v_{i-1,j})$; $c^s_{i,j}$ – the weight of the south edge $(v_{i,j}, v_{i+1,j})$; $c^e_{i,j}$ – the weight of the east edge $(v_{i,j}, v_{i,j+1})$; and $c^w_{i,j}$ – the weight of the west edge $(v_{i,j}, v_{i,j-1})$.

The time taken by a processor, under a mapping, is a function of the execution (computation) and communication times. As defined before, the execution time is the sum of the weights of the nodes in the partition. The communication time is in general a function of the amount of communication and the distance which a message travels. If we place the condition that communicating/adjacent nodes are mapped onto adjacent processors, then this time can be modelled as the sum of the edge weights that fall across the adjacent processors. The time taken by a processor will depend also on the the communication model. In this paper, we assume that a processor cannot overlap computation and communication, and furthermore all communication must be synchronized, *i.e.,* we assume a blocking send-receive model of communication. Under this architectural model the time taken by a processor is simply the sum of the execution and communication times.

Under the blocking send-receive model, if an edge with weight $c^x_{i,j}$ in the task graph is mapped to adjacent processors then the cost $c^x_{i,j}$ is incurred by both processors. Now consider the set of exposed edges in the partition $< v_{i,j}, v_{k,l} >$. Firstly, we observe that any partition $< v_{i,j}, v_{k,l} >$ mapped to $P_x$, where $1 < x < p$, will expose a total of $2m + 2$ edges. This is illustrated in Figure 9. The exposed edges consist of six different sets (some of these sets could be empty if $v_{i,j}$ or $v_{k,l}$ is a boundary node in the GSC). Each of the nodes $v_{i,j}$ and $v_{k,l}$ has one exposed edge in their column, namely the edges $(v_{i,j}, v_{i-1,j})$ and $(v_{k,l}, v_{k+1,l})$ respectively. Now consider the "left exposed" column, *i.e.,* the edges on the left boundary of the partition. These edges fall into two sets: edges in $L_u = \{(v_{x,j}, v_{x,j+1}) | 1 \leq x < i\}$ and the edges in $L_d = \{(v_{x,j}, v_{x,j-1}) | i \leq x \leq m\}$. We have a similar situation for the "right exposed" column which consists of edges in the set $R_u = \{(v_{x,l}, v_{x,l+1}) | 1 \leq x \leq k\}$ and $R_d = \{(v_{x,l}, v_{x,l-1}) | k < l \leq m\}$. The communication time $T_c(v_{i,j}, v_{k,l})$ on the processor is the sum of all these edge weights, and so

$$T_c(v_{i,j}, v_{k,l}) = c^n_{i,j} + c^s_{k,l} + \sum_{1 \leq x < i} c^e_{x,j} + \sum_{i \leq x \leq m} c^w_{x,j} + \sum_{1 \leq x \leq k} c^e_{x,l} + \sum_{k < x \leq m} c^w_{x,l}. \tag{3}$$

The time, or load, $T(v_{i,j}, v_{k,l})$ on a processor assigned the partition of modules $< v_{i,j}, v_{k,l} >$ is the sum of the execution and communication times, and thus

$$T(v_{i,j}, v_{k,l}) = T_e(v_{i,j}, v_{k,l}) + T_c(v_{i,j}, v_{k,l})$$

.

The above definition of the load can be used to define the weight of the edges in the assignment graph; thus, weight of the edge $(u_i, u_j)$ is the load $T(u_{i+1}, u_j)$. Now consider the computation of the edge weights by the algorithm. We can use the technique, used earlier, of "precomputing" some values.

To compute $T_c(i,j)$, the communication cost of partition $< i,j >$, we can precompute an array of values. Let $C_i$, for $1 \le i \le m$, be an $m$ element array, and let $C_i[j]$, for $1 \le i,j \le m$, denote the sum of the northernmost $j$ east edges in column $i$ (*i.e.*, the $j$ northernmost edges between modules in column $i$ and column $i+1$). This is defined as:

$$C_i[j] = \sum_{1 \le l \le j} c^e_{l,i}.$$

For any $i$ and $j$, $C_i[j] = C_i[j+1] - c^e_{i,j+1}$. This relation can be used to compute, for any $i$, all $O(m)$ values of $C_i[j]$, for $1 \le j \le m$, in $O(m)$ time (by first computing $C_i[m]$) Computing the set of $m$ arrays $C_i$, for each $i$, $1 \le i \le m$, takes $O(M)$ time. These arrays can be used to compute the sum of the edges between row $k$ to row $l$ (between column $i$ and column $i+1$) in constant time, since this is equal to $C_i[l] - C_i[k-1]$.

Let modules $v_{i_1,j_1}$ and $v_{i_2,j_2}$ correspond to the relabelled nodes $u_i$ and $u_j$ respectively; *i.e.*, $T_c(i,j) = T_c(v_{i_1,j_1}, v_{i_2,j_2})$. Using the precomputed values in the arrays $C_k[l]$, each of the four summation terms in equation 3, for $T_c(v_{i_1,j_1}, v_{i_2,j_2})$, can be rewritten as:

- $\sum_{1 \le l < i_1} c^e_{l,j_1} = C_{j_1}[i_1 - 1]$
- $\sum_{i_1 \le l \le m} c^w_{l,j_1} = (C_{j_1-1}[m] - C_{j_1-1}[i_1 - 1])$
- $\sum_{1 \le l \le i_2} c^e_{l,j_2} = C_{j_2}[i_2]$
- $\sum_{i_2 < l \le m} c^w_{l,j_2} = (C_{j_2-1}[m] - C_{j_2-1}[i_2 - 1])$

All four summation terms can be computed in constant time using the above equations. Therefore the value of $T_c(i,j)$ can be computed in constant time, as can the value of $T_e(i,j)$. Consequently, $D^1(i,j) = T_e(i+1,j) + T_c(i+1,j)$ can be computed in constant time for all $1 \le i < j \le M$, and the algorithm still takes $O(M^2p)$ time.

## 6 Conclusions

This paper discussed the problem of partitioning and mapping a grid of nodes onto a linear array. We showed that finding an optimal mapping which satisfies the contiguity constraint is NP-complete, thereby suggesting that more constraints (on the type of partitions allowed) are required to make the problem tractable. Subsequently we presented a new scheme, part-column partitioning, for partitioning and mapping a grid of $M$ nodes to a linear array of $p$ processors, and developed an $O(M^2p)$ algorithm that finds an optimal mapping satisfying this constraint. We then provided an $O(M)$ heuristic with a worst-case performance bound of 2, but then demonstrated in our experiments that the heuristic performs better than its worst-case bound. The performance of our partitioning scheme was experimentally compared to the previous method of partitioning the grid along entire columns, the full-column partitioning scheme. Experiments revealed that our

Figure 9: Exposed Edges in a Part-Column Partition.

partitioning scheme can provide upto 40% improvements in decreased partitioning cost. Future endeavors include more general task and processor graphs.

## References


[1] M. Annaratone, E. Arnould, T. Gross, H. Kung, M. Lam, O. Menzilcioglu, and J. Webb. The Warp Computer: Architecture, Implementation, and Performance. *IEEE Trans. Comput.*, C-36 (Dec. 1987), pp. 1523–1538.

[2] M. J. Berger and S. H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. Comput.*, C-36 (May 1987), pp. 570–580.

[3] S. H. Bokhari. On the Mapping Problem. *IEEE Trans. Comput.*, C-30 (Mar. 1981), pp.207–214.

[4] S. H. Bokhari. Partitioning Problems in Parallel, Pipelined and Distributed Computing. *IEEE Trans. Comput.*, C-37 (Jan. 1988), pp. 48–57.

[5] S. Bollinger and S. Midkiff. Heuristic Technique for Processor and Link Assignment in Multicomputers. *IEEE Trans. Comput.*, 40 (Mar. 1981) pp.325–336.

[6] T. F. Chan and Y. Saad. Multigrid Algorithms on the Hypercube Multiprocessor. *IEEE Trans. Comput.*, C-35 (1986).

[7] H.-A. Choi, and B. Narahari. Algorithms for Mapping and Partitioning Chain Structured Parallel Computations. in *Proc. of Int. Conf. on Parallel Processing*, St. Charles, Illinois, Aug. 1991.

[8] A. N. Choudhary and J. H. Patel. *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*, *Kluwer Academic Publishers*, Boston, MA, 1990.

[9] S. Even. *Graph Algorithms*, *Computer Science Press*, 1979.

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

[11] M. A. Iqbal and S. H. Bokhari. Efficient Algorithms for a class of Partitioning Problems. *ICASE Report No.90-49*, July 1990.

[12] Y-W. E. Ma and L. Tao. Embeddings among Torus and Meshes. in *Proc. Int. Conf. on Parallel Processing*, St. Charles, Illinois, August 1987.

[13] Y-W. E. Ma and R. Krishnamurti. The Architecture of REPLICA - A Special Purpose Computer System for Active Multi-Sensory Perception og 3-Dimensional Objects. in *Proc. 11th Annual Int. Symp. on Computer Architecture*, 1984, pp.30–37.

[14] L. M. Ni and C-T. King. On Partitioning and Mapping for Hypercube Computing. *Int. Journal of Parallel Programming*, Dec. 1988.

[15] D. M. Nicol. Rectilinear Partitioning of Irregular Data Parallel Computations. *Technical Report*, College of William & Mary, 1991.

[16] D. M. Nicol and D. R. O'Hallaron. Improved Algorithms for Mapping Pipelined and Parallel Computations. *IEEE Trans. Comput.*, March 1991.

[17] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and Problems Partitionings: Their influences on the performance of multiple processor systems. *IEEE Trans. Comput.*, C-36 (july 1987), pp. 845–858.

[18] P. Sadayappan, and F. Ercal. Nearest-neighbor Mappings of Finite element graphs onto Processor Meshes. *IEEE Trans. Comput.*, C-36 (Dec. 1987) ,pp. 1408–1424.

[19] S. Sahni and E. Horowitz. *Fundamentals of Computer Algorithms*. *Computer Science Press*, Boston, MA.

[20] C. C. Shen and W. H. Tsai. A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems. *IEEE Trans. Comput.*, C-34 (Mar. 1987), pp. 197–203.