An efficient dynamic load balancing algorithm for adaptive mesh refinement

Ravi Varadarajan
Dept. of Mathematics,
University of Puerto Rico
Rio Piedras, PR 00931
Email : r_varadaraja@upr1.upr.clu.edu
Fax : (809) 754-0757

Injae Hwang
Comp. and Info. Sciences Department,
University of Florida
Gainesville, FL 32611
Email : ih@cis.ufl.edu

May 26, 1994

Abstract

In numerical algorithms based on adaptive mesh refinement, the computational workload changes during the execution of the algorithms. In mapping such algorithms on to distributed memory architectures, dynamic load balancing is necessary to balance the workload among the processors in order to obtain high performance. In this paper, we propose a dynamic processor allocation algorithm for a mesh architecture that reassigns the workload in an attempt to minimize both the computational and communication costs. Our algorithm is based on a heuristic for a 2D packing problem that gives provably close to optimal solutions for special cases of the problem. We also demonstrate through experiments how our algorithm provides good quality solutions in general.

1 Introduction

Parallel computers with thousands of processors are becoming a commercial reality due to advanced VLSI technolgies. These machines can provide tremendous computational power needed to solve many computationally intensive problems that arise in different fields of science and engineering such as aerodynamics, nuclear physics and weather forecasting. In solving such problems, it is necessary to distribute the computational load among the processors in order to achieve good performance as measured by speed-up, processor utilization etc.. The term "load balancing" refers to this activity of distributing or redistributing the computational load in order to achieve high performance.

In multiprocessor computer systems with distributed memory, load balancing should also take into consideration the interprocessor communication cost that arises when for example, a computation in a processor needs data located in another processor's memory. For simple applications, it is sufficient for the programmer to perform *static* load balancing by properly distributing the data before the parallel program begins its computation. But there are many engineering applications wherein simple static load balancing techniques are not sufficient to ensure high performance. A case in point is the solution of partial differential equations using *adaptive meshes*.

In numerical methods that employ finite differences to solve such problems, a rectangular mesh is imposed on the computational domain and the solution is evaluated at the mesh points; the mesh size (or spacing between mesh points) is related to the required error tolerance in the solution. It is computationally advantageous to have mesh points spaced more closely in regions with steep gradients and have coarser mesh points in the regions where the solution is smooth. In many problems (for example, flame propagation in combustion), the regions of steep gradient are difficult to predict before the computation begins. For this reason, we start with a uniform mesh of coarse size and finer and finer meshes are dynamically imposed on regions with large estimated errors as the solution proceeds. Parallelization of adaptive mesh algorithms thus introduces frequent changes in workload distribution among the processors and hence requires *dynamic load balancing* to achieve high performance. In this paper, we address the issue of dynamic load balancing that arises particularly in mapping adaptive mesh refinement algorithms onto parallel architectures.

There are many load balancing algorithms (such as binary decomposition [7], scatter decomposition [28]) proposed in the literature for numerical algorithms based on finite difference or finite element methods. These algorithms are more suitable for static load balancing as they ignore the interprocessor communication cost that usually results from relocating the intermediate solution values among the processors during rebalancing. There are other

general dynamic load balancing algorithms (such as diffusion scheme [20]) suitable for slowly varying workload distributions, as the time to rebalance the workload is quite high in these methods.

In this paper, we propose dynamic load balancing algorithms especially for adaptive mesh refinement, that *explicitly consider the computational and communication costs in rebalancing the workload.* The grids generated dynamically during the computation are assumed to be rectangular. We use two-dimensional packing problem to allocate a set of processors for each grid in mesh multiprocessor system. The set of processors is chosen in such a way that computational workload among the processors is balanced and intra-grid communication cost is minimized at the same time. We propose an efficient heuristic packing algorithm and show how it can be used for processor allocation.

A word about the organization. In the next section, we briefly discuss some of the previously proposed load balancing algorithms. In Section 3, we formulate the dynamic load balancing problem that arises in mapping adaptive mesh computations onto parallel computers. In Section 4, we propose a simple heuristic and two approaches for implementing it on the parallel machines. In Sections 5 and 6, we describe the two approaches and analyze their complexities for hypercube architectures. In Section 7, we give the results of our experiments performed to evaluate our heuristic methods with respect to some well-known scheduling heuristics. Finally, we summarize our paper with a brief discussion of scope of future work.

2 Previous work

In this section, we discuss some of the load balancing techniques proposed in the literature for mapping numerical computations onto parallel architectures. These techniques partition the computational domain into pieces (subdomains) which are then allocated to the processors. For static load balancing, communication between two processors is usually assumed to be proportional to the number of points on the boundaries between the subdomains allocated to the processors. In these techniques, balancing the load among the processors is given more importance than minimizing the inter-processor communication cost.

In [7], Berger and Bokhari propose *binary decomposition* for distributing the workload associated with non-uniform mesh computations in a multiprocessor system with a mesh topology. It is a recursive partitioning of the domain into vertical and horizontal strips alternatively so that the left and right or top and bottom segments that result from the cut have approximately equal load; load here is measured by the number of mesh points. Number of processors is assumed to be a power of two in this method. This technique,

while useful for static load balancing, has limited applicability as a dynamic load balancing method for adaptive meshes since it ignores the cost of migrating intermediate solution values that occurs during reallocation of mesh points to the processors. For minor imbalances in workload, this technique can perform efficient rebalancing by adjusting the cut lines starting with the smallest-sized strips. It is not specified how binary decomposition can be efficiently implemented in parallel on a multiprocessor system.

Scatter decomposition is another static load balancing technique which accounts for minor fluctuations in workload distribution that tend to concentrate in a few contiguous regions of the computational domain. In this case, the rectangular domain is divided into (say) m rows and n columns of equal width. Let r_{ij} be the region in the i-th row and j-th column, where $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$. Then for a multiprocessor system with $p \times q$ mesh (assuming p divides m and q divides n), we allocate the computations of r_{ij} to processor indexed (s,t) where $s = i \bmod m$ and $t = j \bmod n$, where $0 \leq s \leq p-1$ and $0 \leq t \leq q-1$. Larger the ratio of $\frac{mn}{pq}$, better balanced the workload will be but higher the communication cost among the processors; this trade-off very much depends on the application. Determining this trade-off is a major difficulty with this method. This technique was used as a load balancing method by Williams [32] and was analyzed for a probabilistic workload by Nicol and Saltz [28].

In [29], Simon proposes the application of graph-theoretic techniques for mapping unstructured grid calculations that arise in computational fluid dynamics onto multiprocessor architectures. These techniques recursively apply a procedure that partitions a planar graph into two halves with roughly equal number of nodes such that the number of edges in the cutset (measure of communication cost) is attempted to be minimized. In one technique called "recursive graph bisection", nodes are partitioned based on their distances from an *extremal* vertex which is the root of a breadth-first spanning tree with maximum height. In another technique called "recursive spectral bisection", partitioning is done along the eigen vector associated with the second largest eigen value of the *Laplacian matrix* $A - D$, where A is the adjacency matrix and D is the diagonal matrix of vertex degrees. Recursive spectral bisection is seen to provide a desirable graph partition.

There are many distributed dynamic load balancing strategies of which "diffusion method" is one example. It is a local balancing operation, wherein each processor independently moves (receives) workload to (from) its neighbors based on only the information about its workload and that of its neighbors. This method was used for molecular dynamics simulation applications [11]. A major disadvantage of this method is that it takes a long time to achieve global balancing but this delay is acceptable when the workload distribution changes slowly. Cybenko [20] showed the conditions for convergence of this method and the rate of convergence for arbitrary processor network topologies.

Hanxledon and Scott [25] propose a decentralized dynamic strategy in which each processor does local balancing based on the total workload information that it receives from every other processor. This strategy was tested on Intel iPSC/2 for WaTor, a particle migration application and was found to give good results.

The technique we propose in this paper is somewhat similar to binary decomposition but it *explicitly* takes into account inter-processor communication cost that occurs in rebalancing the workload. We also propose *efficient parallel implementations* of this technique. It is a technique tailor-made for adaptive mesh refinement applications.

3 Proposed load balancing approaches

In a typical adpative mesh refinement algorithm, the given partial differential equation is discretized by imposing initially a uniform mesh on the computational domain; we call this mesh a coarse grid at level 0. Letting i to be equal to 0 initially, the following steps are then repeatedly applied.

1. Obtain solution values at the mesh points of the grids at level i, using finite difference schemes that involve 4 or 8 mesh neighbors.

2. If $i \neq 0$, project solution values of level i grids onto grids at level $i-1$, for updating solution values at level i mesh points.

3. Estimate errors at mesh points of level i grids and if the estimated errors at all the mesh points of level i are within the acceptable limit, terminate the algorithm.

4. Otherwise, in regions of high error, define grids at level $i+1$ with finer mesh spacing (assuming a constant ratio between level i and $i+1$ mesh spacings). One common approach is to flag high-error points, cluster them into groups using a suitable clustering algorithm and define a finer grid for each group.

5. Define boundary values for grids at level $i+1$ by interpolating intermediate solution values at level i.

6. Let $i = i+1$ and go to step 1.

In the formulation of our load balancing problem, we use the following assumptions:

1. A grid at level i $(i > 0)$ is completely contained inside a grid at level $i-1$. This is a most commonly used assumption in adaptive mesh refinement for simplification of numerical algorithms. In this case, we call the two grids "child" and "parent" grids respectively.

2. Grid computations are synchronized such that grid computations at level i start at the same time. Dynamic load balancing is performed whenever finer grids are created and before their computations are started.

3. Computational cost for each grid depends only on the number of mesh points it contains. This cost includes time for steps 1,3, 4 and 5 above. For simplicity, we will assume that the cost is a linear function of the number of points, though our formulation can be used for any *known* function of number of points.

4. There are two types of inter-processor communication called "intra-grid communication" and "inter-grid communication" respectively. Intra-grid communication is needed whenever computation of a value at a mesh point (during step 1 above) in a processor needs a value at a neighboring mesh point located in another processor; both the mesh points belong to the same grid at some level. Inter-grid communication occurs during steps 2 and 5 whenever a child grid is located in a different processor than that of its parent. Communication cost is directly proportional to the distance between the processors in the network and we ignore traffic on the communication links.

We are currently investigating the following three approaches for mapping the adaptive mesh computations. Each approach has its own merits and demerits.

In the first approach we call the "grid indivisibility" approach, we allocate all the mesh points of a grid to the same processor; thus a grid is an indivisible computational entity that can be treated as a *process*. Creation of a finer grid is analogous to the creation of a child process which can either be allocated to the same processor as the parent process or migrated to a different processor by the load balancing process. A major advantage of this approach is that in addition to being simple to implement, intra-grid communication (which occurs more frequently than inter-grid communication) is avoided. But the number of grids at a level must be at least the number of processors to keep all the processors busy and this will not happen at least for the first few levels.

In the second approach we call the "domain decomposition" approach, computational domain is divided into regions (possibly using scatter decomposition) and the mesh computations of the grids at all levels associated with a region are assigned to the same processor. Whenever finer grids are created, we perform new decomposition for balancing the workload associated with finer grids. This new decomposition may require migration of coarse (previous level) grid mesh points to different processors. But this approach eliminates inter-grid communication at the expense of this migration cost. It introduces intra-grid communication however since the mesh points of a grid may be allocated to more than one processor. The

load balancing problem here is similar to the problem considered in [31] for load balancing with *resource migration* in distributed systems if the coarse grids are treated as resources.

The third approach we call the "grid partitioning" approach is a more general approach than the previous two methods but the load balancing problem formulation is more complicated and its solution difficult to implement. In this approach, we allocate a set of processors for each grid and within the set of processors, the grid is uniformly partitioned into pieces and each piece is assign to a processor.

In this paper, we propose a solution to the load balancing problem formulated using the third approach. This approach requires that the number of processor be larger than the number of grid at a level of refinement.

3.1 Problem Formulation

In this section, we consider the grid assignment for a two-dimensional adaptive mesh algorithm. When we formulate our load balancing problem, we assume that all two-dimensional grids are *rectangular*. We also assume that target machines are two-dimensional mesh multiprocessors. Later in this chapter, we will consider hypercube multiprocessors.

Let m be the number of grids at level i. If we allocate $X_j \times Y_j$ submesh for each grid j close to the processors where grid j was generated, the submesh of one grid can overlap with submeshes of other grids especially when grids are closely located to each other. Then, the amount of workload for the overlapping processors will be twice as much as those of other processors. To avoid this workload imbalance, some of the grids may have to be assigned to processors which are distant from the processors where they were generated. These assignments will incur inter-grid communication at some distance which can not be expressed in terms of X_j and Y_j. To define the problem more formally, we introduce the following notations :

$G = (V, E)$ — processor graph where V is the set of processor nodes and E the set of communication links

S — set of fine grids generated at level i

$f : S \rightarrow 2^V$ — $f(a)$ is the neighborhood of processors where grid a was generated.

$f'(a)$ — processors (subset of $f(a)$) having boundary values for grid a before load balancing

U_{comp} — computational cost per mesh point; depends on the finite difference method used

U_{comm} — cost of transmitting a mesh point value between neighboring processors

$d(p_1, p_2)$ — communication distance between processors p_1 and p_2 in G

$d'(V_1, V_2)$ — $\max_{p_1 \in V_1, p_2 \in V_2} d(p_1, p_2)$

F — frequency of intra-grid communication during level i computation

$g : S \rightarrow 2^V$ — grid assignment function to be determined by load balancing

$g'(a)$ — processors (subset of $g(a)$) having boundary values for grid a after load balancing

S_g — set of subgrids created by assignment g

M_b — number of mesh points in the subgrid b

B_b — number of mesh points on the boundary of subgrid b

$h : V \rightarrow S_g$ — $h(p)$ is the subgrid assigned to processor p

After the partitioning and assignment of grids at level i, the computational workload E_p for processor p is given by $U_{comp} M_{h(p)}$. Note that we require that no more than one subgrid be assigned to a processor. For a fine grid a, inter-grid communication cost $C_1(a)$ is estimated to be $U_{comm}\,(d'(f'(a), g'(a))B'(a) + d'(f(a), g(a))M'(a))$ where $B'(a) = \frac{B_a}{\min(|(f'(a)|, |g'(a)|)}$ and $M'(a) = \frac{M_a}{\min(|(f(a)|, |g(a)|)}$; the first term is the cost of migrating the boundary values for fine grid a that occurs before the computation of mesh point values in a and the second term is the cost of sending the fine grid values back to the parent grid for updating. Whenever a grid a is allocated to more than one processor, for each of its subgrids b, intra-grid communication is needed whose cost $C_2(b)$ is estimated to be $U_{comm} F B_b$ where B_b is the number of mesh points (interior to grid a) on the boundary of b. Our problem is then formally stated as follows :

Find g, S_g and hence h so as to minimize $\max_{p \in V} (E_p + C_2(h(p))) + \max_{a \in S} C_1(a)$ **with the restriction that the grids are assigned to** *disjoint sets* **of processors, i.e. for any two grids a_1 and a_2, $g(a_1) \cap g(a_2) = \emptyset$.**

As was stated before, we restrict our attention to the case where each fine grid a_j ($1 \leq j \leq m$) is *rectangular* in shape with dimensions $w_j \times h_j$ and the processor graph G is a *mesh* network with P rows and Q columns (assuming $P \geq Q$ without loss of generality). Since there is *at most one subgrid* assigned to a processor, $m \leq PQ$. The problem is then posed as follows :

Determine for each grid a_j, dimensions $X_j \times Y_j$ of the processor submesh to be allocated to the grid and l_j, the index of the processor in the south-west corner of the submesh so as to minimize $\max_{1 \leq j \leq m} (U_{comp} \frac{w_j h_j}{X_j Y_j} + 2 U_{comm} F (\frac{w_j}{X_j} + \frac{h_j}{Y_j})) + \max_{a_j \in S} C_1(a_j)$**.** In the rest of this chapter, $\frac{x}{y}$ denotes $\lceil \frac{x}{y} \rceil$ when we discuss the formulation of the problem.

For the simplicity of the problem, we ignore inter-grid communication cost and seek solutions which minimize computation cost and intra-grid communication cost. We will

discuss the extension of the approach to include inter-grid communication cost at the end of this chapter. If we remove the term representing inter-grid communication cost from the problem formulation, our objective function decreases as X_j and Y_j increase. Consequently, it is advantageous to assign a submesh which is as large as possible for each grid a_j. It is also necessary that $\frac{w_j h_j}{X_j Y_j}$ be equal to $\frac{w_k h_k}{X_k Y_k}$ for all pairs of j and k to balance the workload among the processors. From the above discussion, we have the following two desirable properties of processor allocation.

1. $\sum_{j=1}^{m} X_j Y_j = N$,
2. $\forall j, k \;\; \frac{w_j h_j}{X_j Y_j} = \frac{w_k h_k}{X_k Y_k}$.

When the total number of grid points at level i is larger than the number of processors, the number of processors $X_j Y_j$ for each grid j is limited by the above two properties. Let $N_j = X_j Y_j$. Then the intra-grid communication cost for grid a_j is as follows.

$$C_2(a_j) \approx 2U_{comm} F \left(\frac{w_j}{X_j} + \frac{X_j h_j}{N_j} \right)$$

Then,

$$\frac{dC_2(a_j)}{dX_j} = 2U_{comm} F \left(\frac{-w_j}{{X_j}^2} + \frac{h_j}{N_j} \right)$$

$\frac{dC_2(a_j)}{dX_j}$ becomes 0 when $X_j = \sqrt{\frac{N_j w_j}{h_j}}$. Since $Y_j = \frac{N_j}{X_j}$, $Y_j = \sqrt{\frac{N_j h_j}{w_j}}$. Then it follows that $\frac{w_j}{X_j} = \frac{h_j}{Y_j}$. To minimize the cost of intra-grid communication separately for each grid (for a fixed number of processors assigned to the grid), we add the third property of processor allocation as follows.

3. $\forall j \;\; \frac{w_j}{X_j} = \frac{h_j}{Y_j}$

Now, our problem is approximated to finding m submeshes and their locations, one for each grid, which satisfy the above three properties. This problem is formally stated as follows :

For each grid a_j of dimension $w_j \times h_j$, find $X_j \times Y_j$ processor submesh to be allocated to the grid and l_j, the index of the processor in the south-west corner of the submesh so as to maximize $\sum_{j=1}^{m} X_j Y_j$ with the restriction that $\forall j, k \;\; \frac{w_j h_j}{X_j Y_j} = \frac{w_k h_k}{X_k Y_k}$ and $\forall j \;\; \frac{w_j}{X_j} = \frac{h_j}{Y_j}$. We will call this problem *submesh allocation problem.*

4 Processor Allocation Using Two-Dimensional Packing

In this section, we propose an efficient approach to find an approximate solution to the problem formulated in the previous section. This approach views the problem as a two-dimensional packing problem wherein rectangles need to be packed in a bin of infinite width and height. We present an approximate heuristic for this packing problem and show how the approximate solution can be used to allocate processor submeshes to the grids. Experimental results on the performance of the packing algorithm are included at the end of this section.

4.1 A Packing Algorithm

Two dimensional packing problem has been studied by many researchers [5, 4, 6, 18]. It arises in a variety of situations such as scheduling of tasks and cutting-stock problems. Cutting-stock problems may involve cutting objects out of a sheet or roll of material so as to minimize waste. The scheduling of tasks with a shared resource involves two dimensions, the resource and time, and the problem is to schedule the tasks so as to minimize the total amount of time used. Memory is a typical example of shared resources. In general, the problem is stated as follows :

Given a rectangular bin with fixed width and infinite height, pack a finite set of rectangles of specified dimensions into the bin in such a way that the rectangles do not overlap and the total bin height used in the packing is minimized. The rectangles should be packed with their sides parallel to the sides of the bin. In one version of the problem (see [5]) the rectangles should be packed in a fixed orientation.

In our processor allocation problem, grids correspond to rectangles to be packed. *But, the problem is slightly different from the above two-dimensional bin packing problem. First, the width as well as the height of the bin is unlimited.* Assume that we are given a $\sqrt{N} \times \sqrt{N}$ mesh multiprocessors for m grids. *The goal of our packing is minimizing the maximum of width and height of bin used for packing m grids starting from the left-bottom corner of the bin. Second, the orientation of grids is not important, so grids can be rotated when they are packed.* Figure 1 shows an example of such packing.

After packing m grids as in Figure 1, we use the two ratios, $\frac{width}{\sqrt{N}}$ and $\frac{height}{\sqrt{N}}$, to allocate a submesh for each grid. The detailed allocation method will be described later in this section. First, we will show that decision version of our 2D packing problem is NP-complete in the following theorem.

Theorem 1 : The following problem is NP-complete :

Given m rectangles of dimension $w_i \times h_i$ $(w_i \geq h_i)$, is there a packing so that the maxi-

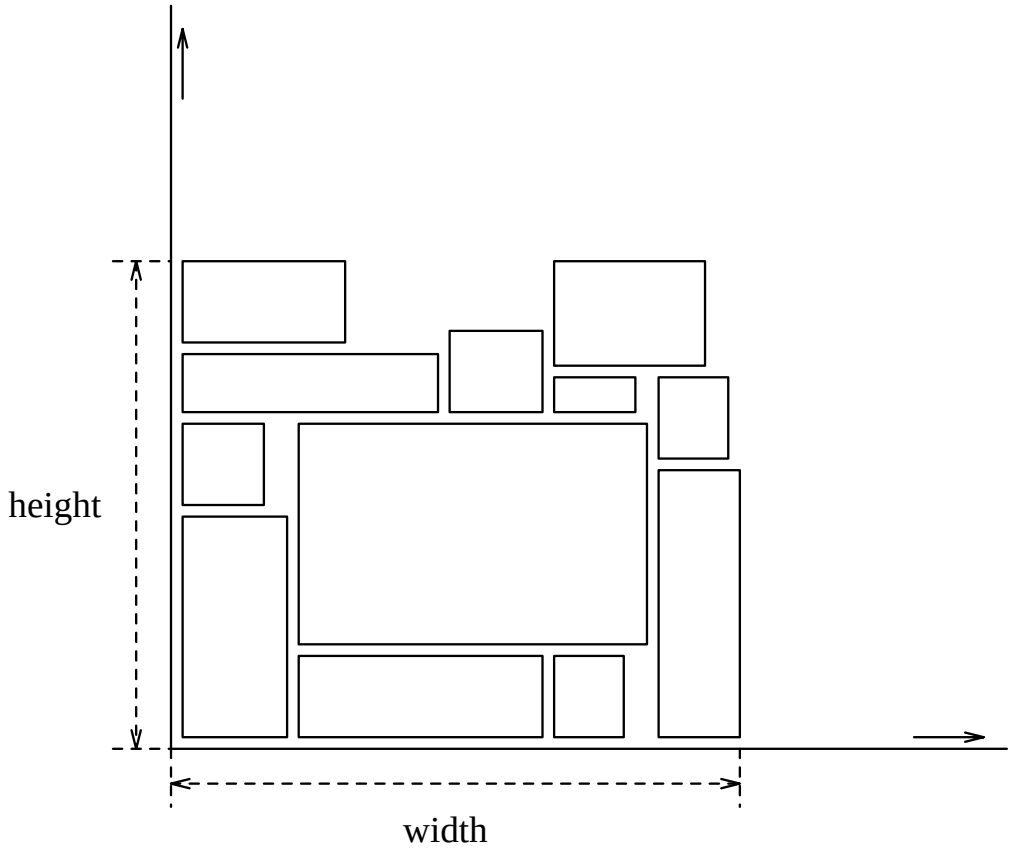

Figure 1: An Example of Packing

mum of height and width of packing is smaller than or equal to L?

Proof : We will reduce the bin packing problem which is known to be NP-complete [24] to our 2D packing problem. The bin packing problem is stated as follows: **Given k bins of capacity B, and a finite set $U = \{u_1, u_2, \ldots, u_m\}$ of items with size of each item $s(u_i) \in Z^+$ $(s(u_i) \leq B)$, is there a partition of U into disjoint sets $U_1, U_2, \ldots, U_k$ such that the sum of sizes of the items in each U_i is no more than B?** We can construct an instance of the 2D packing problem for an instance of the bin packing problem in the following way : Create $m + 1$ rectangles with the following dimensions; $w_0 = k(B+1), h_0 = k(B+1) - B$, $w_1 = B+1, h_1 = s(u_1)$, $w_2 = B+1, h_2 = s(u_2)$, $\cdots$, $w_m = B+1, h_m = s(u_m)$ and $L = k(B+1)$. Then any feasible packing of these rectangles must be in the form as in Figure 2. The largest rectangle in the figure is the first rectangle in the list. Since one dimension of the rectangle is already $k(B+1)$, other rectangles cannot be packed above it. The larger of the two dimensions is $B+1$ for all but the first rectangle, hence they can only be packed so that their longer sides are parallel to the Y-axis. For this instance of the 2D packing problem to have a solution, rectangles should be packed so that the width of the packing does not exceed $k(B+1)$. Note that each level of packing of

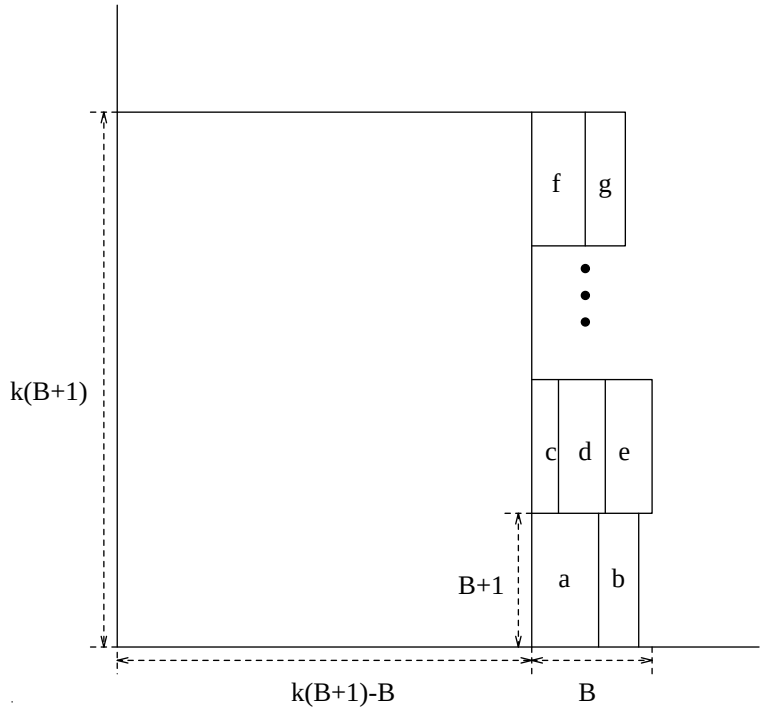

Figure 2: Feasible Packing of m+1 Rectangles

rectangles indexed from 1 to n (for example, rectangles "a", "b" form one level of packing in Figure 2) corresponds to a disjoint set U_i in the bin packing problem. Now, it is obvious that there is a solution for an instance of the bin packing problem if and only if there is a solution for the corresponding instance of the 2D packing problem. ■

Since the decision version of the problem is NP-complete, the original optimization problem is NP-hard. For this reason, we seek an approximation algorithm to solve the problem.

Since our packing problem is somewhat different from the commonly known 2D bin packing problem, the heuristic algorithms developed in [5, 4, 6, 18] are not readily useful for our purpose. One approach to solving our problem is to try different bin widths and for each width, use one of the above packing heuristics to see if the height of the packing does not exceed the desired height for that width. We can find the minimum width that allows the desired packing, by using some form of *interpolation search* on the interval between minimum and maximum widths possible. A better approach, that avoids repacking, is to use a packing heuristic that attempts to satisfy two goals, namely, packing as tightly as possible and making the width/height ratio for the packing closely match the ratio of the processor mesh. To this end, we propose a heuristic which we will call "tight-pack" (TP, for short) heuristic.

The basic idea of TP-heuristic is as follows. First the grids are sorted in some selected

order. Then we start packing grids one by one at the south-west corner of the bin. Let's consider the space of the bin as the first quadrant of X-Y plane. Then the south-west corner of the bin becomes the origin of the coordinate system, that is (0,0). Each packed grid has 4 corners NW,NE,SE and SW with respect to its orientation in the packing. A NW or SE corner of a packed grid is called a **free corner (FC)** if no other item occupies that corner such that the item is above and to the right of that corner. In our algorithm, only *free corners* are considered for packing the new grid and when a new grid is placed in a free corner, it is placed so that it is above and to the right of the corner. After packing the first grid at the origin, the next grid is packed at one of the two corners created by packing the first grid. Figure 3 (a) shows the two corners, "a" and "b". In general, after packing $w_i \times h_i$

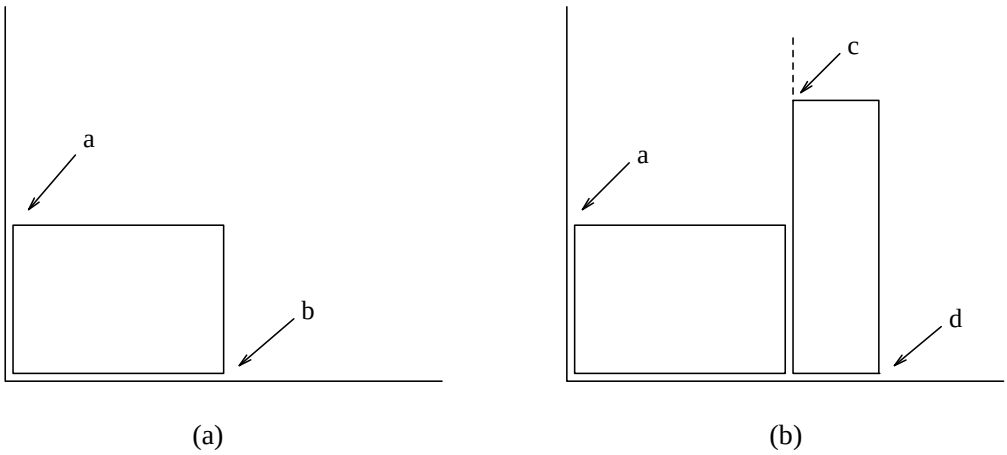

Figure 3: The Corners for Packing the Next Grid

grid at (x_j, y_j), we add two more free corners located at $(x_j, y_j + h_i)$ and $(x_j + w_i, y_j)$ to the list of free corners. We also keep the maximum size of the grid which can be packed in that free corner along with its location. We call it the "size" of free corner. Figure 3 (b) shows two new corners, "c" and "d", after packing the second grid at corner "b". After the second grid is packed, the width of the grid which can be packed at corner "a" is limited to the width of the first grid. Hence, we need to update that information whenever we pack a grid. Let W_i and H_i be the width and height of the space used in packing after packing the first i grids. We choose the corner for the $i+1$-st grid, so that the maximum of W_{i+1} and H_{i+1} is minimized. *If we are given a processor mesh $P \times Q$ with $P \geq Q$ and $\frac{P}{Q} = R$, we choose the corner for the $i+1$-st grid, so that the maximum of W_{i+1} and RH_{i+1} is minimized.* The detailed description of our packing algorithm is given bellow. Assume that we are given the mesh ratio R and m grids, $w_1 \times h_1, w_2 \times h_2, \ldots, w_m \times h_m$. After the following algorithm terminates, global variables, $XLoc_i$ and $YLoc_i$, contain the location of the corner where grid i was packed. $Orient_i$ is set to 1 if grid i was rotated and it is set to 0 otherwise.

Algorithm Packing;

1. $width = 0$
2. $height = 0$
3. Initialize the three lists $CornerList$, $XScanList$ and $YScanList$ to be empty.
4. Insert $[(0,0),(\infty,\infty)]$ into $CornerList$.
   // $[(x,y),(p,q)]$ represents the corner located at (x,y) and the maximum grid size that can be packed is $p \times q$. //
5. for each $w_i \times h_i$ grid do
   begin
6. **FindBestCorner**$((w_i,h_i),k)$
   // In procedure **FindBestCorner**, each corner in $CornerList$ is examined and a corner $[(x_k,y_k),(p_k,q_k)]$ is chosen which minimizes the maximum of $width$ and $height * R$ after $w_i \times h_i$ grid is packed //
7. **UpdateCornerList**$((w_i,h_i),k)$
   // In procedure **FindBestCorner**, the size of each corner in $CornerList$ is updated after $w_i \times h_i$ grid is packed at corner $[(x_k,y_k),(p_k,q_k)]$ //
8. **FindNewCornerSize**$((w_i,h_i))$
   // After $w_i \times h_i$ grid is packed at corner $[(x_k,y_k),(p_k,q_k)]$, two new corners, $[(x_k,y_k+h_i),(p_{k'},q_{k'})]$ and $[(x_k+w_i,y_k),(p_{k''},q_{k''})]$ are created. In procedure **FindNewCornerSize**, the size of each corner is determined. //
   end

end Packing;

Procedure FindBestCorner$((w_i,h_i),k)$;

1. $mindim = \infty$
2. for each element $[(x_j,y_j),(p_j,q_j)]$ in $CornerList$ do
   begin
3. $mx1 = \infty, \quad my1 = \infty, \quad mx2 = \infty, \quad my2 = \infty$
   // Try to pack a grid $(w_i \times h_i)$ at a free corner (p_j,q_j) in both orientations. //
4. if $(p_j - w_i) \geq 0$ and $(q_j - h_i) \geq 0$ then
5. $mx1 = \max(width, x_j + w_i), \quad my1 = \max(height, y_j + h_i)$
6. else if $(p_j - h_i) \geq 0$ and $(q_j - w_i) \geq 0$ then
7. $mx2 = \max(width, x_j + h_i), \quad my2 = \max(height, y_j + w_i)$
8. else goto the next iteration

9. $m1 = \max(mx1, R * my1)$
10. $m2 = \max(mx2, R * my2)$
11. $m = \min(m1, m2)$
12. if $(m < mindim)$ then
    begin
13. $mindim = m, \quad k = j, \quad XLoc_i = x_j, \quad YLoc_i = y_j$
14. if $(m1 \leq m2)$ then
15. $width = mx1, \quad height = my1, \quad Orient_i = 0$
    else
16. $width = mx2, \quad height = my2, \quad Orient_i = 1$
    end
end

end FindBestCorner;

Procedure UpdateCornerList$((w_i, h_i), k)$;

1. Remove $[(x_k, y_k), (p_k, q_k)]$ from $CornerList$
2. if $Orient_i = 1$ then
3. $w'_i = h_i, \quad h'_i = w_i$
   else
4. $w'_i = w_i, \quad h'_i = h_i$

   // If the grid $(w_i \times h_i)$ packed at k-th free corner overlaps with the packing area of a free corner $[(x_j, y_j), (p_j, q_j)]$, update the size of that free corner. //
5. for each element $[(x_j, y_j), (p_j, q_j)]$ in $CornerList$ do
   begin
6. if $(y_k \leq y_j < y_k + h'_i)$ and $(x_j \leq x_k < x_j + p_j)$ then
7. $p_j = x_k - x_j$
8. if $(x_k \leq x_j < x_k + w'_i)$ and $(y_j \leq y_k < y_j + q_j)$ then
9. $q_j = y_k - y_j$
10. if $p_j = 0$ or $q_j = 0$ then
11. remove $[(x_j, y_j), (p_j, q_j)]$ from $CornerList$
    end

end UpdateCornerList;

Procedure FindNewCornerSize$((w_i, h_i))$;

// Find the sizes of two new free corners created after packing a grid (w_i, h_i) //

1. $p = \infty, \quad q = \infty$
2. if $Orient_i = 1$ then

3. $w_i' = h_i, \quad h_i' = w_i$

else

4. $w_i' = w_i, \quad h_i' = h_i$

// Find the closest grid to the new corner packed on line $y = YLoc_i$. //

5. for each element $[(x_j^1, y_j^1), (x_j^2, y_j^2)]$ in $XScanList$ do

// Each element $[(x_j^1, y_j^1), (x_j^2, y_j^2)]$ in $XScanList$ contains the locations of the SW, NW corners of a packed item. (Note that $x_j^1 = x_j^2$) //

begin

6. if $(y_j^1 \leq YLoc_i \leq y_j^2)$ and $(XLoc_i + w_i' \leq x_j^1)$ then
7. if $(x_j^1 < p)$ then
8. $p = x_j^1$

end

// Find the closest grid to the new corner packed on line $x = XLoc_i + w_i'$. //

9. for each element $[(x_j^1, y_j^1), (x_j^2, y_j^2)]$ in $YScanList$ do

// Each element $[(x_j^1, y_j^1), (x_j^2, y_j^2)]$ in $YScanList$ contains the locations of the SW, SE corners of a packed item. (Note that $y_j^1 = y_j^2$) //

begin

10. if $(x_j^1 \leq XLoc_i + w_i' \leq x_j^2)$ and $(YLoc_i \leq y_j^1)$ then
11. if $(y_j^1 < q)$ then
12. $q = y_j^1$

end

13. $p = p - (XLoc_i + w_i')$
14. $q = q - YLoc_i$
15. Insert $[(XLoc_i + w_i', YLoc_i), (p, q)]$ into $CornerList$
16. $p = \infty, \quad q = \infty$

// Find the closest grid to the new corner packed on line $y = YLoc_i + h_i'$. //

17. for each element $[(x_j^1, y_j^1), (x_j^2, y_j^2)]$ in $XScanList$ do

begin

18. if $(y_j^1 \leq YLoc_i + h_i' \leq y_j^2)$ and $(XLoc_i \leq x_j^1)$ then
19. if $(x_j^1 < p)$ then
20. $p = x_j^1$

end

// Find the closest grid to the new corner packed on line $x = XLoc_i$. //

21. for each element $[(x_j^1, y_j^1), (x_j^2, y_j^2)]$ in $YScanList$ do

begin

22. if $(x_j^1 \leq XLoc_i \leq x_j^2)$ and $(YLoc_i + h_i' \leq y_j^1)$ then

23. if $(y_j^1 < q)$ then
24. $q = y_j^1$

end

25. $p = p - XLoc_i$
26. $q = q - (YLoc_i + h_i')$
27. Insert $[(XLoc_i, YLoc_i + h_i'), (p, q)]$ into $CornerList$
28. Insert $[(XLoc_i, YLoc_i), (XLoc_i, YLoc_i + h_i')]$ into $XScanList$
29. Insert $[(XLoc_i, YLoc_i), (XLoc_i + w_i', YLoc_i)]$ into $YScanList$

end FindNewCornerSize;

Every time a grid is packed, a corner is used and two new corners are introduced. Since there is only one corner initially, the number of elements in $CornerList$ is at most $m + 1$. An element is added to $XScanList$ as well as to $YScanList$, each time a grid is packed. Hence, the number of elements in $XScanList$ and $YScanList$ can be at most m. As a result, Procedures **FindBestCorner, UpdateCornerList** and **FindNewCornerSize** take $O(m)$ time. Since there are m grids to be packed, the total time complexity of our packing algorithm is $O(m^2)$.

Remark : A small improvement can be made if we remove corners which are too small to be useful from $CornerList$. Let MINHOLESIZE = $\min(\min_{1 \le i \le m} w_i, \min_{1 \le i \le m} h_i)$. In procedure **UpdateCornerList**, if either $p_j <$ MINHOLESIZE or $q_j <$ MINHOLESIZE, then remove $[(x_j, y_j), (p_j, q_j)]$ from $CornerList$. In procedure **FindNewCornerSize**, if either $p <$ MINHOLESIZE or $q <$ MINHOLESIZE, then do not insert $[(XLoc_i + w_i', YLoc_i), (p, q)]$ or $[(XLoc_i, YLoc_i + h_i'), (p, q)]$ into $CornerList$.

The following theorem shows the correctness of our packing algorithm. In the following theorem, we assume for convenience that we do not make modifications as in above remark.
Theorem 2 : After i-th grid is packed, The following two claims hold.
(1)For any element $[(x_j, y_j), (p_j, q_j)]$ in the $CornerList$, the maximum size of the grid that can be packed at corner (x_j, y_j) is $p_j \times q_j$.
(2)For any point (x,y) in the space of the bin, either it is occupied by a grid or it belongs to at least one corner in the $CornerList$.

Proof : To prove the first part of the theorem, we need to show that procedure **FindNewCornerSize** correctly computes the sizes of the two new corners. We will use induction on j, the number of grids already packed. Base case is trivial. When j is 1, both $XScanList$ and $YScanList$ are empty. Hence, the loops of lines 5 - 12 and 17 - 24 will not be executed. Both p and q remain ∞, which is the correct width and height of the maximum grid that can be packed at the new corners. Now, assume that the sizes of all the corners had been correctly computed until grid $j-1$ was packed. We pack grid j at $(XLoc_j, YLoc_j)$, and two

new corners, namely, $[(XLoc_j + w'_j, YLoc_j), (p', q')]$ and $[(XLoc_j, YLoc_j + h'_j), (p'', q'')]$ are created; here w'_j, h'_j are dimensions of grid j along X and Y directions respectively. Let us see how p' and q' are determined in procedure **FindNewCornerSize**. $XscanList$ contains the coordinates of the SW, NW corners of each grid which was already packed. We scan each element in $XScanList$ and find $[(x^1_k, y^1_k), (x^2_k, y^2_k)]$ such that $(y^1_k \leq YLoc_j \leq y^2_k)$ and $(XLoc_j + w'_j \leq x^1_k)$, and x^1_k is minimum among all such elements; if no such grid exists, then p' is correctly set to ∞. We find $[(x^1_{k'}, y^1_{k'}), (x^2_{k'}, y^2_{k'})]$ in $YScanList$ in the similar way. Figure 4 (a) shows such grids k, k'. In procedure **FindNewCornerSize** p' and q' are set to $x^1_k - (XLoc_j + w'_j)$ and $y^1_{k'} - YLoc_j$ respectively. It is obvious that p' and q' cannot be greater than the above values from the figure. For both p' and q' to be correct, the shaded area in Figure 4 (a) should not be occupied by any grids which were already packed. Suppose that there is a grid l packed at $(XLoc_l, YLoc_l)$, and $YLoc_l < YLoc_{k'}$ and $XLoc_l < XLoc_{k'}$, that is, grid l is in the shaded area. Figure 4 (b) shows such a grid l. Since every grid including grid l was packed at a valid corner, there must be a grid n either to the left of or below grid l. Assume that grid n is located below grid l. Now, we apply the same argument to grid n, and so on. Eventually, there must be a grid t packed at $(XLoc_t, YLoc_t)$ such that $XLoc_t < XLoc_k$ and $(YLoc_t \leq YLoc_j \leq YLoc_t + h'_t)$. Figure 4 (c) shows such a grid t. This contradicts the fact that x^1_k is the minimum among those grids. Therefore, there cannot exist such a grid l and procedure **FindNewCornerSize** correctly determines p' and q'. Similarly, we can prove that p'' and q'' are correctly determined.

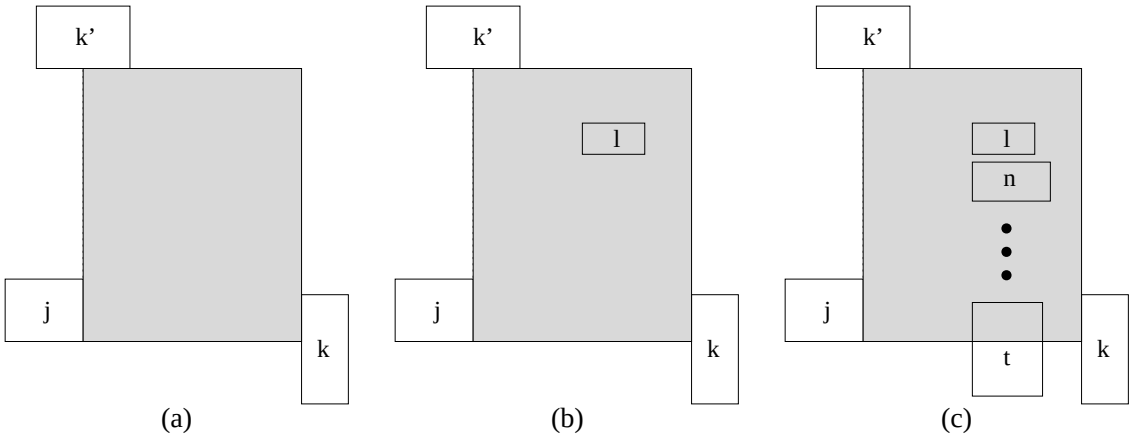

Figure 4: Figures for The Proof of Theorem 5.3.2

Proving the second part of the theorem is straightforward. We showed that sizes of corners are correctly computed. Since grids do not overlap in our packing, any point (x, y) is occupied at most one grid. We can use an induction similar to the one used for the first part of the theorem to prove that (x, y) belongs to the free space of at least one corner if it is not occupied. Base case (i.e. that is, when no grids have been packed) is trivial. Let's

assume that the claim is true after grid $j-1$ had been packed. Now, we pack grid j at corner $[(XLoc_j, YLoc_j), (p, q)]$. This corner is removed from $CornerList$ and two new corners, $[(XLoc_j + w'_j, YLoc_j), (p', q')]$ and $[(XLoc_j, YLoc_j + h'_j), (p'', q'')]$ are added, where w'_j, h'_j are dimensions of grid j along X and Y directions respectively. If p' and q' are correctly computed, p' and q' must be at least $p - w'_j$ and q respectively. From the same reasoning, p'' and q'' must be at least p and $q - h'_j$ respectively. Therefore, for any point (x, y) which belonged to the free space of the corner $[(XLoc_j, YLoc_j), (p, q)]$ before, either it is occupied by grid j or it belongs to one of the two new corners. ■

To show a bound for the accuracy of the solutions provided by our packing algorithm, we impose the following restrictions on packing the grids. When $w_i \times h_i$ $(w_i \geq h_i)$ grid is packed at a corner (x, y), it should be placed so that the side with dimension w_i (long side) is parallel to the X-axis if $x < Ry$ and it should be placed with long side parallel to the Y-axis if $x > Ry$. Suppose there is a free corner that cannot accommodate the item in the allowed orientation but can accommodate the item in the other orientation. Then we disregard this corner though it is possible to get a better packing by placing the item in that corner. If $x = Ry$, then both orientations of the grid are allowed for that corner. We will call this packing algorithm *modified TP-heuristic*. Now we state a result on the solution accuracy bound when $R = 1$ (i.e. for square meshes)

Theorem 3 : Consider the two-dimensional packing problem with $R = 1$ and let $w^* = \max_i w_i$. Let W_{opt}, H_{opt} be the width and height of the optimal packing and let W_{TP}, H_{TP} be the corresponding values for the packing given by the modified TP-heuristic when items are packed in decreasing order of their maximum side lengths. Assume without loss of generality that $W_{opt} \geq H_{opt}$ and $W_{TP} \geq H_{TP}$. Then $W_{TP} \leq \sqrt{2} W_{opt} + 3w^*$.

Proof : Let us denote the regions below and above the line OP (that has unit slope) by R_1 and R_2. First we observe that there is always a corner in R_1 as well as in R_2 that can accommodate a subsequent item in the allowed orientation (i.e. long side parallel to Y-axis in R_1 and parallel to X-axis in R_2). This follows from the fact that we can always pack a subsequent item q abutting to Y-axis (or X-axis). Since the item q' below (or to the left of) q was packed prior to q, the maximum side of q' is longer than the maximum side of q. Hence, there is always enough space to pack item q above (or to the right of) item q'.

Now, we show that $W_{TP} - H_{TP} \leq w^*$ as follows. Let W^i_{TP}, H^i_{TP} be the width and height of the packing after the i-th item is packed. Suppose that $W_{TP} - H_{TP} > w^*$. Then there must be an item i of dimensions (w_i, h_i) such that that $W^{i-1}_{TP} - H^{i-1}_{TP} \leq w^*$ and $W^i_{TP} - H^i_{TP} > w^*$. It also must be true that the i-th item was packed at a corner in R_1, hence $W^{i-1}_{TP} < W^i_{TP}$ and $H^{i-1}_{TP} \leq H^i_{TP}$. Let (x, y) be the location of a corner in R_2 which can accommodate the i-th

item. Then $x < y \leq H_{TP}^{i-1}$. Since $x + w_i < H_{TP}^{i-1} + w^* < W_{TP}^i$ and $y + h_i \leq H_{TP}^{i-1} + w^* < W_{TP}^i$, the maximum of the width and height of the packing would be smaller if the item i were packed at the corner at (x, y). Hence the item i should not have been packed at a corner in R_1. Since we always pack items so that the maximum of the width and height of the packing is minimized, we can conclude that $W_{TP} - H_{TP} \leq w^*$.

We only have to prove our result when $W_{TP} > 3w^*$ in which case $H_{TP} > 2w^*$. Consider the two isosceles right triangles A_1 and A_2 (see Figure 5) in regions R_1 and R_2 with areas $\frac{(W_{TP}-2w^*)^2}{2}$, $\frac{(H_{TP}-2w^*)^2}{2}$ respectively. Note that all items in the region A_1 (A_2) have been packed with their long sides parallel to Y (X) axis. Thus the items are packed in these regions as in bottom-up left-justified (BL for short) strategy of [5]. We can make the similar argument on the occupancy of A_1 and A_2 as in [5]. Note that any vertical (or horizontal) cut through A_1 (or A_2) can be partitioned into alternating segments corresponding to cuts through unoccupied and occupied areas. Using the fact that there are items to the right of A_1 and above A_2 and considering the order in which the items are packed, we can show that the sum of the occupied segments is at least the sum of the unoccupied segments. By integrating the lines over $W_{TP} - 2w^*$ (or $H_{TP} - 2w^*$), we can verify that A_1 (or A_2) is at least half full. This means that $W_{opt}^2 \geq 1/4\,((H_{TP} - 2w^*)^2 + (W_{TP} - 2w^*)^2) \geq \frac{(H_{TP}-2w^*)^2}{2}$ and the result follows from the fact that $W_{TP} - H_{TP} \leq w^*$. ■

The bound can be improved when the items to be packed are square shaped.

Theorem 4 : Consider the same 2D packing problem as in Theorem 5.3.3 except that the items are square shaped. If the items are packed in decreasing order of their sizes in the modified TP-heuristic, then $W_{TP} \leq \sqrt{2}W_{opt} + 2w^*$.

Proof : The proof of this theorem is similar to the proof of the previous theorem. It can be easily shown that $W_{TP} - H_{TP} \leq w^*$. Since all the items are square shaped, they are packed as in BL strategy [5] in the two isosceles right triangles S_1 and S_2 (see Figure 6) with areas $\frac{(W_{TP}-w^*)^2}{2}$, $\frac{(H_{TP}-w^*)^2}{2}$ respectively. Using the fact that there are items to the right of S_1 and above S_2 and considering the order in which the items are packed, we can show that S_1 and S_2 are at least half-occupied. This means that $W_{opt}^2 \geq 1/4\,((H_{TP} - w^*)^2 + (W_{TP} - w^*)^2)$ $\geq \frac{(H_{TP}-w^*)^2}{2}$ and the result follows from the fact that $W_{TP} - H_{TP} \leq w^*$. ■

The order in which we consider the grids for packing will affect the performance of our packing algorithm. Since small grids have a better chance to fit into holes without increasing the width or height of the packing, it is intuitively advantageous to pack them later. Assuming that $w_i \geq h_i$ $(1 \leq i \leq m)$, we consider the following four orderings of grids to be reasonable for good performance.

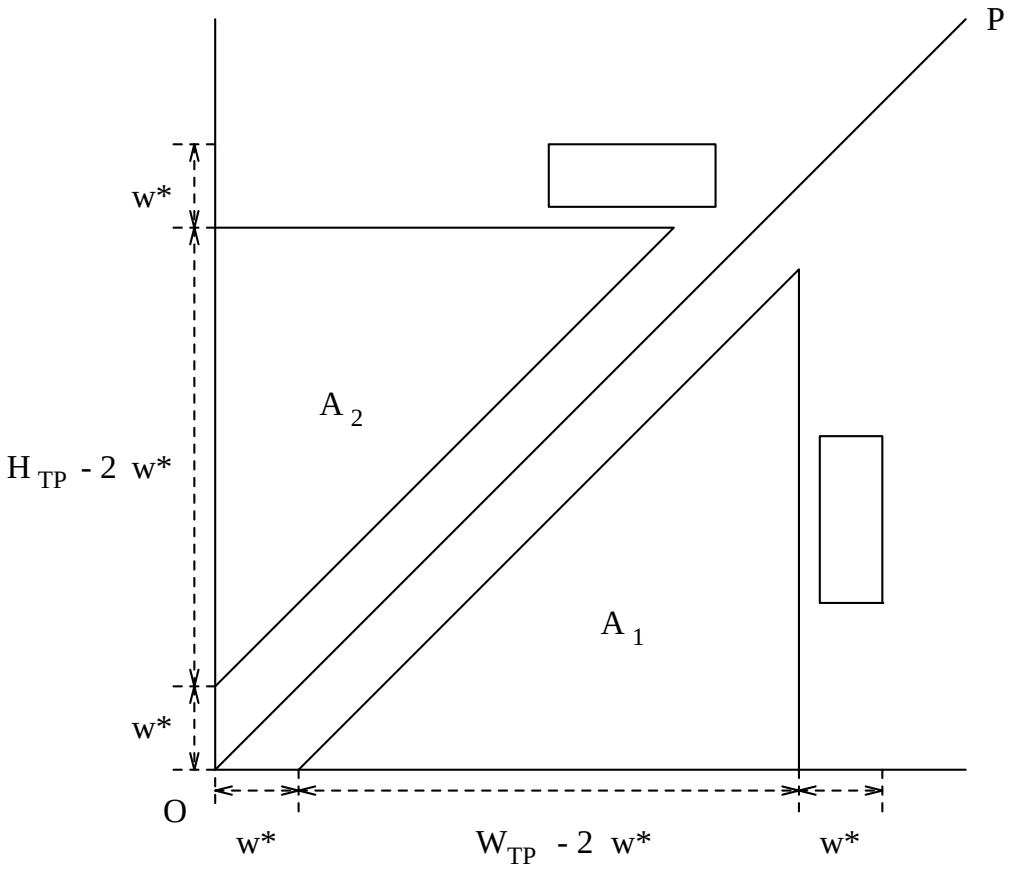

Figure 5: Regions A_1 and A_2 in the Packing (Theorem 5.3.3)

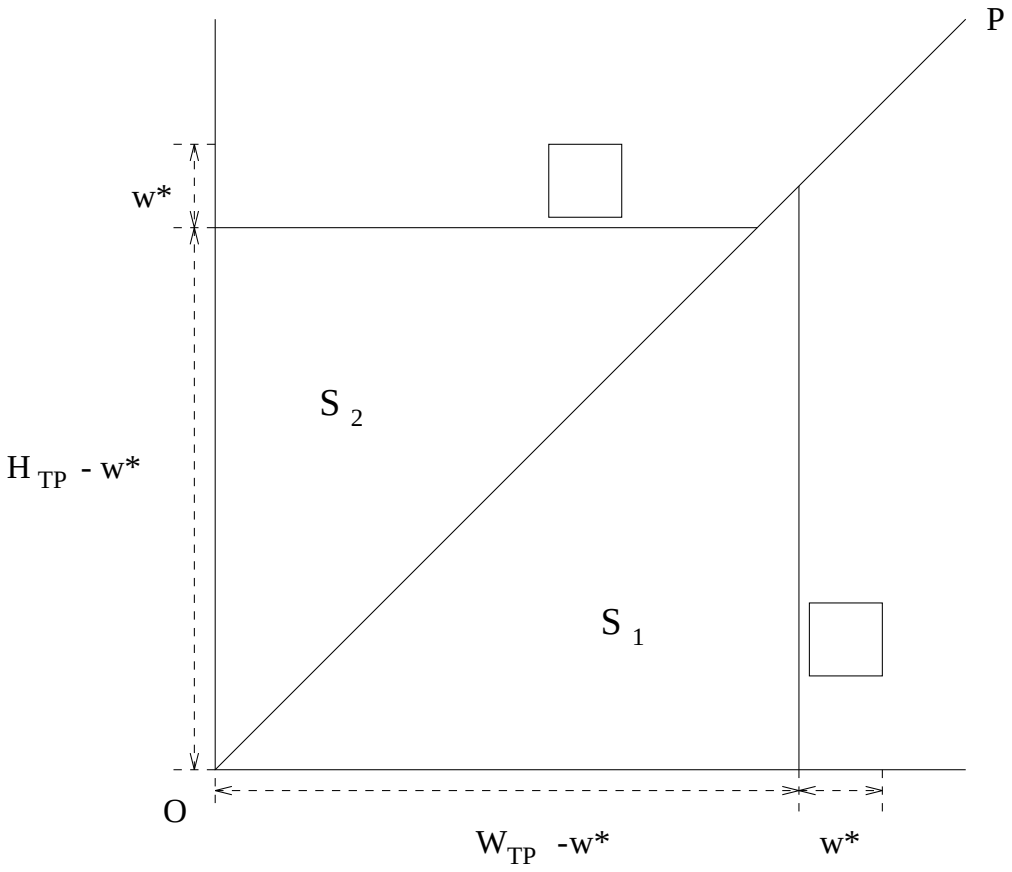

Figure 6: Regions S_1 and S_2 in the Packing (Theorem 5.3.4)

1. Decreasing order of w_i
2. Decreasing order of h_i
3. Decreasing order of $w_i * h_i$
4. Decreasing order of $\frac{w_i}{h_i}$

Since sorting the grids can be done in $O(m \log m)$ time, the time complexity of our packing algorithm will be same regardless of which grid ordering is used.

4.2 A Parallel Packing Algorithm

The packing algorithm described in the previous section is sequential. One processor has to collect all the information about the grids from the other processors and execute the packing algorithm in order to find the processor allocation. The result of allocation should be communicated to all the processors which in turn communicate the boundary values for the fine grids to the corresponding processors. In this section, we present a parallel algorithm in which m processors cooperatively execute the packing algorithm for m grids in order to speed up the algorithm. The same packing method that was used in the sequential algorithm will be used in the parallel algorithm described here.

In the adaptive mesh algorithm, each rectangular grid at level i may be generated across more than one processor. One of those processors (with lowest index) produces a packet $< (w_j, h_j), S_j >$ for grid j, where (w_j, h_j) is the dimension of grid j and S_j is its own processor index. These packets contain the necessary information that forms the input data to our packing and processor allocation algorithms. processors for them. The detailed description of the algorithm is given below. The topology of multiprocessors on which the algorithm runs, is assumed to be either mesh or hypercube.

Algorithm Parallel Packing

(a) Let PS be a set of m processors forming a submesh or a subcube, that is $PS = \{p_{b_i} | 0 \leq i \leq m-1\}$. The processors which produced packets send them to the processors in PS (one packet per processor) using **procedure TokenPacking**. The processors in PS will do the remaining steps of our parallel packing algorithm.

(b) Sort packets according to the packing order using a parallel sorting algorithm. After sorting, assume that processor P_{b_i} is holding packet $< (w_i, h_i), S_i >$.

(c) Store the initial free corner $[(0,0),(\infty,\infty)]$ at processor P_{b_0}. Each processor P_{b_i} set both $width$ and $height$ to 0. Now, pack the grids one by one by performing m iterations where in the i-th iteration ($0 \leq i \leq m-1$) call **procedure GridPacking**(i).

(d) After step (c), each processor P_{b_i} has $(XLoc_i, YLoc_i)$, that is, the free corner where grid (w_i, h_i) was packed and the width and height of the packing. Each processor P_{b_i} include the above information in its packet $< (w_i, h_i), S_i >$ and send it to S_i using **procedure SendPacketToSource**.

end Parallel Packing;

Procedure TokenPacking;
// Here a subset of processors $P_{j_0}, P_{j_1}, \ldots, P_{j_l}$ with $j_0 < j_1 < \cdots < j_l$ has one packet each and it is desired to store the packet of P_{j_k} in P_k for $t \leq k \leq (t+l) \bmod m$ for some $0 \leq t \leq N-1$. Using greedy routing, this operation can be done in $O(\log N)$ time on a hypercube and $O(\sqrt{N})$ time on a mesh [26]. //

Procedure GridPacking(i);

1. Call **procedure Broadcast**($b_i, < (w_i, h_i), S_i >$).

2. Call parallel **procedure FindBestCorner**($(w_i, h_i), k$) to find the corner (x_k, y_k) where the grid (w_i, h_i) is to be packed, and its orientation.

3. Processor P_{b_i} set $XLoc_i$ and $YLoc_i$ to x_k and y_k respectively, and set $Orient_i$ to $Orient_k$. Also P_{b_i} determines the width and height of the packing after (w_i, h_i) is packed.
   Call **procedure Broadcast**($b_i, ((x_k, y_k), (width, height), Orient_i)$).

4. Call parallel **procedure UpdateCorners**($(w_i, h_i), k$) to update the sizes of corners after the grid (w_i, h_i) is packed at (x_k, y_k).

5. Call Parallel **procedure FindNewCornerSize**($(w_i, h_i), k$) to determine the sizes of two new corners.

Procedure FindBestCorner($(w_i, h_i), k$);
1. Each Processor P_{b_j} does the following
   begin
   Let $[(x_j, y_j), (p_j, q_j)]$ the corner P_{b_j} is holding
   if $(p_j - w_i) \geq 0$ and $(q_j - h_i) \geq 0$ then
   $mx1 = \max(width, x_j + w_i), \quad my1 = \max(height, y_j + h_i)$

else if $(p_j - h_i) \geq 0$ and $(q_j - w_i) \geq 0$ then

$mx2 = \max(width, x_j + h_i), \quad my2 = \max(height, y_j + w_i)$

else set m to ∞ and goto the next step

$m1 = \max(mx1, R * my1)$

$m2 = \max(mx2, R * my2)$

$m_j = \min(m1, m2)$

if $(m1 \leq m2)$ then $\quad Orient_j = 0$

else $\quad Orient_j = 1$

end

If P_{b_j} has two corners then choose the one which gives smaller m_j.

If P_{b_j} does not have any corners then set m_j to ∞.

2. Call parallel **procedure FindMin**$(\{m_l\}_{l=0}^{m-1}, m_k, b_i)$

end FindBestCorner;

Procedure UpdateCorners$((w_i, h_i), k)$;

1. Processor P_{b_k} remove $[(x_k, y_k), (p_k, q_k)]$.
2. Each Processor P_{b_j} does the following

begin

Let $[(x_j, y_j), (p_j, q_j)]$ the corner P_{b_j} is holding

if $Orient_i = 1$ then

$w_i' = h_i, \quad h_i' = w_i$

else

$w_i' = w_i, \quad h_i' = h_i$

if $(y_k \leq y_j < y_k + h_i')$ and $(x_j \leq x_k < x_j + p_j)$ then

$p_j = x_k - x_j$

if $(x_k \leq x_j < x_k + w_i')$ and $(y_j \leq y_k < y_j + q_j)$ then

$q_j = y_k - y_j$

if $p_j = 0$ or $q_j = 0$ then

remove $[(x_j, y_j), (p_j, q_j)]$

end

If P_{b_j} has two corners then update the second one in the same way.

end UpdateCorners;

Procedure FindNewCornerSize$((w_i, h_i), k)$;

1. Each Processor $P_{b_j} (j < i)$ does the following

begin

if $Orient_i = 1$ then

$w'_i = h_i, \quad h'_i = w_i$

else

$w'_i = w_i, \quad h'_i = h_i$

if $(YLoc_j \leq y_k \leq YLoc_j + h_j)$ and $(x_k + w'_i \leq XLoc_j)$

then $p_j = XLoc_j$

else $p_j = \infty$

if $(XLoc_j \leq x_k + w'_i \leq XLoc_j + w_j)$ and $(y_k \leq YLoc_j)$

then $q_j = YLoc_j$

else $q_j = \infty$

2. Call parallel **procedure FindMin**$(\{p_l\}_{l=0}^{i-1}, p, b_i)$
3. Call parallel **procedure FindMin**$(\{q_l\}_{l=0}^{i-1}, q, b_i)$
4. Find p' and q' in the same way for the other corner.
5. Store the two new corners, $[(XLoc_i + w'_i, YLoc_i), (p, q)]$ and $[(XLoc_i, YLoc_i + h'_i), (p', q')]$, at Processor P_{b_i}.

end FindNewCornerSize;

Procedure Broadcast(j, V);

// Processor P_j broadcasts value V to all the processors in PS; takes $O(\log N)$ time in a hypercube and $O(\sqrt{N})$ time in a mesh [26]. //

Procedure FindMin$(\{a_l\}_{l=0}^{n-1}, b, j)$;

// Given the a_l values with one value per processor, find the minimum (b) of these values and store it in the processor P_j. This operation can be done in $O(\log N)$ time on a hypercube and $O(\sqrt{N})$ time on a mesh. //

Procedure SendPacketToSource

1. Each processor P_{b_i} include $XLoc_i$, $YLoc_i$, $Orient_i$, $width$ and $height$ in the packet $< (w_i, h_i), S_i >$.

2. For a hypercube, sort the packets according to S_i using a parallel sorting algorithm.

3. Send each packet $< (w_i, h_i), S_i, (XLoc_i, YLoc_i), Orient_i, (width, height) >$ to processor S_i. For this, use the monotone routing (see chapter 3) for a hypercube and one-to-one routing [26] for a mesh.

end SendPacketToSource

Finding a corner to pack the next grid, updating the sizes of corners and determining the sizes of two new corners are done in the same way as in the sequential packing algorithm. After executing the above algorithm, processor S_i can find a submesh for grid (w_i, h_i) using

the information included in the packet.

Time complexity analysis for a hypercube :

Step (a) takes $O(\log N)$ time and step (b) takes $O(\log^2 m)$ time. Both procedure **FindBestCorner** and **FindNewCornerSize** take $O(\log m)$ time. procedure **UpdateCorners** takes a constant time. Hence step (c) takes $O(m \log m)$ time. Step (d) takes $O(\log N + \log^2 m)$ time. The total time complexity for a hypercube is $O(\log N + m \log m)$.

Time complexity analysis for a hypercube :

Step (a) takes $O(\sqrt{N})$ time and step (b) takes $O(\sqrt{m} \log m)$ time. Both procedure **FindBestCorner** and **FindNewCornerSize** take $O(\sqrt{m})$ time. procedure **UpdateCorners** takes a constant time. Hence step (c) takes $O(m\sqrt{m})$ time. Step (d) takes $O(\sqrt{N})$ time. The total time complexity for a mesh is $O(\sqrt{N} + m\sqrt{m})$.

4.3 Allocation of Processors

In the previous sections, we described algorithms for packing grids in such a way that the ratio of width to height is as close to the mesh ratio as possible and the packing area is minimized at the same time. Now, we are ready to allocate a set of processors (submesh) for each grid using the results of packing. Let W and H be the width and height of the packing, and let (x_i, y_i) be the location of the south-west corner of grid i with dimensions $w_i \times h_i$. Assume that we are given a $P \times Q (P \geq Q)$ mesh of multiprocessors and let $W \geq H$. Let $((i,j),(x \times y))$ be a $x \times y$ submesh with its south-west corner processor indexed (i,j). We propose the following two ways to allocate submeshes to grids.

1. Assign submesh
   $\Big((\lfloor x_i \frac{P}{W} \rfloor, \lfloor y_i \frac{Q}{H} \rfloor),\ (\lfloor (x_i + w_i)\frac{P}{W} \rfloor - \lfloor x_i \frac{P}{W} \rfloor) \times (\lfloor (y_i + h_i)\frac{Q}{H} \rfloor - \lfloor y_i \frac{Q}{H} \rfloor)\Big)$ to grid i.
2. If $(\frac{Q}{P} \leq \frac{H}{W})$, then assign submesh
   $\Big((\lfloor x_i \frac{Q}{H} \rfloor, \lfloor y_i \frac{Q}{H} \rfloor),\ (\lfloor (x_i + w_i)\frac{Q}{H} \rfloor - \lfloor x_i \frac{Q}{H} \rfloor) \times (\lfloor (y_i + h_i)\frac{Q}{H} \rfloor - \lfloor y_i \frac{Q}{H} \rfloor)\Big)$ to grid i.
   else assign submesh
   $\Big((\lfloor x_i \frac{P}{W} \rfloor, \lfloor y_i \frac{P}{W} \rfloor),\ (\lfloor (x_i + w_i)\frac{P}{W} \rfloor - \lfloor x_i \frac{P}{W} \rfloor) \times (\lfloor (y_i + h_i)\frac{P}{W} \rfloor - \lfloor y_i \frac{P}{W} \rfloor)\Big)$ to grid i.

The first method tries to use as many processors as possible to reduce the computational workload, but may violate the property that $\frac{w_i}{X_i} = \frac{h_i}{Y_i}$ for all the grids. In the second method, we allocate processors in such a way that $\frac{w_i}{X_i} \approx \frac{h_i}{Y_i}$. In other words, the second method uses uniform scaling in X and Y directions while the first method uses different scaling factors in X and Y directions in order to reduce the packing to the given mesh size. In the second method, we may not utilize all the processors in the mesh. We will call the first and second method *non-uniform scaling* and *uniform scaling* respectively. Both of the methods are identical to each other when $\frac{Q}{P} = \frac{H}{W}$.

To make sure that both X_i and Y_i are not zero, that is, at least one processor is allocated to each grid, we impose the following lower bound on the size of each grid. Let $D = \sum_i \max(w_i, h_i)$. D is a very loose upper bound on $\max(W, H)$ where W and H are width and height of any packing. Then, $\min_i(\min(w_i, h_i)) \geq \frac{D}{\min(P,Q)}$ should be satisfied. To achieve better bound on D, we can quickly pack the grids in the following way. Each time a grid is packed, we examine only two corners, one on X-axis and the other on Y-axis. Then use $\max(width, height)$ as D. Since we always consider only two corners, this packing takes only $O(m)$ time. After finding the lower bound on the size of the grids, change w_i or h_i to this value if they are smaller than that.

4.4 Processor Allocation for Hypercubes

If we are given a hypercube, we can still use the above method to allocate processors for grids assuming that we are given a mesh. Then, we can easily embed the mesh in the given hypercube using reflected Gray code [10]. The reflected Gray code is generated in the following way. We start with the 1-bit Gray code sequence $\{0, 1\}$, and then insert a zero and a one in front of the two elements obtaining the two sequences $\{00, 01\}$ and $\{10, 11\}$. We then reverse the second sequence to obtain $\{11, 10\}$, and then concatenate the two sequences to obtain the 2-bit reflected Gray code.

$$\{00, 01, 11, 10\}$$

Generally, given a $(d-1)$-bit code

$$\{b_1, b_2, \ldots, b_p\},$$

where $p = 2^{d-1}$ and $b_1, b_2, \ldots, b_p$ are binary strings, the corresponding d-bit code is

$$\{0b_1, 0b_2, \ldots, 0b_p, 1b_p, 1b_{p-1}, \ldots, 1b_1\}.$$

The preceding recursive construction of the reflected Gray code sequence can be generalized. Let d_a and d_b be positive integers, and let $d = d_a + d_b$. Suppose that $\{a_1, a_2, \ldots, a_{p_a}\}$ and $\{b_1, b_2, \ldots, b_{p_b}\}$ are the d_a-bit and d_b-bit reflected Gray code sequences, where $p_a = 2^{d_a}$ and $p_b = 2^{d_b}$. Consider the $p_a \times p_b$ matrix of d-bit strings $\{a_i b_j | i = 1, 2, \ldots, p_a, j = 1, 2, \ldots, p_b\}$.

$$\begin{pmatrix} a_1b_1 & a_1b_2 & \cdots & a_1b_{p_b} \\ a_2b_1 & a_2b_2 & \cdots & a_2b_{p_b} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p_a}b_1 & a_{p_a}b_2 & \cdots & a_{p_a}b_{p_b} \end{pmatrix}$$

The preceding construction also shows that the nodes of a d-cube can be arranged along a two-dimensional mesh with p_a and p_b nodes in the first and second dimensions, respectively. The (i,j)-th element of the mesh, where $i = 1, 2, \ldots, p_a$ and $j = 1, 2, \ldots, p_b$, is the d-cube node with identity number $a_i b_j$.

In N-node hypercube, we can embed $\log\sqrt{N}+1$ different meshes with power of two nodes in each dimension, such as $(2^{\log\sqrt{N}} \times 2^{\log\sqrt{N}})$, $(2^{\log\sqrt{N}+1} \times 2^{\log\sqrt{N}-1})$, $\cdots$, $(2^{\log N} \times 2^{0})$. We can pick one of the above meshes and pack the grids using its mesh ratio. Or, we can try for all the above meshes and pick the one which gives $\frac{height}{width}$ closest to the corresponding mesh ratio. Since we use only m processors in the parallel packing algorithm, we can pack the grids using $\lfloor \frac{N}{m} \rfloor$ different mesh ratios at the same time. If we try all $\log\sqrt{N}+1$ meshes using the parallel packing algorithm, we have to repeat the packing algorithm $\lceil \frac{m(\log\sqrt{N}+1)}{N} \rceil$ times. This takes $O(\frac{m^2 \log m \log\sqrt{N}}{N})$ time since one execution of the packing algorithm takes $O(m \log m)$ time.

4.5 Experimental Results

We performed experiments to measure the performance of our approximation algorithm based on the TP-heuristic. As in the experiments for the first approach, we start with some number of (coarse) grids and assume that at each level of adaptive mesh refinement, the number of fine grids (child grids) created within each coarse grid region is exponentially distributed with a mean value of 1. The number of mesh points in each fine grid is uniformly distributed in the range [MINSIZE,MAXSIZE]. The aspect ratios of fine grids, namely $\frac{w_i}{h_i}$ $(w_i \geq h_i)$, are also assumed to be uniformly distributed in the range [1,MAXRATIO]. In our experiments, we went through 200 levels of refinement. For assignment of fine grids at each level, we first solve the two-dimensional packing problem and then allocate a submesh for each grid using the methods described in the previous section.

We compared the performance of our algorithm with that of an algorithm based on the *modified first-fit level heuristic* (LP) discussed in [18] for packing rectangles into a bin of fixed width. In this heuristic, rectangles (grids) are considered in decreasing order of height and packed level by level as in the first-fit heuristic. But successive levels are packed left-right and right-left in the bin of fixed width so that tallest item in a level will be above the shortest in the previous level. This will allow the items to drop down after packing which may in turn reduce the total height of the packing. Since in our problem, the width is not fixed, we have to apply an iterative process where in each iteration, we use the LP-heuristic to determine the height of the packing for some fixed width. The initial width is set equal to $\sqrt{RS}$ where $S = \sum_i (w_i * h_i)$ and R is the mesh ratio. Then, the width is increased by a small amount (1 % of the width) each time until we get a ratio for $\frac{W}{H}$ that is *just* above the

processor mesh ratio $\frac{P}{Q}$.

The objective function values were computed using the following formula.

$$T_i = \max_{1 \le j \le m} \left(U_{comp} \frac{w_j h_j}{X_j Y_j} + 2 U_{comm} F \left(\frac{w_j}{X_j} + \frac{h_j}{Y_j} \right) \right)$$

where U_{comp}, U_{comm} and F are assumed to be 1 for simplicity. First, we compare the solution values obtained by the non-uniform and uniform scaling processor allocation methods in Figure 7. Here, we started with 40 grids. MINSIZE and MAXSIZE are set to $k * 0.7$ and $k * 1.3$ where the constant k is set based on the average number of mesh points per processor which is assumed to be 300. 32×32 processor mesh was used for assignment. The experiment was repeated for different aspect ratios of grids. Figure 7 shows that the non-uniform scaling method performs slightly better than the uniform scaling method. The reason for the narrow difference in the performances of the two methods is attributed to the fact that our algorithm gives a packing whose $\frac{W}{H}$ ratio is close to $\frac{P}{Q}$ ratio of the mesh. In our remaining experiments, we use only the non-uniform scaling method for allocation of submeshes. Figure 8 shows the solution values obtained after applying four different packing orders of grids in TP-heuristic. From the results, we see that packing grids in decreasing order of grid size ($w_i * h_i$) performs the best, while packing them in decreasing order of their aspect ratio $\left(\frac{w_i}{h_i}\right)$ performs the worst. We use only decreasing order of grid size as a packing order in our remaining experiments.

Figure 9 - Figure 24 show comparison of TP and LP heuristics with different values of various parameters. In the figures, we show the cumulative sum (across 200 levels) for (a) computational cost, (b) communication cost, (c) sum of (a) and (b) and (d) average processor utilization which indicates how tightly the grids are packed in the packing algorithm. The processor utilization is the ratio of the number of processors used in allocation to the total number of processors. Figure 9 and Figure 12 show how the performance of each heuristic varies with *variance in grid size* (i.e. number of grid points). In this experiment, MINSIZE and MAXSIZE are set to $k * (1-\text{VAR})$ and $k * (1+\text{VAR})$ respectively, and VAR varies from 0.1 to 0.9. We used the same values for other parameters as in the previous two experiments. Figure 13 and Figure 16 show how the performance of each heuristic varies with the aspect ratios of the grids. Figure 17 and Figure 20 show how the performance of each heuristic varies with processor mesh ratio. In this experiment, we use 32*MESHRATIO$\times$32 mesh for assignment, where MESHRATIO varies from 1 to 3. Figure 21 and Figure 24 show how the performance of each heuristic varies with initial number of grids. Below we give a summary of our observations.

1. As can be seen in Figure 9 and Figure 12, the TP-heuristic performs better than the LP-heuristic when the variation in grid sizes is reasonably large. The processor utilization

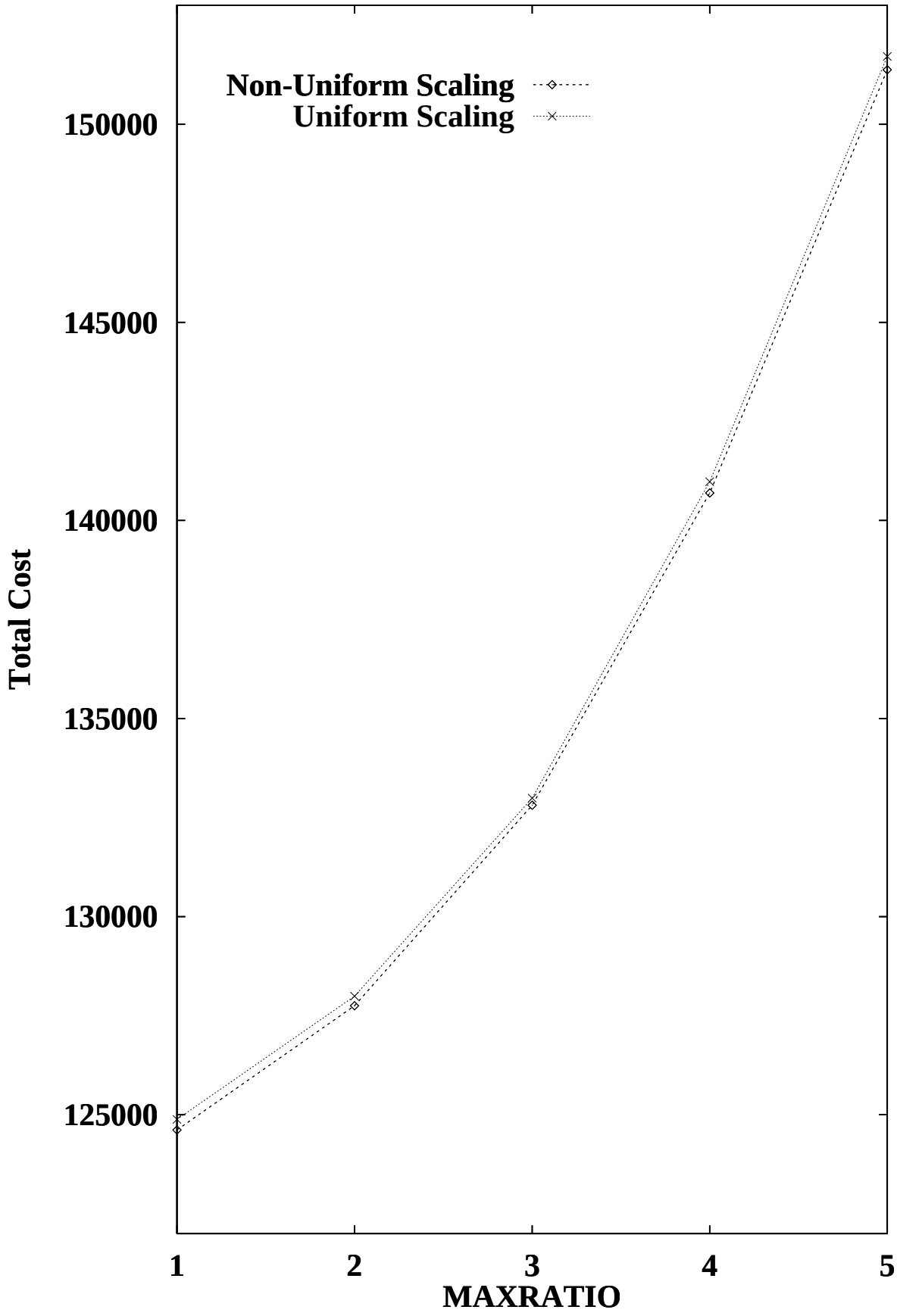

Figure 7: The Comparison of Two Different Allocation Methods

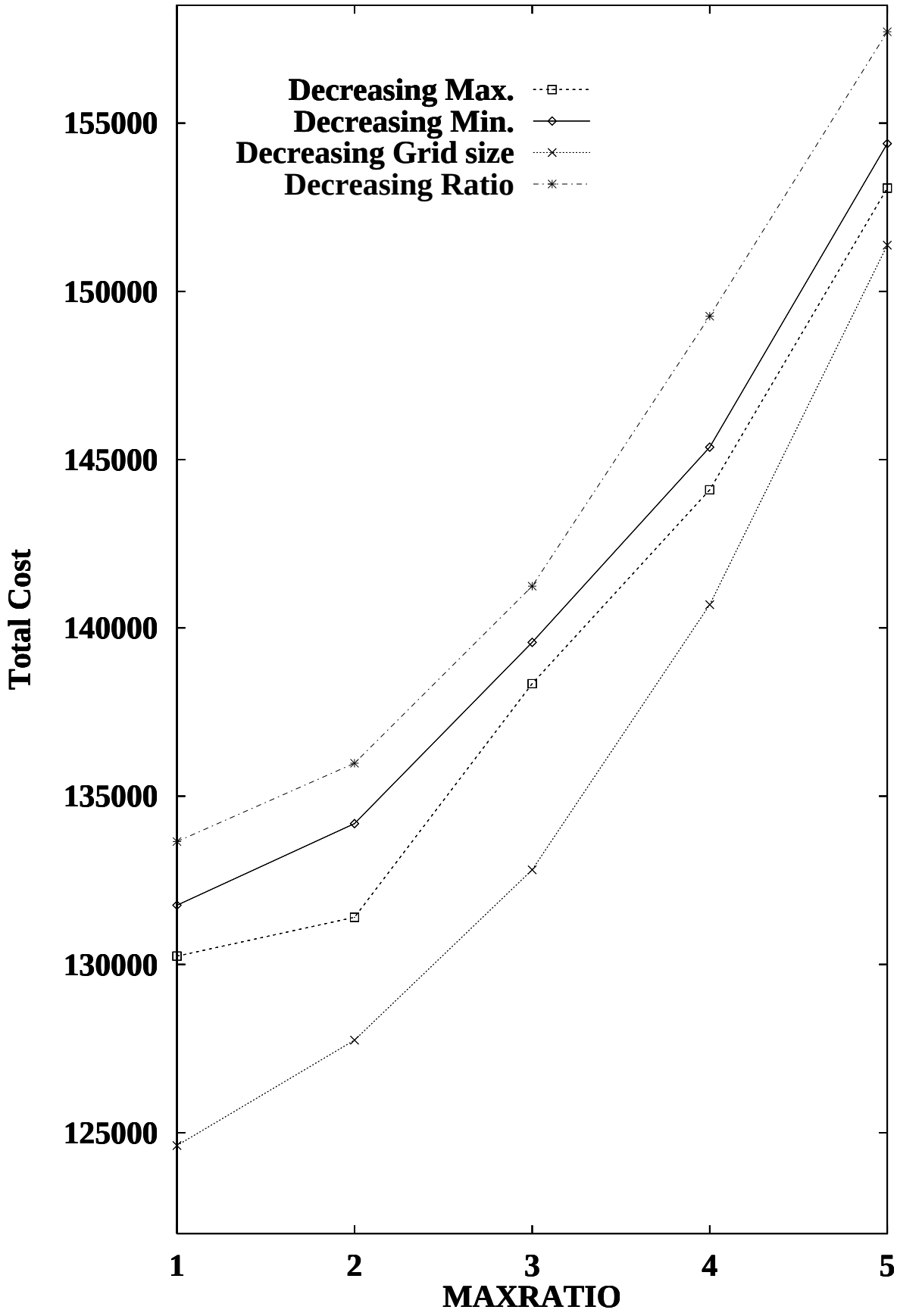

Figure 8: The Comparison of Four Different Packing Orders

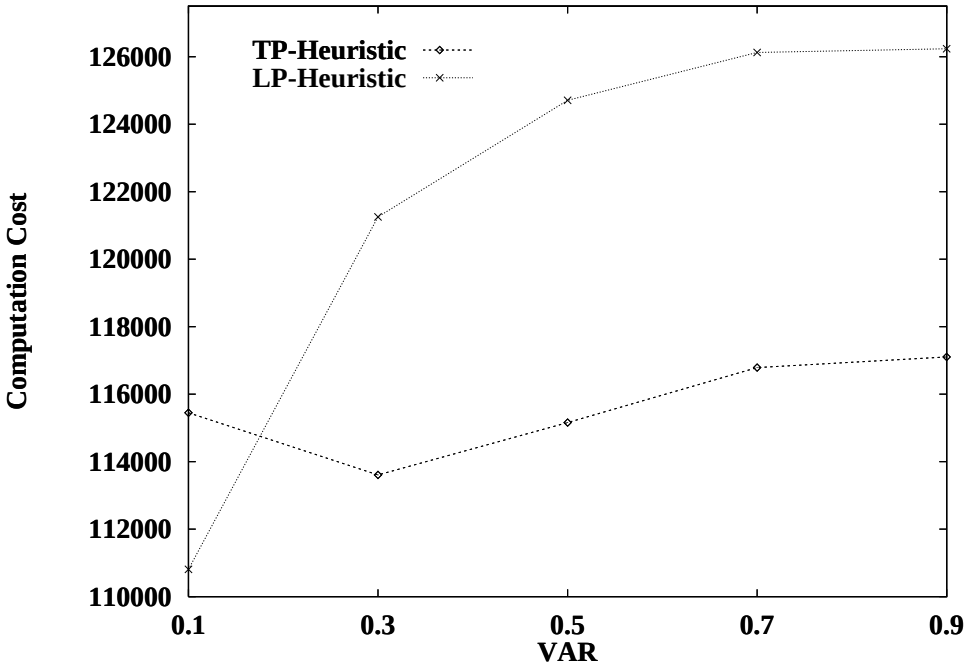

Figure 9: Computation Costs for Different Grid Size Variances

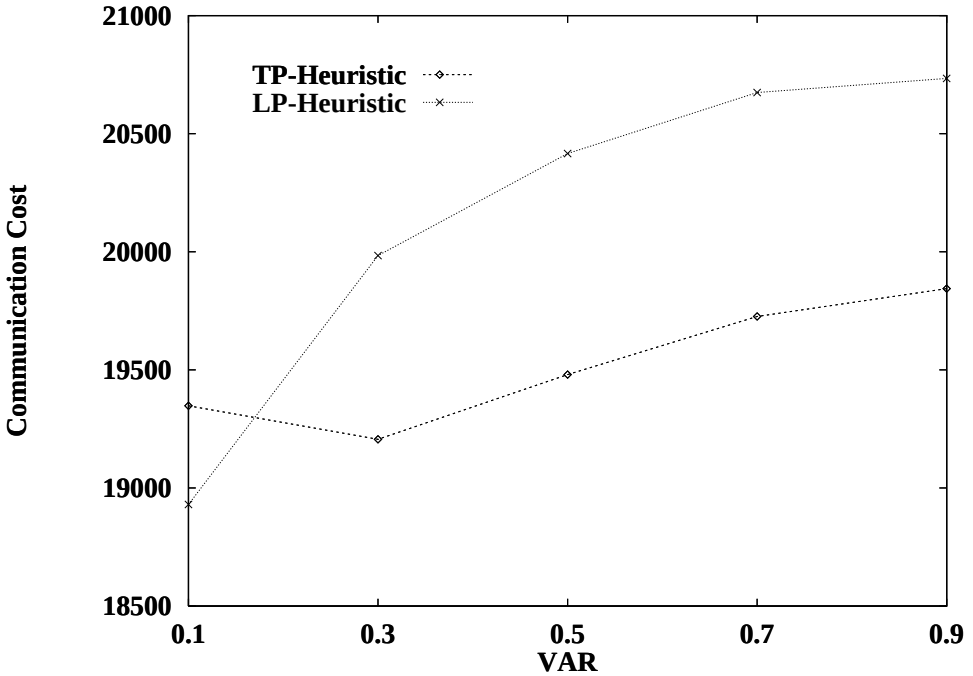

Figure 10: Communication Costs for Different Grid Size Variances

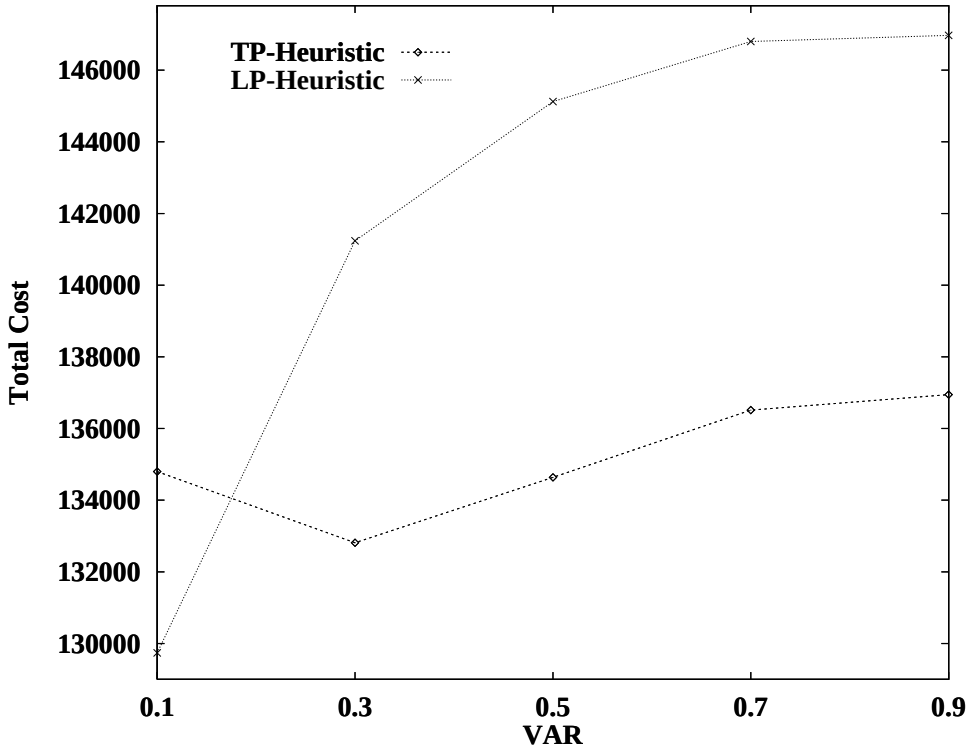

Figure 11: Total Costs for Different Grid Size Variances

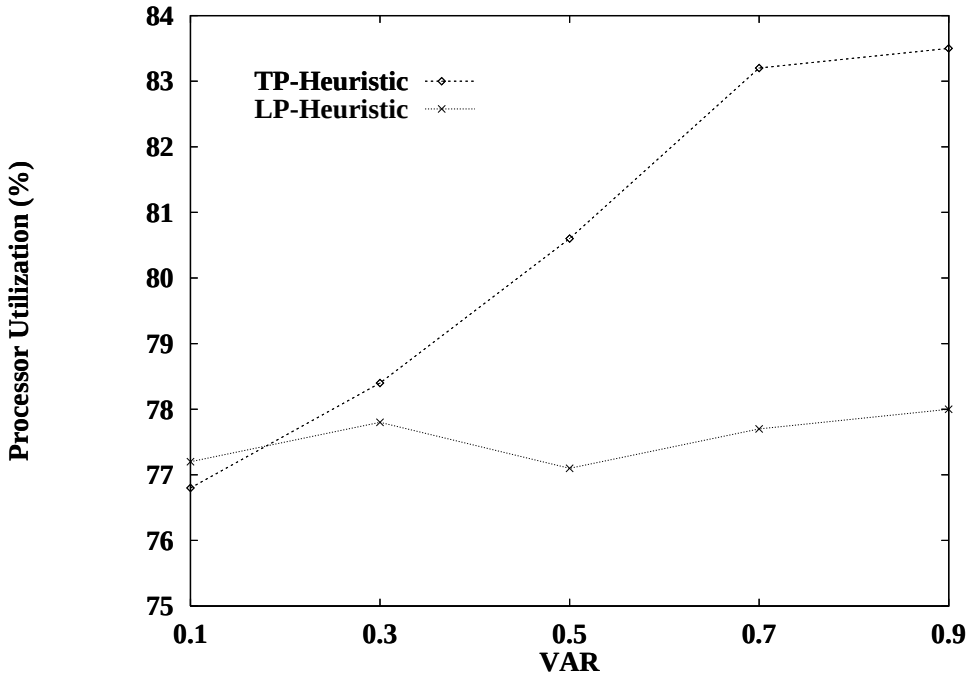

Figure 12: Processor Utilizations for Different Grid Size Variances

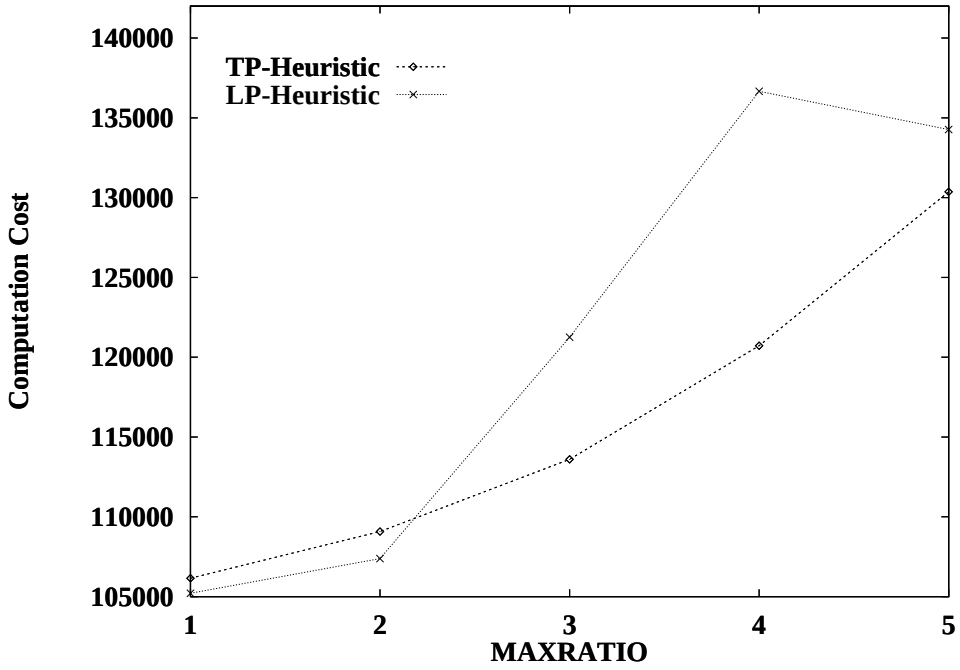

Figure 13: Computation Costs for Different Aspect Ratios of Grids

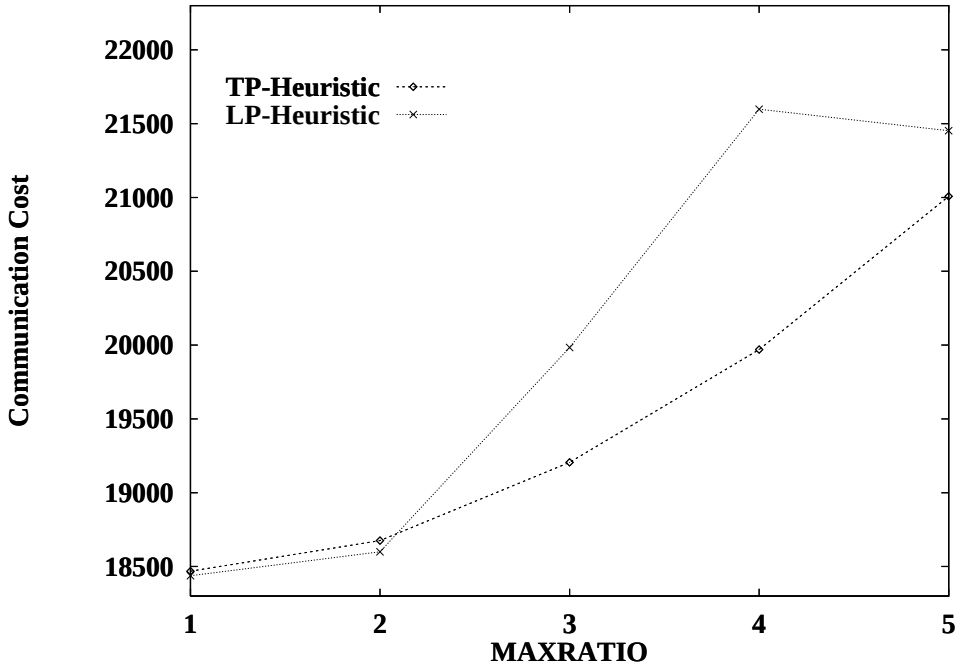

Figure 14: Communication Costs for Different Aspect Ratios of Grids

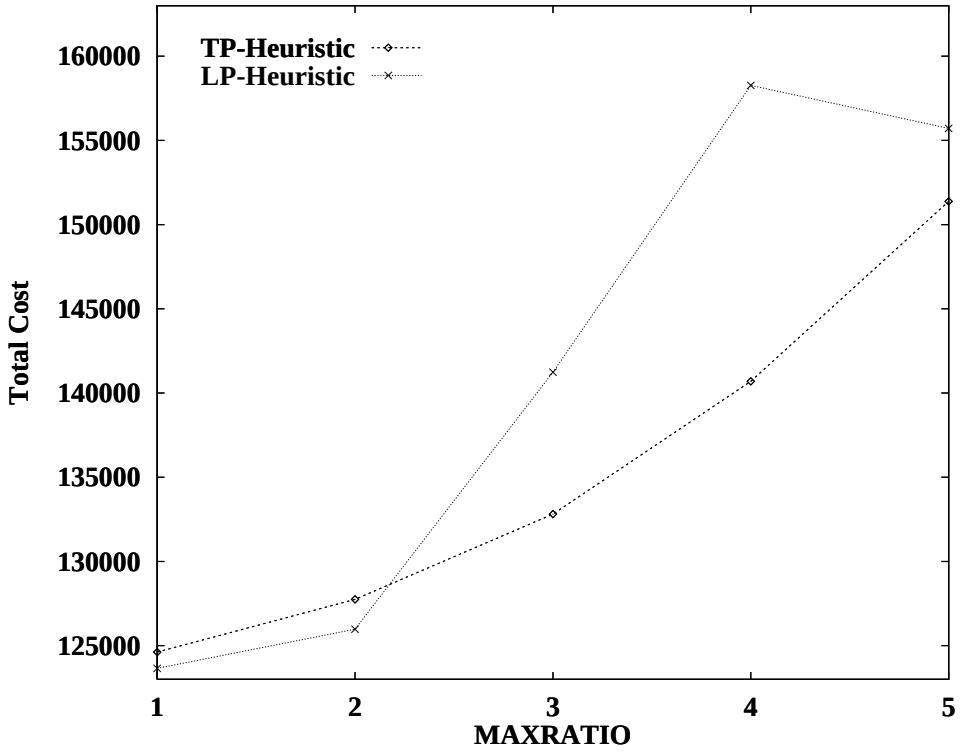

Figure 15: Total Costs for Different Aspect Ratios of Grids

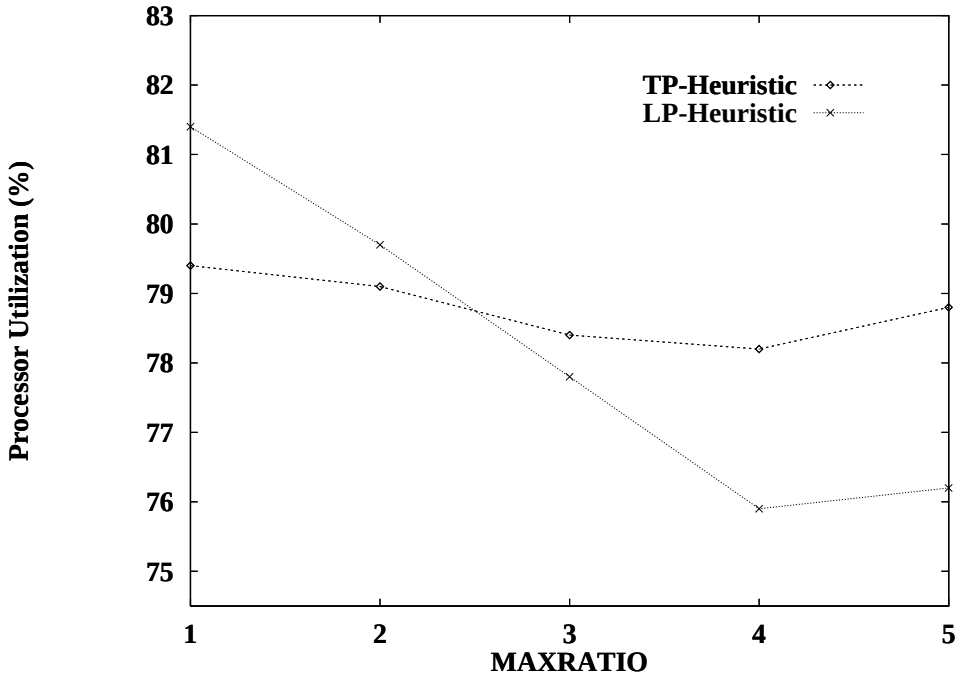

Figure 16: Processor Utilizations for Different Aspect Ratios of Grids

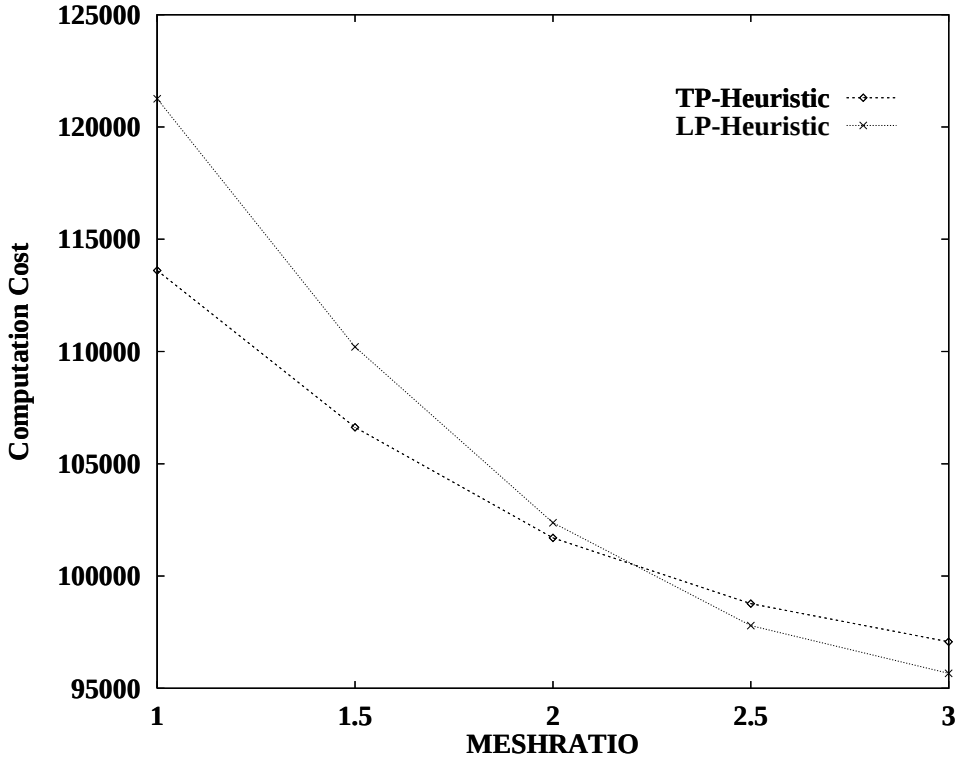

Figure 17: Computation Costs for Different Aspect Ratios of Processor Meshes

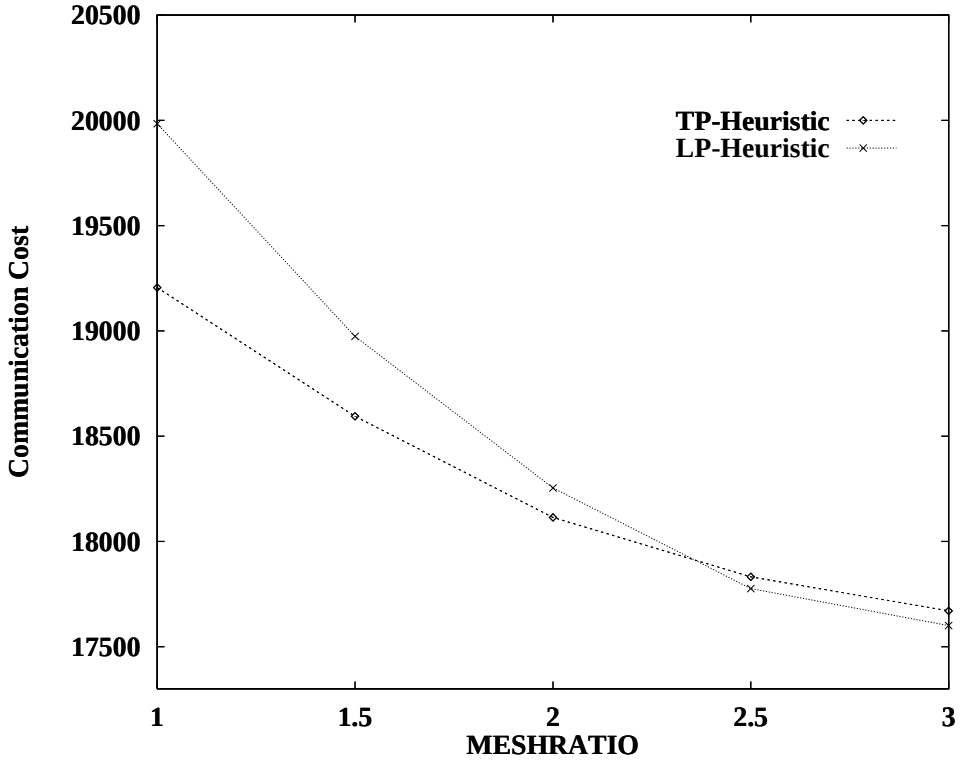

Figure 18: Communication Costs for Different Aspect Ratios of Processor Meshes

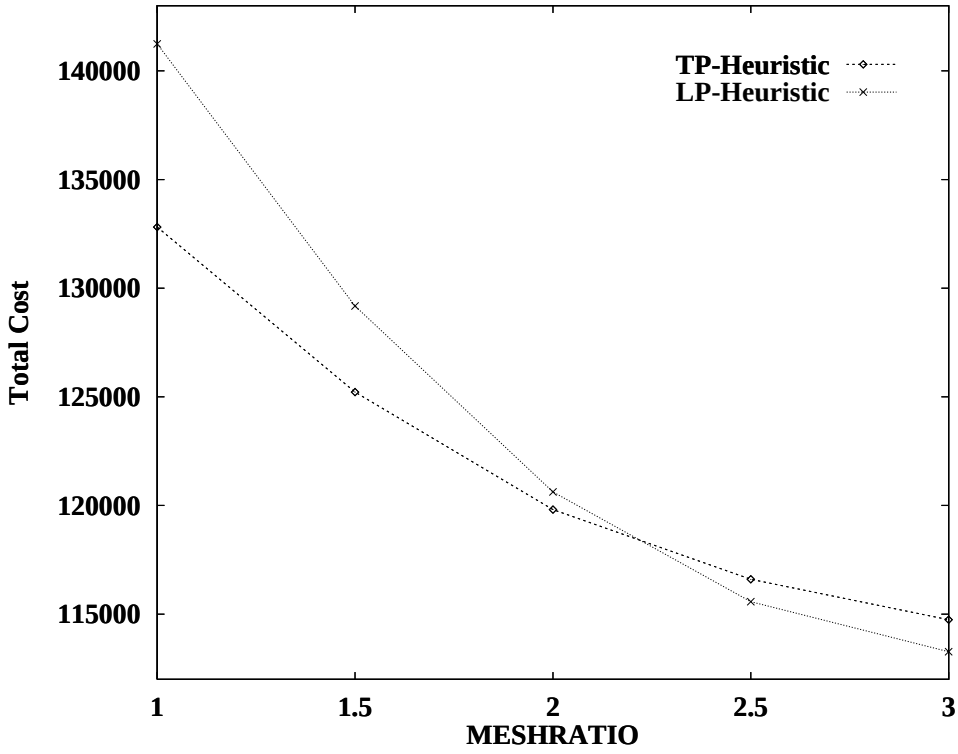

Figure 19: Total Costs for Different Aspect Ratios of Processor Meshes

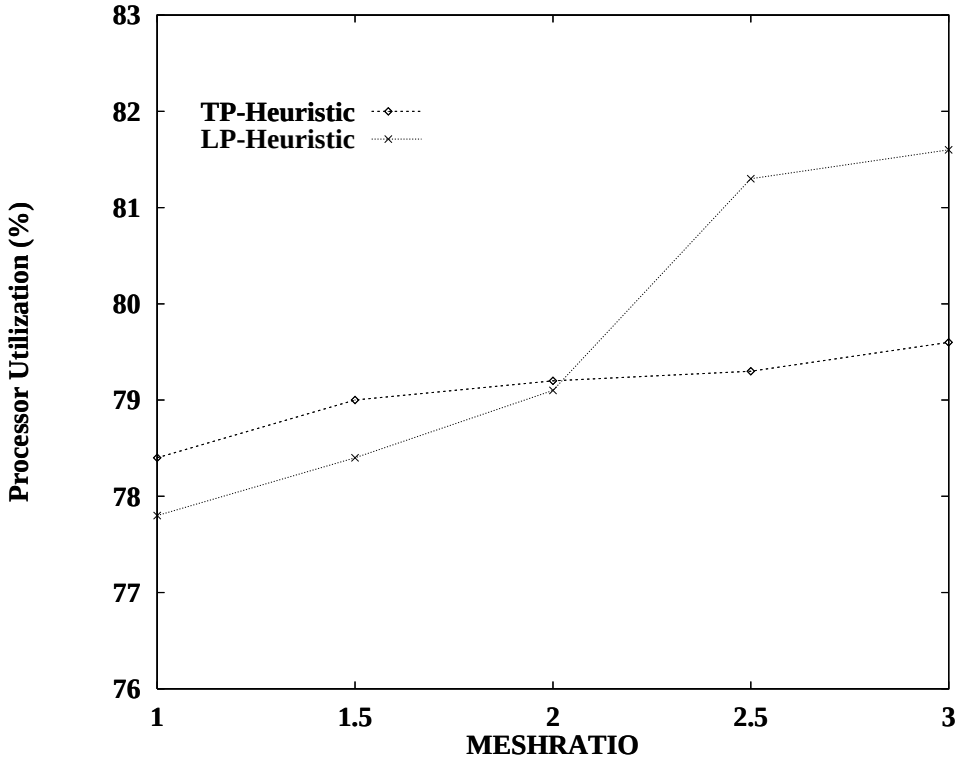

Figure 20: Processor Utilizations for Different Aspect Ratios of Processor Meshes

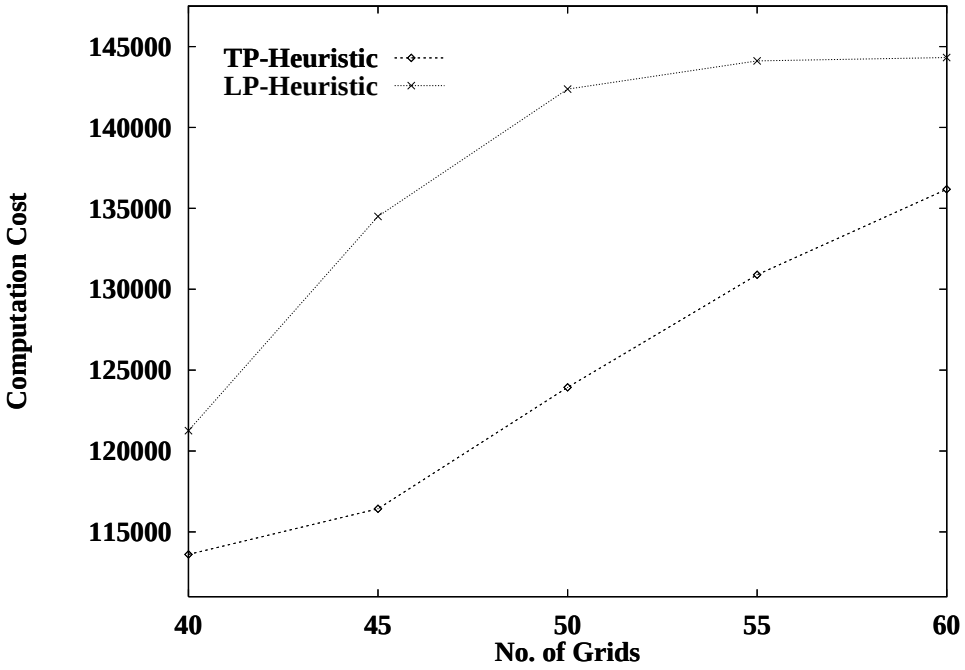

Figure 21: Computation Costs for Different Numbers of Grids

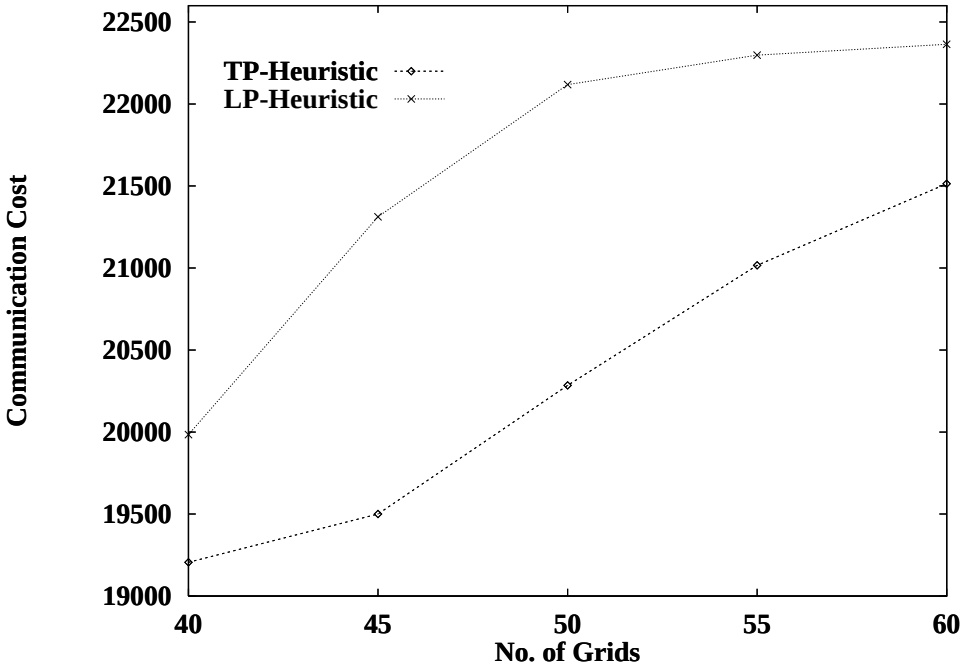

Figure 22: Communication Costs for Different Numbers of Grids

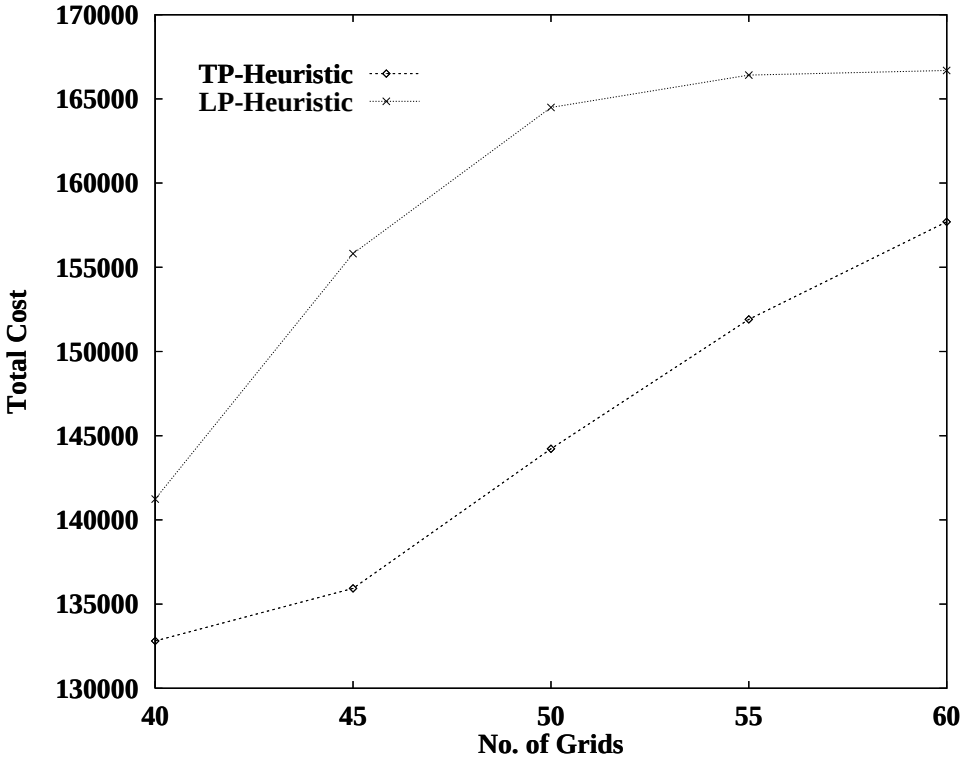

Figure 23: Total Costs for Different Numbers of Grids

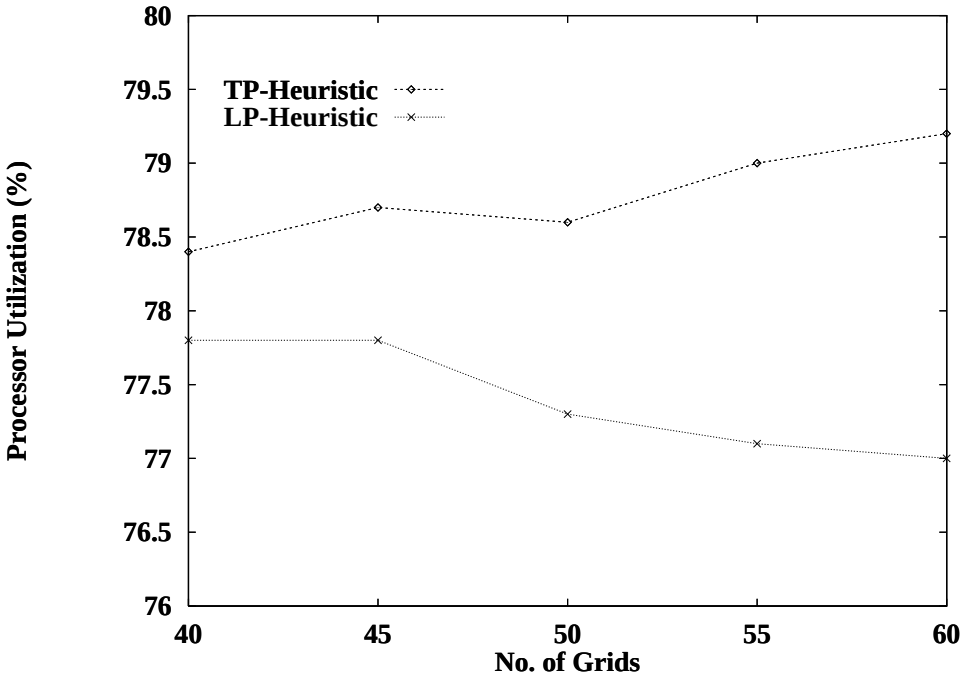

Figure 24: Processor Utilizations for Different Numbers of Grids

also increases with the variation in grid sizes for TP-heuristic. Since the processor utilization indicates how tightly the grids are packed in the packing algorithm, this result demonstrates that our proposed heuristic gives good packings in practice.

2. Figure 13 and Figure 16 show that LP outperforms TP only when the grids are square or close to being square shaped. But as the aspect ratio increases, TP produces better solutions than LP as indicated by smaller computational and communication costs as well as higher processor utilization. In both TP and LP heuristics, the total cost increases with the maximum aspect ratio of the grids. This implies that it is difficult to pack the grids tightly when the variance in aspect ratios is large.

3. The following reason can be attributed to the better performance of TP over LP for large variation in grid sizes or larger aspect ratios : TP allows smaller grids that are packed later on, to be packed into holes created during the packing of larger grids and this has the effect of keeping the width and height of packing as small as possible.

4. Figure 17 and Figure 20 show that LP outperforms TP when processor mesh ratio is larger than 2. Higher processor utilization for LP heuristic indicates that LP can pack rectangles tightly when the width of the bin is large. In LP-heuristic, the number of levels in packing is small when the width of the bin is large.

5. Figure 21 and Figure 24 indicate that TP performs always better than LP regardless of the number of grids to be packed. Processor utilization of TP seems to increase slightly as the number of grids increases.

6. In all the figures, both computation and communication costs show the same behavior. The reason for this is that both computation and communication costs are inversely proportional to the number of processors allocated and processors are uniformly allocated to the grids according to the number of grid points.

7. In most of the cases, TP outperforms LP in both total cost and processor utilization. One of the reasons for this is that TP-heuristic takes advantage of the fact that two possible orientations are allowed in the packing. TP has the additional advantage over LP in that it avoids repacking and hence has less time complexity.

5 Extension for Inter-grid Communication Cost

The solution method based on two-dimensional packing ignores inter-grid communication cost. In this section, we discuss possible approaches for minimizing inter-grid communication

cost as well as computation and intra-grid communication costs. First we consider one-dimensional problem and propose an iterative approach. Then we discuss the possibility of applying it and other approaches to solving the two-dimensional case.

5.1 One-Dimensional Problem

Processor allocation for the one-dimensional problem is relatively easier than two-dimensional problem. We can find an optimal solution when there is only one grid at each level. Let X be the number of contiguous processors (in one-dimensional mesh) allocated for the newly generated fine grid. Then, the length of the communication path for inter-grid communication is $X + \alpha$ in the worst case, where α depends on the actual locations of these X processors. When there is only one fine grid at each level, we can allocate X processors including the processors where the fine grid was generated. Then α becomes a constant. For the fine grid at level 2, in Figure **??** (a) $X = 2$ and $\alpha = -2$, and in Figure **??** (b) $X = 4$ and $\alpha = -2$. If we uniformly divide the fine grid among X processors, the number of grid points each processor is responsible for is $\frac{M}{X}$, where M is the total number of grid points in the fine grid. Then, the problem is finding X which minimizes

$$T_i = U_{comp}\frac{M}{X} + U_{comm}\frac{M}{X}(X + \alpha) + C_I$$

The first term is the computation cost for each processor to which $\frac{M}{X}$ grid points are assigned. The second term is the worst case inter-grid communication cost when the data have to travel through a path of length $X + \alpha$. The third term C_I is the cost of intra-grid communication. It is a constant because regardless of X, it is necessary to exchange only two grid points (left end and right end) values with neighboring processors. In the above equation T_i decreases indefinitely up to C_I as X increases. But, as we increase X, $\frac{M}{X}$ reaches packet size S_P and communication cost is only proportional to $X + \alpha$. Then, the above equation becomes

$$T_i = U_{comp}\frac{M}{X} + U_{comm}S_P(X + \alpha) + C_I, \text{ when } \frac{M}{X} \leq S_P$$

If we differentiate T_i,

$$\frac{dT_i}{dX} = \frac{-U_{comp}M}{X^2} + U_{comm}S_P$$

$\frac{dT_i}{dX}$ becomes 0 when X is equal to $\sqrt{\frac{U_{comp}M}{U_{comm}S_P}}$. Therefore, T_i is minimized when X is equal to $\max(\frac{M}{S_P}, \sqrt{\frac{U_{comp}M}{U_{comm}S_P}})$.

This analysis is valid when there is only one fine grid at level i. The problem becomes more difficult when there are more than one fine grid at level i. If each grid is assigned to a set of X processors that are in the neighborhood of the set of processors where it was

generated, a grid can overlap with other nearby grids in some processors. The workload in these processors will be twice as much as that in the other processors. To avoid this workload imbalance, some grids have to be assigned to processors far from those where they were generated. This will make the maximum length of data path greater than X. Let us assume that there are m fine grids (grid 1, grid 2, $\cdots$, grid m from left to right in the one-dimensional domain) at level i and the number of grid points in grid j is M_j. We also assume that grid j is assigned to $P_{l_j}, P_{l_j+1}, \ldots, P_{r_j}$, and $X_j = r_j - l_j + 1$. If we allow overlapping of grids in some processors, workload imbalance among processors is inevitable. Hence, we avoid assigning more than one grid to any processor. Now, when there are m grids at level i, the computation time T_i becomes as follows:

$$T_i = \max_{1 \le j \le m} \left(U_{comp} \frac{M_j}{X_j} + U_{comm} f(X_j)(X_j + \alpha_j) + C_I \right)$$

where $f(X_j) = \max(\frac{M_j}{X_j}, S_P)$. α_j is the extra path length caused by assigning the grid far from the processors where it was generated to avoid overlapping. Here, each α_j depends on the assignments of the other grids at the same level, unlike the case when there is only one grid in which case it is a constant. Therefore, we cannot compute individual values of X_j in the same way as before. In general, it is difficult to find each X_j without knowing α_j. Here, we propose a simple heuristic using an iterative procedure.

1. For each j, $X_j = \max(\frac{M}{S_P}, \sqrt{\frac{U_{comp} M}{U_{comm} S_P}})$.
   This is the optimal number of processors allocated to grid j if we ignore the other grids at the same level. If $\sum_j X_j$ is greater than N, each X_j is reduced proportionally according to M_j.

2. Let s_j be the index of the processor where grid j was generated. If grid j was generated in a set of processors, then let s_j be the index of the leftmost processor. Allocate $P_{l_j}, P_{l_j+1}, \ldots, P_{r_j}$ to grid j in the following way.

   $ptr = 0$
   for $j = 1$ to m
   $\quad l_j = \max(ptr, s_j), \quad r_j = l_j + X_j - 1, \quad ptr = r_j + 1$
   if $(ptr > N)$ then
   $\quad ptr = N - 1$
   $\quad$ for $j = m$ to 1
   $\quad\quad r_j = \min(ptr, r_j), \quad l_j = r_j - X_j + 1, \quad ptr = l_j - 1$

3. Using the above processor allocations, compute T_i and $\alpha_j = l_j - s_j$. Also find j_{max} such that $T_i = U_{comp} \frac{M_{j_{max}}}{X_{j_{max}}} + U_{comm}(\frac{M_{j_{max}}}{X_{j_{max}}} + 2)(X_{j_{max}} + \alpha_{j_{max}}) + C_I$. Since each X_j was

obtained assuming α_j to be 0, we may be able to reduce T_i by reducing $\alpha_{j_{max}}$. Assume that $\alpha_{j_{max}} > 0$ and let $k = \max_{(\alpha_l=0)\ \&\ (l<j_{max})} l$. Reduce $X_k, X_{k+1}, \ldots, X_{j_{max}-1}$ by one and reallocate processors to each grid j $(k \leq j \leq j_{max})$.

4. Recompute T_i using the above reallocation of processors. If it was improved then go to step 1.

5.2 Two-Dimensional Problem

Considering inter-grid communication cost for two-dimensional problem is very complicated. When the grids clustered on the computational domain, allocation of processors for one grid cannot be determined independently since grids are closely located to each other. The iterative heuristic used for one-dimensional problem previously can be considered for solving the two-dimensional problem. But, after we find $X_i \times Y_i$ submesh for each grid i assuming there is only one grid at a level, we still need to find the locations of these submeshes. For the one-dimensional problem, we assigned m linear subarrays in the same order as the grids. But, we need to solve a packing problem to arrange these submeshes without overlapping. This packing problem is different from the one we considered in the previous section since we have to assign submeshes that are somewhat close to the original locations of the grids. This problem need to be solved again and again after we reduce the sizes of submeshes by some quantities. For this reason, we cannot apply the same heuristic to the two-dimensional problem without major changes.

One possible solution for assigning submeshes is starting with a submesh for a grid located at the center, then assigning other submeshes around it using a simple packing algorithm. But, after reducing the sizes of the submeshes, we have to rearrange the submeshes so that maximum inter-grid communication distance is reduced. This may not be a trivial problem without overlapping submeshes.

Another possible way of allocating submeshes considering inter-grid communication cost is imposing a constraint on it as we did in the first approach. When the grids are clustered, we can compute the size of a submesh which all m grids will be assigned to. Let $D(PS_i)$ be the diameter of a submesh PS_i, then $\max_{a \in S} U_{comm} D(PS_i)(B_a + M_a) \leq C_{max}$ where all the other notations are defined in chapter 3. The size of the submesh should be small enough so that maximum inter-grid communication cost cannot exceed the limit no matter which processors are allocated for a grid within the submesh. Then, we can use the two-dimensional packing approach described in the previous section to allocate processors for each grid within the submesh. Since we do not use all the processors, computation time will increase. When there are more than one cluster of grids, we have to find a submesh for each cluster. This

problem is the same as finding a submesh for each grid, but the number of submeshes we need to find is much smaller because we need one submesh per a cluster of grids. If the number of clusters is reasonably small, it will be easier than solving the original problem.

There is another alternative solution when the grids are scattered instead of clustered. Consider mesh processors as domain and fine grids generated by them as workload. Then we partition the processors as we decompose the domain using binary decomposition (see Figure 25). The processors are partitioned in such a way that the computational requirement

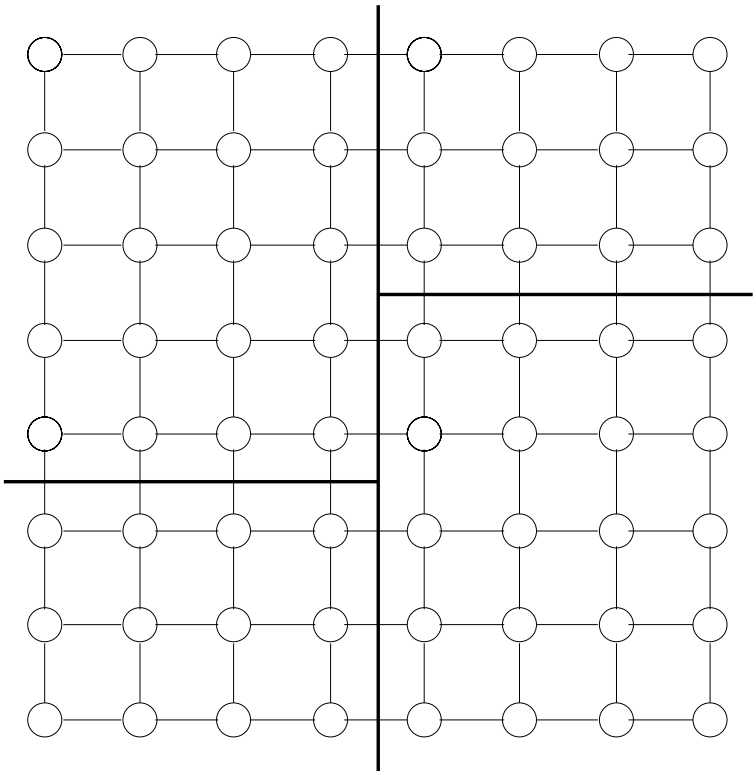

Figure 25: Partitioning of Processors

of grids are balanced among the different groups of processors. We need to repeat partitioning the processors until the maximum inter-grid communication cost cannot exceed the limit when processors are allocated within the group. Then we again use the packing approach for each group of processors. This approach will work when the fine grids are reasonably scattered.

We considered a few possible approaches to include inter-grid communication cost in our objective function. Each approach seems to be suitable for a particular type of coarse grid distribution among the processors. But deciding which approach to apply after generating a set of fine grids is a challenging problem. Thus, the inclusion of inter-grid communication cost makes the load balancing problem complicated and requires further study.

6 Conclusions

We have formulated the dynamic processor allocation algorithm that arises in mapping adaptive mesh computations onto multiprocessors. For mesh (and hypercube) architectures, the problem can be transformed into a 2D packing problem. We proposed an efficient approximation heuristic and derived accuracy bounds for square meshes. We also demonstrated through experiments how our approximation algorithm can provide solutions with smaller computational and communication costs than one of the well-known packing heuristics adapted to this problem. We derived the bounds for our heuristic when the grids are packed in decreasing order of their maximum side lengths. We plan to analyze its performance for other orders such as decreasing order of areas, aspect ratios etc.. One of our challenging future tasks is to address the load balancing problem when inter-grid communication cost is also considered in the objective function.

References

[1] S. Acharya and F Moukalled, "An Adaptive Grid Solution Procedure for Convection-Diffusion Problems," *Journal of Computational Physics*, Vol.91, 1990, pp.32-54.

[2] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1974.

[3] I. Altas and J. Stephenson, "A Two-Dimensional Adaptive Mesh Generation Method," *Journal of Computational Physics*, Vol.94, 1991, pp.201-224.

[4] B. S. Baker, D. J. Brown and H. P. Katseff, "A 5/4 Algorithm for Two-Dimensional Packing," *Journal of Algorithms*, Vol.2, 1981, pp.348-368.

[5] B. S. Baker, E. G. Coffman and R. L. Rivest, "Orthogonal Packings in Two Dimensions," *SIAM Journal on Computing*, Vol.9, No.4, August, 1980, pp.846-855.

[6] B. S. Baker and J. S. Schwarz, "Shelf Algorithms for Two-Dimensional Packing Problems," *SIAM Journal on Computing*, Vol.12, No.3, August, 1983, pp.508-525.

[7] M. Berger and S. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Transactions on Computers*, Vol.C-36, 1987, pp.570-580.

[8] M. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations," *Journal of Computational Physics*, Vol.53, 1984, pp.484-512.

[9] M. Berger and I. Rigoutsos, "An Algorithm for Point Clustering and Grid Generation," *IEEE Transactions on Systems, Man and Cybernetics*, Vol.21, no.5, 1991, pp.1178-1286.

[10] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[11] J. Boillat, F. Bruce and P. Kropf, "A Dynamic Load-Balancing Algorithm for Molecular Dynamics Simulation on Multi-processor Systems," *Journal of Computational Physics*, Vol.96, 1991, pp.1-14.

[12] S. H. Bokhari, "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed System," *IEEE Trans. Software Engineering*, Vol.7, No.6, November 1981, pp.583-589.

[13] S. W. Bollinger and S. F. Midkiff, "Processor and Link Assignment in Multicomputers Using Simulated Annealing," *Proceedings of the 1988 International Conference on Parallel Processing*, Vol.1, 1988, pp.1-7.

[14] R. G. Casey, "Allocation of Copies of a File in an Information Network," *Proc. AFIPS National Computer Conference*, Vol.43, 1972, pp.371-374.

[15] W. W. Chu, "Optimal File Allocation in a Computer Network," in *Computer Communication Systems*, N. Abramson and F. F. Kuo eds., Prentice-Hall, Englewood Cliffs, NJ, 1973, pp.82-94.

[16] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, Inc. New York, 1976.

[17] E. G. Coffman, M. R. Garey and D. S. Johnson, "An Application of Bin-packing to Multiprocessor Scheduling," *SIAM Journal on Computing*, Vol.7, No.1, February, 1978, pp.1-17.

[18] E. G. Coffman, M. R. Garey, D. S. Johnson and R. E. Tarjan, "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms," *SIAM Journal on Computing*, Vol.9, No.4, August, 1980, pp.808-826.

[19] W. Y. Crutchfield, "Load Balancing Irregular Algorithms," Lawrence Livermore National Laboratory Report, 1991.

[20] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol.7, 1989, pp.279-301.

[21] K. D. Devine, J. E. Flaherty, S. R. Wheat and A. B. Maccabe, "A Massively Parallel Adaptive Finite Element Method with Dynamic Load Balancing," *Proceedings of Supercomputing '93 Conference*, Portland, Oregon, 1993, pp.2-11.

[22] A. K. Dewdney, "Computer Recreations," *Scientific American*, Dec. 1984, pp.14-18.

[23] L. W. Dowdy and D. V. Foster, "Comparative Models of the File Assignment Problem," *ACM Computing Surveys*, Vol.14, No.2, June 1982, pp.287-313.

[24] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, San Francisco, CA, 1983.

[25] R. Hanxleden and L. R. Scott, "Load Balancing on Message Passing Architectures," *Journal of Parallel and Distributed Computing*, Vol.13, 1991, pp.312-324.

[26] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann Publishers, Inc. San Mateo, CA, 1991.

[27] D. Nicol and N. Saltz, "Dynamic Remapping of Parallel Computations with Varying Resource Demands," *IEEE Transactions on Computers*, Vol.37, No.9, 1988, pp.1073-1087.

[28] D. Nicol and N. Saltz, "An Analysis of Scatter Decomposition," *IEEE Trnasactions on Computers*, Vol.39, No.11, 1991, pp.1337-1345.

[29] H. D. Simon, "Partitioning of Unstructured Problems for Parallel Processing," *Computing Systems in Engineering*, Vol.2, No.2/3, 1991, pp.135-148.

[30] P. Sadayappan and F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes," *IEEE Transactions on Computers*' Vol.C-36, No.12, 1987, pp.1408-1424.

[31] Ravi Varadarajan and Eva Ma, "An Approximation Load Balancing Model with Resource Migration in Distributed Systems," *Proceedings of the 1988 International Conference on Parallel Processing*, Vol.1, 1988, pp.13-17.

[32] W. Williams, "Load Balancing and Hypercubes: A Preliminary Look," *Hypercube Multiprocessors 1987*, SIAM, Philadelphia, PA, 1897, pp.108-113.

[33] J. Woo and S. Sahni, "Load Balancing on a Hypercube," *Proceedings of the Fifth International Parallel Processing Symposium*, Anaheim, CA, 1991, pp.525-530.