Cluster Partitioning Approaches to Mapping Parallel Programs Onto a Hypercube

P. Sadayappan, F. Ercal[*] and **J. Ramanujam**
Department of Computer and Information Science
The Ohio State University, Columbus Ohio 43210

Abstract

The task-to-processor mapping problem is addressed in the context of a local-memory multiprocessor with a hypercube interconnection topology. Two heuristic cluster-based mapping strategies are compared - 1) a nearest-neighbor approach and 2) a recursive-clustering scheme. The nearest-neighbor strategy is shown to be more effective on hypercube systems with high communication start-up costs, especially for finite element graphs; the recursive partitioning heuristic is generally better on hypercubes with lower communication start-up costs and is more effective on random task graphs.

[*]Current address: Computer Engineering & Information Science Department, Bilkent University, Ankara, Turkey.

1 Introduction

The effective exploitation of the potential power of parallel computers requires efficient solutions to the task-to-processor mapping problem. The problem is that of optimally allocating the tasks of a parallel program among the processors of the parallel computer in order to minimize execution time of the program. The mapping problem is NP-complete [8, 9], except in a few specific contexts with very unrealistic constraints that typically do not hold in practice. Hence the various approaches that have been proposed all seek to obtain satisfactory sub-optimal solutions in a reasonable amount of time [1-4,6-9,11-18]. This paper discusses two heuristic approaches based on cluster-partitioning - 1) a nearest-neighbor approach and 2) a recursive-clustering approach. We compare and contrast the way in which these two approaches use heuristics to limit the configuration space of possible mappings that is (partially) searched to derive satisfactory mappings in reasonable time.

The mapping problem has been investigated in two distinct settings that we refer to as the Task Precedence Graph (TPG) model and the Task Interaction Graph (TIG) model. With the TPG model, the parallel program is modeled as a collection of tasks with known (or estimated) computation times and explicit execution dependences expressed in the form of precedence relations. An example of a TPG is shown in Fig. 1(a). The parallel program is represented as a graph whose vertices represent tasks and the directed edges represent execution dependences. In the example shown in Fig. 1(a), task **h** can only commence execution after tasks **f** and **g** have completed execution. The weight associated with a vertex represents the relative computation time required for its execution and the weight associated with an edge represents the relative amount of data to be communicated between two tasks. Thus task **f** requires twice as much computation as **b** and the amount of data to be transferred between **f** and **h** is thrice as much as that between **a** and **b**. This model is very appropriate in the context of parallel programs that may essentially be characterized by an acyclic task graph.

With the TIG model, the parallel program is again modeled as a graph where the vertices represent the parallel tasks and vertex-weights represent known or estimated computation costs of the tasks. In contrast to the TPG model, however, execution dependences are not explicitly captured; thus, the edges of a TIG represent communication requirements between tasks, with edge-weights reflecting the relative amounts of communication involved, but do not capture any temporal execution dependences. Thus in Fig. 1(b), the relative amount of data communication required between tasks **a** and **b** is 2, but nothing is stated regarding the temporal dependences between these two tasks. The TIG approach can be used to approximately model the behavior of complex parallel programs by lumping together all estimated communications between pairs of tasks and ignoring the temporal execution dependences. There is also a class of parallel programs, that we call iterative parallel programs, where the TIG model is quite accurate [1]. With this class of programs, execution proceeds as a sequence of sequential iterations. In each iteration, all parallel tasks can execute independently, but each task then needs to communicate values computed during

Figure 1: Two approaches to modeling of parallel programs

that iteration with tasks it is connected to in the TIG, before it can commence its next iteration. Given a mapping of tasks to processors, over a large number of iterations, the slowest processor will clearly control program completion time. Thus minimizing iteration step completion time of the slowest processor (sum of computation time and inter-processor communication time for all inter-task communications that need to go across to other processors) will minimize execution time of the parallel program. Hence there is no need to explicitly model temporal dependences and their satisfaction on a per-task basis, as the sequence of iterations of the iterative parallel program proceeds. This is the nature of the mapping problem addressed in this paper; an example of its use is presented in [1] in the context of parallelizing a finite element modeling program employing iterative techniques for the solution of large sparse systems of linear equations.

Given a parallel program that is characterized by a TIG, the mapping of the tasks to processors may either be performed statically (before program execution) or dynamically in an adaptive manner as the parallel program executes. The appropriate approach depends on the nature of the TIG. If the TIG that characterizes the parallel program is static, i.e. the vertex and edge weights can be accurately estimated a priori, then a static approach is more attractive, since the mapping computation need only be performed once. This is the case, for example, with many iterative parallel programs. We only consider static mapping schemes in this paper.

If the physical communication costs of the parallel computer system can be quantitatively characterized, then by using it in conjunction with the parallel program's TIG, a cost function can be formulated to evaluate the effectiveness of any particular mapping of the tasks onto the processors. The mapping problem can thus be formulated as a problem of finding a mapping that minimizes this mathematical cost function, and indeed many researchers have done so [7, 8, 11, 13, 16, 17, 18]. In some contexts, mapping schemes have been proposed that utilize intuition about the specific mapping context and do not use any explicit cost functions in the mapping procedure – for example, the scattered decomposition scheme proposed in [12] and heuristics proposed in [15]. Such schemes, which we call "domain-heuristic" schemes, are computationally very efficient. In contrast, schemes that explicitly optimize cost functions are often computationally time consuming, but are more generally applicable and potentially capable of obtaining better mappings. The primary problem with most of the approaches based on explicit minimization of a cost function is the exponentially large configuration space of possible mappings that must be selectively searched in seeking to optimize the function. Many proposed heuristics to explicitly minimize cost functions have been shown to work relatively well on small task graphs (with a few tens of tasks), but have not been evaluated on large task graphs (with hundreds or thousands of nodes) [16, 17, 18]. The "flat" and unstructured view of task allocation as a mapping of each task onto a processor leads to an unmanageably large search space. If the task graph has N nodes and the parallel computer has P processors, the space of possible mappings has P^N configurations.

We propose that hybrid approaches that combine characteristics of domain-heuristics and the use of an explicit cost function are most attractive for the mapping problem. Instead of using "flat and unstructured" strategies aimed at minimizing a mathematically formulated cost function, domain-heuristics are used in conjunction with an explicit cost function to significantly reduce the search space of potentially good candidate mappings.

Our approach to mapping is motivated by the following observations:

- **Clustering:** Pairs of tasks that require much communication are better grouped together and mapped to the same processor in order to reduce communication costs. Thus the task-to-processor mapping problem may be viewed instead as a problem of forming clusters of tasks with high intra-cluster communication and low inter-cluster communication, and allocating these clusters to processors in a way that results in low inter-processor communication costs. This view of mapping in terms of formation of clusters of closely coupled tasks helps reduce significantly the space of mappings that is selectively searched for a satisfactory solution.

- **Load Balancing:** The clusters should be formed in such a way as to distribute the total computational load as uniformly as possible among the clusters. This requirement can often conflict with the requirement for minimal inter-cluster communication. We may hence identify computational load balancing and minimization of inter-processor communication as the two key objectives of the mapping. The component terms in typical cost functions used for mapping often quantify one or the other of these two requirements.

- **Complexity:** The complexity of the mapping problem essentially arises due to the need to simultaneously achieve both the objectives of load-balancing and minimization of communication. Rather than use two explicit terms in the cost function, one representing load balancing and another representing communication costs, the search space can be reduced if only one of these two is explicitly modeled while the other is used implicitly in guiding the search. As explained later, the two schemes used in this paper represent two alternate ways of doing so. The nearest-neighbor mapping scheme explicitly attempts load balancing among clusters, whereas low communication costs are achieved implicitly through the use of a domain-heuristic. In contrast, the recursive-clustering approach explicitly attempts to minimize communication costs, while load balancing is achieved implicitly by the search strategy.

The paper is organized as follows. In section 2, we formalize the mapping problem that we address. In section 3, we discuss the nearest-neighbor approach to mapping and in section 4, we develop the recursive-clustering scheme. In section 5, we compare the effectiveness of the two schemes on a number of sample task graphs and summarize in section 6.

2 The Mapping Problem

In this section, we formalize the mapping problem considered and develop the cost function that is used as a measure of goodness of the mapping. The parallel program is characterized by a Task Interaction Graph $G(V, E)$, whose vertices, $V = \{1, 2, \ldots, N\}$, represent the tasks of a program, and edges E, characterize the data communication requirements between tasks. The weight of a task i, denoted w_i, represents the computational load of the task. The weight of an edge (i, j) between i and j, denoted c_{ij}, represents the relative amount of communication required between the two tasks.

The parallel computer is represented as a graph $G(P, E_p)$. The vertices $P = \{1, 2, \ldots, K\}$ represent the processors and the edges E_p represent the communication links. The system is assumed to be homogeneous, with identical processors. Hence, in contrast to the Task Interaction Graph (TIG), no weights are associated with the vertices or edges of the Processor Interconnection Graph. The processors are assumed to either execute a computation or perform a communication at any given time, but not simultaneously do both.

The cost of communication between a processor and its neighbor is assumed to consist of two components – 1) a startup (setup) cost T_s, that is independent of message size, and 2) a data transmission cost $l \times T_t$, that is proportional to the number of bytes transmitted. This is characteristic of all commercially available hypercube computers, with T_s representing the cost of a kernel call and context switch to process the message and T_t characterizing the physical rate of data transmission on the inter-processor communication link. For communication between remote processors, with hypercube systems that use a packet-switched store-and-forward mechanism, the source processor, destination processor and the intermediate processors expend time corresponding to message setup and data transfer.

The task-to-processor mapping is a function $M : V \rightarrow P$. $M(i)$ gives the processor onto which task i is mapped. The **Task Set** (TS_q) of a processor q is defined as the set of tasks mapped onto it:

$$TS_q = \{j | M(j) = q\}, \qquad q = 1, \ldots, K$$

The **Work Load** (WL_q) of processor q is the total computational weight of all tasks mapped onto it:

$$WL_q = \sum_{j \in TS_q} w_j, \qquad q = 1, \ldots, K$$

and the idealized average load is given by $\overline{WL} = \frac{1}{K}\sum_{i=1}^{K} WL_i$. The **Communication Set** (CS_q) of processor q is the set of the edges of the TIG whose images under the mapping go through the processor q, based on the routing used for messages on the hypercube.

The **Communication Load** (CL_q) of processor q is the total time spent by processor q in performing communication. This includes the communication start-up time and data transfer time for each message originating or terminating at q and for each message routed through q.

Cost functions that have been used with the task allocation problem may be broadly categorized as belonging to one of two models: a *minimax cost* model [11, 16] or a *summed total cost* model [3, 7, 8]. With the minimax cost model, the total time required (the execution time + communication time + idle time) by each processor under a given mapping is estimated and the maximum cost(time) among all processors is to be minimized. By making a simplifying assumption that the maximally loaded processor does not suffer significantly from idle time, the function to be minimized is:

$$T_{\text{minimax}} = \min_{M} \left\{ \max_{q} (CL_q + WL_q) \right\}, \qquad q = 1, \ldots, K \tag{1}$$

3 Nearest-Neighbor Mapping

The nearest-neighbor mapping strategy was proposed in [13] as an effective approach to mapping finite element graphs onto processor meshes. Given a regular $m \times n$ rectangular mesh of processors $P_{ij}, i = 1 \ldots m, j = 1 \ldots n$, a *nearest-neighbor* of a processor P_{ij} is any processor P_{kl}, where k is either $i-1, i,$ or $i+1$, and l is either $j-1, j,$ or $j+1$. A nearest-neighbor mapping is one where any pair of nodes that share an edge in the TIG are mapped onto nearest-neighbor processors. The essential idea behind the nearest-neighbor mapping approach is that if TIG nodes are assigned to processors in a manner that results in just nearest-neighbor communication, then the total communication cost should be low. Starting with an initial nearest-neighbor mapping, successive incremental modification of the mapping is done to improve load-balancing among the processors while always maintaining the nearest-neighbor property. The search strategy reduces the configuration space of possible mappings by incorporating a "cluster" viewpoint of the mapping as opposed to a "flat" individual-task-to-processor viewpoint. Moreover, the search space is further reduced by being explicitly concerned only with the load-balancing aspect of the mapping, while the implicit nearest-neighbor constraint in conjunction with node-clustering aids in keeping communication costs low. In this section we explain the approach through an example and refer the reader to [13] for algorithmic details.

The nearest-neighbor mapping algorithm proceeds in two phases:

1. An initial mapping is first generated by grouping nodes of the TIG into clusters and assigning clusters to processors in a manner that the nearest-neighbor property is satisfied.

2. The initial mapping is successively modified using a boundary-refinement procedure where nodes are reassigned among processors in a manner that improves load balancing but always maintains the nearest neighbor-property.

Allowing for the possibility that some of the processors do not get any nodes mapped onto them, a nearest-neighbor mapping of any TIG onto a processor mesh is clearly always possible. In the extreme case, all TIG nodes form a single cluster that is assigned to one of the processors, while all other processors are assigned no tasks at all. The initial mapping algorithm of course attempts to do considerably better than that extreme case. The initial-mapping procedure may be understood through the example in Fig. 2, requiring the mapping of a 40-node TIG onto a 2×4 mesh of 8 processors. Here the TIG is a "mesh graph", a graph that can be embedded onto some subset of a uniform 2-dimensional grid with a dilation of 1, i.e. the nodes of the TIG can be mapped onto the grid points of a uniform 2-dimensional grid in such a way that any two nodes of the TIG that share an edge are mapped onto adjacent (vertically or horizontally) grid points. We first use an example of a mesh graph to simplify the explanation of the initial mapping scheme and later point out how the scheme can be generalized for arbitrary graphs.

The basis for the initial mapping procedure is the following 1-dimensional strip mapping scheme. Let us imagine that all the processors in each row of the processor mesh are lumped together to create a "macro-processor". The 8-processor mesh in Fig. 2 can be so treated as a linear chain of 4 macro-processors, as seen in Fig. 2(a). The essential idea behind the 1-D strip method is to partition the mesh graph into "strips", where each strip encompasses one or more contiguous rows (columns) of the graph. The mesh graph is covered by strips in such a way that each strip is adjacent to at most two other strips, one on either side of it. The number of strips used equals the number of processors (macro-processors) in the processor chain, and the strips can be made to each contain an equal number of mesh graph vertices (plus or minus one, if the number of graph vertices is not a perfect multiple of the number of strips created). Given a mesh with N nodes, to be mapped onto a linear chain of P processors, a load-balanced mapping will assign either $\lceil \frac{N}{P} \rceil$ or $\lfloor \frac{N}{P} \rfloor$ nodes to each processor (the former to $(N \bmod P)$ processors and the latter number to $(P - (N \bmod P))$ processors). Starting with the leftmost node in the uppermost row of the mesh-graph, nodes are checked off and assigned to a processor by proceeding across the row

Figure 2: Illustration of the Nearest-Neighbor approach to mapping

Figure 3: Example of 1-D Strip-Partitioning of a Non-Mesh TIG

of the mesh graph. If the first row is completely exhausted, then we begin at the left end of the next row and continue assigning nodes to the processor. When the required number of nodes has been assigned, we begin assigning the remaining nodes in that row to the next processor in the linear chain of processors. Proceeding in this manner, all nodes in the problem mesh are assigned to processors. Thus in Fig. 2(a), the graph is partitioned into 4 strips containing 10 tasks each, to be assigned to each of the 4 macro-processors. This could be thought of as a symmetric contraction of the graph along one dimension.

A similar procedure can be used to create a vertical 1-D strip partition of the same graph, as shown in Fig. 2(b). Now we group all processors in a column together, to form a chain of two macro-processors. The TIG is partitioned into two vertical strips. By overlapping the two orthogonal strip-partitions generated, and forming the intersections, we can generate a number of regions, that equals the number of processors in the processor mesh, as shown in Fig. 2(c). The nature of the construction guarantees that the generated partitions satisfy the nearest-neighbor property. The two independent partitions of such a 2-D partition are each individually load balanced, but the intersection partitions are generally not load-balanced, as can be seen from the table of assigned loads in Fig. 2(c). However, this serves as a good initial nearest-neighbor mapping that is then refined by the load-balancing boundary refinement procedure.

One way of balancing the computational loads of the processors is to reassign some of the tasks among the processors; for example, by transferring one task from P_{22} to each of P_{12} and P_{21}, one task from P_{11} to P_{21} and one task from P_{31} to P_{32}, as shown graphically in Fig. 2(d) as a Load Transfer Graph (LTG). In general, given an initial mapping, it is possible to determine a set of load transfers that will balance the load, by setting up and solving a system of linear equations in integer variables.

A heuristic boundary-refinement procedure (described in detail in [13] iteratively attempts to transfer tasks between processors using the LTG. An "active" processor list is formed, sorted in decreasing order of current task-load, comprising all processors that source an edge in LTG. A task is sought in the most heavily loaded active processor (preferably on the partition boundary), to be transferred to a neighbor processor in the LTG, ensuring that the nearest-neighbor constraint is not violated. If no such task can be found, the sorted active processor list is scanned in decreasing order of processor loads till an active processor with a transferable task is found. The task transfer is made and the LTG is updated by decreasing the relevant edge-weight by one and removing it if its edge-weight becomes zero. The algorithm thus proceeds incrementally to refine the mapping by perturbing the boundaries of the partitions in attempting to balance the load. While its effectiveness depends on the initial partition generated, it has been found to be good in practice. Even initial mappings of finite element graphs, where one or more processors get no assigned nodes have typically been load-balanced after boundary refinement. The final mapping after application of the reassignment procedure is shown in Fig. 2(e) for the chosen example.

A generalization of the 1-dimensional strip method to non-mesh graphs is possible. This is illustrated through an example in Fig. 3. The essential idea again is to create strip-like regions to cover the graph, such that if a node is on a certain strip, all other nodes sharing an edge with that node in the TIG should lie either on the same strip or on an adjacent strip. In the case of mesh graphs, the process of generating strips was facilitated by the regularity of the mesh-graph and the natural grouping of nodes into columns and rows. This is not so for a general non-mesh TIG. Such a grouping is hence created by a levelization process that assigns a unique level to each node. Starting with a single randomly selected node, or if possible a peripheral node, or a set of connected peripheral nodes that are assigned to level 1, all nodes directly connected to these level 1 nodes, and that have not yet been assigned a level, are assigned a level number of 2. The same procedure is carried out with the nodes at level 2, and continued thus till all nodes are levelized. The nature of the levelization procedure ensures that a neighbor node of any node assigned level l, will necessarily be assigned level $l-1, l$ or $l+1$. Now, strip partitioning can be performed using the levels similar to using columns(rows) in the earlier described procedure for 1-dimensional strip partitioning for mesh-graphs, as seen in figure 3.

In the case of mesh graphs, mapping onto an $m \times n$ processor mesh was achieved by performing two independent 1-D partitions - one n-way (in the horizontal direction) and the other m-way (in the vertical direction). In the case of non-mesh graphs, the difficulty is in the generation of a second levelization that is orthogonal to the first one. This is because, unlike with mesh graphs, it is not meaningful to associate physical directions such as vertical and horizontal with the levels generated, as can be seen with many of the levels in figure 3. A heuristic is hence used to attempt the generation of a second levelization that is as orthogonal to the first one as possible. This is done by first identifying so called "corner" nodes of the TIG - these are peripheral nodes that form end-points of the maximum diameter(s) of the TIG, i.e. are nodes with maximum distance of separation in the TIG. Starting with an arbitrary corner node, one fourth of the peripheral nodes visited upon traversing the periphery in one direction are used as the nodes in level 1 of the first levelization; the same number of nodes visited upon traversing the periphery in the opposite direction are used for level 1 of the second levelization.

The nearest-neighbor approach can be expected to be quite effective in mapping TIG's that exhibit a high degree of locality, as is the case with parallel programs modeling physical systems using finite element methods, finite difference methods etc. When the TIG does not represent purely local interactions, but has "non-local" connectivity, the nearest-neighbor restriction becomes overly constraining and load balancing becomes extremely difficult. An alternate cluster-based mapping scheme is developed in the next section that is not so restrictive. Instead of using a search strategy that explicitly attempts load balancing while implicitly keeping communication costs low, the opposite view is taken, of explicitly attempting to minimize communication costs by the use of an explicit cost function, while achieving load balance implicitly through the nature of the developed algorithm.

```
Algorithm Recursive_Clustering ($G_I, G_P, M$)
    /* $G_I = (T, E)$ is the input TIG along with vertex and edge weights */
    /* $G_P = (P, E_P)$ is the Processor Interaction Graph of the target hypercube */
    /* $M$ is a mapping $V \rightarrow P$ that is the output of the algorithm */

    /* Phase 1 */

    Set Depth = 0 and Maxdepth = $\log_2 |P|$
    Form_Clusters($G_I$, Depth, Maxdepth,S)
            /* $S$ is a set of $|P|$ clusters returned by Form_Clusters by */
            /* partitioning the graph $G_I$. Minimization of inter-cluster */
            /* communication volume is attempted by Form_Clusters */

    - From the set $S$ of clusters, form graph $G_S$ that characterizes the
        inter-cluster communication.
            /* $G_S = G(C, E_S)$ has $|P|$ vertices, one for each cluster in $S$ */
            /* $G_S$ has an edge between two vertices if the corresponding clusters */
            /* contain a vertex each of $G_I$ that have an edge between them in $G_I$ */
            /* The edges of $G_S$ have weights that are the sum of the weights */
            /* of the relevant edges in $G_I$ */

    /* Phase 2 */

    Allocate_Processors($G_S, G_P, M$)
            /* The vertices of $G_S$ are mapped onto processors of $G_P$ attempting */
            /* to minimize the total inter-processor communications volume, */
            /* accounting for physical distances */
```

Figure 4: Algorithm for Recursive-Clustering

4 Mapping by Recursive-Clustering

A recursive-clustering scheme is developed in this section to map arbitrary TIG's onto a local memory machine with a hypercube interconnection topology. The algorithm proceeds in two phases:

1. **Cluster Formation:** The TIG is first partitioned into as many clusters as the number of processors. Starting with the entire TIG as a single cluster, this is done recursively, by successively dividing a cluster into two equally vertex-weighted (as nearly as possible) partitions with minimal total weight of inter-partition edges.

2. **Processor Allocation:** The clusters generated in the first phase are each allocated to some processor, one cluster per processor, in a manner that attempts to minimize the total inter-processor communication volume.

Both the phases of the algorithm – the cluster-formation phase and the processor-allocation phase –

```
Algorithm Form_Clusters ($G$,Depth,Maxdepth,$S$)

  if (Depth = MaxDepth)
  then /* No more partitioning of $G$; add it to the set of clusters $S$ */
  else /* Recursively partition */
     {
        Mincut($G, G_L, G_R$) /* Partition $G$ into two equal parts $G_L$ and $G_R$ */
        Form_Clusters($G_L$,Depth+1, Maxdepth,$S$)
        Form_Clusters($G_R$,Depth+1, Maxdepth,$S$)
     }
```

Figure 5: Algorithm for Forming Clusters

explicitly attempt to minimize communication volume through the use of an iterative improvement heuristic based on the Kernighan-Lin mincut algorithm [10] and load balancing is achieved implicitly in the first phase of the algorithm.

Kernighan and Lin [10] proposed an extremely effective *mincut* heuristic for graph bisection, with an empirically determined time complexity of $O(n^{2.4})$. Their algorithm is based on finding a favorable sequence of vertex-exchanges between the two partitions to minimize the number of inter-partition edges. The evaluation of sequences of perturbations instead of single perturbations endows the method with the *hill-climbing* ability, rendering it superior to simple local search heuristics. Fiduccia and Mattheyses [5] used efficient data structures and vertex displacements instead of exchanges to derive a linear time heuristic for graph partitioning, based on a modification of the algorithm in [10]. While the original mincut algorithm of Kernighan and Lin applied only to graphs with uniform vertex weights, the Fiduccia-Mattheyses scheme can handle graphs with variable vertex weights, to divide it into partitions with equi-total vertex weights.

The basic mincut algorithm used here is similar in spirit to the Fiduccia-Mattheyses variant of the Kernighan-Lin heuristic. An initial two-way partition is created by assigning the nodes of the graph, one by one, always to the partition with lesser total weight (randomly in case both are equal). After creating the initial partition, a sequence of maximally improving node transfers from the partition with currently greater load to the partition with lower load are tried. The iterative improvement heuristic is otherwise very similar to the Kernighan-Lin mincut heuristic, except for the use of one-way node transfers instead of node exchanges. The use of node transfers in this fashion guarantees load-balance even though the individual vertex weights are variable. The mincut bipartitioning procedure is used recursively to perform a P-way partition of a graph if P is a power of 2 – by first creating two equal sized partitions, then independently dividing each of these into two sub-partitions each, and so on till P partitions are created. The recursive partitioning procedure is illustrated using an example in Figure 8(a) - 8(c). The parallel processor system has 8 processors and so the recursive partitioning procedure is applied upto depth $\log_2 8 = 3$.

```
**Algorithm Mincut** $(G_I, G_L, G_R)$

    /* accepts a graph $G_I$ as input and partitions it into */
    /* two clusters $G_L$, $G_R$ such that the cutsize is minimal */

    - Form an initial load-balanced partition of $G_I$
            assigning its nodes into one of two clusters $C_1^*$ and $C_2^*$.
    - Mark all nodes as unlocked.
    - Associate a gain value $g_v$ with each node $v$ and initialize $g_v$ to zero

    do {
        $C_1 \leftarrow C_1^*$; $C_2 \leftarrow C_2^*$;
        - Compute $\forall v \in V, g_v$ = total reduction in cost of the cut when $v$ is
                moved from its currently assigned cluster to the other cluster.
        - Compute $W_1$ and $W_2$, the total load in $C_1$ and $C_2$ respectively
        seqno $\leftarrow$ 0 ;
        do {
            seqno $\leftarrow$ seqno + 1
            - Let $C_i$ be the cluster with greater total weight and $C_j$ the lesser
                    total weight i.e., $W_i \geq W_j$, $i, j \in \{1, 2\}$, $i \neq j$
            - Among the unlocked vertices, identify $v^* \in C_i$ with maximal gain $g_v^*$
            - If no such vertex exists, exit this loop.
            - Assume that $v^*$ is moved to $C_j$, update the gain for all unlocked nodes
                    and calculate loads $W_1$ and $W_2$ for $C_1$ and $C_2$
            - Lock $v^*$ and record the status of the movement and $gain[seqno] \leftarrow g_v^*$
            } while (there is at least one vertex to move)
        Let $\mathcal{G}^* = \max_l \sum_{i=1}^{l} gain[i] = \sum_{i=1}^{l^*} gain[i]$
        where $l^*$ is the value of $l$ that maximizes the cumulative gain
        if [$(\mathcal{G}^* > 0)$ OR $((\mathcal{G}^* = 0)$ AND (better load balancing))]
            then perform all moves from 1 to $l^*$ i.e., form $C_1^*$ and $C_2^*$.
        } while $\mathcal{G}^* > 0$
    $G_L \leftarrow C_1^*$
    $G_R \leftarrow C_2^*$
```

Figure 6: Mincut Algorithm

```
Algorithm Allocate_Processors ($G_S, G_P, M$)

    /* accepts two graphs $G_S$, $G_P$ as input and returns mapping M */
    /* $G_P = G(P, E_P)$ is a graph representing the interconnection of the hypercube */
    /* $G_S = G(C, E_S)$ is a graph with $|P|$ nodes, each node is a cluster of TIG nodes */
    /* M is the mapping of clusters to processors */

    - Start with an initial mapping M that assigns cluster $C_i$ to processor $P_i$
    - Unlock all processor nodes
    do {
        - Calculate the gain for all processor pairs $P_i$ and $P_j$ assuming
                the clusters currently assigned to $P_i$ and $P_j$ are swapped.
        - seqno $\leftarrow$ 0 ;
        do {
            seqno $\leftarrow$ seqno + 1
            - Among the unlocked processor-pairs, pick the one, say, $(P_k, P_l)$ with maximum gain
            - If no such pair exists, exit this loop.
            - Assume that $P_k$ and $P_l$ are swapped, update the gain for all unlocked processor-pairs
            - Lock $P_k$ and $P_l$
        } while (there is at least one unlocked processor)
        Let $\mathcal{G}^* = \max_k \sum_{i=1}^{k} gain[i] = \sum_{i=1}^{k^*} gain[i]$
                where $k^*$ is the value of $k$ that maximizes the cumulative gain
        if ($\mathcal{G}^* > 0$) then perform all moves from 1 to $k^*$ by changing $M$
    } while $\mathcal{G}^* > 0$
    return ($M$)
```

Figure 7: Processor Assignment Algorithm

The second phase attempts to minimize total inter-processor communication volume when the clusters generated in the first phase are allocated to processors, one cluster per processor. Whereas in the first phase, clusters are formed with minimal inter-cluster communication volume, in the second phase, actual communication distances required with the cluster-to-processor mapping are taken into account in determining the mapping with minimal total inter-processor communication volume. An iterative improvement heuristic that uses the "hill-climbing" ability of Kernighan-Lin-like local search methods is used in algorithm **Allocate_Processors** (See Figure. 7). Pairs of clusters that maximally decrease the total inter-processor communication volume, are considered for swapping. This is illustrated in Figure. 8(d), where it can be seen that the cluster initially assigned to P_{41} is finally assigned to P_{22} and vice-versa, and the assignments to P_{31} and P_{12} are likewise exchanged. The total volume of inter-processor communication decreases from 164 to 144 as a result of this second phase optimization.

5 Comparison of Effectiveness of Mapping Schemes

In this section, we compare the effectiveness of the Nearest Neighbor (NN) and Recursive-Clustering (RC) schemes using a number of sample TIG's. Four of the samples are finite element graphs and two are random graphs. The first two samples are mesh-graphs (taken from [12]) and are shown in Fig. 9(a) and 9(b). These graphs exhibit a high degree of locality and are locally regular and uniform but very irregular at the outer periphery. These are representative of the kinds of graphs that result from exploiting parallelism from computations modeling physical systems by finite element and finite difference methods. Sample 3 (taken from [6]) and sample 4 (similar to those in [2]), shown in Fig. 9(c) and 9(d) respectively, are non-mesh graphs. Unlike mesh graphs, these graphs are not uniform, but nevertheless exhibit considerable locality of interconnection. They are similar to graphs obtained with the use of multi-grid methods. The last two samples are completely random graphs, generated using a random generator. These graphs are hence quite non-uniform and do not exhibit any locality of interconnection either.

Mappings were generated both using NN and RC for a target 16-processor hypercube system. The generated mappings were evaluated under two settings. Table 1 presents estimates of the speedup that would be obtained on executing iterative parallel programs characterized by the TIG's on a hypercube system with a message setup time of 1150 μs, data transmission time of 10 μs per word, $w_i = 1200\mu s$ and $e_{ij} = 1$. These parameters are representative for the parallel solution of a system of linear equations by a Gauss-Jacobi scheme. Table 2 contains estimates of speedups in an idealized setting with message setup time of zero, the other parameters being the same as for Table 1. The estimated speedup reported in Tables 1 and 2 is defined by:

$$\text{Speedup} = \frac{T_{seq}}{T_{par}} = \frac{\sum_{i=1}^{n} w_i}{T_{\text{minimax}}}$$

Figure 8: Illustration of the Recursive-Clustering Algorithm

Figure 9: Sample problem graphs used for performance evaluation

Cost of mappings produced by NN & RC						
No.	Graph Characteristics		Estimated Speedup			
	$\|V\|$	Description	RC (5 runs)			NN
			best	worst	mean	
1.	505	Sample 1 (mesh)	11.31	9.97	10.49	13.02
2.	1449	Sample 2 (mesh)	13.89	12.84	13.31	14.64
3.	602	Sample 3 (non-mesh)	10.63	9.31	10.06	12.38
4.	256	Sample 4 (non-mesh)	7.25	5.94	6.60	7.62
5.	400	Sample 5 (random)	4.48	4.47	4.48	3.96
6.	800	Sample 6 (random)	6.92	6.91	6.92	4.69

Table 1: Comparison of solution quality of NN and RC on sample graphs: Setup cost = 1150 μs; Number of processors in hypercube = 16.

Cost of mappings produced by NN & RC						
No.	Graph Characteristics		Estimated Speedup			
	$\|V\|$	Description	RC (5 runs)			NN
			best	worst	mean	
1.	505	Sample 1 (mesh)	15.23	15.07	15.15	14.44
2.	1449	Sample 2 (mesh)	15.60	15.47	15.53	15.51
3.	602	Sample 3 (non-mesh)	15.15	14.98	15.09	14.70
4.	256	Sample 4 (non-mesh)	14.76	13.96	14.21	9.87
5.	400	Sample 5 (random)	14.32	14.23	14.29	4.28
6.	800	Sample 6 (random)	14.76	14.68	14.72	4.91

Table 2: Comparison of solution quality of NN and RC on sample graphs: Setup cost = 0; Number of processors in hypercube = 16.

With RC, since pseudo-random numbers are used for generating random initial partitions, different runs on the same sample TIG will generally produce different final partitions. Hence, the best-case, worst-case and average-case speedups are reported for sets of five runs. With the finite-element graphs (samples 1–4), the nearest-neighbor mapping is clearly superior. This has been the case with every finite-element graph sample tested. This is a consequence of the high degree of locality and planar nature of the TIG that permits very good mappings onto a mesh (and hence onto a hypercube). Even though only a mesh-subset of the links of the hypercube are utilized, the planar nature of the TIG suggests that higher degrees of processor connectivity may not be of much use. In the case of random graphs, poor speedups are obtained with both schemes, with the RC mapping being slightly better. Both approaches result in mappings requiring many inter-processor communication messages.

Table 2 presents results for an idealized hypercube system with zero communication start-up cost. In this case, the total communication cost is entirely due to data transmission time. Speedups with both approaches improve as the startup time decreases but the improvement with RC is in all cases significantly greater. In fact, the estimated speedup with RC is better than the NN

Comparison of mapping times for NN & RC.				
No.	Graph Characteristics		Mapping times in sec.	
	$\|V\|$	Description	RC(per run)	NN
1.	505	Sample 1 (mesh)	10.12	5.0
2.	1449	Sample 2 (mesh)	37.42	11.6
3.	602	Sample 3 (non-mesh)	4.95	2.3
4.	256	Sample 4 (non-mesh)	12.78	6.3
5.	400	Sample 5 (random)	13.20	7.3
6.	800	Sample 6 (random)	20.23	12.6

Table 3: Comparison of mapping times (in seconds) for NN and RC on sample graphs on a Pyramid 90x processor running Pyramid OSx/4.0

mapping for all samples when start-up cost is zero. This is because RC explicitly minimizes total communication volume. Whereas NN minimizes total number of messages and achieves reasonably low communication volume, it does not attempt explicit minimization of the volume.

Table 3 reports the execution time of NN and RC mapping schemes on a Pyramid 90x processor running Pyramid OSx/4.0 operating system. It can be seen that NN is faster than each run of RC by a factor of 2 to 3. Howeve, the time taken for mapping by either scheme is small enough that the generation time should not be a critical factor in choosing one or the other of the two approaches. Overall, RC appears to be preferable for random TIG's, independent of communication start-up cost. With finite-element graphs, NN provides better mappings for machines with a relatively high communication start-up cost, while RC is more attractive for machines with low communication start-up cost.

6 Summary

In this paper, the task-to-processor mapping problem was addressed. Since the problem is NP-complete, efficient heuristics to obtain satisfactory sub-optimal solutions in reasonable time are required. A *minimax* cost function was formulated to evaluate the effectiveness of proposed mapping strategies. Two mapping strategies were described – the Nearest-Neighbor (NN) approach and the Recursive-Clustering (RC) approach. The former approach uses a procedure that explicitly attempts to improve load balance and implicitly keeps communication costs low. The latter method on the other hand explicitly attempts to minimize communication costs while guaranteeing load balance implicitly. The effectiveness of the mapping schemes was evaluated by using different sample TIG's. The nearest-neighbor strategy is found to be more effective on hypercube systems with high message start-up times, especially for finite element graphs; the recursive partitioning heuristic is generally better on hypercubes with lower message start-up times and is more effective on random task graphs.

References

[1] C. Aykanat, F. Ozguner, F. Ercal and P. Sadayappan, "Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes," *IEEE Trans. on Computers,* Vol. C-37, No. 12, Dec. 1988, pages 1554-1568.

[2] M.J. Berger and S.H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Trans. on Computers,* Vol. C-36, No. 5, May 1987, pages 570-580.

[3] F. Berman, "Experience with an Automatic Solution to the Mapping Problem," in *Characteristics of Parallel Algorithms,* Editors: L. Jamieson, D. Gannon and R. Douglass, MIT Press Scientific Computation Series, 1987, pages 307-334.

[4] S.H. Bokhari, "On the Mapping Problem," *IEEE Trans. on Computers,* Vol. C-30, No. 3, March 1981, pages 207-214.

[5] C.M. Fiduccia and R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. of the 19th Design Automation Conference,* Jun. 1982, pages 175-181.

[6] J.W. Flower, S.W. Otto and M.C. Salama, "A Preprocessor for Finite Element Problems," in *Proc. Symposium on Parallel Computations and their Impact on Mechanics, ASME Winter Meeting, Dec. 1987;* also as Caltech Concurrent Computation Project (C^3P) Report-292, Jun. 1985.

[7] G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon and D.W. Walker, *Solving Problems on Concurrent Processors: General Techniques and Regular Problems,* Vol. 1, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[8] B. Indurkhya, H.S. Stone and L.Xi-Cheng, "Optimal Partitioning of Randomly Generated Distributed Programs," *IEEE Trans. Software Eng.,* Vol. SE-12, No. 3, Mar. 1986, pages 483-495.

[9] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. Computers,* Vol. C-33, No. 11, Nov. 1984, pages 1023-1029.

[10] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Systems Technical Journal,* Vol. 49, No. 2, 1970, pages 291-308.

[11] V.M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans. Computers,* Vol. C-37, No. 11, Nov. 1988, pages 1384-1397.

[12] R. Morison and S. Otto, "The Scattered Decomposition for Finite Elements," *Journal of Scientific Computing 2,* (1986); also as Caltech Concurrent Computation Project (C^3P) Report-286, May 1985.

[13] P. Sadayappan and F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs Onto Processor Meshes," *IEEE Trans. on Computers,* Vol. C-36, No. 12, Dec. 1987, pages 1408-1424.

[14] K. Schwan and C. Gaimon, "Automating Resource Allocation in the Cm^* Multiprocessor," *Proc. of the 5th International Conference on Distributed Computing Systems,* May 1985, pages 310-320.

[15] K. Schwan, W. Bo, N. Bauman, P. Sadayappan and F. Ercal, "Mapping Parallel Applications on a Hypercube," *Hypercube Multiprocessors 1987,* Ed: M. Heath, SIAM, Philadelphia, 1987, pages 141-151.

[16] C. Shen and W. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minmax Criterion," *IEEE Trans. on Computers,* Vol. C-34, No. 3, Mar. 1985, pages 197-203.

[17] J.B. Sinclair and M. Lu, "Module Assignment in Distributed Systems," *Proc. 1984 Computer Networking Symposium,* Gaithesburg, MD, Dec. 1984, pages 105-111.

[18] J.B. Sinclair, "Efficient Computation of Optimal Assignments for Distributed Tasks," *Journal of Parallel and Distributed Computing,* Vol. 4, No. 4, Aug. 1987, pages 342-362.