Hypergraph Model for Mapping Repeated Sparse Matrix-Vector Product Computations onto Multicomputers *

Ümit V. Çatalyürek

Computer Engineering Department
Bilkent University
Ankara, 06533, Turkey
cumit@cs.bilkent.edu.tr

Cevdet Aykanat

Computer Engineering Department
Bilkent University
Ankara, 06533, Turkey
aykanat@cs.bilkent.edu.tr

Graph model of computation has deficiencies in the mapping of repeated sparse matrix-vector computations to multicomputers. We propose a hypergraph model of computation to avoid these deficiencies. We also propose one-phase Kernighan-Lin based mapping heuristics for the graph and hypergraph models. The proposed mapping heuristics are used for the experimental evaluation of the validity of the proposed hypergraph model on sparse matrices selected from Harwell-Boeing collection and NETLIB suite. The proposed heuristic using the hypergraph model finds drastically better mappings than the one using the graph model on test matrices with unstructured sparsity pattern.

1 Introduction

Iterative solvers are widely used for the solution of large, sparse, linear system of equations on distributed memory architectures (multicomputers). Consider the solution of a linear system of m equations with an $m \times m$ sparse coefficient matrix. Three basic types of operations are repeatedly performed at each iteration. These are linear operations on dense m-vectors (such as scalar-vector product and vector addition), inner product(s) of dense m-vectors, and sparse matrix-vector product of the form $\mathbf{y} = \mathbf{A}\mathbf{x}$, where $\mathbf{y}$ and $\mathbf{x}$ are dense m-vectors, and $\mathbf{A}$ is an $m \times m$ matrix with the sparsity structure of the coefficient matrix [1, 9].

All of these basic operations can be performed concurrently by distributing the rows of the $\mathbf{A}$ matrix and the components of the dense vectors in the same way. This data distribution scheme induces a computational distribution such that each processor is held responsible for updating the values of those vector components assigned to itself. With this data distribution scheme, linear vector operations and inner-product operations can be easily and efficiently parallelized by an even distribution of vector components to processors [1, 9]. Linear vector operations do not necessitate communication, whereas inner-product operations necessitate global communication. However, the global communication overhead due to these inner product computations become negligible with increasing problem size.

Sparse matrix-vector product computations constitute the most time consuming operation in iterative solvers. In the parallel matrix-vector product computation, processors need some nonlocal components of the global $\mathbf{x}$-vector, depending on the sparsity pattern of the local rows. Hence, the row partitioning scheme mentioned earlier necessitates communication just before the local matrix-vector product computations. Each processor send some of its local $\mathbf{x}$-vector components to those processor(s) which need them. After receiving the needed nonlocal $\mathbf{x}$ components, each processor can concurrently compute its local components of the global $\mathbf{y}$-vector by performing a local matrix-vector product. Load balance during concurrent local matrix-vector product computations necessitates a row-partitioning which assigns equal number of nonzero entries to each processor. Hence, by weighting each row by its nonzero entry count, load balancing problem can be considered as the *number partitioning* problem which is known to be NP-hard. However, different row partitionings with good load balance, and different assignments of these row partitions to processors may also significantly differ the communication requirement. Unfor-

*This work is partially supported by the Commission of the European Communities, Directorate General for Industry under contract ITDC 204-82166.

tunately, the communication requirement scales up with increasing problem size. The minimization of the communication overhead while maintaining the computational load balance reduces to the NP-hard *mapping problem* for coefficient matrices with unstructured and irregular sparsity pattern [7].

Simultaneous *single-hop* communications between distinct adjacent pairs of processors can be performed concurrently. However, simultaneous *multi-hop* communications between distant pairs of processors may introduce congestion to the interconnection network, thus increasing the communication overhead. Multi-hop communications between distant processors are usually routed over the shortest paths of links between the communicating pairs of processors. Hence, multi-hop messages are usually weighted with the distances between the respective pairs of processors in the network, while considering their contribution to the overall communication cost [2, 3, 10]. Here, distance refers to the number of communication links and switching elements along the communication route in *static* and *dynamic* interconnection networks, respectively.

The mapping methods proposed in the literature employ graph model of computation [2, 3, 10]. In this work, we show the deficiencies of the graph model for mapping sparse matrix-vector product computations, and propose a hypergraph model which avoids these deficiencies. Kernighan-Lin [8] (KL) based heuristics are fast heuristics widely used for solving the mapping problem [2, 10]. However, KL-based heuristics proposed in the literature solve the mapping problem in two-phase. In these approaches, a *clustering* phase is followed by a *one-to-one mapping* phase. In this work, we also propose one-phase (*many-to-one mapping*) *KL*-based mapping heuristics for the graph and hypergraph models. The proposed *KL*-based mapping heuristics are used for the experimental evaluation of the validity of proposed hypergraph model on symmetric sparse matrices selected from *Harwell-Boeing* collection [4] and *NETLIB* suite [6].

2 Graph Model of Computation

A symmetric sparse matrix $\mathbf{A}$ can be represented as an undirected graph $G_A(V, E)$. The vertices in the vertex set V correspond to the rows/columns of the $\mathbf{A}$ matrix. In the edge set E, $(v_i, v_j) \in E$ if and only if a_{ij} and a_{ji} of the $\mathbf{A}$ matrix are nonzeros. Hence, the vertices in the adjacency list of a vertex v_i denote the column (row) indices of the off-diagonal nonzeros in row i (column i) of $\mathbf{A}$. Each nonzero entry of $\mathbf{A}$ incurs a multiply/add operation in the matrix-vector product computation. Hence, $w_i = d_i + 1$ denotes the computational load of mapping row i to a processor, where d_i denotes the degree of vertex v_i.

In the graph model, the mapping problem is to find a many-to-one mapping function M which assigns each vertex of the graph G_A to a unique processor of the multicomputer, and minimizes the total communication cost

$$CC = \sum_{(v_i,v_j)\in E, M(i)\neq M(j)} 2 \times D_{M(i),M(j)} \qquad (1)$$

while maintaining the computational load balance. The computational load W_p of a processor p is the summation of the weights of the tasks assigned to that processor. That is, $W_p = \sum_{v_i \in V, M(i)=p} w_i$ for $p = 1, 2, \ldots, P$, where, $M(i) = p$ denotes the label of the processor that row i is mapped to. In (1), D_{pq} denotes the distance between the processors p and q in the interconnection network. An edge $(v_i, v_j) \in E$ is said to be *cut* if vertices v_i and v_j are mapped to two different processors, i.e., $M(i) \neq M(j)$, otherwise *uncut*. Only cut edges incur communication. The amount of contribution of a cut edge (v_i, v_j) is equal to twice the distance between processors $M(i)$ and $M(j)$. The factor 2 appears since the local x_i and x_j values should be exchanged between processors $M(i)$ and $M(j)$, respectively.

Figure 1(a), which illustrates different cases of adjacent vertex mappings to a 2×3 mesh, is given in order to reveal the deficiencies of the graph model. Mapping adjacent vertices v_1 and v_2 to the same processor P_1 does not incur any communication since (v_1,v_2) is an uncut edge. Mapping adjacent vertices v_{10} and v_3 to different processors P_1 and P_6, respectively, incurs a communication cost of 6 since (v_{10}, v_3) is a cut edge and $D_{1,6}$=3. Similarly, mapping adjacent vertices v_6 and v_7, and v_8 and v_9 to adjacent processors P_4 and P_5, respectively, will incur a total communication cost of 4 since both edges (v_6,v_7) and (v_8,v_9) are cut edges and $D_{4,5}$=1. Hence, the graph model correctly represents these cases. However, consider vertices v_4 and v_5 in P_2 which are adjacent to vertex v_2 in P_1. This case will incur a communication cost of 4 in the graph model, since both edges (v_2,v_4) and (v_2,v_5) are cut edges and $D_{1,2} = 1$. However, it is obvious that processor P_1 should send x_2 only once to processor P_2, whereas P_2 should send both x_4 and x_5 to P_1. Hence, the actual contribution to the communication cost should be 3 instead of 4. That is, graph model does not differentiate between the cut-edge pairs $\{(v_2, v_4), (v_2, v_5)\}$ and $\{(v_6, v_7), (v_8, v_9)\}$.

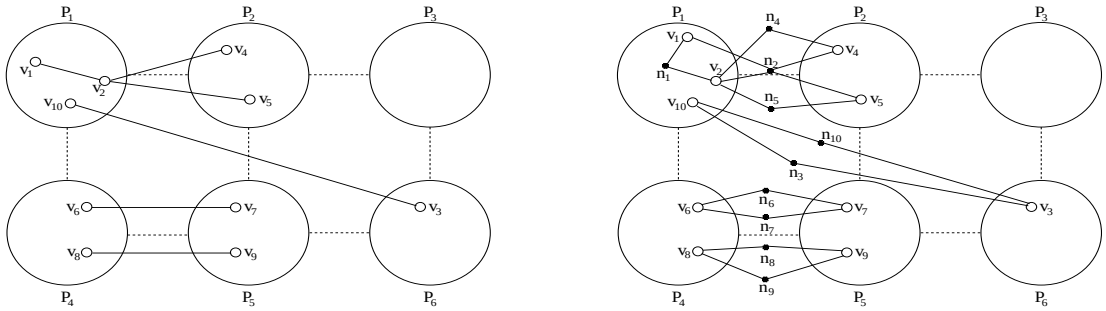

Figure 1: Different cases for mapping adjacent vertex pairs to a 2x3 mesh in (a) graph, (b) hypergraph models

3 Hypergraph Model of Computation

In this work, we exploit hypergraphs for a more accurate modeling of parallel sparse matrix-vector product computations. A hypergraph $H(V, N)$ consists of a finite non-empty set V of vertices and a finite non-empty set $N \subseteq 2^V$ of hyperedges (*nets*), where 2^V is power set of vertex set V. Each net n_i in N is a subset of V. Vertices in a net n_j are called its *pins* and denoted as $pins(n_j)$. The set of nets connected to a vertex v_i is denoted as $nets(v_i)$

In the proposed model, matrix $\mathbf{A}$ is represented with a hypergraph $H_A(V, N)$. There is one vertex v_i and one net n_j in V and N for each row i and column j of $\mathbf{A}$, respectively. Net n_j contains the vertices corresponding to the rows which have a nonzero entry on column j. Formally, $v_i \in n_j$ if and only if $a_{ij} \neq 0$. The following relation exists between graph (G_A) and hypergraph (H_A) models of a symmetric $m \times m$ matrix $\mathbf{A}$. The vertex sets are the same, and the net set of H_A is $N = \{n_i : n_i = adj_{G_A}(v_i) \cup \{v_i\}$ for $i = 1, 2, \ldots, m\}$.

In the hypergraph model, a net that has at least one pin (vertex) in a processor is said to *connect* that processor. The set of processors that a net n_i connects is denoted by its *connectivity set* $\mathcal{C}(i)$. A net that connects more than one processor is said to be *cut*, otherwise *uncut*. Set of cut nets are called *external nets* (N_E). Thus, total interprocessor communication cost can be formulated as

$$CC = \sum_{n_i \in N_E} \sum_{p \in \mathcal{C}(i) \ni p \neq M(i)} D_{M(i),p} \qquad (2)$$

Only cut nets contribute to the communication cost. A cut net n_i indicates that processor $M(i)$ should send its local x_i to those processor in the connectivity set of net n_i except itself. Therefore, amount of communication contribution of a cut net n_i is the sum of the distances between the source processor $M(i)$ and the processors in the set $\mathcal{C}(i) - \{M(i)\}$.

Figure 1(b) displays the partial hypergraph representation of a symmetric matrix whose partial graph representation is given in Figure 1(a). Net n_1 has no contribution to the communication cost since it is an uncut net. Consider the vertices v_4 and v_5 in processor P_2 which are adjacent to vertex v_2 in processor P_1. Respective nets n_4, n_5 and n_2 are all cut nets, and hence they contribute $D_{2,1}$=1, $D_{2,1}$=1 and $D_{1,2}$=1, respectively, to the communication cost. Hence, their total contribution will be 3 thus correctly modeling the actual communication requirement.

4 One-Phase KL-Based Heuristics

KL algorithm is an iterative improvement heuristic originally proposed for 2-way graph partitioning (bipartitioning) [8]. This algorithm became the basis for most of the subsequent partitioning algorithms, all of which we call the KL-based algorithms. KL algorithm performs a number of passes until it finds a locally minimum partition. Each pass consists of a sequence of vertex swaps. Fiduccia-Mattheyses (FM) [5] introduced a faster implementation of KL algorithm for hypergraph partitioning. They proposed vertex move concept instead of vertex swap. This modification as well as proper data structures, e.g., bucket lists, reduced the time complexity of a single pass of KL algorithm to linear in the size of the graph. Here, *size* refers to the number of edges and pins in a graph and hypergraph, respectively. Sanchis [11] proposed a multiway hypergraph partitioning algorithm which directly handles the partitioning of a hypergraph into more than two parts. Note that all the previous approaches before Sanchis' algorithm (SN algorithm) are originally bipartitioning algorithms.

As mentioned earlier, all KL-based mapping heuristics proposed in the literature employ two-phase approach. Here, we propose one-phase KL-based mapping heuristics for the graph and hypergraph models. The proposed algorithms adopt the nice features of FM and SN algorithms such as bucket lists,

```
1     construct a random, initial, feasible mapping;
2     repeat
2.1     unlock all vertices;
2.2     compute P − 1 move gains of each vertex v ∈ V
          by invoking Gcompute(G, v)/Hcompute(H, v)
          for graph/hypergraph model;
2.3     mcnt = 0;
2.4     while there exists a feasible move of
          an unlocked vertex do
2.4.1     select a feasible move with max gain g_max of
            an unlocked vertex v
            from processor s to processor t;
2.4.2     mcnt = mcnt + 1;
2.4.3     G[mcnt] = g_max;
2.4.4     Moves[mcnt] = {v, s, t};
2.4.5     tentatively realize the move of vertex v;
2.4.6     lock vertex v;
2.4.7     recompute the move gains of unlocked vertices
            in Adj(v)/pins(nets(v)) by invoking
            Gcompute(G, v)/Hcompute(H, v)
            for graph/hypergraph model;
2.5     perform prefix sum on the array G[1 ... mcnt];
2.6     select 1 ≤ i* ≤ mcnt with max gain G_max = G[i*];
2.7     if G_max > 0 then
2.7.1     permanently realize the moves in Moves[1 ... i*]
      until G_max ≤ 0
```

Figure 2: One-phase KL-based mapping heuristic

vertex move concept, multiple $(P-1)$ move directions, and operates on *feasible* mappings. Here, P denotes the number of processors. A mapping is said to be feasible if it satisfies the load balance criterion $W_{avg}(1-\varepsilon) \leq W_p \leq W_{avg}(1+\varepsilon)$, for each processor $p=1,2,\ldots P$. Here, $W_{avg}=(\sum_{i=1}^{n} w_i)/P$ denotes the computational load of each processor under perfect load balance condition, and ε represents the predetermined maximum load imbalance ratio allowed. Each vertex is associated with $(P-1)$ possible moves. Each move is associated with a *gain*. The *move gain* of a vertex v_i in processor s with respect to processor t $(t \neq s)$, i.e., the gain of the move of v_i from the home (source) processor s to the destination processor t, denotes the amount of decrease in the overall communication cost to be obtained by making that move. Positive gain refers to a decrease, whereas negative gain refers to an increase in the communication cost.

Figure 2 illustrates the proposed KL-based mapping heuristic. The algorithm starts from a random feasible mapping (Step 1), and iterates a number of passes over the vertices of the graph/hypergraph until a locally optimum mapping is found (*repeat-loop* at Step 2). Figures 3 and 4 illustrate the move gain computation algorithms for the graph and hypergraph models, respectively. In Figure 4, $\delta_j(s)$ denotes the number of pins of the net n_j that lie in processor s, i.e., $\delta_j(s) = |\{v_i \in n_j : M(i) = s\}|$.

```
Gcompute(G, v_i)
1       s ← M(i);
2       for each processor t ≠ s do
2.1         g_i(t) ← 0;
3       for each v_j ∈ Adj(v_i) do
3.1         q ← M(j);
3.2         for each processor t ≠ s do
3.2.1           g_i(t) ← g_i(t) + 2 × (D_sq − D_qt);
```

Figure 3: Gain computation for Graph Model

```
Hcompute(H, v_i)
1       s ← M(i);
2       for each processor t ≠ s do
2.1         g_i(t) ← 0;
3       for each n_j ∈ nets(v_i) do
3.1         if n_j ≠ n_i then
3.1.1           q ← M(j);
3.1.2           if δ_j(s) = 1 then
3.1.2.1             for each processor t ≠ s do
3.1.2.1.1               g_i(t) ← g_i(t) + D_qs;
3.1.3           for each processor t ≠ s do
3.1.3.1             if δ_j(t) = 0 then
3.1.3.1.1               g_i(t) ← g_i(t) − D_qt;
3.2         else
3.2.1           for each processor t ≠ s do
3.2.1.1             for each processor q ∈ C(i)
                      and q ≠ s do
3.2.1.1.1               g_i(t) ← g_i(t) + D_sq − D_qt;
3.2.1.2             if δ_i(s) > 1 then
3.2.1.2.1               g_i(t) ← g_i(t) − D_st;
```

Figure 4: Gain computation for Hypergraph Model

At the beginning of each pass, all vertices are *unlocked* (Step 2.1), and initial $P-1$ move gains for each vertex are computed (Step 2.2). At each iteration (*while-loop* at Step 2.4) in a pass, a feasible move with the maximum *gain* is selected, tentatively performed, and the vertex associated with the move is *locked* (Steps 2.4.1–2.4.6). The locking mechanism enforces each vertex to be moved at most *once* per pass. That is, a locked vertex is not selected any more for a move until the end of the pass. After the move, the move gains affected by the selected move are updated so that they indicate the effect of the move correctly. In the graph model, move gains of only those unlocked vertices which are adjacent to the vertex moved should be updated. In the hypergraph model, move gains of only those unlocked vertices which share nets with the vertex moved should be updated. In the current implementation, these updates are performed by recomputing the move gains of those local vertices for the sake of simplicity (Step 2.4.7).

At the end of the pass, we have a sequence of tentative vertex moves and their respective gains. We then construct from this sequence the *maximum prefix subsequence* of moves with the *maximum prefix sum* (Steps 2.5 and 2.6). That is, the gains of the moves in the maximum prefix subsequence give the

Table 1: Communication cost averages and standard deviations (σ), and execution time averages (in seconds). $|V|$ is the number of vertices, $|E|$ is the number of edges and d_{avg} is the average vertex degree in the graph model.

mapping problem					KL-G (graph model)			KL-H (hypergraph model)		
sparse matrix				2D	comm. cost		exec.	comm. cost		exec.
name	$\|V\|$	$\|E\|$	d_{avg}	mesh	avg.	σ	time	avg.	σ	time
BCSPWR06	1454	1923	2.65	2x2	443	36	0.8	**246** (0.56)	67	4.0 (4.89)
				2x4	887	79	2.4	**509** (0.57)	85	8.0 (3.35)
				4x4	1489	78	9.2	**841** (0.56)	104	21.6 (2.36)
				4x8	2614	181	45.0	**1627** (0.62)	191	70.1 (1.56)
BCSPWR07	1612	2106	2.61	2x2	466	46	0.9	**268** (0.57)	64	4.2 (4.55)
				2x4	1005	74	2.5	**541** (0.54)	79	8.7 (3.48)
				4x4	1626	99	10.9	**971** (0.60)	117	25.7 (2.36)
				4x8	2912	245	43.6	**1907** (0.65)	160	80.3 (1.84)
BCSPWR08	1624	2213	2.73	2x2	505	41	0.9	**271** (0.54)	64	4.5 (4.88)
				2x4	1045	84	2.8	**565** (0.54)	90	10.2 (3.71)
				4x4	1708	132	11.4	**996** (0.58)	108	25.2 (2.21)
				4x8	3143	265	47.7	**2091** (0.67)	181	78.8 (1.65)
BCSPWR09	1723	2394	2.78	2x2	555	54	1.1	**332** (0.60)	75	5.1 (4.55)
				2x4	1140	103	2.9	**663** (0.58)	103	10.4 (3.62)
				4x4	1811	137	11.2	**1138** (0.63)	143	27.2 (2.43)
				4x8	3269	231	58.6	**2211** (0.68)	217	94.4 (1.61)
BCSPWR10	5300	8271	3.12	2x2	1904	102	6.1	**1026** (0.54)	195	23.1 (3.78)
				2x4	3641	312	15.4	**2003** (0.55)	337	47.3 (3.06)
				4x4	5761	483	62.0	**3433** (0.60)	196	117.1 (1.89)
				4x8	10236	982	289.1	**6325** (0.62)	568	410.1 (1.42)
NESM	662	4116	12.4	2x2	800	221	1.6	**458** (0.57)	63	8.5 (5.32)
				2x4	1833	404	3.0	**1122** (0.61)	172	18.8 (6.33)
				4x4	3375	322	8.6	**2537** (0.75)	336	43.7 (5.10)
				4x8	5935	863	33.3	**5185** (0.87)	533	119.9 (3.60)
80BAU3B	2262	10074	8.91	2x2	2170	196	4.5	**1550** (0.71)	211	44.0 (9.73)
				2x4	4313	379	11.1	**3230** (0.75)	341	99.3 (8.95)
				4x4	7142	486	34.3	**5844** (0.82)	318	206.6 (6.03)
				4x8	13530	1083	125.8	**11065** (0.82)	773	524.3 (4.17)
JAGMESH9	1349	3876	5.75	2x2	**154**	38	1.4	186 (1.21)	52	4.7 (3.46)
				2x4	512	127	4.1	**503** (0.98)	115	10.6 (2.61)
				4x4	995	136	12.8	**988** (0.99)	137	30.0 (2.34)
				4x8	2252	499	51.2	**2091** (0.93)	315	72.7 (1.42)
				8x8	4430	717	179.5	**3838** (0.86)	408	205.5 (1.15)
LSHP2614	2614	7683	5.88	2x2	**281**	29	2.9	310 (1.10)	58	10.8 (3.66)
				2x4	**685**	144	8.6	908 (1.33)	232	26.5 (3.07)
				4x4	**1553**	346	30.2	1730 (1.11)	305	60.1 (1.99)
				4x8	4091	1345	124.9	**3678** (0.90)	507	179.6 (1.44)
				8x8	7902	2314	413.0	**6166** (0.78)	773	471.5 (1.14)

maximum decrease in the communication cost among all prefix subsequences of the moves tentatively performed. Then, we permanently realize the moves in the maximum prefix subsequence and start the next pass if the maximum prefix sum is positive. The mapping process terminates if the maximum prefix sum is not positive, i.e., no further decrease in the communication cost is possible, and we then have found a locally optimum mapping. Note that moves with negative gains, i.e., moves which increase the communication cost, might be selected during the iterations in a pass. These moves are tentatively realized in the hope that they will lead to moves with positive gains in the future iterations. This feature together with the maximum prefix subsequence selection brings the *hill-climbing* capability to the KL-based algorithms.

5 Experimental Results

The proposed one-phase *KL*-based mapping heuristics are used for the experimental evaluation of the validity of the proposed hypergraph model on symmetric sparse matrices selected from *Harwell-Boeing* collection [4] and linear programming problems in *NETLIB* suite [6]. Table 1 illustrates the performance results for the mapping of the selected sparse matrices. *BCSPWR06–10* matrices come from the sparse matrix representation of various power networks. *JAGMESH9* and *LSHP2614* matrices come from the finite element discretizations of pinched hole and L-shaped regions, respectively. The sparsity patterns of *NESM* and *80BAU3B* are obtained from the *NETLIB* suite by multiplying the respective constraint matrices with their transposes. Power and *NETLIB* matrices have unstructured sparsity pattern. Finite element matrices have structured but irregular sparsity pattern.

The graph and hypergraph representations of these matrices are mapped onto 2×2, 2×4, 4×4 and 4×8 $2D$-meshes by running the proposed KL-based heuristics on a SunSparc 10. The maximum load imbalance ratio is selected as $\varepsilon = 0.1$. We will refer to the KL-based heuristics using the graph and hyper-

graph models as *KL-G* and *KL-H*, respectively. Both KL-G and KL-H heuristics are executed 20 times for each mapping instance starting from different, random initial mappings. Communication cost averages and their standard deviations (σ), and execution time averages (in seconds) are displayed in Table 1. The communication cost averages displayed in Table 1 correspond to the averages of the communication costs computed according to (2). The values in parentheses in the "comm. cost" column of KL-H, give the ratio of the average communication costs of the mappings found by KL-H to those of KL-G. A bold value in a row correspond to the best value for the respective mapping instance. Values in parentheses in the "exec. time" column of KL-H, give the ratio of the average execution times of KL-H to those of KL-G.

As seen in Table 1, KL-H always finds drastically better mappings than KL-G on test matrices with unstructured sparsity pattern (i.e., power and NETLIB matrices). However, KL-H cannot perform better than KL-G on the mappings of structured matrices (i.e., finite element matrices) to small size meshes. This experimental finding can be attributed to the following reason. KL-H algorithm encounters large number of zero move gains during the mapping of these matrices to small size meshes. KL-H algorithm randomly resolves these ties. However, on such cases, KL-G algorithm tends to gather the adjacent vertices to near processors although they do not decrease the communication requirement at that point in time. Various tie braking strategies might be considered to improve the performance of the proposed KL-H algorithm on such mapping instances. However, as is also seen in Table 1, the relative performance of the KL-H algorithm increases with the increasing mesh size for these structured matrices. The increase in the number of processors is expected to decrease the number of such ties. As seen in Table 1, KL-H performs better than KL-G algorithm on the mappings of JAGMESH9 and LSHP2614 to $P \geq 8$ and $P \geq 32$ processors, respectively. Note that only these structured matrices are mapped to 8×8 mesh just to show this tendency. As is seen in Table 1, KL-H is somewhat slower than KL-G algorithm because of the more expensive local gain update computations. However, relative speed difference decreases with increasing number of processors.

6 Conclusion

We have proposed a hypergraph model for mapping repeated sparse matrix-vector computations to multicomputers. The proposed hypergraph model avoids the deficiencies of the conventional graph model. We have also proposed one-phase Kernighan-Lin based mapping heuristics for the graph and hypergraph models. We have used these heuristics to evaluate the validity of proposed hypergraph model on sparse matrices selected from *Harwell-Boeing* collection and *NETLIB* suite. Experimental results justify the validity of the proposed hypergraph model.

References

[1] Aykanat, C., Özgüner, F., Ercal, F., and Sadayappan, P., "Iterative algorithms for solution of large sparse systems of linear equations on hypercubes," *IEEE Trans. on Computers*, vol. 37, pp. 1554–1567, 1988.

[2] Bultan, T., and Aykanat, C., "A new mapping heuristic based on mean field annealing," *Journal of Parallel and Distributed Computing*, vol. 10, pp. 292–305, 1992.

[3] Camp, W. J., Plimpton, S. J., Hendrickson, B. A., and Leland, R. W., "Massively parallel methods for Engineering and Science problem," *Communication of ACM*, vol. 37, no. 4, pp. 31–41, April 1994.

[4] Duff, I.S., and Grius, R. G., "Sparse matrix test problems," *ACM Trans. on Mathematical software*, vol. 17, no. 1, pp. 1–14, March 1989.

[5] Fiduccia, C. M., and Mattheyses, R. M., "A linear heuristic for improving network partitions," *Proc. Design Automat. Conf.*, pp. 175–181, 1982.

[6] Gay, D. M., "Electronic mail distribution of linear programming test problems" *Mathematical Programming Society COAL Newsletter*, 1985.

[7] Indurkhya, B,, Stone, H. S. and Xi-Cheng, L., "Optimal partitioning of randomly generated distributed programs," *IEEE Trans. Software Engineering*, vol. 12(3), pp. 483–495, 1986.

[8] Kernighan, B. W., and Lin, S., "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, pp. 291–307, 1970.

[9] Pommerell, C., Annaratone, M., and Fichtner, W., "A Set of New Mapping and Coloring Heuristics for Distributed Memory Parallel Architectures". *SIAM J. on Scientific and Statistical Computing*. vol. 13(1), pp. 194–226, 1992.

[10] Sadayappan, P., Ercal, F., and Ramanujam, J., "Cluster partitioning approaches to mapping parallel programs onto hypercube," *Parallel Computing*, vol. 13, pp. 1–16, 1990.

[11] Sanchis, L. A., "Multiple-way network partitioning," *IEEE Trans. on Computers*, vol. 38(1), pp. 62–81, 1989.