Appears in the Proceedings of the International Parallel Processing Symposium 1996

Parallel Multilevel Graph Partitioning *

George Karypis and Vipin Kumar
University of Minnesota, Department of Computer Science, Minneapolis, MN 55455

Abstract

In this paper we present a parallel formulation of a graph partitioning and sparse matrix ordering algorithm that is based on a multilevel algorithm we developed recently. Our parallel algorithm achieves a speedup of up to 56 on a 128-processor Cray T3D for moderate size problems, further reducing its already moderate serial run time. Graphs with over 200,000 vertices can be partitioned in 128 parts, on a 128-processor Cray T3D in less than 3 seconds. This is at least an order of magnitude better than any previously reported run times on 128-processors for obtaining an 128-partition. This also makes it possible to use our parallel graph partitioning algorithm to partition meshes dynamically in adaptive computations. Furthermore, the quality of the produced partitions and orderings are comparable to those produced by the serial multilevel algorithm that has been shown to substantially outperform both spectral partitioning and multiple minimum degree.

1 Introduction

Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing, VLSI design, task scheduling, geographical information systems, and operations research. The problem is to partition the vertices of a graph in p roughly equal parts, such that the number of edges connecting vertices in different parts is minimized. The efficient implementation of many parallel algorithms usually requires the solution to a graph partitioning problem, where vertices represent computational tasks, and edges represent data exchanges. A p-way partition of the computation graph can be used to assign tasks to p processors. Because the partition assigns equal number of computational tasks to each processor the work is balanced among p processors, and because it minimizes the edge-cut, the communication overhead is also minimized. For example, the solution of a sparse system of linear equations $Ax = b$ via iterative methods on a parallel computer gives rise to a graph partitioning problem. A key step in each iteration of these methods is the multiplication of a sparse matrix and a (dense) vector. Partitioning the graph that corresponds to matrix A, is used to significantly reduce the amount of communication [18]. If parallel direct methods are used to solve a sparse system of equations, then a graph partitioning algorithm can be used to compute a fill reducing ordering that lead to high degree of concurrency in the factorization phase [18, 6].

The graph partitioning problem is NP-complete. However, many algorithms have been developed that find a reasonably good partition. Spectral partitioning methods [21, 11] provide good quality graph partitions, but have very high computational complexity. Geometric partition methods [9, 20] are quite fast but they often provide worse partitions than those of more expensive methods such as spectral. Furthermore, geometric methods are applicable only if coordinate information for the graph is available. Recently, a number of researches have investigated a class of algorithms that are based on multilevel graph partitioning that have moderate computational complexity [4, 11, 14, 15, 13]. Some of these multilevel schemes [4, 11, 14, 15, 13] provide excellent (even better than spectral) graph partitions. Even though these multilevel algorithms are quite fast compared with spectral methods, performing a multilevel partitioning in parallel is desirable for many reasons including adaptive grid computations, computing fill reducing orderings for parallel direct factorizations, and taking advantage the aggregate amount of memory available on parallel computers.

Significant amount of work has been done in developing parallel algorithms for partitioning unstructured graphs and for producing fill reducing orderings for sparse matrices [2, 5, 8, 7, 12]. Only moderately good speedups have been obtained for parallel formulation of graph partitioning algorithms that use geometric methods [9, 5] despite the fact that geometric partitioning algorithms are inherently easier to parallelize. All parallel formulations presented so far for spectral partitioning have reported fairly small speedups [2, 1, 12] unless the graph has been distributed to the processors so that certain degree of data locality is achieved [1].

In this paper we present a parallel formulation of a graph partitioning and sparse matrix ordering algorithm that is based on a multilevel algorithm we developed recently [14]. A key feature of our parallel formulation (that distinguishes it from other proposed parallel formulations of multilevel algorithms [2, 1, 22]) is that it partitions the vertices of the graph into $\sqrt{p}$ parts while distributing the overall adjacency matrix of the graph among all p processors. As

*This work was supported by NSF: CCR-9423082 and by the Army Research Office contract DA/DAAH04-95-1-0538, and by Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by Minnesota Supercomputer Institute, Cray Research Inc, and by the Pittsburgh Supercomputing Center. Related papers are available via WWW at URL: **http://www.cs.umn.edu/users/kumar/papers.html**

shown in [16], this mapping is usually much better than one-dimensional distribution, when no partitioning information about the graph is known. Our parallel algorithm achieves a speedup of up to 56 on 128 processors for moderate size problems, further reducing the already moderate serial run time of multilevel schemes. Furthermore, the quality of the produced partitions and orderings are comparable to those produced by the serial multilevel algorithm that has been shown to outperform both spectral partitioning and multiple minimum degree [14]. The parallel formulation in this paper is described in the context of the serial multilevel graph partitioning algorithm presented in [14]. However, nearly all of the discussion in this paper is applicable to other multilevel graph partitioning algorithms [4, 11, 15].

2 Multilevel Graph Partitioning

The ***p-way*** graph partitioning problem is defined as follows: Given a graph $G = (V, E)$ with $|V| = n$, partition V into p subsets, $V_1, V_2, \ldots, V_p$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = n/p$, and $\bigcup_i V_i = V$, and the number of edges of E whose incident vertices belong to different subsets is minimized. A p-way partition of V is commonly represented by a partition vector P of length n, such that for every vertex $v \in V$, $P[v]$ is an integer between 1 and p, indicating the partition at which vertex v belongs. Given a partition P, the number of edges whose incident vertices belong to different subsets is called the ***edge-cut*** of the partition.

The p-way partition problem is most frequently solved by recursive bisection. That is, we first obtain a 2-way partition of V, and then we further subdivide each part using 2-way partitions. After $\log p$ phases, graph G is partitioned into p parts. Thus, the problem of performing a p-way partition is reduced to that of performing a sequence of 2-way partitions or bisections. Even though this scheme does not necessarily lead to optimal partition [15], it is used extensively due to its simplicity [6].

The basic idea behind the multilevel graph bisection algorithm is very simple. The graph G is first coarsened down to a few hundred vertices, a bisection of this much smaller graph is computed, and then this partition is projected back towards the original graph (finer graph), by periodically refining the partition. Since the finer graph has more degrees of freedom, such refinements usually decrease the edge-cut. This process, is graphically illustrated in Figure 1. The reader should refer to [14] for further details.

3 Parallel Multilevel Graph Partitioning Algorithm

There are two types of parallelism that can be exploited in the p-way graph partitioning algorithm based on the multilevel bisection algorithms. The first type of parallelism is due to the recursive nature of the algorithm. Initially a single processor finds a bisection of the original graph. Then, two processors find bisections of the two subgraphs just created and so on. However, this scheme by itself can use only up to $\log p$ processors, and reduces the overall run time of the algorithm only by a factor of $O(\log p)$. We will refer to this type of parallelism as the parallelism associated with the ***recursive step***.

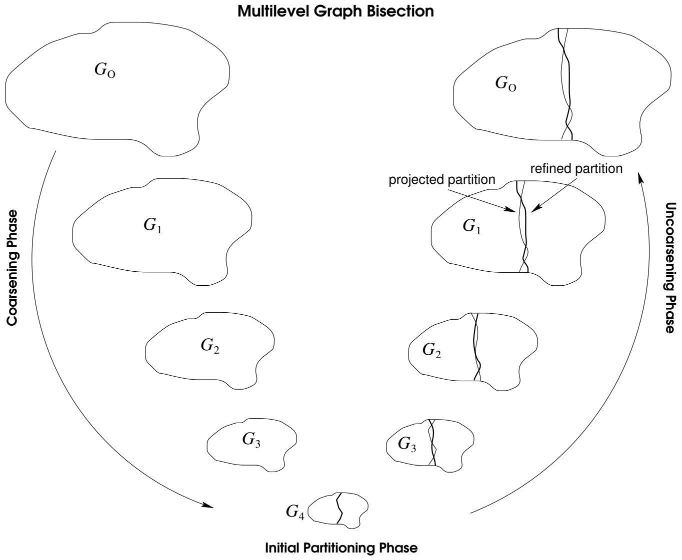

Figure 1: The various phases of the multilevel graph bisection. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a bisection of the smaller graph is computed; and during the uncoarsening phase, the bisection is successively refined as it is projected to the larger graphs. During the uncoarsening phase the light lines indicate projected partitions, and dark lines indicate partitions that were produced after refinement.

The second type of parallelism that can be exploited is during the ***bisection step***. In this case, instead of performing the bisection of the graph on a single processor, we perform it in parallel. We will refer to this type of parallelism as the parallelism associated with the bisection step. By parallelizing the divide step, the speedup obtained by the parallel graph partitioning algorithm is not bounded by $O(\log p)$, and can be significantly higher than that.

The parallel graph partitioning algorithm we describe in this section exploits both of these types of parallelism. Initially all the processors cooperate to bisect the original graph G, into G_0 and G_1. Then, half of the processors bisect G_0, while the other half of the processors bisect G_1. This step creates four subgraphs G_{00}, G_{01}, G_{10}, and G_{11}. Then each quarter of the processors bisect one of these subgraphs and so on. After $\log p$ steps, the graph G has been partitioned into p parts.

In the next three sections we describe how we have parallelized the three phases of the multilevel bisection algorithm.

3.1 Coarsening Phase

During the coarsening phase, a sequence of coarser graphs is constructed. A coarser graph $G_{l+1} = (V_{l+1}, E_{l+1})$ is constructed from the finer graph $G_l = (V_l, E_l)$ by finding a maximal matching M_l and contracting the vertices and edges of G_l to form G_{l+1}. This is the most time consuming phase of the three phases; hence, it needs be parallelized effectively. Furthermore, the amount of communication required during the contraction of G_l to form G_{l+1} depends on how the matching is computed.

On a serial computer, computing a maximal matching can be done very efficiently using randomized algorithms. However, computing a maximal matching in parallel, and particularly on a distributed memory parallel computer, is hard. A direct parallelization of the serial randomized algorithms or algorithms based on depth first graph traversals require significant amount of communication. Communication overhead can be reduced if the graph is initially partitioned among processors in such a way so that the number of edges going across processor boundaries are small. But this requires solving the p-way graph partitioning problem, that we are trying to solve in the first place.

Another way of computing a maximal matching is to divide the n vertices among p processors and then compute matchings between the vertices locally assigned within each processor. The advantages of this approach is that no communication is required to compute the matching, and since each pair of vertices that gets matched belongs to the same processor, no communication is required to move adjacency lists between processors. However, this approach causes problems because each processor has very few nodes to match from. Also, even though there is no need to exchange adjacency lists among processors, each processor needs to know matching information about all the vertices that its local vertices are connected to in order to properly form the contracted graph. As a result significant amount of communication is required. In fact this computation is very similar in nature to the multiplication of a randomly sparse matrix (corresponding to the graph) with a vector (corresponding to the matching vector).

In our parallel coarsening algorithm, we retain the advantages of the previous scheme, but minimize its drawbacks by computing the matchings between groups of $n/\sqrt{p}$ vertices. This increases the size of the computed matchings, and also, as discussed in [16], the communication overhead for constructing the coarse graph is decreased. Specifically, our parallel coarsening algorithm treats the p processors as a two-dimensional array of $\sqrt{p} \times \sqrt{p}$ processors (assume that $p = 2^{2r}$). The vertices of the graph $G_0 = (V_0, E_0)$ are distributed among this processor grid using a cyclic mapping [18]. The vertices V_0 are partitioned into $\sqrt{p}$ subsets, $V_0^0, V_0^1, \ldots, V_0^{\sqrt{p}-1}$. Processor $P_{i,j}$ stores the edges of E_0 between the subsets of vertices V_0^i and V_0^j. Having distributed the data in this fashion, the algorithm then proceeds to find a matching. This matching is computed by the processors along the diagonal of the processor-grid. In particular, each processor $P_{i,i}$ finds a heavy-edge matching M_0^i using the set of edges it stores locally. The union of these $\sqrt{p}$ matchings is taken as the overall matching M_0. Since the vertices are split into $\sqrt{p}$ parts, this scheme finds larger matchings than the one that partitions vertices into p parts.

The coarsening algorithm continues until the number of vertices between successive coarser graphs does not substantially decrease. Assume that this happens after k coarsening levels. At this point, graph $G_k = (V_k, E_k)$ is folded into the lower quadrant of the processor subgrid. The coarsening algorithm then continues by creating coarser graphs. Since the subgraph of the diagonal processors of this smaller processor grid contains more vertices and edges, larger matchings can be found and thus the size of the graph is reduced further. This process of coarsening followed by folding continues until the entire coarse graph has been folded down to a single processor, at which point the sequential coarsening algorithm is employed to coarsen the graph.

Since, between successive coarsening levels, the size of the graph decreases, the coarsening scheme just described utilizes more processors during the coarsening levels in which the graphs are large and fewer processors for the smaller graphs. As our analysis in [16] shows, decreasing the size of the processor grid does not affect the overall performance of the algorithm as long as the graph size shrinks by a certain factor between successive graph foldings.

3.2 Initial Partitioning Phase

At the end of the coarsening phase, the coarsest graph resides on a single processor. We use the Greedy Graph Growing (GGGP) algorithm described [14] to partition the coarsest graph. We perform a small number of GGGP runs starting from different random vertices and the one with the smaller edge-cut is selected as the partition. Instead of having a single processor performing these different runs, the coarsest graph can be replicated to all (or a subset of) processors, and each of these processors can perform its own GGGP partition. We did not implement it, since the run time of the initial partition phase is only a very small fraction of the run time of the overall algorithm.

3.3 Uncoarsening Phase

During the uncoarsening phase, the partition of the coarsest graph G_m is projected back to the original graph by going through the intermediate graphs $G_{m-1}, G_{m-2}, \cdots, G_1$. After each step of projection, the resulting partition is further refined by using vertex swap heuristics (based on Kernighan-Lin [17]) that decrease the edge-cut [14].

For refining the coarser graphs that reside on a single processor, we use the boundary Kernighan-Lin refinement algorithm (BKLR) described in [14]. However, the BKLR algorithm is sequential in nature and it cannot be used in its current form to efficiently refine a partition when the graph is distributed among a grid of processors [8]. In this case we use a different algorithm that tries to approximate the BKLR algorithm but is more amenable to parallel computations. The key idea behind our parallel refinement algorithm is to select a group of vertices to swap from one part to the other instead of selecting a single vertex. Refinement schemes that use similar ideas are described in [5];. However, our algorithm differs in two important ways from the other schemes: (i) it uses a different method for selecting vertices; (ii) it uses a two-dimensional partition to minimize communication.

The parallel refinement algorithm consists of a number of phases. During each phase, at each diagonal processor a group of vertices is selected from one of the two parts and is moved to the other part. The group of vertices selected by each diagonal processor corresponds to the vertices that lead to a decrease in the edge-cut. This process continues by alternating the part from where vertices are moved, until either no further improvement in the overall edge-cut can be made, or a maximum number of iterations has been reached. In our experiments, the maximum number of iterations was set to six. Balance between partitions is maintained by (a) starting the sequence of vertex swaps from the heavier part of the partition, and (b) by employing an explicit balancing iteration at the end of each refinement phase if there is more than 2% load imbalance between the parts of the partition.

Our parallel variation of the Kernighan-Lin refinement algorithm has a number of interesting properties that positively affect its performance and its ability to refine the partition. First, the task of selecting the group of vertices to be moved from one part to the other is distributed among the diagonal processors instead of being done serially. Secondly, the task of updating the internal and external degrees of the affected vertices is distributed among all the p processors. Furthermore, by restricting the moves in each phase to be unidirectional (*i.e.*, they go only from one partition to other) instead of being bidirectional (*i.e.*, allow both types of moves in each phase), we can guarantee that each vertex in the group of vertices being moved reduces the edge-cut.

In the serial implementation of BKLR, it is possible to make vertex moves that initially lead to worse partition, but eventually (when more vertices are moved) better partition is obtained. Thus, the serial implementation has the ability to climb out of local minima. However, the parallel refinement algorithm lacks this capability, as it never moves vertices if they increase the edge-cut. Also, the parallel refinement algorithm, is not as precise as the serial algorithm as it swaps groups of vertices rather than one vertex at a time. However, our experimental results show that it produces results that are not much worse than those obtained by the serial algorithm. The reason is that the graph coarsening process provides enough global view and the refinement phase only needs to provide minor local improvements.

4 Experimental Results

We evaluated the performance of the parallel multilevel graph partitioning and sparse matrix ordering algorithm on a wide range of matrices arising in finite element applications. The characteristics of these matrices are described in Table 1.

Matrix Name	No. of Vertices	No. of Edges	Description
4ELT	15606	45878	2D Finite element mesh
BCSSTK31	35588	572914	3D Stiffness matrix
BCSSTK32	44609	985046	3D Stiffness matrix
BRACK2	62631	366559	3D Finite element mesh
CANT	54195	1960797	3D Stiffness matrix
COPTER2	55476	352238	3D Finite element mesh
CYLINDER93	45594	1786726	3D Stiffness matrix
ROTOR	99617	662431	3D Finite element mesh
SHELL93	181200	2313765	3D Stiffness matrix
WAVE	156317	1059331	3D Finite element mesh

Table 1: Various matrices used in evaluating the multilevel graph partitioning and sparse matrix ordering algorithm.

We implemented our parallel multilevel algorithm on a 128-processor Cray T3D parallel computer. Each processor on the T3D is a 150Mhz Dec Alpha chip. The processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 150Bytes per second, and a small latency. We used SHMEM message passing library for communication. In our experimental setup, we obtained a peak bandwidth of 90MBytes and an effective startup time of 4 microseconds.

Since, each processor on the T3D has only 64MBytes of memory, some of the larger matrices could not be partitioned on a single processor. For this reason, we compare the parallel run time on the T3D with the run time of the serial multilevel algorithm running on a SGI Challenge with 1.2GBytes of memory and 150MHz Mips R4400. Even though the R4400 has a peak integer performance that is 10% lower than the Alpha, due to the significantly higher amount of secondary cache available on the SGI machine (1 Mbyte on SGI versus 0 Mbytes on T3D processors), the code running on a single processor T3D is about 15% slower than that running on the SGI. The computed speedups in the rest of this section are scaled to take this into account[1]. All times reported are in seconds. Since our multilevel algorithm uses randomization in the coarsening step, we performed all experiments with a fixed seed.

4.1 Graph Partitioning

The performance of the parallel multilevel algorithm for the matrices in Table 1 is shown in Table 2 for a p-way partition on p processors, where p is 16, 32, 64, and 128. The performance of the serial multilevel algorithm for the same set of matrices running on an SGI is shown in Table 3. For both the parallel and the serial multilevel algorithm, the edge-cut and the run time are shown in the corresponding tables. In the rest of this section we will first compare the quality of the partitions produced by the parallel multilevel algorithm, and then the speedup obtained by the parallel algorithm.

Figure 2 shows the size of the edge-cut of the parallel multilevel algorithm compared to the serial multilevel algorithm. Any bars above the baseline indicate that the parallel algorithm produces partitions with higher edge-cut than the serial algorithm. From this graph we can see that for most matrices, the edge-cut of the parallel algorithm is worse than that of the serial algorithm. This is due to the fact that the coarsening and refinement performed by the parallel algorithm are less powerful. But in most cases, the difference in edge-cut is quite small. For nine out of the ten matrices, the edge-cut of the parallel algorithm is within 10% of that of the serial algorithm. Furthermore, the difference in quality decreases as the number of partitions increases. The only exception is 4ELT, for which the edge-cut of the parallel 16-way partition is about 27% worse than the serial one. However, even for this problem, when larger partitions are considered, the relative difference in the edge-cut decreases; and for the of 128-way partition, parallel multilevel does slightly better than the serial multilevel.

Figure 3 shows the size of the edge-cut of the parallel algorithm compared to the Multilevel Spectral Bisection al-

[1] The speedup is computed as $1.15 * T_{SGI}/T_{T3D}$, where T_{SGI} and T_{T3D} are the run times on SGI and T3D, respectively.

	$p = 16$			$p = 32$			$p = 64$			$p = 128$		
Matrix	T_p	EC_{16}	S	T_p	EC_{32}	S	T_p	EC_{64}	S	T_p	EC_{128}	S
4ELT	0.48	1443	6.0	0.48	1995	7.0	0.48	3210	8.5	0.48	4734	11.1
BCSSTK31	1.28	27215	10.7	1.02	43832	17.0	0.87	67134	23.6	0.78	98675	31.6
BCSSTK32	1.69	43987	12.0	1.33	71378	19.2	1.05	104532	28.4	0.92	155321	37.9
BRACK2	2.14	14987	8.6	1.83	21545	12.2	1.56	32134	16.8	1.35	45345	21.9
CANT	3.20	199567	13.4	2.29	322498	23.7	1.71	441459	38.0	1.47	575231	49.7
COPTER2	2.05	22498	7.4	1.78	32765	11.1	1.59	45230	14.0	1.42	60543	18.2
CYLINDER93	2.35	131534	14.3	1.71	198675	24.5	1.34	288340	39.2	1.05	415632	56.3
ROTOR	3.16	26532	11.0	2.89	39785	14.4	2.40	57540	20.0	2.10	77450	26.4
SHELL93	5.80	54765	13.9	4.40	86320	22.5	3.25	130856	35.3	2.67	200057	49.9
WAVE	5.10	57543	10.3	4.70	76785	13.3	3.73	101210	19.9	3.09	138245	26.8

Table 2: The performance of the parallel multilevel graph partitioning algorithm. For each matrix, the performance is shown for 16, 32, 64, and 128 processors. T_p is the parallel run time for a p-way partition on p processors, EC_p is the edge-cut of the p-way partition, and S is the speedup over the serial multilevel algorithm.

Matrix	T_{16}	EC_{16}	T_{32}	EC_{32}	T_{64}	EC_{64}	T_{128}	EC_{128}
4ELT	2.49	1141	2.91	1836	3.55	2965	4.62	4600
BCSSTK31	11.96	25831	15.08	42305	17.82	65249	21.40	97819
BCSSTK32	17.62	43740	22.21	70454	25.92	106440	30.29	152081
BRACK2	16.02	14679	19.48	21065	22.78	29983	25.72	42625
CANT	37.32	199395	47.22	319186	56.53	442398	63.50	574853
COPTER2	13.22	21992	17.14	31364	19.30	43721	22.50	58809
CYLINDER93	29.21	126232	36.48	195532	45.68	289639	51.39	416190
ROTOR	30.13	24515	36.09	37100	41.83	53228	48.13	75010
SHELL93	69.97	51687	86.23	81384	99.65	124836	115.86	185323
WAVE	45.75	51300	54.37	71339	64.44	97978	71.98	129785

Table 3: The performance of the serial multilevel graph partitioning algorithm on an SGI, for 16-, 32-, 64-, and 128-way partition. T_p is the run time for a p-way partition, and EC_p is the edge-cut of the p-way partition.

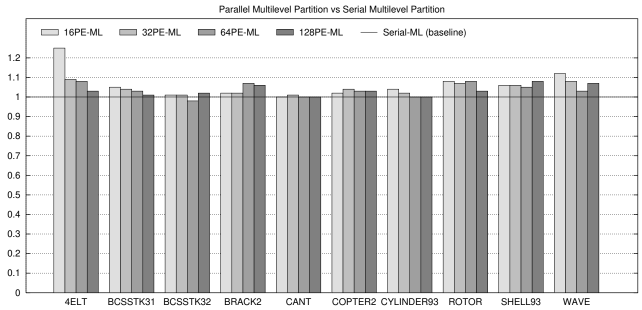

Figure 2: Quality (size of the edge-cut) of our parallel multilevel algorithm relative to the serial multilevel algorithm.

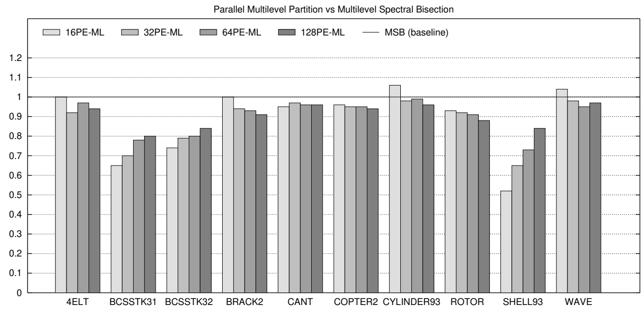

Figure 3: Quality (size of the edge-cut) of our parallel multilevel algorithm relative to the multilevel spectral bisection algorithm.

gorithm (MSB) [3]. The MSB algorithm is a widely used algorithm that has been found to generate high quality partitions with small edge-cuts. We used the Chaco [10] graph partitioning package to produce the MSB partitions. As before, any bars above the baseline indicate that the parallel algorithm generates partitions with higher edge-cuts. From this figure we see that the quality of the parallel algorithm is almost never worse than that of the MSB algorithm. For eight out of the ten matrices, the parallel algorithm generated partitions with fewer edge-cuts, up to 50% better in some cases. On the other hand, for the matrices that the parallel algorithm performed worse, it is only by a small factor (less than 6%). This figure (along with Figure 2) also indicates that our serial multilevel algorithm outperforms the MSB algorithm. An extensive comparison between our serial multilevel algorithm and MSB, can be found in [14].

Tables 2 and 3 also show the run time of the parallel algorithm and the serial algorithm, respectively. A number of conclusions can be drawn from these results. First, as p increases, the time required for the p-way partition on p-processors decreases. Depending on the size and characteristics of the matrix this decrease is quite substantial. The decrease in the parallel run time is not linear to the increase in p but somewhat smaller for the following reasons: (a) As p increases, the time required to perform the p-way partition also increases; (there are more partitions to perform). (b) The parallel multilevel algorithm incurs communication and idling overhead that limits the asymptotic speedup to $O(\sqrt{p})$ unless a good partition of the graph is available even before the partitioning process starts [16].

4.2 Sparse Matrix Ordering

We used the parallel multilevel graph partitioning algorithm to find a fill reducing ordering via nested dissection. The performance of the parallel multilevel nested dissection algorithm (MLND) for various matrices is shown in Table 4. For each matrix, the table shows the parallel run time and the number of nonzeros in the Cholesky factor L of the resulting matrix for 16, 32, and 64 processors. On p processors, the ordering is computed by using nested dissection for the first $\log p$ levels, and then multiple minimum degree [19] (MMD) is used to order the submatrices stored locally on each processor.

Matrix	T_{16}	$\|L\|$	T_{32}	$\|L\|$	T_{64}	$\|L\|$
BCSSTK31	1.7	5588914	1.3	5788587	1.0	6229749
BCSSTK32	2.2	7007711	1.7	7269703	1.3	7430756
BRACK2	2.9	7788096	2.5	7690143	1.8	7687988
CANT	4.4	29818759	2.8	28854330	2.2	28358362
COPTER2	2.6	12905725	2.1	12835682	1.6	12694031
CYLINDER93	3.5	15581849	2.2	15662010	1.7	15656651
ROTOR	6.1	23193761	4.0	24196647	3.0	24624924
SHELL93	8.5	40968330	5.7	40089031	4.5	35174130
WAVE	8.7	87657783	6.3	85317972	4.8	87243325

Table 4: The performance of the parallel MLND algorithm on 16, 32, and 64 processors for computing a fill reducing ordering of a sparse matrix. T_p is the run time in seconds and $|L|$ is the number of nonzeros in the Cholesky factor of the matrix.

Figure 4 shows the relative quality of both serial and parallel MLND versus the MMD algorithm. These graphs were obtained by dividing the number of operations required to factor the matrix using MLND by that required by MMD. Any bars above the baseline indicate that the MLND algorithm requires more operations than the MMD algorithm. From this graph, we see that in most cases, the serial MLND algorithm produces orderings that require fewer operations than MMD. The only exception is BCSSTK32, for which the serial MLND requires twice as many operations.

Comparing the parallel MLND algorithm against the serial MLND, we see that the orderings produced by the parallel algorithm requires more operations (see Figure 4). However, as seen in Figure 4, the overall quality of the parallel MLND algorithm is usually within 20% of the serial MLND algorithm. The only exception in Figure 4 is SHELL93. Also, the relative quality changes slightly as the number of processors used to find the ordering increases.

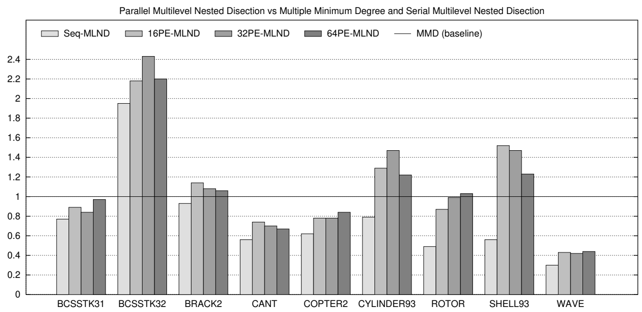

Figure 4: Quality of our parallel MLND algorithm relative to the multiple minimum degree algorithm and the serial MLND algorithm.

Comparing the run time of the parallel MLND algorithm (Table 4) with that of the parallel multilevel partitioning algorithm (Table 2) we see that the time required by ordering is somewhat higher than the corresponding partitioning time. This is due to the extra time taken by the approximate minimum cover algorithm and the MMD algorithm used during ordering. But the relative speedup between 16 and 64 processors for both cases are quite similar.

References

[1] Stephen T. Barnard. Pmrsb: Parallel multilevel recursive spectral bisection. In *Supercomputing 1995*, 1995.

[2] Stephen T. Barnard and Horst Simon. A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes. In *Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, pages 627–632, 1995.

[3] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proceedings of the sixth SIAM conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[4] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. In *6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452, 1993.

[5] Pedro Diniz, Steve Plimpton, Bruce Hendrickson, and Robert Leland. Parallel algorithms for dynamically partitioning unstructured grids. In *Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, pages 615–620, 1995.

[6] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[7] Madhurima Ghose and Edward Rothberg. A parallel implementtaion of the multiple minimum degree ordering heuristic. Technical report, Old Dominion University, Norfolk, VA, 1994.

[8] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *Internation Journal of Parallel Programming*, (16):498–513, 1987.

[9] M. T. Heath and Padma Raghavan. A Cartesian parallel nested dissection algorithm. Technical Report 92-1772, Department of Computer Science, University of Illinois, Urbana, IL, 1992. To appear in *SIAM Journal on Matrix Analysis and Applications*, 1994.

[10] Bruce Hendrickson and Rober Leland. The chaco user's guide, version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, 1993.

[11] Bruce Hendrickson and Rober Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[12] Zdenek Johan, Kapil K. Mathur, S. Lennart Johnsson, and Thomas J. R. Hughes. Finite element methods on the connection machine cm-5 system. Technical report, Thinking Machines Corporation, 1993.

[13] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. Technical Report TR 95-037, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/mlevel_analysis.ps.

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/mlevel_serial.ps. A short version appears in Intl. Conf. on Parallel Processing 1995.

[15] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical Report TR 95-064, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/mlevel_kway.ps.

[16] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. Technical Report TR 95-036, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/˜karypis/papers/mlevel_parallel.ps.

[17] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 1970.

[18] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[19] J. W.-H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11:141–153, 1985.

[20] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of 31st Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[21] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11(3):430–452, 1990.

[22] Padma Raghavan. Parallel ordering using edge contraction. Technical Report CS-95-293, Department of Computer Science, University of Tennessee, 1995.