# Efficient Nested Dissection for Multicore Architectures

Dominique LaSalle and George Karypis

Department of Computer Science & Engineering,
University of Minnesota,
Minneapolis, MN 55455, USA
{lasalle,karypis}@cs.umn.edu

**Abstract.** Sparse matrices are common in scientific computing and machine learning. By storing and processing only the non-zero elements of a matrix containing mostly zeros, sparse matrix algorithms often reduce computation and storage requirements of operations by an order of complexity. The order of the rows and columns of the matrix can have a significant impact on the efficiency of sparse direct methods. For example, in a Cholesky decomposition, it is desirable to re-order the input matrix so as to reduce the number of non-zeros in the factors. One of the most effective methods for re-ordering is nested dissection, where vertex separators are recursively found in the graph representation of the matrix and are used to permute the rows and columns. In this work we investigate the creation of vertex separators on shared memory parallel architectures and their use in nested dissection. We introduce a new effective scheme for refining a vertex separator in parallel, and a specialized parallel task scheduling scheme for the nested dissection problem. These algorithms have been implemented in the mt-Metis framework. Our experiments show that mt-Metis is 1.5× faster than ParMetis and PT-Scotch while producing orderings with 3.7% fewer non-zeros and 14.0% fewer operations.

## 1 Introduction

Sparse matrices are used in a variety of scientific computing and machine learning applications. Because sparse matrices do not store the zero-valued elements which make up the majority of their entries, their use results in significant savings of storage space as well as computation. Fill reducing orderings are permutations on the input matrix which decrease the number of non-zero elements (fill-in) in the output matrix of direct sparse methods [6]. For a Cholesky decomposition, we want to find a re-ordering such that the Cholesky factor will have as little fill-in as possible. One of the most effective methods for creating a fill reducing ordering is that of nested dissection [9, 10]. In nested dissection, balanced minimum vertex separators are recursively found in the graph representing the non-zero pattern of the sparse matrix. The quality of the resulting ordering depends upon being able to find small separators.

The problem of finding minimum size balanced vertex separators is known to be NP-Hard [2]. Heuristic multilevel methods have been developed to find small vertex separators in near linear time [12, 16, 17, 5]. Many of these approaches include scalable distributed memory algorithms. While these algorithms work well when each processor has its own memory hierarchy, their execution on modern multicore systems result in large degrees of memory contention and duplication. For generating edge separators, it has been shown that shared-memory parallel algorithms can result in significant runtime and memory usage reductions [5, 4, 19].

Vertex separators pose several additional challenges to parallelism beyond those of edge separators. Whereas most applications for edge separators demand that the partitioning be generated quickly and place only moderate importance on the quality of the separator, nested dissection places a much higher importance on quality. While the higher levels of recursion in nested dissection result in independent tasks, they are still bounded by memory bandwidth on multicore systems and are often unbalanced in their associated work. An effective approach must effectively balance these tasks while achieving high cache utilization.

In this paper, we present shared memory parallel algorithms for generating vertex separators and using those vertex separators to generate a fill reducing ordering via nested dissection in parallel. Our contributions build on the previous work for creating edge separators using the multilevel paradigm on shared memory architectures [19]. We adapt these algorithms for vertex separators and introduce a new method for refining a vertex separator in parallel while making minimal sacrifices in terms of separator size. We introduce specialized task scheduling to maximize cache efficiency for the nested dissection problem. We achieve up to 10× speedup on 16 cores, while producing orderings with only 1.0% more fill-in and requiring only 0.7% more operations than the serial ND-Metis. This is 1.5× faster, 3.7% less fill-in, and 14.0% fewer operations than ParMetis [16].

This paper is organized into the following sections. Section 2 introduces the notation used in this paper. Section 3 discusses relevant prior work. In Section 4, we detail our contributions. Section 5 describes the conditions of our experiments and our results follow in Section 6. Finally, we summarize the contributions of this work in Section 7.

## 2 Definitions & Notation

In this work we deal with a simple undirected graph $G = (V, E)$, consisting of a set of vertices $V$, and a set of edges $E$. Each edge is composed of an unordered pair of vertices (i.e., $v, u \in V$).

We will denote the size of the vertex set by the scalar $n = |V|$, and the size of the edge set by the scalar $m = |E|$. Vertices and edges can have non-negative integer weights associated with them. The weight of a vertex $v$ is denoted by $\eta(v)$, and the weight of an edge $e$ is denoted by $\theta(e)$. If there are no weights associated

with the edges, then their weights are assumed to be one. The *neighborhood* of a vertex $v$, that is the set of vertices adjacent to $v$, is denoted by $\Gamma(v)$.

A vertex separator is a set of vertices of the graph $S \subset V$, such that when removed it leaves two components $A$ and $B$. Finding a vertex separator is often subject to a balance constraint, $\epsilon$. That is, we want to minimize $|S|$ while satisfying:

$$2\frac{\max(|A|,|B|)}{|A|+|B|} \leq 1+\epsilon.$$

## 3 Background

For over two decades multilevel methods have been used with great success for graph partitioning. These methods have been shown to be both extremely fast and produce results of high quality [11, 15, 21, 23]. First, increasingly coarser graphs $G_1, \ldots, G_s$ are generated from the original graph $G_0$. This process is known as the *coarsening* phase. Next, in the *initial partitioning* phase, a partitioning of the coarsest graph $G_s$, is made via some direct partitioning algorithm (e.g., spectral bisection [22] or KL [18]). This initial solution is then projected through the multiple graph levels, and is refined at each level as the degrees of freedom are increased. This is known as the *uncoarsening* phase. Buluç et al. [3] provide a thorough overview of modern multilevel approaches to graph partitioning.

The use of threads to exploit shared memory parallelism has recently been used to decrease runtimes and memory usage compared to that of traditional parallel distributed memory codes. Chevalier and Pellegrini [5] presented PT-Scotch, a parallel partitioning library exploiting both shared and distributed memory parallelism. Threads are used to parallelize the coarsening phase, which provides significant speedup even with refinement and several other steps being performed serially. Çatalyürek et. al. [4] similarly explored parallelizing the coarsening of hypergraphs via shared memory parallelism. LaSalle and Karypis [19] investigated methods for effectively parallelizing all three phases of the multilevel paradigm.

Originally proposed by George [9, 10], nested dissection is a recursive algorithm for generating fill reducing orderings of sparse matrices. The algorithm works by recursively partitioning the graph representation of a symmetric sparse matrix via vertex separators, ordering the rows and columns with partition $A$ first, $B$ second, and $S$ last. This new ordering can greatly reduce the required memory and number of computations for performing Cholesky factorization. Because at each level the vertex separators induce two disconnected components, $A$ and $B$, parallelism can efficiently be extracted by ordering $A$ and $B$ in parallel.

As such, the creation of vertex separators for nested dissection can be parallelized by processing $A$ and $B$ independently. The popular parallel partitioning packages ParMetis [16] and PT-Scotch [5] both follow similar multilevel approaches to performing nested dissection. All $p$ processors work cooperatively to create the first $\log p$ levels of separators in parallel, before each processor performs nested serial dissection on its subgraph.

## 4 Methods

This paper builds upon the previous work for multi-threaded multilevel graph partitioning [19]. We use the same parallelization and coarsening strategies. Each thread is assigned a set of vertices and their associated edges, and is responsible for the computations on them.

### 4.1 Coarsening

The coarsening phase consists of two steps: *matching* and *contraction*. During matching, each vertex is either paired with a neighbor vertex, or itself. During contraction, paired vertices are merged together to form coarse vertices in the next coarser graph $G_{i+1}$.

The matching scheme we use is known as Heavy Edge Matching (HEM) [16]. This prioritizes edges for matching across based on their weight. Then, in a matching vector $M$, the matches of vertices $v$ and $u$ are recorded, $M(v) = u$ and $M(u) = v$. As this matching is done without locks, it is possible for race conditions to exist in determining if a vertex is unmatched. To resolve this issue, the strategy proposed by Çatalyürek et al. [4] is used. Each thread re-iterates over its set of vertices, and any vertex for which $M(M(v)) \neq v$, is matched with itself ($M(v) = v$). Because the number of vertices is orders of magnitude greater than the number of threads, the number of broken matchings is extremely small.

Contraction is an inherently parallel process, as each coarse vertex in $G_{i+1}$ can be independently constructed given $G_i$ and $M$. This process of matching and contraction repeats until $G_i$ is sufficiently small for the initial partitioning phase.

### 4.2 Vertex Separators

The generation of vertex separators differs from edge separators in the initial partitioning and uncoarsening phases.

**Initial Separator Selection** A widely used method of generating a vertex separator from an edge separator is to find a vertex cover of the set of cut edges [22]. Because we apply refinement to the separator, we instead take all boundary vertices as the initial separator of the coarsest graph $G_s$, and let refinement thin the separator and possibly move it away from the boundary set of vertices. We repeat this process several times and select the minimum balanced separator. As these separators are generated and refined independently, the process in inherently parallel. As the input graph is the same across the generation of different separators, waiting until $G_s$ is sufficiently small so as to fit into shared cache is desirable.

**Separator Refinement** After the current separator is projected from $G_i$ to $G_{i-1}$, it is refined. Refinement of a vertex separator consists of moving vertices from the separator $S$ into either partition $A$ or partition $B$. If a vertex being moved is connected to vertices on the opposite side of the separator, those vertices are then pulled into the separator. The reduction in separator size from moving vertex $v \in S$ to $A$ is

$$gain = \eta(v) - \sum_{u \in \Gamma(v) \cap B} \eta(u). \tag{1}$$

**FM Refinement:** The Fidducia-Mattheyses refinement (FM) algorithm [7], as applied to the vertex separator problem [12], works as follows. First, priority queues for moving vertices out of the separator to either partition are initialized and filled with vertices in $S$. The priority of vertices in these queues is determined by equation (1). Vertices are selected from either priority queue in order of gain, except when one partition is overweight, in which case the vertex at the top of the priority queue for the lower weight partition is selected. Once a vertex is selected, it is moved out of the separator, and its neighbors in the opposite partition are pulled into the separator. If the neighbors being pulled into the separator have not been moved yet in this refinement pass, they are added to the priority queue. Once both priority queues are emptied, the best observed state is restored. To reduce runtime, this process is terminated early if a certain number of moves past the best state have been made. Keeping track of the best state and reverting to it, makes the FM algorithm inherently serial.

**Greedy Refinement:** The greedy algorithm moves vertices through the separator to one side at a time. This is done so that at any given moment, the current state of the separator is valid. First, the lowest weight side of the separator is selected as the side to which all moves will be made in the first pass. Then, each thread adds the vertices it owns that are part of the separator to its own priority queue, using equation (1) for the priority. Each thread makes a local copy the current partition weights which it uses to keep track of moves and enforce the balance constraint. These weights are periodically synchronized with the global weights as moves are made. While this makes it possible for refinement to violate the balance constraint if enough vertices are moved before partition weights are synchronized, it is unlikely as it is desirable for the balance constraint on vertex separators in nested dissection to be large [13]. In practice we have not observed Greedy refinement to cause imbalance.

Each thread then extracts vertices from its priority queue. If the vertex can be moved out of the separator without violating the balance constraint, and has a positive gain associated with it, it is moved. The neighboring vertices that the thread owns have their connectivity information updated and are added to the separator as applicable. Messages are sent to the threads owning the remote vertices to notify them of the move.

Once the queue is empty, or the gain associated with moving the top vertex is negative, the thread waits for the other threads to finish. The thread then reads its messages, and updates its vertices accordingly. Finally, the threads

synchronize once more, and the process repeats with the other side selected. While efficient, this method often results in lower quality than the serial FM algorithm as it cannot break out of local minima.

**Segmented FM Refinement:** Because we want the improved quality that results from breaking out of local minima, one possible solution is to have threads perform FM on internal vertices (vertices which are not connected to vertices owned by another thread). We will refer to this approach as Segmented FM (SFM), which for these internal vertices works the same as the serial FM algorithm and allows us to break out of local minima in parallel. External vertices, those that have neighbors belonging to other threads, are prevented from moving out of the separator. This ensures that as long as each thread maintains a valid separator for its vertices, the global separator will also be valid. Each thread saves its best locally observed state, and independently reverts back to it at the end of each pass.

For this method to be effective, each thread must have a large number of internal vertices and few external vertices. To accomplish this, as a pre-processing step, we create a $k$-way edge separator of the graph using the method described in [19]. While this increases the runtime, it is a parallel step and scales well. Furthermore, this pre-partitioning improves data locality, which is particularly beneficial for nested dissection where we can use a single pre-partitioning for the entire process. We select a value for $k$ that is several times larger than the number of threads and assign partitions to threads via hashing so that each thread owns vertices in several locations of the graph. This is done so that many of the threads will own vertices that will be part of the separator, and the work during refinement will be distributed across multiple threads. We found using a value of $k$ that is five times the number of threads to be effective.

While this method allows us to find high quality local separators, the inability to move external vertices prevents the separator from moving significantly. For more than a few threads, this can have a significant impact on separator size as is shown in Section 6.

**Greedy with Segmented FM Refinement:** Both Greedy refinement and SFM refinement have their advantages and disadvantages. Greedy refinement's ability to move both internal and external vertices allows it to move the separator freely, but it cannot break out of local minima. SFM refinement can break out of local minima for a thread's internal vertices, however external vertices anchor the separator in place, limiting the improvement. As quality is one of our primary concerns, these disadvantages make both Greedy and SFM refinement unattractive options on their own.

For this reason, we investigated a hybrid refinement strategy by overlapping Greedy and SFM refinement passes. The first greedy pass thins the separator and moves it to a local minima. Next, the SFM pass moves the sections of the separator on internal vertices out of the local minima. The next Greedy pass then allows the external vertices to catch up with the moved internal ones. This process repeats until neither the Greedy pass nor the SFM pass move any vertices. This provides an effective refinement scheme that can break out

**Algorithm 1** Parallel Nested Dissection

```
1: function ND(G)
2:     if Number of threads is greater than 1 then
3:         A, B, S ← vertex separator of G, in parallel
4:         P_A ← half the threads call ND(A)
5:         P_B ← half the threads call ND(B)
6:     else
7:         A, B, S ← vertex separator of G, serial
8:         Add ND(A) to work pool
9:         Add ND(B) to work pool
10:        Wait for ND(A) and ND(B) to finish
11:    end if
12:    return {P_A, P_B, S}
13: end function
```

of local minima and move external vertices in parallel, without leading to an invalid separator.

### 4.3 Nested Dissection

Our parallel nested dissection algorithm is described in Algorithm 1. At the first level, all threads call the function `ND`. The threads then induce a vertex separator cooperatively, and use this to split the graph into parts $A$ and $B$. The threads then split into two groups, with one group recursing on $A$ and the other recursing on $B$, generating the orderings $P_A$ and $P_B$ respectively. This repeats until each thread group contains only a single thread. Each thread then spawns tasks for processing $A$ and $B$, and adds them to the work pool. Once both $A$ and $B$ have been ordered, the ordering of $G$ is computed by placing $A$ first, $B$ second, and $S$ last. When $|A|$ is small enough, it is ordered via the Multiple Minimum Degree algorithm [20]. This is omitted due to space constraints from Algorithm 1.

**Task Scheduling** Splitting the recursive calls on the graph parts $A$ and $B$ into parallel tasks, allows us to dynamically balance the computational load. However, we need to effectively utilize the cache to overcome memory bandwidth restrictions. The task tree of nested dissection has several properties that we want to keep in mind when scheduling the tasks. 1) The lower a task is on the tree (the earlier it is generated), the larger the graph that is associated with it. 2) The graph associated with a given task is a subgraph of the graph associated with its parent's task, thus the best cache use is achieved by having a task processed immediately after its parent.

To maximize our cache use, we propose a task scheduling scheme specifically for the nested dissection problem, that takes advantages of these properties. Our scheduling scheme operates on two levels. Each thread maintains a local list of tasks that it generates. It processes the tasks in its list in Last-In First-Out order to ensure that whatever subgraph is currently cached is used by the

Table 1: Graphs Used in Experiments

| Graph | # Vertices | # Edges | Graph | # Vertices | # Edges |
|---|---|---|---|---|---|
| auto | 448,695 | 3,314,611 | delaunay_n24 | 16,777,216 | 50,331,601 |
| NLR | 4,163,763 | 12,487,976 | large_fe | 7,221,643 | 83,149,197 |
| med_fe | 1,752,854 | 20,552,976 | nlpkkt240 | 27,993,600 | 373,239,376 |

next scheduled task as often as possible. When a thread runs out of tasks in its own list, it steals tasks from neighboring threads in First-in First-out order (the largest tasks). This not only ensures stolen tasks have enough work associated with them to achieve cache re-use, also ensures that the stolen tasks are the ones least likely to have their associated graph resident in another thread's cache. In Section 6.3 we compare this scheduling scheme against the generic scheme implemented in the OpenMP runtime.

## 5 Experimental Methodology

The experiments in this paper were run on a HP ProLiant BL280c G6 with 2x 8-core Xeon E5-2670 @ 2.6 GHz system with 64GB of memory. We used Intel C Compiler, version 13.1, and the GNU GCC compiler 4.9.2. The algorithms evaluated here are implemented in mt-Metis 0.4.0, which is available from `http://cs.umn.edu/~lasalle/mt-metis`. We will refer to the new vertex separator and nested dissection functionality as mt-ND-Metis in the following experiments. For comparison, we also used Metis [16] version 5.1.0 (referred to in the experiments as ND-Metis) from `http://cs.umn.edu/~metis`, ParMetis [17] version 4.0.3 from `http://cs.umn.edu/~metis`, and PT-Scotch [5] version 6.0.3 from `http://www.labri.fr/perso/pelegrin/scotch`.

The results presented for vertex separators are the geometric means from 25 runs using different random seeds. The results presented for nested dissection are the geometric means from 10 runs using different random seeds.

Table 1 details the graphs used for evaluation in Section 6. We opted to use these graphs for varying sizes and domains. The auto, NLR, delaunay_n24, and nlpkkt240 graphs were obtained from the 10th DIMACS Implementation Challenge [1]. The graphs med_fe and large_fe are 3D finite element meshes used in physics simulations.

## 6 Results

### 6.1 Vertex Separators

Table 2 shows the effect on separator size of the different refinement schemes. We compare the three parallel methods run with 16 threads to that of serial FM. SFM refinement resulted in large separators compared to that of serial FM, due to its inability to move external vertices. Greedy refinement did much better, finding separators only 6.1% larger than serial FM. The refinement scheme

Table 2: Size of Vertex Separators

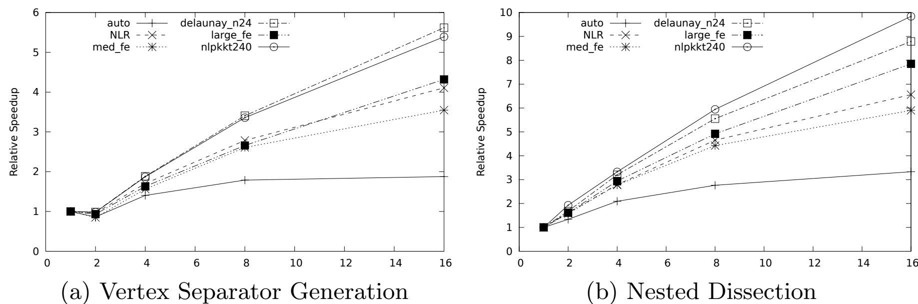
| | auto | NLR | med_fe | delaunay_n24 | large_fe | nlpkkt240 |
|---|---|---|---|---|---|---|
| FM (serial) | **2,133** | **1,811** | 2,166 | 3,507 | 6,421 | 156,564 |
| Greedy | 2,277 | 1,918 | 2,281 | 4,167 | 6,717 | 148,665 |
| SFM | 2,985 | 2,264 | 5,882 | 4,302 | 1,2430 | 262,243 |
| Greedy+SFM | 2,205 | 1,821 | **2,071** | **3,492** | **6,024** | **146,523** |

Table 3: Refinement Time in Seconds

| | auto | NLR | med_fe | delaunay_n24 | large_fe | nlpkkt240 |
|---|---|---|---|---|---|---|
| FM (serial) | 0.044 | 0.178 | 0.104 | 0.898 | 0.336 | 3.183 |
| Greedy | 0.048 | 0.091 | 0.071 | 0.130 | 0.181 | 1.251 |
| SFM | **0.030** | **0.069** | 0.068 | **0.115** | 0.185 | **1.153** |
| Greedy+SFM | 0.050 | 0.101 | **0.062** | 0.147 | **0.134** | 2.678 |

combining both Greedy and SFM refinement passes, produced separators of comparable size to FM, and for several graphs found slightly smaller separators on average. The number of external vertices that are prevented from being moved when trying to break out of a local minima in this scheme is quite small due to our pre-partitioning.

Table 3 shows the effect on runtime of the different refinement schemes. The runtime of serial FM is included for comparison against the other three refinement schemes run with 16 threads. None of the parallel refinement schemes exhibit significant speedup over FM consistently. There are two reasons for this. First, refinement operates on a small portion of the graph, and requires frequent synchronization. Second, the parallel refinement schemes make more passes before they settle on a separator. This also explains why the Greedy+SFM scheme is sometimes faster than the SFM and Greedy schemes. It performs more work per pass than either Greedy or SFM, but settles on a separator in fewer passes.




(a) Vertex Separator Generation (b) Nested Dissection

Fig. 1: Strong Scaling of mt-ND-Metis on 16 Cores

Table 4: Improvement over OpenMP Task Scheduling

| | auto | NLR | med_fe | delaunay_n24 | large_fe | nlpkkt240 |
|---|---|---|---|---|---|---|
| ICC OMP | 68.9% | 38.3% | 48.7% | 30.4% | 39.9% | 25.9% |
| GCC OMP | 62.2% | 39.0% | 60.2% | 25.6% | 40.0% | 23.0% |

Figure 1a shows the strong scaling of mt-ND-Metis generating vertex separators using up to 16 cores. The time shown includes the cost of pre-partitioning the graph, which is why there is a slowdown observed between one and two threads. The speedup achieved is largely dependent upon the size of the graph, and how effectively the amount of work between synchronization points can hide the parallel overhead. Looking beyond two threads, the larger graphs achieve speedups nearing 6× overall. Discounting the pre-partitioning time, the largest and third largest graphs exhibit super linear scaling with speedups over 17×. This is due to improved locality that comes from the pre-partitioning, and the extra cache available on the second processor. This shows the importance of having a well distributed graph, even on shared memory architectures.

### 6.2 Task Scheduling

Table 4 shows the percent improvement of our nested dissection task scheduling scheme, over that of the implementation schemes provided by ICC [14] and GCC [8]. Our scheme was on average 41.1% faster than the ICC scheduler and 40.6% faster than the GCC scheduler. This is because these schedulers are designed to handle tasks with varying properties, whereas our specialized scheduler takes advantage of the nature of the nested dissection task tree.

### 6.3 Nested Dissection

Figure 1b shows the strong scaling of mt-ND-Metis performing nested dissection. For the smallest graph, auto, the achieved speedup is limited to 3.3×, as the parallel overhead plays a significant role in the runtime. For the larger graphs, the different graph operations performed dominate the runtime and hide the parallel overhead. As a result, speedup of 6–10× is achieved on the other five graphs. We see a greater speedup here than on just vertex separators as the cost of performing nested dissection is significantly greater than that of creating a $k$-way edge separator, and better hide its added cost.

Table 5 compares the orderings of mt-ND-Metis with that of ND-Metis, ParMetis, and PT-Scotch, in terms of number of non-zeros in the Cholesky factor and the operations required to compute it. The runtimes to generate these orderings are also included (excluding I/O, but including preprocessing). Making efficient use of the multicore system, mt-ND-Metis was on average 1.5× faster than the other two parallel methods, and 10.1× faster than the serial ND-Metis. The number of operations required by orderings produced by mt-ND-Metis were only 0.7% higher than those required by mt-ND-Metis, and 14.0% lower than

Table 5: Comparison of Nested Dissection

| | auto | NLR | med_fe | delaunay_n24 | large_fe | nlpkkt240 |
|---|---|---|---|---|---|---|
| ND-Metis | | | | | | |
| Fill-in | **2.22e+08** | **2.05e+08** | 2.88e+08 | **7.24e+08** | 1.61e+09 | **1.98e+11** |
| Operations | **4.53e+11** | **1.25e+11** | 3.83e+11 | **7.39e+11** | 4.57e+12 | **1.93e+16** |
| Time (s) | 7.94 | 51.82 | 39.26 | 248.83 | 184.58 | 1148.52 |
| mt-ND-Metis 16 Threads | | | | | | |
| Fill-in | 2.31e+08 | 2.06e+08 | **2.87e+08** | 7.30e+08 | **1.55e+09** | 2.07e+11 |
| Operations | 5.06e+11 | 1.28e+11 | **3.71e+11** | 7.46e+11 | **3.94e+12** | 2.04e+16 |
| Time (s) | 1.44 | **4.67** | **4.44** | **17.85** | **16.34** | **93.80** |
| ParMetis 16 Processes | | | | | | |
| Fill-in | 2.29e+08 | 2.13e+08 | 3.10e+08 | 7.58e+08 | 1.60e+09 | 2.17e+11 |
| Operations | 4.94e+11 | 1.52e+11 | 4.98e+11 | 9.40e+11 | 4.51e+12 | 2.30e+16 |
| Time (s) | 1.60 | 6.21 | 6.43 | 29.52 | 31.17 | 169.84 |
| PT-Scotch 16 Processes | | | | | | |
| Fill-in | 2.52e+08 | 2.73e+08 | 3.84e+08 | 9.72e+08 | 1.93e+09 | 2.62e+11 |
| Operations | 5.89e+11 | 3.39e+11 | 8.70e+11 | 2.00e+12 | 8.57e+12 | 2.79e+16 |
| Time (s) | **1.12** | 5.83 | 7.46 | 26.82 | 39.33 | 678.65 |

those required by ParMetis or PT-Scotch. The hybrid refinement of mt-ND-Metis enables these high quality results, close to that of ND-Metis. The high-speed parallel vertex separator generation during the low levels of the nested dissection tree coupled with the specialized task scheduling in the higher levels enables mt-ND-Metis to produce orderings the fastest for all datasets except the smallest.

## 7 Conclusion

In this work we presented new shared-memory parallel methods for producing minimal balanced vertex separators and fill reducing orderings of sparse matrices. Specifically, we introduced a new parallel refinement scheme that can break out of local minima. We also introduced a task scheduling scheme specifically designed for the nested dissection problem that outperforms OpenMP task schedulers by 40.8%. We implemented these algorithms in mt-ND-Metis, and show that produces orderings 1.5× faster than ParMetis [16] and PT-Scotch [5], and 10.1× faster than ND-Metis [16]. The orderings produced by mt-ND-Metis result in only 1.0% more fill-in and require only 0.7% more operations than those of ND-Metis.

## Acknowledgment

This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities

was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

## References

1. Bader, D.A., Meyerhenke, H., Sanders, P., Wagner, D. (eds.): Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings, Contemporary Mathematics, vol. 588. American Mathematical Society (2013)
2. Bui, T.N., Jones, C.: Finding good approximate vertex and edge partitions is np-hard. Information Processing Letters 42(3), 153 – 159 (1992)
3. Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent advances in graph partitioning. CoRR abs/1311.3144 (2013), http://dblp.uni-trier.de/db/journals/corr/corr1311.html#BulucMSSS13
4. Çatalyürek, Ü.V., Deveci, M., Kaya, K., Ucar, B.: Multithreaded clustering for multi-level hypergraph partitioning. In: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. pp. 848–859. IEEE (2012)
5. Chevalier, C., Pellegrini, F.: Pt-scotch: A tool for efficient parallel graph ordering. Parallel Computing 34(6), 318–331 (2008)
6. Davis, T.A.: Direct methods for sparse linear systems, vol. 2. Siam (2006)
7. Fiduccia, C., Mattheyses, R.: A linear-time heuristic for improving network partitions. In: Design Automation, 1982. 19th Conference on. pp. 175 –181 (june 1982)
8. Free Software Foundation: The GNU OpenMP Implementation (2014), https://gcc.gnu.org/onlinedocs/gcc-4.9.2/libgomp.pdf
9. George, A.: Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis 10(2), 345–363 (1973)
10. George, A., Liu, J.W.: An automatic nested dissection algorithm for irregular finite element problems. SIAM Journal on Numerical Analysis 15(5), 1053–1069 (1978)
11. Hendrickson, B., Leland, R.: A multilevel algorithm for partitioning graphs. In: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM). Supercomputing '95, ACM, New York, NY, USA (1995), http://doi.acm.org/10.1145/224170.224228
12. Hendrickson, B., Rothberg, E.: Effective sparse matrix ordering: Just around the bend. In: Proc. of 8th SIAM Conf. Parallel Processing for Scientific Computing. Citeseer (1997)
13. Hendrickson, B., Rothberg, E.: Improving the run time and quality of nested dissection ordering. SIAM Journal on Scientific Computing 20(2), 468–489 (1998)
14. Intel: Intel OpenMP Runtine Library (2014), https://www.openmprtl.org/sites/default/files/resources/libomp_20141212_manual.pdf
15. Karypis, G., Kumar, V.: Multilevel graph partitioning schemes. In: ICPP (3). pp. 113–122 (1995)
16. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. 20(1), 359–392 (Dec 1998), http://dx.doi.org/10.1137/S1064827595287997
17. Karypis, G., Kumar, V.: A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. Journal of Parallel and Distributed Computing 48(1), 71–95 (1998)
18. Kernighan, B.W., Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs. The Bell system technical journal 49(1), 291–307 (1970)

19. LaSalle, D., Karypis, G.: Multi-threaded graph partitioning. In: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. pp. 225–236. IEEE (2013)
20. Liu, J.W.: Modification of the minimum-degree algorithm by multiple elimination. ACM Transactions on Mathematical Software (TOMS) 11(2), 141–153 (1985)
21. Pellegrini, F., Roman, J.: Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking. pp. 493–498. HPCN Europe 1996, Springer-Verlag, London, UK, UK (1996), http://dl.acm.org/citation.cfm?id=645560.658570
22. Pothen, A., Simon, H.D., Liou, K.P.: Partitioning sparse matrices with eigenvectors of graphs. SIAM Journal on Matrix Analysis and Applications 11(3), 430–452 (1990)
23. Sanders, P., Schulz, C.: Engineering multilevel graph partitioning algorithms. In: Demetrescu, C., Halldrsson, M. (eds.) Algorithms - ESA 2011, Lecture Notes in Computer Science, vol. 6942, pp. 469–480. Springer Berlin / Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-23719-5_40