A Fast Parallel Graph Partitioner for Shared-Memory Inspector/Executor Strategies

Christopher D. Krieger and Michelle Mills Strout

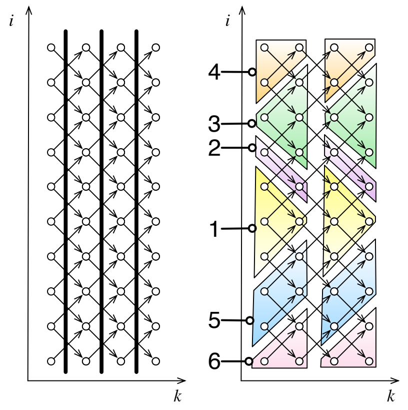
Colorado State University, Fort Collins CO 80523, USA
krieger|mstrout@cs.colostate.edu

Abstract. Graph partitioners play an important role in many parallel work distribution and locality optimization approaches. Surprisingly, however, to our knowledge there is no freely available parallel graph partitioner designed for execution on a shared memory multicore system. This paper presents a shared memory parallel graph partitioner, ParCubed, for use in the context of sparse tiling run-time data and computation reordering. Sparse tiling is a run-time scheduling technique that schedules groups of iterations across loops together when they access the same data and one or more of the loops contains indirect array accesses. For sparse tiling, which is implemented with an inspector/executor strategy, the inspector needs to find an initial seed partitioning of adequate quality very quickly. We compare our presented hierarchical clustering partitioner, ParCubed, with GPart and METIS in terms of partitioning speed, partitioning quality, and the effect the generated seed partitions have on executor speed. We find that the presented partitioner is 25 to 100 times faster than METIS on a 16 core machine. The total edge cut of the partitioning generated by ParCubed was found not to exceed 1.27x that of the partitioning found by METIS.

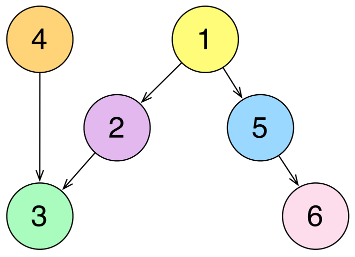
Keywords: inspector/executor strategies, graph partitioning, irregular applications, sparse tiling

1 Introduction

Computations involving irregular data access patterns figure prominently in many important scientific applications. These include solving partial differential equations over irregular grids, molecular dynamics simulations, and computations over sparse matrices. Often, key loops in these computations are largely free of loop carried dependencies and can be performed using doall parallelism across all or a subset of elements. Additionally, loops performing reductions can often be parallelized. Unfortunately, in many cases these straightforward parallelization strategies encounter performance problems due to the irregularity of the data accesses. Irregular accesses that jump around in memory decrease the efficiency of caching and data prefetching and therefore increase the demand on memory bandwidth.



(a) At left, the Jacobi iteration space for a tri-diagonal sparse matrix with a doall parallelization. On the right, a full sparse tiling of the computation.



(b) Task graph for the full sparse tiled Jacobi computation.

Fig. 1. Transformation of Jacobi computation from doall to task graph parallelism.

To avoid memory bandwidth bottlenecks, these algorithms are modified by performance programmers to improve the algorithm's temporal and spatial locality. These optimization techniques include irregular cache blocking [5], full sparse tiling [11], and communication avoiding algorithms [10].

The benefits realized by these approaches can be illustrated by applications as simple as a sparse Jacobi solver, which has the common nearest neighbor dependence structure that can be found in many irregular or sparse applications. The sparse Jacobi solver algorithm is an iterative algorithm for determining solutions to a system of linear equations. Given a sparse matrix A, and two vectors $\boldsymbol{u}$ and $\boldsymbol{f}$ related by $A\boldsymbol{u} = \boldsymbol{f}$, one iteration of the sparse Jacobi method produces an approximation to the unknown vector $\boldsymbol{u}$. This method is known to converge if A is strictly diagonally dominant. The recurrence equation that describes Jacobi is as follows:

$$u_i^{(k)} = \frac{1}{A_{ii}} \left(f_i - \sum_{j \neq i} A_{ij} * u_j^{(k-1)} \right)$$

While the Jacobi method admits a straightforward doall parallel solution on the i loop, this approach quickly hits scalability issues due to memory bandwidth demands. Figure 1(a) shows the iteration space for sparse Jacobi when the matrix is tri-diagonal so the matrix graph is a line. The left-side of Figure 1(a) shows a doall parallelization with barriers between each iteration of the outer k loop. The right side of Figure 1(a) shows a possible sparse tiling of Jacobi. That tiling results in the task graph shown in Figure 1(b). Figure 2 shows a comparison of the execution time between a doall parallelized Jacobi solver and a full sparse

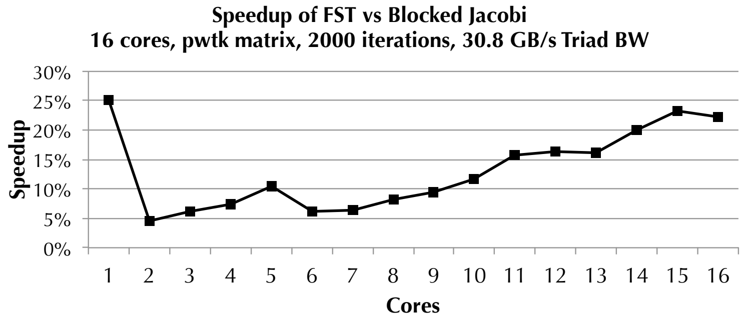


Fig. 2. Performance difference between full sparse tiled and blocked Jacobi solvers.

tiled [11, 12] version as threads are increased from one to sixteen on the 16 core machine (described in Section 4). The benefit ranges from 5% to 25% as cores are added and memory bandwidth demand increases.[1]

A common theme among optimization approaches such as full sparse tiling is the reduction of communication or data sharing between execution units through the aggregation of computation that accesses common data. As such, it is common to create a graph where each node represents some computation (e.g. an iteration point in 1(a)) and edges between such points represent that those computations share data. For example, the matrix graph for a sparse matrix, where each row/column is a node and there is an edge between nodes i and j if $A_{ij} \neq 0$, can be partitioned to create a seed partitioning for growing sparse tiles.

The graph partitioning problem is to divide a graph $G = (V, E)$ with $|V| = n$ into k partitions V_i such that $\bigcup_{i=1}^{k} V_i = V$ and $\bigcap_{i=1}^{k} V_i = \emptyset$ with the added constraint that the *edge cut*, the number of edges whose incident vertices are in different partitions, is minimized.

Given the importance of graph partitions, we were surprised to find no freely available parallel graph partitioner designed for execution on a shared memory multicore system. Therefore in this paper we present a shared memory parallel graph partitioner called *ParCubed* (Par Parallel Partitioner or PAR^3), emphasizing that the algorithm is parallel and produces results roughly on par with other partitioners. Section 2 presents the graph partitioning algorithm, and Section 3 describes how the algorithm is efficiently implemented. We evaluate the new partitioner by comparing it with METIS and GPart in Section 4. Section 5 presents related work, and Section 6 concludes the paper.

[1] Performance monitoring hardware indicates that the drop from one core to two cores is due to a doubling of last level cache available as a second 24MB shared cache on a second socket entered into use.

```
Inflate(initialNodeID , maxSize)
{
 // initialization
 enqueue initialNodeID into FIFO
 initialize currentSize to 1

 // main inflation loop
  while ( (currentSize < maxSize) && FIFO not empty )
  {
    dequeue nodeID from front of FIFO
    if (size of partition containing nodeID is 1)
    {
         if ( (currentSize + 1) <= maxSize )
         {
             // merge this node into the partition
             merge nodeID and initialNodeID partitions
             increment currentSize by one

             // add next generation of adjacent nodes to the FIFO
             foreach (node adjacent to node nodeID)
             {
                  enqueue adjacent node at back of FIFO
             }
} } } }
```

Fig. 3. Pseudocode of the inflation phase used to create partitions from starting nodes

2 Overview of the *ParCubed* Graph Partitioning Algorithm

In this section, we present an overview of our graph partitioning approach, ParCubed. At the highest level, the partitioning processing consists of three phases: *inflate*, *join*, and *fold*. First, single vertices are used as seeds for partitions. These partitions are *inflated* by adding adjacent vertices using a layered approach. In the second phase, these subpartitions are *joined* together to create fewer, but larger, partitions. Finally, if the generated number of partitions still exceeds the desired number, the excess partitions are *folded* into other partitions to bring the partition count down to the target number. In this section we detail each of these phases, and in Section 3 we parallelize the overall ParCubed partitioning algorithm

2.1 Description of the Inflation Phase

The inflation subroutine is expressed in pseudocode in Figure 3. It is essentially a breadth first growth. The algorithm arbitrarily selects a node from the graph to serve as the seed of a partition. It then adds all of that node's adjacent neighbors, reducing the edge cut contribution from the original node as much as possible, ideally to zero. It then proceeds to add all the neighbors of each of the original node's neighbors, and so forth, until the partition size limit is reached.

As the partition is inflated, neighbors may be encountered that are already in another partition. In that case, the inflation stops expanding in that direction,

```
GPartJoin(initialVertex, maxSize)
{
  currentPar = partition that contains initialVertex
  currentSize = size of currentPar

  foreach neighbor neighborVertex of initialVertex
  {
    neighborPar = partition that contains vertex neighborVertex
    neighborSize = size of neighborPar
    if (neighborSize + currentSize <= maxSize)
    {
      merge currentPar and neighborPar
      currentSize += neighborSize
    } } }
```

Fig. 4. Pseudocode of the GPart-like join phase

yielding to the other partition. Because of this, it is sometimes impossible to reduce the edge cut contribution of a layer to zero.

If at any time a growing partition cannot expand, such as when it is surrounded by other partitions, the process stops for this initial node. The algorithm then continues with another node taken as the seed of another subpartition.

2.2 Description of the Join Phase

It is possible that a large number of small partitions may result from the inflation process. To deal with this, the partitioner follows up the inflation step with a hierarchical partition joining step similar to GPart [7]. This phase is expressed in pseudocode in Figure 4. During this phase, each node is visited once again. If any neighbor of that node is found to be in a partition that is small enough that merging it with the currently visited node's partition would not exceed the maximum partition size, the two partitions are merged. This differs from inflation in that entire subpartitions are being merged, rather than single vertices being added to a partition.

2.3 Description of the Folding Phase

Commonly, the first two phases produce more partitions than are desired. A third, final step is used to trim the number of partitions. During the *folding* step, the partitions are ordered by partition size from smallest to largest. If k partitions are desired, then the *adopting partition* P_k and *extra partition* P_{k-1} are merged, P_{k+1} and P_{k-2} are merged, and so forth. This combines increasingly smaller extra partitions with increasingly larger adopting partitions. If more than twice the desired number of partitions was originally found, the folding process functions in a modulo fashion, wrapping as needed.

Extra and adopting partitions are matched solely based on size. Due to this fact, unconnected partitions can be created. If the matching process between

```
Fold(partNums : array of partition numbers
{
  origNumFoundPartitions = number of found partitions
  numberExtraPartitions = origNumFoundPartitions − num desired partitions;
  firstAdoptingPartitionIndex = numberExtraPartitions

  // figure out the point at which to fold
  adoptingPartitionIndex = firstAdoptingPartitionIndex;
  extraPartitionIndex = firstAdoptingPartitionIndex −1;

  // merge the partitions
  foreach extra partition
  {
    extraPartition = partNums[extraPartitionIndex−−]
    adoptingPartition = partNums[adoptingPartitionIndex++]

    // do modulo wrap
    if (adoptingPartitionIndex >= origNumFoundPartitions)
    {
      adoptingPartitionIndex = numberExtraPartitions;
    }

    // merge
    merge extraPartition and adoptingPartition
}
```

Fig. 5. Pseudocode of the folding step

adopting and extra partitions were to consider adjacency, higher quality partitions may be produced. In the interest of reducing runtime, adjacency is currently ignored.

Also note that during folding, the maximum partition size is ignored, so partitions that exceed the desired size can be produced. In practice, we saw approximately 5% of the partitions to be oversized. In general, the oversized partitions were within 15% of the target size, but a handful of extreme outliers were observed that were as large as 150% of the target.

3 Parallel Implementation of the Partitioner

Having provided an overview of our graph partitioning algorithm, we now turn to the details of how the algorithm can be efficiently implemented for parallel execution.

3.1 Parallel Disjoint Set Data Structure

Since much of the algorithm consists of identifying which partition a node is in, adding nodes to partitions, or merging two partitions, having an efficient way to do these operations in parallel is critical. The backbone of our implementation is therefore a parallel implementation of a disjoint set data structure. This structure is also known as a *union-find* data structure because of its efficient support for those two operations. It provides an $O(n)$, where n is the number of graph nodes, determination of which set contains a particular node and an $O(1)$ set merge operation.

The disjoint set data structure is built on the concept of a *forest of trees*. Each element of the disjoint set is either a top level root or else points to another element in the structure, indicating membership in that set. The element at which it points may be a root or may in turn point at another element, creating a hierarchy. To perform a merging of two sets, the root of one set is pointed to any element of the other set, usually the root.

Determining the set to which any given element belongs is a `find` operation. In a `find` on a given element, that element is visited. If it is a root, then the element belongs to that set. If it is not a root, the node at which it points is recursively visited until a root is reached.

While the `merge` operation is very efficient, it can lead to deep trees that must be traversed repeatedly during `find` operations. A `find` operation is always $O(n)$, but several optimizations can reduce the runtime cost in practice. *Path halving* involves linking each node to its grandparent during a `find` operation, effectively halving each node's depth. *Path flattening* is similar, but finds the root node for a given node, then points all elements between the initial node and the root node directly to the root node. As explained below, our approach does not use either optimization.

Our disjoint set structure was originally taken from Berman's thesis [2]. We subsequently modified that implementation to better suit our application. Our implementation consists of a one dimensional array of integers sized to hold one array element for each vertex in the adjacency graph. The value of each array element can represent one of two things. If the value is negative, then this element is a root of a set and the absolute value of the stored value is the size of the set. If the value is positive, it represents the array index of the parent of this element.

In general, disjoint set structures are not safe for parallel operation. To resolve this, we made a number of straightforward modifications to the standard disjoint set structure and its usage. First, `find` operations can proceed in parallel without any synchronization. This allows for many concurrent `find` operations to occur without the overhead of locking. A detrimental effect is that `find` operations cannot perform path flattening or path halving optimizations. However, these optimizations are largely unneeded by our algorithm. During the inflation process, we merge the original seed set only with nodes that are not yet merged with any other nodes. As a result, each of these other nodes is the root in its own set. When it is merged with the seed set, it creates a tree with depth two. When the inflation step completes, the entire disjoint set structure has maximum depth two and cannot be flattened further. During the join step, the disjoint set depth can grow, but since it is starting with a very shallow tree the depth typically does not exceed a depth of four, with depth eight being the greatest observed in our testing.

On a `merge` operation, a lock is acquired for both root nodes in the merge. Since a node's set may change after a find operation has returned its set, set membership information may be stale. To handle this, after locking the nodes passed to a `merge`, they are checked to see if they are still truly root nodes. If not, their locks are released and the nodes' paths are traversed until a root is again found. Those roots are locked and once again checked to determine if they are root nodes. This continues until both nodes are locked and are roots. The two sets are then merged and the locks are released.

The performance and scalability of the inflation step proved to be sensitive to the number of locks used by the disjoint set. If too coarse grained locks were used, lock contention hurt performance. If too fine grained locks were used, performance suffered in some cases because the locks were polluting the per-core caches and creating memory traffic. Time to initialize the locks also contributed significantly to total algorithm runtime when excessive locks were used.

To tune the number of locks, we used a process of gradual lock refining. We varied the number of locks and swept the thread count. At each point, we measured the amount of time spent in the `merge` algorithm as a rough proxy for lock contention. We also examined total partitioner runtime. Based on these data, we determine the lock count using a simple linear function of thread count and graph node count. The average degree of nodes in the adjacency graph had a secondary effect but is ignored in our current lock count calculation.

3.2 Overview of Partitioner Parallelization

The general parallelization strategy used for partitioning is a straightforward SPMD approach. Each thread is assigned a block of nodes.

Each thread immediately begins inflating from nodes in its block. As they grow, partitions can pull in nodes outside the thread's range, but because all operations done during the inflate phase use the disjoint set structure described above, they require no additional synchronization. There is, however, a barrier between the inflate and the join phases of our algorithm.

As with the inflate phase, any operation on shared data during the join phase consists entirely of disjoint set operations. Each thread attempts to join partitions within its chunk with adjacent partitions.

The final folding phase is currently done serially. It could be parallelized, but at present takes between 1% - 4% of the total algorithm runtime on a 16 core machine.

This approach to parallelization results in non-deterministic partitionings. The order in which nodes are initially inflated, and in which they are joined, impacts the final partitioning results. This order depends on several factors. First, the number of threads directly impacts the visitation ordering. A node that is visited first by some core when using N threads will most likely not be visited first when using $N + 1$ threads, simply because it will no longer be the first node in a block. Also, slight differences in operating system scheduler behavior cause different interleavings between threads, resulting in a different global ordering of node visitation.

Table 1. Characteristics of sparse matrices used in the performance evaluation.

Name	Rows	Avg Nonzeroes/Row	Memory (MB)
xenon2	157464	24	48
thermal2	1228045	7	130
pwtk	217918	53	138
nd24k	72000	399	345
audikw_1	943695	82	913

4 Evaluation

There are several aspects of performance that were considered when evaluating the ParCubed graph partitioner. We first wanted to evaluate the usefulness of each of the three distinct phases. In the context of a shared memory inspector, the runtime of the partitioner is crucial and was evaluated. Lastly, the edge cut of the partitioning is one well understood measure of the partitioning quality. We supplement edge cut results by also measuring the runtime of the executor phase while performing a full sparse tiling inspector/executor strategy. This runtime is the bottom line measure of partitioning quality.

The tiled computation used in this performance evaluation is a sparse Jacobi solver. The Jacobi algorithm is described in detail in Section 1. In these tests, the Jacobi kernel is tiled across two convergence iterations of the main loop. Each tile was sized to access approximately 200kB of data, so as to fit within the mid-level cache of the processors used.

All of the sparse matrices were drawn from the University of Florida Matrix Market and are listed in Table 1. The tests were run on a two socket 16 core Xeon E7-4830 server with 256kB mid level data caches per core and 24 MB of shared last level cache per socket. The Intel icpc compiler, version 12.1.2 (20111128) was used at optimization level -O3.

The GPart algorithm used for comparison purposes was run in three passes. The first pass created partitions of up to 50% the size of the final maximum partition size. The second pass permits partitions up to 75% of the final size, while the third pass accepts partitions up to 125% of the maximum size. We experimented with a variety of different size progressions and found this progression to give the lowest geometric mean of edge cuts across all input graphs. Also note that GPart typically does not generate the desired number of partitions. Using this progression, GPart on average returned a number of partitions equal to 97.5% of the requested number.

The serial METIS algorithm is the `METIS_PartGraphKway()` algorithm, with the partition size as the only balance constraint. Neither edge nor vertex weights were used and the objective function was set to minimize edge cut rather than communication volume.

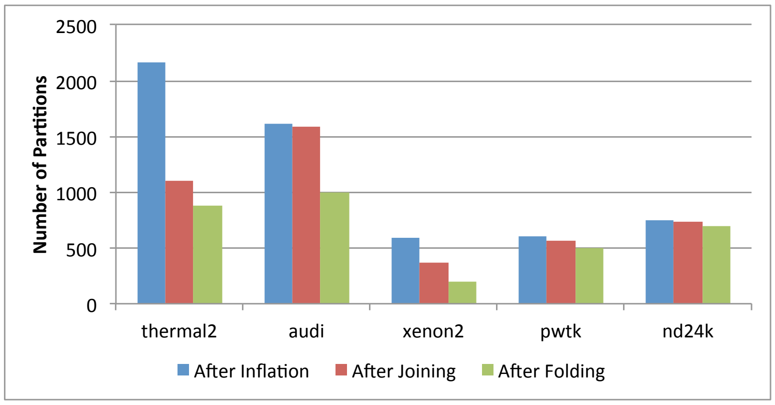


Fig. 6. Number of partitions after each phase.

4.1 Benefit derived from each phase of the partitioning process

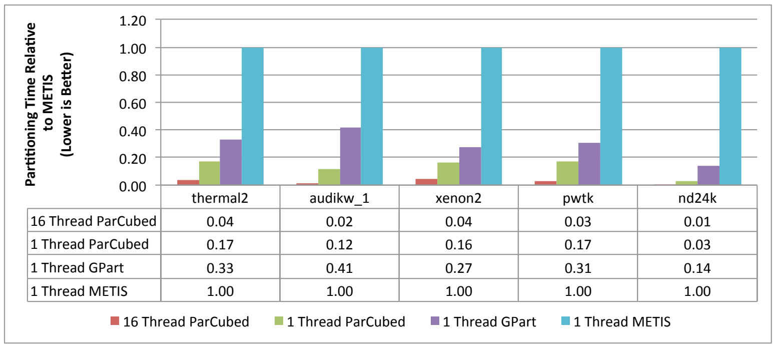
As seen in Figure 6, the number of partitions affected by each of the three phases (inflation, joining, folding) varies greatly between sparse matrices. The figure shows how many partitions exist after each phase of the algorithm. The requested number of partitions is always reached after folding.

In general, if the input matrix is sufficiently connected, meaning that it has a relatively high number of non-zero elements per row, than the inflation phase is able to do a good job of generating partitions. In these cases, the hierarchical joining phase provides little benefit. However, when the matrix is more sparse, such as the thermal2 matrix in this evaluation, the join step is able to combine smaller partitions and bring the total number of partitions more in line with the target. The audikw_1 matrix results shows a large number of partitions being combined during folding, demonstrating the value of the folding step. Note that METIS always generates the desired number of partitions for these input sets, while GPart regularly produces more or fewer partitions than requested.

4.2 Speed of partitioner compared with METIS and GPart

A major requirement for partitioners in a shared memory multicore environment is extremely low runtime. In Figure 7, we compare the total partitioning time of 16 core multithreaded ParCubed, single threaded ParCubed, serial METIS, and serial GPart on the five different sparse matrices in our test suite. All times are normalized to the METIS time. METIS is consistently the slowest algorithm, on the order of 5 to 30 times slower than serial ParCubed and 25 to 100 times slower than 16 thread ParCubed.

A full comparative study of parallel ParMETIS performance is a work in progress. However, due to the time required to either read in sparse matrix data on each rank or communicate portions of the data between ranks, the performance will not be comparable. For example, partitioning a trivial matrix



	thermal2	audikw_1	xenon2	pwtk	nd24k
16 Thread ParCubed	0.04	0.02	0.04	0.03	0.01
1 Thread ParCubed	0.17	0.12	0.16	0.17	0.03
1 Thread GPart	0.33	0.41	0.27	0.31	0.14
1 Thread METIS	1.00	1.00	1.00	1.00	1.00

Fig. 7. Runtimes of different partitioning algorithms relative to serial METIS

using ParMETIS on a multicore machine using a shared memory MPI transport took 1.025 seconds. By comparison, 16 thread ParCubed on a similar machine took just 0.18 seconds to fully partition the audikw_1 matrix, the largest in our study.

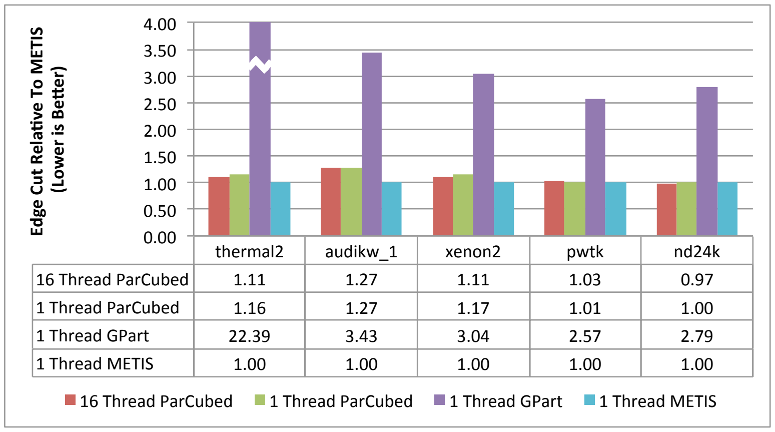
4.3 Comparison of Total Partitioning Edge Cut

One common metric of graph partitioning quality is the edge cut of the set of partitions. Figure 8 shows the edge cuts that were obtained using 16 way parallel ParCubed, serial ParCubed, GPart, and METIS. As explained earlier, in parallel ParCubed, the order in which nodes are visited varies with the number of threads used. As these results show, the quality of the partitioning is not greatly impacted by this effect in general, with results varying slightly in either direction.

When comparing the edge cut, note that METIS always generated the desired number of partitions for these input sets, while GPart regularly produced more partitions than requested.

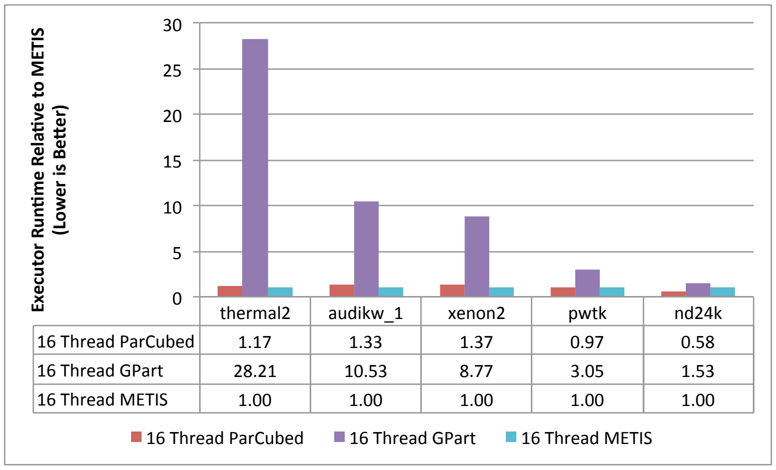
4.4 Executor Runtime

The ultimate purpose of partitioning the executor work is to reduce runtime. The runtime of the executor when run with 16 threads is shown in Figure 9. The results largely track the edge cut results of Figure 8, as expected. The nd24k sparse matrix is an exception. That matrix is the most highly connected in our test suite. It also exhibits the largest run to run variation in executor runtime. We believe that because of the large amount of shared data, it is very sensitive to task placement within the machine. If two tiles that share significant amounts of data are executed back to back on the same core, that core's cache will contain some of the shared data. This means that the assumption that each edge between partitions is equally costly does not always hold true. If communication between tiles requires accessing main memory, it is more expensive than data cache accesses. We believe that this effect partially explains the miscorrelation between edge cut and executor performance in this case.



	thermal2	audikw_1	xenon2	pwtk	nd24k
16 Thread ParCubed	1.11	1.27	1.11	1.03	0.97
1 Thread ParCubed	1.16	1.27	1.17	1.01	1.00
1 Thread GPart	22.39	3.43	3.04	2.57	2.79
1 Thread METIS	1.00	1.00	1.00	1.00	1.00

Fig. 8. Relative edge cuts achieved by different partitioning algorithms. Bars are normalized to METIS edge cut values. For thermal2, the GPart value of 22.39x is off the top of the chart. Lower values are better.



	thermal2	audikw_1	xenon2	pwtk	nd24k
16 Thread ParCubed	1.17	1.33	1.37	0.97	0.58
16 Thread GPart	28.21	10.53	8.77	3.05	1.53
16 Thread METIS	1.00	1.00	1.00	1.00	1.00

Fig. 9. Relative performance of the Jacobi executor. Bars are normalized to METIS executor runtime values. Lower values are better.

5 Related Work

The ParCubed graph partitioner presented in this paper depends on a parallel union-find data structure, therefore we summarize some of the work related to such data structures. Additionally we summarize other graph partitioners with similar approaches to the ParCubed graph partitioner, but that are not parallel or are only parallelized for distributed memory machines.

5.1 Parallel Union Find Algorithms and Data Structures

A major enabler of the algorithm presented here is the *disjoint set* or *union-find* data structure. A number of different implementations have been developed for use in parallel applications. Wait-free implementations are described in [1] and rely on atomic compare and exchange operations for correctness. Additional methods of implementing parallel disjoint set data structures are presented in [4]. In [2], the code complexity of a number of different parallel implementations of union-find is surveyed from a software engineering perspective. This work includes software transactional memory, coarse and fine grained locking, and wait free approaches.

5.2 Graph Partitioners

The field of graph partitioning has been extensively researched for decades. While many different techniques have been developed, the approaches most related to this research focus either on multi-level graph partitioning or on hierarchical partitioning.

Multi-level graph partitioners

Multi-level graph partitioners, such as the METIS [9] algorithm that was used in our comparison above, deliver high quality partitions. They typically involve a coarsening phase in which the graph is reduced, or coarsened, by combining multiple nodes into clusters or supernodes. This step is followed by a direct partitioning of the coarse graph. The results of the partitioning are then projected back onto the original graph. Multi-level partitioners differentiate themselves in the method used to perform each of these phases and in additional refinement steps added to improve the quality of the final partitioning.

To speed up the partitioning process, many multi-level partitioners have been parallelized. ParMETIS [8] is a set of multi-level parallel partitioning algorithms related to METIS. It is parallelized using MPI and is designed for use in distributed memory environments. METIS has also been parallelized for use on shared memory machines using the Galois system, but that implementation is presently slower than ParMETIS [13].

Jostle [14] is another MPI parallelized multi-level graph partitioner. Published results show that it can produce partitionings equal to or better than ParMETIS, but it is typically slower. Another multi-level graph partitioner, PT-Scotch [3], is also parallelized using MPI and likewise is slower than ParMETIS.

Hierarchical clustering or growth based partitioners
Another general approach to graph partitioning is to combine graph nodes together into clusters. These clusters are then combined to form larger clusters and so on until the desired partition size has been achieved.

GPart [7] is a hierarchical clustering partitioner that is significantly faster than METIS, but to the best of our knowledge it has not been parallelized. Its use has focused more on data reordering to improve locality [6] and it has features that allow the partitioning to target multiple levels of cache. The join phase of our partitioner is essentially a slight variation on the GPart technique. TLayout [15] is a growth based parallel graph partitioner, but it is specifically designed for execution on a GPGPU and performs poorly on multicore CPUs.

6 Conclusions and Future Work

ParCubed is a fast, parallel graph partitioner for shared memory systems. It delivers results comparable to those of METIS in a fraction of the runtime. This makes it an attractive option for applications such as inspector/executor optimizations in which the execution time of the partitioner must be minimized, even at the cost of a slight increase in graph partition edge cut.

Work on the ParCubed partitioner is ongoing. We are currently investigating the optimal settings for parameters such as target partition sizes used in each step. Setting the target size to less than the final desired partition size during inflation, then setting it to slightly larger than the target size during the join phase has shown some promise. Testing on a larger variety of sparse matrices continues as well. We are also investigating improvements to the inflation phase of the algorithm, searching for techniques to further reduce the total edge cut of the partitioning. We will also compare the partitioning times of this algorithm with those of ParMETIS running a shared memory based MPI communication layer on a multicore system.

Future work will include using the partitioner on problems from additional problem domains, such as molecular dynamics. In these cases, the adjacency graph is not based on a sparse matrix, but comes from an earlier phase of the inspector that determines adjacency based on physical proximity of atoms in the simulation space.

Acknowledgments

We thank Xipeng Shen and Bo Wu of the College of William and Mary for their help with the TLayout partitioner.

This project is supported by a Department of Energy Early Career Grant DE-SC0003956, by a National Science Foundation CAREER grant CCF 0746693, and by the Department of Energy CACHE Institute grant DE-SC04030.

References

1. R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, STOC '91, pages 370–380, New York, NY, USA, 1991. ACM.
2. I. Berman. Multicore programming in the face of metamorphosis: Union-find as an example. Master's thesis, Tel-Aviv University, July 2010.
3. C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331, July 2008.
4. G. Cybenko, T. G. Allen, and J. E. Polito. Practical parallel union-find algorithms for transitive closure and clustering. *Int. J. Parallel Program.*, 17(5):403–423, Oct. 1989.
5. C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Electronic Tranactions on Numerical Analysis*, 10:21–40, 2000.
6. H. Han and C.-W. Tseng. A comparison of locality transformations for irregular codes. In *5th International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR)*. Springer, 2000.
7. H. Han and C.-W. Tseng. Improving locality for adaptive irregular scientific codes. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, LCPC '00, pages 173–188, London, UK, UK, 2001. Springer-Verlag.
8. G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.
9. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.
10. M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 36:1–36:12, New York, NY, USA, 2009. ACM.
11. M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck. Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, College Park, Maryland, July 2002.
12. M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications*, 18(1):95–114, February 2004.
13. X. Sui, D. Nguyen, M. Burtscher, and K. Pingali. Parallel graph partitioning on multicore architectures. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 246–260, Berlin, Heidelberg, 2011. Springer-Verlag.
14. C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Comput.*, 26(12):1635–1660, Nov. 2000.
15. B. Wu, E. Z. Zhang, and X. Shen. Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 243–252, Washington, DC, USA, 2011. IEEE Computer Society.