Abstract

The METIS graph partitioning algorithm has three stages. The first stage, coarsening, takes a large graph, with vertices \( |V| \) and edges \( |E| \), and creates successively smaller graphs that are good representations of the original graph. The second stage partitions the small graph. The third stage, refinement, projects and refines the partition of the smaller graph onto the original graph. We present implementation details of a parallel coarsening algorithm that adds \( \Theta(|V|) \) parallelizable work to the original algorithm. On ygg.lcs.mit.edu, we add \( \Theta(|E|) \) serial work for optimal performance that limits our program to being 2.72 times faster on 8 processors than 1 processor, and 1.22 times faster on 8 processors than the original algorithm. We present issues with parallelizing refinement along with suggestions towards dealing with the issues.

1 Introduction

Consider a graph \( G_0 \) with sets of vertices \( V \) and edges \( E \). Edges and vertices may have weights. Partitioning into \( n \) partitions entails dividing the vertices of \( V \) into \( n \) disjoint subsets such that the sum of the vertex weights in each partition is roughly equal. The goal is to find a partition with a low edge-cut. The edge-cut of a partition is the sum of the weights of edges whose vertices lie in different partitions.

1.1 METIS Algorithm \([1]\)

METIS is a graph partitioning algorithm developed at the University of Minnesota by George Karypis. METIS consists of three stages: coarsening, initial partitioning, and refinement. The idea behind METIS is to create successively smaller graphs \( G_1, G_2, \ldots, G_k \) from \( G_0 \), partition \( G_k \) in very little time, and project the partition back onto the \( G_0 \), while refining at each step. Figure 1 illustrates the method.

METIS has three stages:

1. Coarsening. The original graph, \( G_0 \), is transformed into sequentially smaller graphs \( G_1, G_2, \ldots, G_k \) such that \( |V_0| > |V_1| > |V_2| > \ldots > |V_k| \). \( G_k \) is meant to be a good representation of \( G_0 \). Theoretically, a great partitioning of \( G_k \) represents a fairly good partitioning of \( G_0 \). Coarsening along with refinement constitutes roughly 95% of the serial runtime.

2. Initial Partition. \( G_k \) is partitioned. \( G_k \) is small enough such that this stage is completed very quickly. This partitioning constitutes roughly 5% of the serial runtime, thus its details are unimportant.

3. Refinement. The partition \( P_k \) is projected back onto \( P_{n-1}, \ldots, P_0 \). After each projection \( P_i \), the partitioning is refined using a greedy algorithm. Each partition \( P_i \) for \( 0 \leq i \leq k - 1 \) is refined before projecting to \( P_{i+1} \).

1.2 Graph Representation in METIS

In METIS, graphs are represented with arrays. Given a graph of \( v \) vertices, vertices are implicitly labeled 0, 1, 2, \ldots, \( v - 1 \). Information on edges is held in two arrays: \texttt{xadj} and \texttt{adjncy}. For all vertices \( i \), the list of vertices adjacent to \( i \) are listed in the array \texttt{adjncy} from elements \texttt{xadj}[i] inclusive to \texttt{xadj}[i + 1]) exclusive. Thus, for \texttt{xadj}[i] \( \leq j < \texttt{xadj}[i + 1] \), the value of \texttt{adjncy}[j] is connected to the vertex \( i \). The array \texttt{adjwgt} holds the edge weights, thus \texttt{adjwgt}[j] holds the edge weight of \texttt{adjncy}[j]. Figure 2 illustrates an example of a graph. Figure 3 is its associated representation.
**Figure 1:** Various phases of partitioning algorithm.

**Figure 2:** Sample graph.

**Figure 3:** Array representation of sample graph.
2 Implementation of Parallel Coarsening

As explained in Section 1, the coarsening stage of METIS consists of creating successive coarse representations of a graph. We focus on parallelizing the process of creating a coarse graph $G_1$ from graph $G_0$.

2.1 Coarsening Algorithm.

![Coarsening Algorithm Diagram](image)

The graph $G_1$ is created by collapsing as many pairs of connected vertices of $G_0$. A simple example is illustrated in Figure 4. Formally, coarsening consists of three stages: matching, mapping and creating.

The first stage is matching. We greedily find as many pairs of connected vertices as possible, calling each pair a match. Selecting the pair $(i, j)$, means $i$ is matched with $j$ and vice versa. Pairs of vertices are matched until no vertex with an unmatched adjacent vertex remains. If a vertex $i$ has no unmatched adjacent vertices, $i$ is matched with itself. The matching stage of Figure 4 is composed of finding pairs of vertices to circle.

Vertices are matched two ways. A vertex may be matched to a random unmatched adjacent vertex, (called random matching), or to the unmatched vertex with the heaviest edge weight (called heavy-edge matching). The information is stored in the array match. We require for all $i$ such that $0 \leq i < |V|$, $\text{match}[\text{match}[i]] = i$. A consistent matching satisfies this requirement, an inconsistent matching does not. The match array of Figure 4 is in Figure 5.

![Match Array](image)

The second stage is mapping. Each matched pairs of vertices of $G_0$ will represent one vertex of $G_1$. Thus, we need implicit vertex labels for the vertices of $G_1$. If $G_1$ is to have $v_1$ vertices, matched vertices $i$ and $j$ must map to a unique $k$ such that $0 \leq k < v_1$. The numbers outside matched pairs in Figure 4 are the mapping. The information is stored in the array cmap. We require $\text{cmap}[i] = \text{cmap[match[i]]}$. The resulting array cmap of the example above is illustrated in Figure 6.

![Cmap Array](image)

The third stage is creating the coarser graph. The mapping determines the number of vertices of $G_1$. An upper bound on the number of edges of $G_1$ is the number of edges in $G_0$. All information needed is in
the arrays described above. Thus all that is required is to iterate through the edges of \( G_0 \) and write the appropriate information for \( G_1 \).

2.2 Parallelizing of Creating the Coarse Graph.

We focus on parallelizing the final stage first because its runtime is the longest of the three stages. We cross two issues while parallelizing this stage. The first issue is the data representation. The current data representation does not lend itself to parallelization, thus we modify it. The second issue is massive data writing in parallel. We analyze data allocation and data placement policies to increase parallelism and scalability.

2.2.1 Modifying the Data Representation

If the array \( \text{adjncy} \) for \( G_1 \) may be written, then so can the array \( \text{adjwgt} \) for \( G_1 \). So we focus on issues filling \( \text{xadj} \) and \( \text{adjncy} \). Figure 3 the data representation of a graph. Suppose one processor is to write the adjacency list for vertex 0 and another processor is to write the adjacency list for vertex 2. We know the list for vertex 0 is at the beginning of the array \( \text{adjncy} \), but without a preprocessed \( \text{xadj} \) array, a processor will not know where in \( \text{adjncy} \) to write the information for vertex 2. Thus, \( \text{xadj} \) must be preprocessed to parallelize this stage.

Preprocessing \( \text{xadj} \) is expensive. To know \( \text{xadj}[i] \), we must know exactly how many edges vertices 0, 1, ..., \( i - 1 \) have. Thus, for each matched pair \((j, k)\), this requires iterating through all vertices adjacent to \( j \) and \( k \). The runtime of this additional work for all matched pairs is \( \Theta(|E|) \).

Calculating an upper bound on the number of edges a vertex will have is simpler. For matched vertices \((j, k)\) of \( G_0 \), the number of edges its mapped vertex of \( G_1 \) will have is at most the sum of the number of edges \( j \) and \( k \) have in \( G_0 \). This is easily calculated in \( \text{xadj} \). Needing to only calculate an upper bound, as will be shown in Section 2.3, requires \( \Theta(|V|) \) additional parallelizable work. Because \( E > V \), this is preferable.

Consider the following representation. For graph \( G_0 \) with \( v_0 \) vertices, let \( \text{xadj} \) have \( 2v_0 \) elements. Currently, for \( j \) such that \( \text{xadj}[i] \leq j < \text{xadj}[i+1] \), \( \text{adjncy}[j] \) is adjacent to \( i \). Instead, let for \( j \) such that \( \text{xadj}[2i] \leq j < \text{xadj}[2i+1] \), \( \text{adjncy}[j] \) is adjacent to \( i \). An illustrated example of the new representation is in Figure 7.

\[
\begin{align*}
\text{xadj:} & \quad 0 \quad 2 \quad 3 \quad 6 \quad 8 \quad 10 \ldots \\
\text{adjncy:} & \quad 1 \quad 3 \quad 0 \quad 4 \quad 2 \quad 1 \quad 5 \quad \ldots 
\end{align*}
\]

**Figure 7:** Modified graph representation with two pointers into adjncy array per vertex instead of one.

The advantage to this representation is \( \text{xadj}[2i] \) needs to be at least how many edges the vertices 0, 1, ..., \( i - 1 \) have. Preprocessing such an array is simple. We preprocess the array in the mapping stage (Section 2.3).

In this stage, \( T_1 = \Theta(|E|) \) and \( \infty = \Theta(\log |E|) \).

2.2.2 Data Writing

Consider the following piece of cilk code that allocates and fills a large amount of memory with some value. Because the current stage of the coarsening algorithm requires mostly data writing, analysis of a procedure such as this is necessary to achieve speedup in this stage.

\[
cilk\text{void fill}(\text{int } *\text{array}, \text{int val, int len})\{
\text{if(len <= (1<<18)))}\{
\]
memset(array, val, len*4);
} else {
    spawn fill(array, val, len/2);
    spawn fill(array+len/2, val, len-len/2);
}
}

enum { N = 200000000 };
int main(int argc, char *argv[]){
    x = (int *)malloc(N*sizeof(int));
    mt_fill(context, x, 25, N);/*******Print time taken***********/
    mt_fill(context, x, 25, N);/*******Print time taken***********/
}

Note we run the procedure fill twice. Table 1 has results achieved on ygg.lcs.mit.edu.

<table>
<thead>
<tr>
<th></th>
<th>1 proc</th>
<th>2 proc</th>
<th>4 proc</th>
<th>8 proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Run</td>
<td>6.94</td>
<td>5.8</td>
<td>5.3</td>
<td>5.45</td>
</tr>
<tr>
<td>Second Run</td>
<td>3.65</td>
<td>2.8</td>
<td>1.6</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Table 1: Experimental timing results of both runs of fill on ygg.lcs.mit.edu on array of length $2 \cdot 10^8$. All values are in seconds.

Note that the procedure fill scales better when being run on memory that has already been set to some value. As a result, before writing the coarse graph, we initially set the data that will represent the coarse graph to 0. This adds $\Theta(|E|)$ serial work to the coarsening algorithm. Reasons for this are not fully known. It is possible this action is not required on other multi-processor machines.

Data placement policies are also an issue. The default policy, first touch placement, allocates pages of memory on the memory module on which the process that first touches it is run. Note that in the above code, one processor allocates memory. Thus, with first touch placement, the memory is allocated on one memory module. When all other processors try accessing the memory, memory contention occurs.

A better policy is round-robin placement, which allocates pages of memory on all memory modules in a round-robin fashion. Table 2 has results achieved on ygg.lcs.mit.edu using round-robin placement.

<table>
<thead>
<tr>
<th></th>
<th>1 proc</th>
<th>2 proc</th>
<th>4 proc</th>
<th>8 proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Run</td>
<td>6.9</td>
<td>6.2</td>
<td>6.5</td>
<td>6.6</td>
</tr>
<tr>
<td>Second Run</td>
<td>4.0</td>
<td>2.6</td>
<td>1.3</td>
<td>.79</td>
</tr>
</tbody>
</table>

Table 2: Experimental timing results of both runs of fill on ygg.lcs.mit.edu on array of length $2 \cdot 10^8$ using round-robin placement. All values are in seconds.

We achieve best performance with round-robin placement run on memory that has already been set to some value.

2.3 Parallelizing Matching

Pseudo-code for random matching and heavy-edge matching are similar:

```c
foreach vertex u
    if(vertex u unmatched){
        find unmatched adjacent vertex v; /***whether v random or heaviest such vertex irrelevant**/
        match[u] = v;
```
match[v] = u;
}

Determinacy races are the only issue. If \( u \) tries to match with \( v \) while \( v \) concurrently tries to match with some other vertex \( w \), our match array may have inconsistencies. We choose to correct inconsistencies in the mapping stage as opposed to use locks to ensure inconsistencies do not occur. In the mapping stage, for each vertex \( i \), we check if match[match[i]] == i. If not, we set match[i] = i. This adds \( \Theta(|V|) \) parallelizable work, which is preferable to using locks. Using locks requires initialization and allocation overhead, along with meticulous coding to avoid deadlock contention. Lock contention, an inevitable result from using locks, limits speedup.

2.4 Parallelizing Mapping

Three tasks compose this stage: correcting inconsistencies in match array, creating cmap array, and preprocessing information for xadj of \( G_1 \). For now, let us assume the match array has no inconsistencies and xadj of \( G_1 \) need not be preprocessed. Consider the following pseudo-code that performs mapping and matching concurrently:

```c
if(node u unmatched){
    find unmatched adjacent node v;
    LOCK(maplock);
    match[u] = v;
    match[v] = u;
    cmap[u] = cmap[v] = num;
    num++;
    UNLOCK(maplock);
}
```

This code does not scale well on multiple processors because each processor will try to access the same lock. For this reason we separate the matching and mapping phases. Given a consistent match array, we can create cmap using a variant on a parallel prefix algorithm.

2.4.1 Parallel Prefix Algorithm

Given an array \( a \), of \( n \) elements, the goal is to modify \( a \) such that \( a[i] = \sum_{j=1}^{i} a[j] \). If all elements in \( a \) are initially 1, once modified, \( a[i] = i + 1 \) for \( 0 \leq i < n \). A trivial serial algorithm is presented below:

```c
void prefix_serial(int* array, int len){
    int i;
    for(i = 1; i < len; i++){
        array[i] += array[i-1];
    }
}
```

We present a parallel algorithm. We use a divide and conquer algorithm. For array \( a_0 \) and base case of size \( b \), run prefix_serial on blocks of size \( b \) and store the sum of each block in another array \( a_1 \). Recursively solve the parallel prefix of the array \( a_1 \). Lastly, increment all values in a block of size \( b \) by its associated value in \( a_1 \). An illustrated example on an array of length 9 and base case of size 3 is in Figure 8.

The parallel algorithm has twice as much work as the serial algorithm, but has critical path \( \Theta(\log n) \). The total work is \( \Theta(n) \).
2.4.2 Mapping Scheme

Our scheme for mapping is as follows.

1. Initially, for all $i$, let $cmap[i] = 1$ if $match[i] \leq i$, -1 otherwise.

2. Run a parallel prefix algorithm on values of $cmap$ that are not -1.

3. For all $i$ such that $cmap[i] = 1$, let $cmap[i] = cmap[match[i]]$.

2.4.3 Mapping Algorithm

Our algorithm uses a variant on the parallel prefix algorithm presented in Section 3.4.1. It makes three passes over the data. For clarity purposes, we assume a consistent $match$ array.

1. First Pass:
   - For blocks of size $b$, sequentially enumerate values of $cmap[i]$ for $i$ such that $i \leq match[i]$.
   - For all other $i$ in block, set $cmap[i] = -1$.
   - Store number of such $i$ in array $helper$.

2. Recursive Case: Run prefix algorithm on resulting array.

3. Second Pass:
   - For $i$ such that $i \leq match[i]$, increment $cmap[i]$ by its associated value in $helper$, thus setting the correct value for $cmap[i]$.

4. Third Pass:
   - For $i$ such that $i > match[i]$, set $cmap[i] = cmap[match[i]]$.

For a matched pair $(i, j)$, assuming $i < j$, define $i$ to be the major component and $j$ to be the minor component. We take a third pass over the data to set $cmap$ of minor components of matchings to avoid false sharing. During the second pass, a minor component has a $cmap$ value of -1. We cannot set the correct value of $cmap$ at that moment because the $cmap$ value of the major component may or may not be evaluated by another processor yet. Thus, to correctly set the value in the second pass, we would need it to be set along with the major component. That is, we not only increment $cmap[i]$, but then set $cmap[match[i]] = cmap[i]$. $cmap[match[i]]$ may be in another processor’s cache. Changing its value would invalidate that processor’s cache, thus leading to false sharing.

Figure 8: Example of parallel prefix.
2.4.4 Fixing match and preprocessing xadj

We can fix match in the first pass of our mapping algorithm. During the first pass, we check if \( i \leq \text{match}[i] \). Before doing so, we can check if \( \text{match}[\text{match}[i]] == i \). If not, we set \( \text{match}[i] = i \).

To preprocess xadj, we create another array \( \text{numedges} \). \( \text{numedges}[k] \) holds an upper bound on the number of edges vertex \( k - 1 \) of \( G_1 \) will hold. Set \( \text{numedges}[0] \) to 0. Say matched pair \((i, j)\) maps to \( k \). Thus \( \text{cmap}[i] = k \). In the second pass of our mapping algorithm, we set \( \text{numedges}[k] \) to be the number of edges of \( i \) in addition to the number of edges of \( \text{match}[i] \). This information can be found in constant time from xadj of \( G_0 \).

We run a prefix algorithm on \( \text{numedges} \) to attain the preprocessed list we desire. Thus, \( \text{numedges}[i] = \text{xadj}[2i] \) for xadj of \( G_1 \). The prefix algorithm we run can be serial because its total runtime is very small compared to the rest of the coarsening algorithm.

2.4.5 Mapping Conclusions

In serial coarsening, mapping and matching are done concurrently, thus this entire stage is additional necessary work to parallelize the algorithm. The work of this stage is \( \Theta(|V|) \) and the critical path is \( \Theta(\lg |V|) \).

3 Parallelizing Refinement

As explained in Section 1, refinement entails projecting a partition of a coarse graph, \( G_1 \) to a finer graph \( G_0 \) and refining the partition of \( G_0 \). We present the serial algorithm for refinement, issues with parallelizing refinement, and possible ideas to explore for solving them.

3.1 Refinement algorithm

The first step is projecting the partition of \( G_1 \) onto \( G_0 \). Say \( G_1 \) is partitioned into \( n \) disjoint subsets. For \( i \) such that \( 0 \leq i < |V_1| \), we have \( \text{where}[i] = x \), where \( 0 \leq x < n \). \( x \) represents the partition vertex \( i \) belongs to. \( \text{where} \) holds the partitioning information. Given matched pair \((i, j)\) of \( G_0 \) maps to \( k \) of \( G_1 \), projecting the partition entails setting \( \text{where}[i] \) and \( \text{where}[j] \) to \( \text{where}[k] \).

The second step is refining the partition of \( G_0 \). Define a boundary vertex to be connected to some vertex in a different partition and an internal vertex to be connected to only nodes in the same partition. We iterate over all boundary vertices, testing if moving the vertex to another partition lowers the edge-cut. If so, we move the vertex to the partition that reduces the edge-cut the most. We iterate over the boundary vertices a constant number of times.

To facilitate this step, during projection, we preprocess the list of boundary vertices into an array. For each boundary vertex, we calculate the sum of the weights of edges that lie in its partition, called its internal degree. For each other partition the boundary vertex is connected to, we calculate the sum of the weights of edges that lie in that partition, called the external degree. Each boundary vertex has one internal degree and up to \( n - 1 \) external degrees. These degrees are stored in an array.

3.2 Algorithmic Parallelizing Issues

Projecting the partition entails only data writing and very little computation. A straightforward parallelization of this step seems feasible using the same methods in parallelizing the graph writing stage of coarsening.

Parallel refinement can lead to increasing the edge-cut. Consider Figure 9. Concurrently moving nodes 2 and 3 leads to an increase in the edge-cut.

To ensure moving vertex \( i \) does not increase the edge-cut, we must lock \( i \) along with all adjacent vertices. This creates huge locking overhead and is not recommended. Any parallel refinement solution should assume such cases are infrequent (assume \(|V| >> p \) where \( p \) is the number of processors). Should something like this
occur, not only will its effect to the edge-cut most likely be minimal, but future refinements of the graph will likely correct the error. Thus, we suspect allowing such errors are tolerable.

3.3 Implementation Parallelizing Issues

The current implementation lends itself very well for serial execution, but has issues for parallelization.

3.3.1 Maintaining List of Boundary Vertices

Currently, boundary vertices are stored in an array. When a vertex’s partition changes, all previously internal adjacent vertices become boundary vertices. We wish to check these vertices in future iterations. Appending them to the end of an array will be difficult because other processors will be trying to append other vertices as well, causing concurrency issues. This seems only to be resolved with a unique lock. Once again, this will cause a bottleneck and limit parallelism.

If possible, we would like to be able to maintain some data structure that holds all boundary vertices such that it may be iterated in parallel and appending is cheap. A simple array does not satisfy these requirements. When the number of partitions is small, in practice, the number of boundary vertices will be significantly less than the number of total vertices. If possible, we would like to avoid iterating through all vertices.

3.3.2 Keeping Degrees Updated

Examples such as Figure 9 create determinacy races for external and internal degrees of vertices. Inaccurate degree information may prevent necessary vertex moves or lead to harmful vertex moves. We do not know how often this might occur during refinement. We wish to avoid inaccuracies in degree information because correcting them would be expensive. We may use lock each degree before modifying it, but this entails allocating and initializing $|V|n$ locks.

Degrees are created to save repeated computation over repeated iterations. An efficient parallel implementation of refinement that uses these degrees is currently open to research.

4 Timing Results

Table 3 presents timing results on the total work and critical path of the parallel stages of coarsening a 1600x1600 grid. Because critical path is being measured, the values measured are higher than the values in practice. Still, they demonstrate potential for parallelism.
Table 4 presents timing results of coarsening a 1600 by 1600 grid.

One experiment uses first-touch placement and memset all arrays to 0 before operating. Table 5 breaks down runtime of each stage.

Another experiment uses round-robin placement and memset all arrays to 0 before operating. Table 6 breaks down runtime of each stage.

Note that despite having more total work than first-touch placement, round-robin placement achieves better speedup and as good performance on multiple processors.

5 Conclusion

We present implementation details of a parallel coarsening algorithm. We add $\Theta(|V|)$ parallelizable work to the coarsening stage to enable parallelization. To run efficiently on ygg.lcs.mit.edu we add $\Theta(|E|)$ serial work so that the algorithm scales more efficiently. The $\Theta(|E|)$ work severely limits scalability.

We present issues with parallelizing refinement and present paths for future research towards handling the issues. For a Masters of Engineering thesis, I propose to develop and test an efficient parallel refining algorithm along with explore optimizations to the current coarsening algorithm.

References


\[
\begin{array}{|l|l|l|l|}
\hline
& T_1 & T_\infty & T_1/T_\infty \\
\hline
\text{matching} & 1.82 \text{ s} & 517 \text{ us} & 3111 \\
\text{mapping} & 1.44 \text{ s} & 1.09 \text{ ms} & 692 \\
\text{coarsening} & 2.67 \text{ s} & 1.7 \text{ ms} & 1565 \\
\hline
\end{array}
\]

Table 3: Total work and critical path timing results of a 1600x1600 grid.
<table>
<thead>
<tr>
<th>serial runtime</th>
<th>1 proc</th>
<th>2 proc</th>
<th>4 proc</th>
<th>8 proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1600x1600</td>
<td>3.57</td>
<td>6.35</td>
<td>4.05</td>
<td>2.92</td>
</tr>
</tbody>
</table>

**Table 4:** Experimental timing results of coarsening a 1600 x 1600 grid on ygg.lcs.mit.edu. All values are in seconds.

<table>
<thead>
<tr>
<th>memset for matching</th>
<th>1 proc</th>
<th>2 proc</th>
<th>4 proc</th>
<th>8 proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>matching</td>
<td>.39</td>
<td>.39</td>
<td>.39</td>
<td>.39</td>
</tr>
<tr>
<td>mapping</td>
<td>.87</td>
<td>.44</td>
<td>.27</td>
<td>.16</td>
</tr>
<tr>
<td>numedges</td>
<td>1.07</td>
<td>.56</td>
<td>.33</td>
<td>.24</td>
</tr>
<tr>
<td>memset for writing</td>
<td>.95</td>
<td>.95</td>
<td>.95</td>
<td>.95</td>
</tr>
<tr>
<td>writing coarse graph</td>
<td>2.51</td>
<td>1.30</td>
<td>.74</td>
<td>.41</td>
</tr>
</tbody>
</table>

**Table 5:** Experimental timing results of coarsening a 1600x1600 grid on ygg.lcs.mit.edu using first-touch placement. All values are in seconds.

<table>
<thead>
<tr>
<th>memset for matching</th>
<th>1 proc</th>
<th>2 proc</th>
<th>4 proc</th>
<th>8 proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>matching</td>
<td>.43</td>
<td>.43</td>
<td>.43</td>
<td>.43</td>
</tr>
<tr>
<td>mapping</td>
<td>.99</td>
<td>.49</td>
<td>.28</td>
<td>.15</td>
</tr>
<tr>
<td>numedges</td>
<td>1.13</td>
<td>.59</td>
<td>.36</td>
<td>.22</td>
</tr>
<tr>
<td>memset for writing</td>
<td>1.08</td>
<td>1.08</td>
<td>1.08</td>
<td>1.08</td>
</tr>
<tr>
<td>writing coarse graph</td>
<td>2.65</td>
<td>1.39</td>
<td>.70</td>
<td>.38</td>
</tr>
</tbody>
</table>

**Table 6:** Experimental timing results of coarsening a 1600x1600 grid on ygg.lcs.mit.edu using round-robin placement. All values are in seconds.