6.895 Term Project Proposal

Zardosht Kasheff

October 8, 2003

Proposal: Parallelizing METIS

I plan to design and implement a parallelized version of the graph partitioning algorithm, METIS, using CILK.

Description of METIS

Graph Partitioning. Consider a graph G with vertices V and edges E. Vertices and edges can have weights associated with them. A ***partition*** of size n divides the vertices into n disjoint subsets. The ***edge cut*** is the number of edges whose vertices lie in different subsets. METIS aims to find a low edge-cut.

Algorithm. METIS performs in three stages: coarsening, partitioning, and refinement.

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

Partitioning. G_n is partitioned. G_n 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.

Refinement. The partition P_n is projected back onto $P_{n-1}, \ldots, P_0$. After each projection P_i, the partitioning is refined using a greedy algorithm.

Parallelizing tasks

Coarsening.

The process of coarsening G_i to G_{i-1} can be parallelized. Coarsening involves two stages: matching and creating.

Matching. The goal of matching is given G_{i-1}, define the vertices of G_i. Every vertex in G_i maps to one or two vertices of G_{i-1}. The process of ***matching*** is to randomly select pairs of *connected* vertices in G_{i-1} to map to a vertex we create for G_i. If we select connected vertices j and k to map to a vertex in G_i, we say the vertices j and k are ***matched***. If all adjacent vertices of vertex j are already matched, we match j with itself.

Each processor picks a vertex v at random and checks if it is unmatched. If so, vertices adjacent to v are searched until another unmatched vertex is found, and the two are matched. If no such vertex is found, v is matched with itself. Obvious concurrency issues have been dealt with and implemented. The process is continues until all vertices are matched.

Creating G_i. Given matched vertices (v_a, v_b) of G_{i-1} are mapped to v_c of G_i, adjacent nodes of v_c as follows. We iterate through adjacent vertices of v_a and v_b, find their mapped values in G_i, and define these to be the adjacent vertices of v_c. Processors can do subsets of vertices of G_i independently. No concurrency issues arise.

Initial Partitioning

Due to the small runtime of this portion, this task will not be parallelized.

Projection and Refinement.

For all vertices of G_i, if matched vertices (v_a, v_b) of G_i map to v_c of G_{i+1}, and v_c belongs to subset j of P_{i+1}, then v_a and v_b initially belong to subset j of P_i. This defines a ***projection***. This can be done in parallel with no concurrency issues because no data is modified. Data is only written.

Refinement involves iteratively attempting to move each vertex from its current subset to another subset if and only if the move results in a decrease in the edge cut. Attempts are made until no possible movements result in a decrease in edge-cut. Processors can attempt to move vertices in parallel, but concurrency issues arise.

No parallelization has been implemented as of yet.

Tasks for Project

Tasks Completed

1. Designed, implemented and tested parallelized coarsening.

2. Designed initial version of parallelized refinement.

Tasks Remaining

1. Obtain speedup for parallelized coarsening.

   Currently, although logically sound, coarsening does not demonstrate any speedup on more processors. The reason is that currently data cannot be written in parallel.

METIS is mostly a comparison based algorithm. The majority of computation is memory operations. For instance, creating G_i mentioned above involves writing data to a large array. Unfortunately, I have been unable to scale algorithms composed of *only* memory operations. Take the following piece of cilk code:

```
cilk void fill(int *array, int val, int len){
  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);
    sync;
  }
}

enum { N = 200000000 };
int main(int argc, char *argv[]){
  int *x;
  x = (int *)malloc(N*sizeof(int));
  spawn fill(x, 25, N);
}
```

This code fills an array in parallel, yet shows practically no speedup in runtime when run on multiple processors. If this code can be parallelized, then coarsening can as well. The reason is almost certainly in the data placement resulting in malloc. That is, *all* the data is located close to one processor. Other processors trying to access the data is causing a bottleneck. This task is currently being worked on.

2. Implement, test, and most likely redesign parallelized refinement.

   The algorithm is non-trivial because many concurrency issues arise. Moving two adjacent vertices concurrently may result in an increase in edge cut, even if moving only one would result in a decrease. A naive solution would be to lock all adjacent vertices when attempting to move a given vertex. This results in a large locking overhead. A better solution shall be investigated once the task above is completed.

   Also, much space is accessed and allocated sequentially. Devising a way to not have to allocate all memory sequentially so that processors can write to memory in parallel, while causing only a small increase in total runtime of the serial algorithm, is a challenge.

Scope and Backup.

Rectifying the first task has been challenging, perhaps because of my inexperience with systems. I believe the problem can be solved soon, but I cannot be sure. As a backup, I know I can have the second task completed by the end of the term. At worst, and this is if all else fails, I will implement a naive parallelization of refinement.

Papers Read.

1. George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. March 1998.

2. George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. March 1998.

3. George Karypis and Vipin Kumar. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. March 1998.

4. Jeffrey S. Gibson. Memory Profiling on Shared Memory Multiprocessors. July 2003.