Parallelizing METIS

A Graph Partitioning Algorithm

Zardosht Kasheff
- **Goal**: Partition graph into $n$ equally weighted subsets such that edge cut is minimized.
- **Edge-cut**: Sum of weights of edges whose nodes lie in different partitions.
- **Partition weight**: Sum of weight of nodes of a given partition.
METIS Algorithm

95% of runtime is spent on Coarsening and Refinement
Graph Representation

All data stored in arrays
- xadj holds pointers to adjncy and adjwgt that hold connected nodes and edge weights
- for j, such that xadj[i] <= j < xadj[i+1]:
  adjncy[j] is connected to i,
  adjwgt[j] is weight of edge connecting

```
xadj: 0 2 5 7 ...
```
```
adjncy: 1 3 0 4 2 1 5 ...
```
```
adjwgt: 2 2 2 2 2 2 2 ...
```
Coarsening Algorithm

Matching

Writing Coarse Graph

Node Labels of Coarser Graph
Coarsening: Writing Coarse Graph

Issue: Data Representation

```plaintext
Node 0  Node 1  Node 2

xadj: 0 2 5 7 ...

adjncy: 1 3 0 4 2 1 5 ...
```
Coarsening: Writing Coarse Graph

Issue: Data Representation

Before:
for j, such that
xadji <= j < xadj[i+1]:
adjncy[j] connected to i.

After:
for j, such that
xadj[2i] <= j < xadj[2i+1]:
adjncy[j] connected to i.
Coarsening: Writing Coarse Graph

Issue: Data Representation

- Now, only need upper bound on number of edges per new vertex
  - If match\((i,j)\) map to \(k\), then \(k\) has at most \(|\text{edges}(i)| + |\text{edges}(j)|\)
  - Runtime of preprocessing xadj only \(O(|V|)\).
Coarsening: Writing Coarse Graph

Issue: Data writing

- Writing coarser graph involves writing massive amounts of data to memory
  - \( T_1 = O(|E|) \)
  - \( T_\infty = O(\lg |E|) \)
  - Despite parallelism, little speedup
Coarsening: Writing Coarse Graph

Issue: Data writing

Example of filling in array:

Cilk void fill(int *array, int val, int len) {
    if (len <= (1<<18)) {
        memset(array, val, len*4);
    } else {
        /*************************************************************************/
    }
}

enum {N = 200000000};
int main(int argc, char* argv[]) {
    x = (int*)malloc(N*sizeof(int));
    mt_fill(context, x, 25, N); gettimeofday(&t2); print_tdiff(&t2, &t1);
    mt_fill(context, x, 25, N); gettimeofday(&t3); print_tdiff(&t3, &t2);
}
Coarsening: Writing Coarse Graph

Issue: Data writing

- Parallelism increases on second fill

After first malloc, we fill array of length $2 \times 10^8$ with 0's:

| ? | ? | ? | ? | ? | ? | ? | ? | ... | ⇒ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |

1 proc: 6.94s
2 proc: 5.8s  speedup: 1.19
4 proc: 5.3s  speedup: 1.30
8 proc: 5.45s speedup: 1.27

Then we fill array with 1's:

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | ⇒ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... |

1 proc: 3.65s
2 proc: 2.8s  speedup: 1.30
4 proc: 1.6s  speedup: 2.28
8 proc: 1.25s speedup: 2.92
Coarsening: Writing Coarse Graph
Issue: Data writing

- Memory Allocation
  - Default policy is First Touch:
    - Process that first touches a page of memory causes that page to be allocated in node on which process runs

Result: Memory Contention
Coarsening: Writing Coarse Graph

Issue: Data writing

- Memory Allocation
  
  - Better policy is Round Robin:
    
    - Data is allocated in round robin fashion.

Result:
More total work but less memory contention.
Coarsening: Writing Coarse Graph

Issue: Data writing

- Parallelism with round robin placement on ygg.

After first malloc, we fill array of length $2 \times 10^8$ with 0's:

| ? | ? | ? | ? | ? | ? | ? | ? | ... | ⇒ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| 1 proc: 6.94s | 1 proc: 6.9s |
| 2 proc: 5.8s | speedup: 1.19 | 2 proc: 6.2s | speedup: 1.11 |
| 4 proc: 5.3s | speedup: 1.30 | 4 proc: 6.5s | speedup: 1.06 |
| 8 proc: 5.45s | speedup: 1.27 | 8 proc: 6.6s | speedup: 1.04 |

Then we fill array with 1's:

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | ⇒ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| 1 proc: 3.65s | 1 proc: 4.0s |
| 2 proc: 2.8s | speedup: 1.3 | 2 proc: 2.6s | speedup: 1.54 |
| 4 proc: 1.6s | speedup: 2.28 | 4 proc: 1.3s | speedup: 3.08 |
| 8 proc: 1.25s | speedup: 2.92 | 8 proc: .79s | speedup: 5.06 |
Coarsening: Matching

Matching          Writing  Coarse Graph

match:  3  2  1  0  5  4  7  6  8

cmap:    3  1  1  3  4  4  0  0  2

numedges: 0  5  5  2  5  7
Coarsening: Matching
Phase: Finding matching

- Can use divide and conquer
  - For each vertex:
    ```
    if(node u unmatched) {
      find unmatched adjacent node v;
      match[u] = v;
      match[v] = u;
    }
    ```
  - Issue: Determinacy races. What if nodes $i,j$ both try to match $k$?
  - Solution: We do not care. Later check for all $u$, if
    $match[match[u]] = u$. If not, then set $match[u] = u$. 
Coarsening: Matching
Phase: Finding mapping

- Serial code assigns mapping in order matchings occur. So for:

```
Matchings occurred in following order:
1) (6,7)
2) (1,2)
3) (8,8) /*although impossible in serial code, error caught in last minute*/
4) (0,3)
5) (4,5)
```
Coarsening: Matching
Phase: Finding mapping

- Parallel code cannot assign mapping in such a manner without a central lock:
  - For each vertex:
    ```java
    if(node u unmatched) {
        find unmatched adjacent node v;
        LOCKVAR;
        match[u] = v;
        match[v] = u;
        cmap[u] = cmap[v] = num;
        num++;
        UNLOCK;
    }
    
    This causes bottleneck and limits parallelism.
Coarsening: Matching Phase: Finding mapping

- Instead, can do variant on parallel-prefix
  
  - Initially, let $\text{cmap}[i] = 1$ if $\text{match}[i] \geq i$, -1 otherwise:

  $\text{cmap:} \begin{array}{cccccccc}
  1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 \\
  \end{array}$

  - Run prefix on all elements not -1:

  $\text{cmap:} \begin{array}{cccccccc}
  0 & 1 & -1 & -1 & 2 & -1 & 3 & -1 & 4 \\
  \end{array}$
Coarsening: Matching
Phase: Finding mapping

- Correct all elements that are -1:

- We do this last step after the parallel prefix to fill in values for cmap sequentially at all times. Combining the last step with parallel-prefix leads to false sharing.
Coarsening: Matching
Phase: Parallel Prefix

\[ -T_1 = 2N \]

\[ -T_{\text{infinity}} = 2 \log N \text{ where } N \text{ is length of array.} \]
Coarsening: Matching
Phase: Mapping/Preprocessing xadj

- Can now describe mapping algorithm in stages:
  - First Pass:
    - For all \( i \), if \( \text{match}[	ext{match}[i]] \neq i \), set \( \text{match}[i] = i \)
    - Do first pass of parallel prefix as described before
  - Second Pass:
    - Set \( \text{cmap}[i] \) if \( i \leq \text{match}[i] \),
      - set \( \text{numedges}[\text{cmap}[i]] = \text{edges}[i] + \text{edges}[\text{match}[i]] \)
  - Third Pass:
    - Set \( \text{cmap}[i] \) if \( i > \text{match}[i] \)
- Variables in blue mark probable cache misses.
Coarsening: Preliminary Timing Results

On 1200x1200 grid, first level coarsening:

Serial:
Matching: .4s
Writing Graph: 1.2s

Parallel:
1proc: 2 proc 4 proc 8 proc
memsetting for matching: .17s .23s .16s .11s
matching: .42s .31s .17s .16s
mapping: .50s .64s .20s .13s
memsetting for writing: .44s .52s .52s .52s
coarsening: 1.2s .71s .44s .24s

Round Robin Placement:
1proc: 2 proc 4 proc 8 proc
memsetting for matching: .20s .27s .16s .09s
matching: .51s .35s .20s .13s
mapping: .64s .64s .20s .13s
memsetting for writing: .52s .52s .52s .52s
coarsening: 1.42s .75s .39s .20s