

Graphs, Parallel Algorithms and Approximation

Alicia A. Thorsen, Ph.D. Candidate

Department of Computer Science
Michigan Technological University
athorsen@mtu.edu

Committee Members:

Dr. Phillip Merkey

Dr. Steven Seidel

Dr. Charles Wallace

Dr. Fredrik Manne, University of Bergen, Norway



Combinatorial Scientific Computing (CSC)

- ❑ Interdisciplinary field of CS
- ❑ Focuses on combinatorial problems found in CS&E
- ❑ Applications in:
 - ❑ Mesh generation
 - ❑ Sparse linear systems
 - ❑ Computational chemistry
 - ❑ Bioinformatics
 - ❑ Statistical physics
- ❑ Need for highly scalable parallel algorithms

Parallel Graph Processing

- ❑ Graph algorithms are central to many areas of CSC
 - ❑ Matching
 - ❑ Coloring
- ❑ Difficult to write parallel programs
 - ❑ Computation is data-driven
 - ❑ Hard to partition data
 - ❑ High data-access to computation ratio
 - ❑ More exploring the graph than computation
 - ❑ Performance dominated by linked loads
- ❑ Graph problems are highly unstructured

Parallel Programming Paradigms

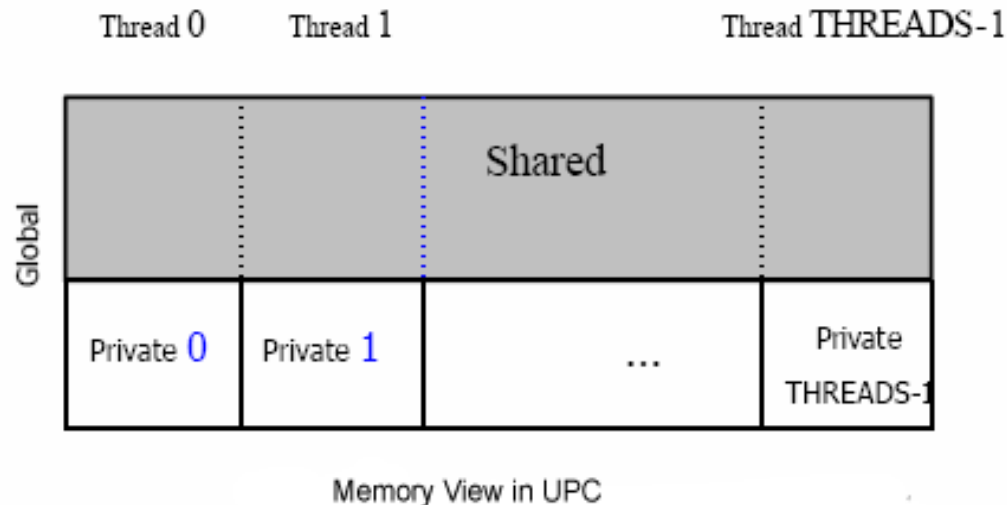
- Message passing
 - Widely used
 - Private memory
 - Explicit communication using messages
 - Suitable for regular problems
 - Structured computation & communication
 - Data can be easily partitioned
 - Not suitable for graph problems
 - Explicit communication required is non-intuitive
 - Code is difficult to develop and maintain

Parallel Programming Paradigms cont'd

- Shared Memory
 - Global memory
 - Implicit communication through reads & writes
 - Suitable for graph problems
 - Implicit partitioning of data
 - Implicit communication
 - High correlation with sequential algorithms
 - Hidden overheads from communication
 - Difficult to analyze
 - Affects scalability

Partitioned Global Address Space Programming Model

- ❑ Aims to address problems with shared memory
- ❑ Partitions the shared space so that a portion is local to each processor
- ❑ Allows programmers to exploit data locality
- ❑ One such language is UPC

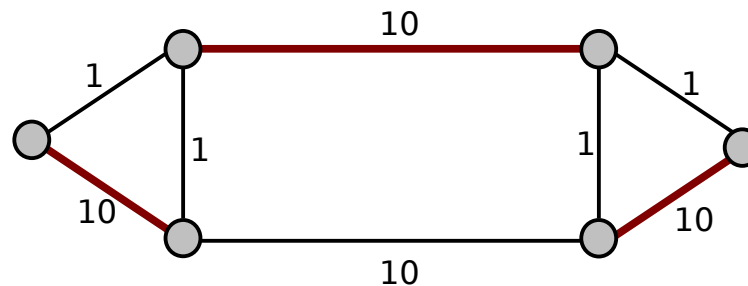


Research Goals

- ❑ Develop highly scalable parallel graph algorithms for CSC problems
 - ❑ Weighted Matching
 - ❑ Vertex Coloring
- ❑ Exploit both fine-grained and coarse-grained communication
- ❑ Evaluate algorithms with respect to a performance model for anticipated UPC platforms

Weighted Matching Problem

- Given a weighted graph $G(V,E)$
 - Find a set M of non-adjacent edges with maximum weight
- Applications
 - Scheduling
 - Network routing
 - Load balancing



Weight of matching is 30

Exact Algorithms

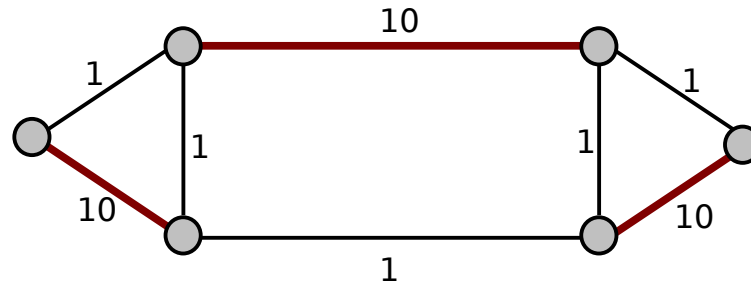
- Best sequential algorithm
 - $O(n^2m)$ - Edmonds, 1965
 - Optimized $O(nm + n^2 \log n)$
- No practical polynomial-time parallel algorithms
 - Algorithms developed for PRAM model
 - Requires exponential processors

Approximation Algorithms

- Greedy algorithm
 - ▢ Sort edges, iteratively choose heaviest & discard neighbors
 - ▢ $O(m \log n)$
 - ▢ $\frac{1}{2}$ approximation ratio
- Preis' LAM algorithm
 - ▢ Finds same matching as greedy without sorting
 - ▢ $O(m)$
 - ▢ Uses locally dominant edges
 - ▢ Edges that are heavier than their incident edges
- Hoepman's distributed protocol
 - ▢ Distributed version of Preis' LAM algorithm
 - ▢ Average case $O(\log m)$

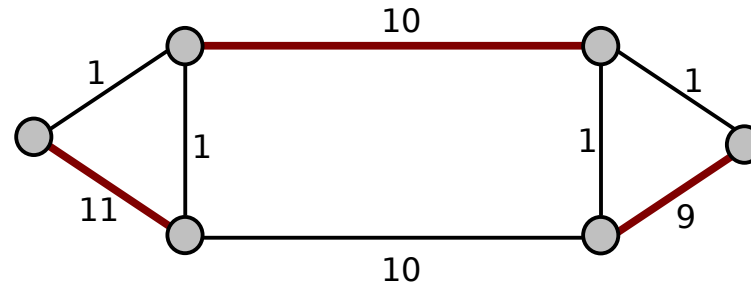
Dominating Edges

- Initially dominant



Weight of matching is 30

- Subsequently dominant



Weight of matching is 30

Sequential Manne-Bisseling Algorithm

```
foreach u ∈ V
    mate(u) = heaviest-available(u)
    if (mate(mate(u)) == u)
        M = M ∪ {(u, mate(u))}
        Q = Q ∪ {u, mate(u)}

while Q ≠ ∅
    Remove u from Q
    foreach v ∈ V s.t. mate(v) == u and v ∉ M
        mate(v) = heaviest-available(v)
        if (mate(mate(v)) == v)
            M = M ∪ {(v, mate(v))}
            Q = Q ∪ {v, mate(v)}

return M
```

Parallel Manne-Bisseling Algorithm

- Find initially dominant edges

```
foreach u ∈ myV
    mate(u) = heaviest-available(u)
synchronize

foreach u ∈ myV
    if (mate(mate(u)) == u)
        M = M ∪ {(u, mate(u))}

        myQ = myQ ∪ {u}
        if (mate(u) ∈ myV)
            myQ = myQ ∪ {mate(u)}

synchronize
```

Parallel Manne-Bisseling Algorithm

- Find subsequently dominant edges

```
while myQ ≠ ∅
    Remove u from myQ
    foreach v ∈ V s.t. mate(v) == u and v ∉ M
        if (v ∉ myV)
            notify owner(v) to add u to its Q
        else
            mate(v) = heaviest-available(v)
            if (mate(mate(v)) == v)
                M = M ∪ {(v, mate(v))}
                myQ = myQ ∪ {v, mate(v)}

synchronize
return M
```

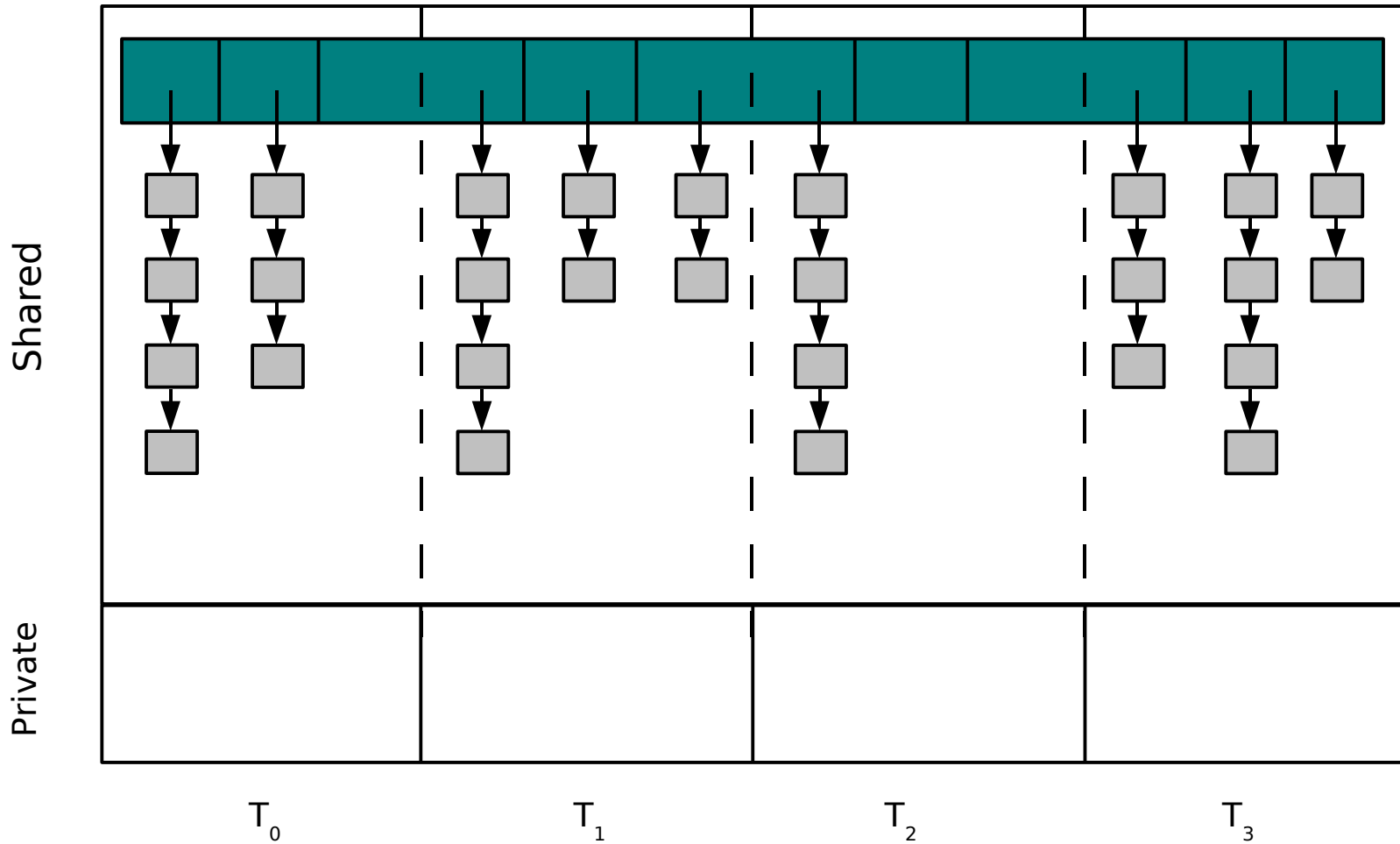
MPI Implementation

- ❑ Create ghost vertices for boundary edges
 - ❑ Run sequential algorithm on local portion of graph
 - ❑ Update owners of ghost vertices
- ❑ Uses Bulk Synchronous Processing (BSP) model
 - ❑ Compute - Communicate supersteps
- ❑ Problems
 - ❑ Keeping track of ghost vertices
 - ❑ Efficiently packaging messages
 - ❑ Lots of arithmetic

UPC Implementation

- Graph distribution
 - Use Metis graph partitioning library
 - Returns a vertex based partitioning
 - Create an array of vertex nodes
 - Max # of vertices in a partition is the blocking factor
 - Add dummy vertices
 - Renumber vertices
- Each vertex node is a struct which contains
 - Vertex id
 - Desired Mate
 - Flag indicating if it has been matched with its mate
 - Linked list of neighbors
 - Each neighbor node contains
 - Id of vertex and weight on incident edge
 - Edges are repeated

Data Structure



UPC Implementation

- Synchronous implementation
 - Similar to MPI compute – communicate supersteps
 - Processes write all notifications in one step
 - Coarse grained communication
 - Pros:
 - Program runs smoothly and gives correct results
 - Cons:
 - Requires 2 barriers
 - Processes wait even when they have work to do
 - Scalability is bad on local machines
 - Is this the way we want to write UPC applications?

UPC Implementation

- Asynchronous implementation
 - ▢ Processes write notifications as they are discovered
 - ▢ Fine-grained communication
 - ▢ Pros:
 - ▢ Easy to program with atomic ops
 - ▢ Program is simpler, looks a lot like sequential algorithm
 - ▢ Theoretically should be faster
 - ▢ Cons:
 - ▢ Doesn't work! Incorrect results.
 - ▢ Process spinning on a variable prevents an atomic write
 - ▢ We promote using UPC this way but no actual support

Performance Model

- Initial mates precomputed in graph input stage
 - ▢ Finding initially dominant edges is embarrassingly parallel
 - ▢ Performance is dominated by small set of remote reads
 - ▢ Influenced by quality of the partition
 - ▢ No remote writes
- Finding subsequently dominant edges is harder
 - ▢ Many remote reads
 - ▢ Remote writes can be either fine or coarse grained
 - ▢ Synchronization

What Can Help?

- Using cache
 - But what about atomic ops?
 - Locks are bad
- True shared memory machine
 - No Cray X1
- Theoretical machine of the future?
 - Aka anticipated UPC platforms
- Asynchronous version
 - Data structures for the “spin lock” problem?