ParaMeter: A profiling tool for amorphous data-parallel applications
Now what?

- Simplest thing to do: write your program using the Galois model, then analyze the parallelism
“Available Parallelism”

• How many active nodes can be processed in parallel over time

• Profile the algorithm, not the system
  • Disregard communication/synchronization costs, run-time overheads and locality concerns

• Can expose common structure between algorithms
Example: Spanning Tree

- Problem: given an unweighted graph and a starting node, construct a spanning tree rooted at that node.
Example: Spanning Tree

• Problem: given an unweighted graph and a starting node, construct a spanning tree rooted at that node
Example: Spanning Tree

- Problem: given an unweighted graph and a starting node, construct a spanning tree rooted at that node
Algorithm + Data Structure

- Algorithm written using Galois foreach operator to represent worklist iteration
- Specifies whether worklist is ordered or unordered
- Data structures implemented in terms of graph ADT
- Provided by Galois class library
Algorithm + Data Structure

• Algorithm
  • Choose node from worklist
  • Iterate over neighbors
  • If neighbor not in ST, add edge to ST, mark node and add to worklist

• Data structures
  • Graph is a “local computation” graph
  • Spanning tree is a Set of edges

Graph \texttt{graph} = \textit{read graph from file}
Node \texttt{startNode} = \textit{pick random node from graph}
\texttt{startNode}.\texttt{inSpanningTree} = \texttt{true}
Worklist \texttt{worklist} = \textit{create worklist containing startNode}
List \texttt{result} = \textit{create empty list}

\textbf{foreach} \texttt{src : worklist}
  \textbf{foreach} \texttt{Node dst : src.neighbors}
    \textbf{if not} \texttt{dst.inSpanningTree}
      \texttt{dst.inSpanningTree} = \texttt{true}
      \texttt{Edge edge} = \texttt{new} \texttt{Edge(src,dst)}
      \texttt{result.add(edge)}
      \texttt{worklist.add(dst)}
Finding Parallelism

- Active nodes: nodes on the frontier of ST (those that have just been added)
- Neighborhood: the immediate neighbors of the active node
- Neighborhoods are small → Can expand ST from several places at the same time
Finding Parallelism

- Active nodes: nodes on the frontier of ST (those that have just been added)
- Neighborhood: the immediate neighbors of the active node
- Neighborhoods are small → Can expand ST from several places at the same time
Finding Parallelism

- Active nodes: nodes on the frontier of ST (those that have just been added)
- Neighborhood: the immediate neighbors of the active node
- Neighborhoods are small → Can expand ST from several places at the same time
- Parallelism can be seen after the fact in structure of tree
How Much Parallelism is There?

• What would happen if we ran the program on an infinite number of processors?
  • Assume every activity takes the same amount of time (one “step”)
  • Assume perfect knowledge of neighborhood and ordering constraints
• How many steps would it take to execute the program?
• How many activities would we be able to execute in one step?
  • Computing upper bound on parallelism
A guess:

Available Parallelism

Time
A guess:

Available Parallelism vs. Time
A guess:
Demo:
Spanning tree

Spanning tree over 120K node graph
So how does this work?
Measuring Parallelism

- Represent program as a DAG
  - Nodes: operations
  - Edges: dependences
- Execution strategy
  - Assume operations take unit time
  - Execute “greedily” – process all ready operations in each step
- Parallelism profile: # of operations executed in each step
Measuring Parallelism

- Represent program as a DAG
  - Nodes: operations
  - Edges: dependences
- Execution strategy
  - Assume operations take unit time
  - Execute “greedily” – process all ready operations in each step
- Parallelism profile: # of operations executed in each step
Measuring Parallelism

- Represent program as a DAG
  - Nodes: operations
  - Edges: dependences
- Execution strategy
  - Assume operations take unit time
  - Execute “greedily” – process all ready operations in each step
- Parallelism profile: # of operations executed in each step
Measuring Parallelism

• Represent program as a DAG
  • Nodes: operations
  • Edges: dependences

• Execution strategy
  • Assume operations take unit time
  • Execute “greedily” – process all ready operations in each step

• Parallelism profile: # of operations executed in each step
Amorphous Data
Parallel Algorithms

- No notion of ordering
  - Represent program as a graph, not a DAG
- Execution: choose set of independent elements to process
- Different scheduling choices lead to different amounts of parallelism
  - Even with unlimited resources!
Amorphous Data
Parallel Algorithms

- No notion of ordering
- Represent program as a graph, not a DAG
- Execution: choose set of independent elements to process
- Different scheduling choices lead to different amounts of parallelism
- Even with unlimited resources!
Greedy scheduling

- Finding schedule to maximize parallelism is \( NP \)-hard

⇒ Solution: Schedule greedily

- Attempt to maximize work done in current step
- Choose a maximal independent set in conflict graph
Incremental Execution

- Conflict graph can change during execution
- New work generated
- New conflicts
- Cannot perform scheduling \textit{a priori}

⇒ Solution: execute in stages, recalculate conflict graph after each stage
Incremental Execution

- Conflict graph can change during execution
- New work generated
- New conflicts
- Cannot perform scheduling \textit{a priori}

\textbf{Solution:} execute in stages, recalculate conflict graph after each stage
Incremental Execution

- Conflict graph can change during execution
- New work generated
- New conflicts
- Cannot perform scheduling \textit{a priori}

$\Rightarrow$ Solution: execute in stages, recalculate conflict graph after each stage
High Level Parameter Execution Strategy

• While work left
  • Generate conflict graph for current worklist
  • Execute maximal independent set of nodes in graph
  • Add newly generated work to worklist
High Level Parameter Execution Strategy

- While work left
  - Generate conflict graph for current worklist
  - Execute maximal independent set of nodes in graph
  - Add newly generated work to worklist

Generate parallelism profile by tracking # of nodes executed in each step
Algorithm classification

- Spanning tree is a “refinement morph” algorithm
- Typical parallelism profile:
  - Little parallelism to start, as data structure gets refined
  - Most parallelism in the middle, as more activities become independent
  - Little parallelism at the end, as algorithm runs out of work
- Does this pattern hold for other refinement algorithms?
Example: Delaunay mesh refinement

- Active nodes: bad triangles
- Neighborhoods: cavities
- Refinement-like algorithm
  - As bad triangles get fixed, mesh gets larger (fixing a bad triangle replaces $N$ triangles with $N+2$ triangles)
  - Cavity sizes stay roughly the same (~6 triangles)
  - As mesh grows, cavities less likely to overlap → more parallelism
Refine 550K triangle mesh, ~261K badly shaped
Example: Agglomerative clustering

- Goal: cluster a set of points together according to distance to build a *dendrogram*
- Points cluster together if they are one another’s nearest neighbor
- Dendrogram is build bottom-up
Example: Agglomerative clustering

- Goal: cluster a set of points together according to distance to build a dendrogram
- Points cluster together if they are one another's nearest neighbor
- Dendrogram is build bottom-up
Example: Agglomerative clustering

• Goal: cluster a set of points together according to distance to build a dendrogram

• Points cluster together if they are one another’s nearest neighbor

• Dendrogram is build bottom-up
Example: Agglomerative clustering

- Goal: cluster a set of points together according to distance to build a dendrogram
- Points cluster together if they are one another’s nearest neighbor
- Dendrogram is built bottom-up
Example: Agglomerative clustering

- **Goal:** cluster a set of points together according to distance to build a dendrogram
- **Points cluster together if they are one another’s nearest neighbor**
- **Dendrogram is build bottom-up**
Expected Parallelism

- Expect amount of parallelism to match “bushiness” of dendrogram
  - If dendrogram is long and skinny, not much parallelism
  - If dendrogram is short and bushy, more parallelism
- Dendrogram built bottom-up
  - “Coarsening” morph
  - Expect parallelism to decrease as tree gets connected
Cluster set of 100K randomly generated points
MST over 300x400 2-D grid
Other Parameter capabilities
Ordered algorithms

• Can profile parallelism in ordered algorithms
• Active nodes must be processed in some order
• Intuition: execute nodes as in out-of-order processor, retire in-order through reorder buffer
• ParaMeter tracks when an activity executes not when it retires
• *cf* measuring ILP
Other metrics

- Parallelism intensity
  - Measures amount of parallelism relative to worklist size

- Neighborhood statistics
  - Minimum, maximum and average neighborhood sizes
Thank you!

https://engineering.purdue.edu/~milind
milind@purdue.edu