Introduction to Parallel Algorithm Analysis

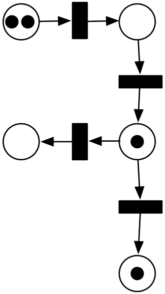
Jeff M. Phillips

October 4, 2013

Petri Nets

C. A. Petri [1962] introduced analysis model for concurrent systems.

- Flow chart
- Described data flow and dependencies.
- Very low level (we want something more high-level)
- Reachability EXP-SPACE-HARD, Decidable



Critical Regions Problem

Edsger Dijkstra [1965]

- Mutex: “Mutual exclusion” of variable
- Semaphores : Locks/Unlocks access to (multiple) data.
- Semaphore more general - keeps a count. Mutex binary.

Critical Regions Problem

Edsger Dijkstra [1965]

- Mutex: “Mutual exclusion” of variable
- Semaphores : Locks/Unlocks access to (multiple) data.
- Semaphore more general - keeps a count. Mutex binary.

Important, but lower level details.

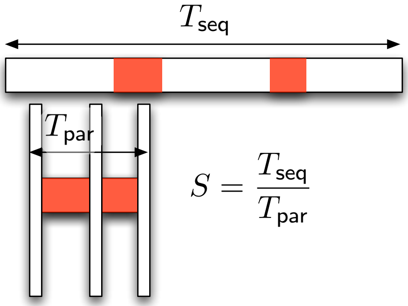
Amdahl's and Gustafson's Laws

Amdahl's Law : Gene Amdahl [1967]

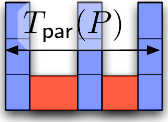
- Small portion (fraction α) non-parallelizable
- Limits max speed-up $S = 1/\alpha$.

Gustafson's Law : Gustafson+Barsis [1988]

- Small portion (fraction α) non-parallelizable
- P processors
- Limits max speed-up $S(P) = P - \alpha(P - 1)$.

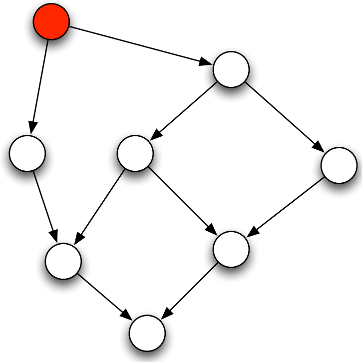


$$S = \frac{T_{\text{seq}}}{T_{\text{par}}}$$



$$S(P) = \frac{T_{\text{seq}}}{T_{\text{par}}(P)}$$

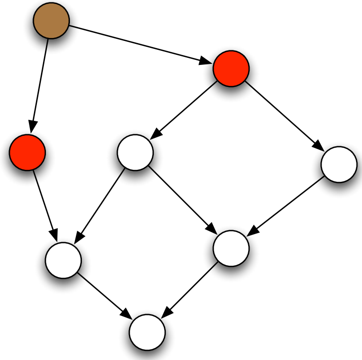
Logical Clocks



Leslie Lamport [1978]

- Posed parallel problems as finite state machine
- Preserved (only) partial order: "happens before" mutex

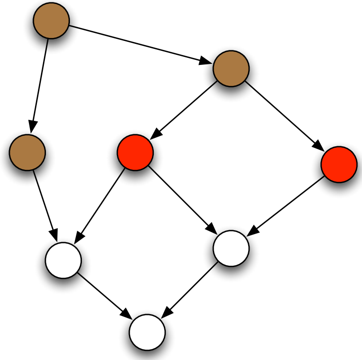
Logical Clocks



Leslie Lamport [1978]

- Posed parallel problems as finite state machine
- Preserved (only) partial order: “happens before” mutex

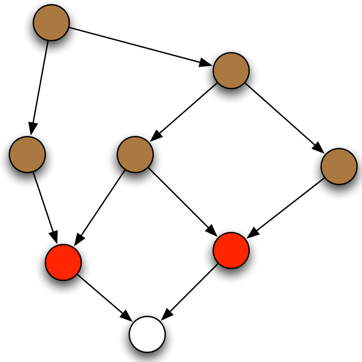
Logical Clocks



Leslie Lamport [1978]

- Posed parallel problems as finite state machine
- Preserved (only) partial order: "happens before" mutex

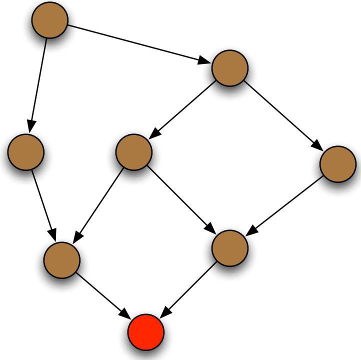
Logical Clocks



Leslie Lamport [1978]

- Posed parallel problems as finite state machine
- Preserved (only) partial order: "happens before" mutex

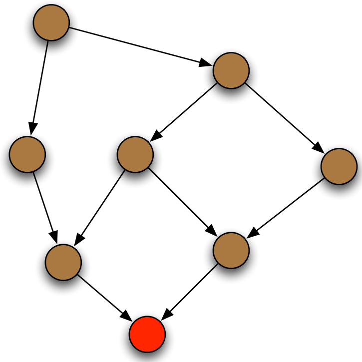
Logical Clocks



Leslie Lamport [1978]

- Posed parallel problems as finite state machine
- Preserved (only) partial order: "happens before" mutex

Logical Clocks



Leslie Lamport [1978]

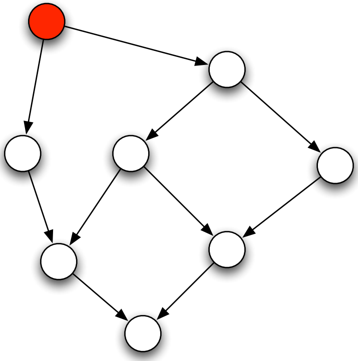
- Posed parallel problems as finite state machine
- Preserved (only) partial order: “happens before” mutex

Highlights nuances and difficulties in clock synchronization.

DAG Model

Directed Acyclic Graph:

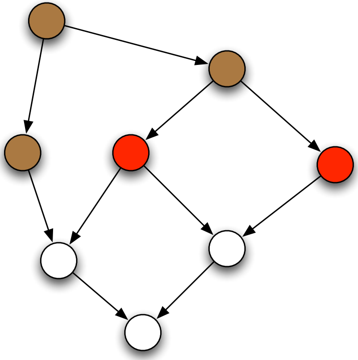
- Each node represents a chunk of computation that is to be done on a single processor
- Directed edges indicate that the **from** node must be completed before the **to** node
- The longest path in the DAG represents the total amount of parallel time of the algorithm
- The width of the DAG indicates the number of processors that can be used at once



DAG Model

Directed Acyclic Graph:

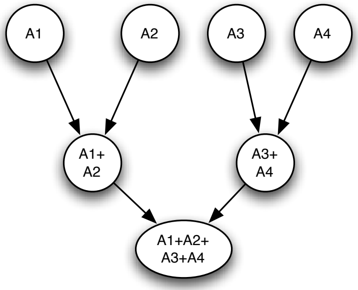
- Each node represents a chunk of computation that is to be done on a single processor
- Directed edges indicate that the **from** node must be completed before the **to** node
- The longest path in the DAG represents the total amount of parallel time of the algorithm
- The width of the DAG indicates the number of processors that can be used at once



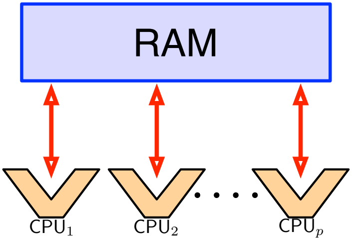
DAG Model

Directed Acyclic Graph:

- Each node represents a chunk of computation that is to be done on a single processor
- Directed edges indicate that the **from** node must be completed before the **to** node
- The longest path in the DAG represents the total amount of parallel time of the algorithm
- The width of the DAG indicates the number of processors that can be used at once



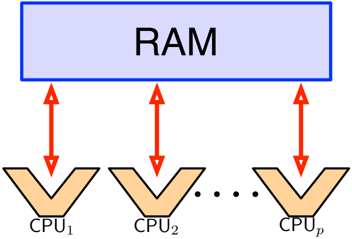
PRAM Model



Steve Fortune and James Wyllie [1978].
"shared memory model"

- P processors which operate on a shared data
- For each processor read, write, op (e.g. $+$, $-$, $\times$) constant time.

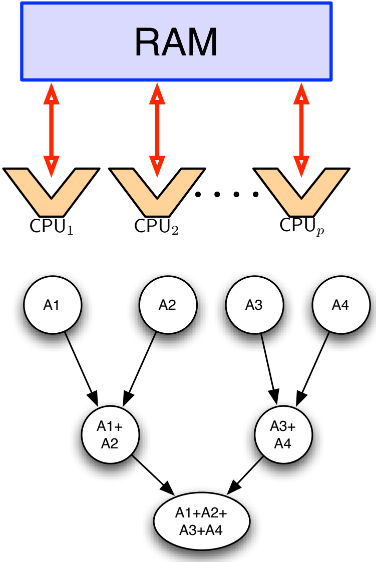
PRAM Model



Steve Fortune and James Wyllie [1978].
"shared memory model"

- P processors which operate on a shared data
- For each processor read, write, op (e.g. $+$, $-$, $\times$) constant time.
- **CREW** : Concurrent read, exclusive write
- **CRCW** : Concurrent read, concurrent write
- **EREW** : Exclusive read, exclusive write

PRAM Model



Steve Fortune and James Wyllie [1978].
"shared memory model"

- P processors which operate on a shared data
- For each processor read, write, op (e.g. $+$, $-$, $\times$) constant time.

- **CREW** : Concurrent read, exclusive write
- **CRCW** : Concurrent read, concurrent write
- **EREW** : Exclusive read, exclusive write

Message Passing Model

Emphasizes Locality

- **send**(X, i) : sends X to P_i
- **receive**(Y, j) : receives Y from P_j
- Fixed topology, can only **send**/**receive** from neighbor

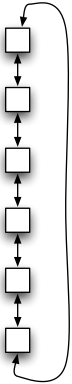
Message Passing Model

Emphasizes Locality

- **send**(X, i) : sends X to P_i
- **receive**(Y, j) : receives Y from P_j
- Fixed topology, can only **send**/**receive** from neighbor

Common Topologies:

- Array/Ring Topology
  -
- Mesh Topology
  -
- Hypercube Topology
  -



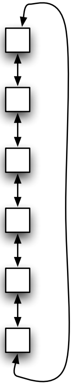
Message Passing Model

Emphasizes Locality

- **send**(X, i) : sends X to P_i
- **receive**(Y, j) : receives Y from P_j
- Fixed topology, can only **send**/**receive** from neighbor

Common Topologies:

- Array/Ring Topology
  - ($\Omega(p)$ rounds)
- Mesh Topology
  -
- Hypercube Topology
  -



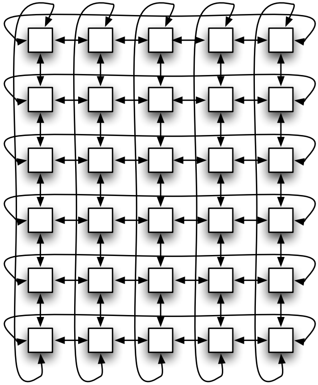
Message Passing Model

Emphasizes Locality

- **send**(X, i) : sends X to P_i
- **receive**(Y, j) : receives Y from P_j
- Fixed topology, can only **send**/**receive** from neighbor

Common Topologies:

- Array/Ring Topology
  - $(\Omega(p)$ rounds)
- Mesh Topology
  -
- Hypercube Topology
  -



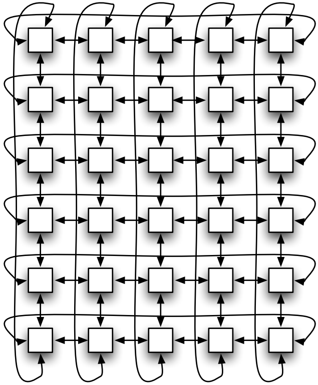
Message Passing Model

Emphasizes Locality

- **send**(X, i) : sends X to P_i
- **receive**(Y, j) : receives Y from P_j
- Fixed topology, can only **send**/**receive** from neighbor

Common Topologies:

- Array/Ring Topology
  - $(\Omega(p)$ rounds$)$
- Mesh Topology
  - $(\Omega(\sqrt{p})$ rounds$)$
- Hypercube Topology
  -



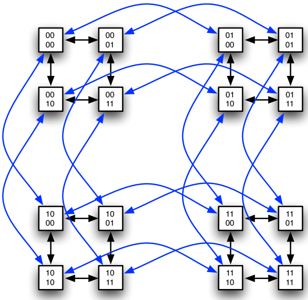
Message Passing Model

Emphasizes Locality

- **send**(X, i) : sends X to P_i
- **receive**(Y, j) : receives Y from P_j
- Fixed topology, can only **send**/**receive** from neighbor

Common Topologies:

- Array/Ring Topology
  - $(\Omega(p)$ rounds)
- Mesh Topology
  - $(\Omega(\sqrt{p})$ rounds)
- Hypercube Topology
  -



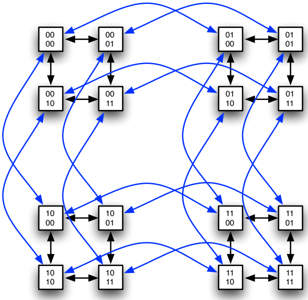
Message Passing Model

Emphasizes Locality

- **send**(X, i) : sends X to P_i
- **receive**(Y, j) : receives Y from P_j
- Fixed topology, can only **send**/**receive** from neighbor

Common Topologies:

- Array/Ring Topology
  - $(\Omega(p)$ rounds)
- Mesh Topology
  - $(\Omega(\sqrt{p})$ rounds)
- Hypercube Topology
  - $(\Omega(\log p)$ rounds)



Programming in MPI

Open MPI :

- (Open Source High Performance Computing).
- `http://www.open-mpi.org/`

Programming in MPI

Open MPI :

- (Open Source High Performance Computing).
- http://www.open-mpi.org/

When to use MPI?

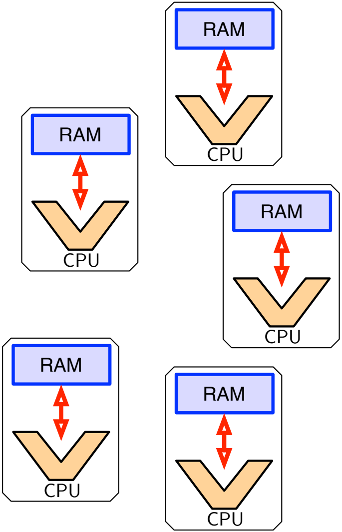
- Critical to exploit locality (i.e. scientific simulations)
- Complication in only talking to neighbor

Bulk Synchronous Parallel

Les Valiant [1989] BSP
Creates "barriers" in parallel algorithm.

1. Each processor computes on data
2. Processors send/receive data
3. Barrier : All processors wait for communication to end globally

Allows for easy synchronization. Easier to analyze since handles many messy synchronization details if this is emulated.

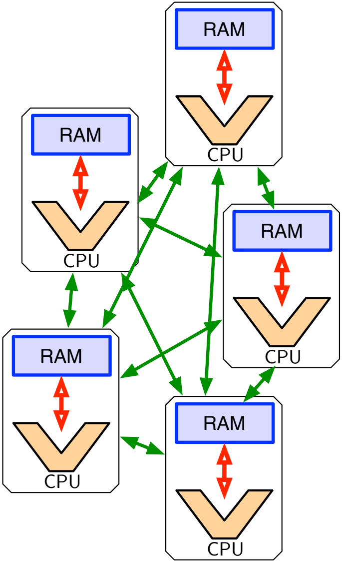


Bulk Synchronous Parallel

Les Valiant [1989] BSP
Creates "barriers" in parallel algorithm.

1. Each processor computes on data
2. Processors send/receive data
3. Barrier : All processors wait for communication to end globally

Allows for easy synchronization. Easier to analyze since handles many messy synchronization details if this is emulated.

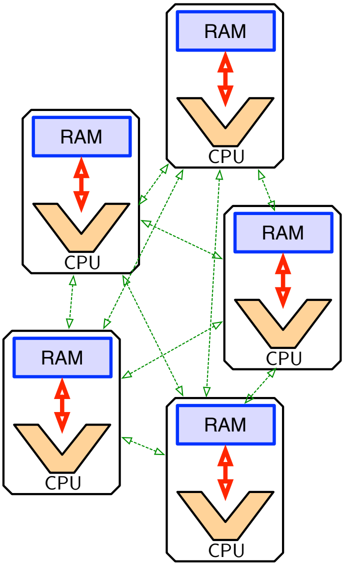


Bulk Synchronous Parallel

Les Valiant [1989] BSP
Creates "barriers" in parallel algorithm.

1. Each processor computes on data
2. Processors send/receive data
3. Barrier : All processors wait for communication to end globally

Allows for easy synchronization. Easier to analyze since handles many messy synchronization details if this is emulated.



Bridging Model for Multi-Core Computing

Les Valiant [2010] Mulit-BSP
Many parameters:

- P : number of processors
- M : Memory/Cache Size
- B : Block Size/Cost
- L : Synchronization Costs

Argues: any portable and efficient parallel algorithm, must take into account all of these parameters.

Bridging Model for Multi-Core Computing

Les Valiant [2010] Mulit-BSP
Many parameters:

- P : number of processors
- M : Memory/Cache Size
- B : Block Size/Cost
- L : Synchronization Costs

Argues: any portable and efficient parallel algorithm, must take into account all of these parameters.

Advantages:

- Analyzes all levels of architecture together
- Like Cache-Oblivious, but not oblivious

Bridging Model for Multi-Core Computing

Les Valiant [2010] Mulit-BSP
Many parameters:

- P : number of processors
- M : Memory/Cache Size
- B : Block Size/Cost
- L : Synchronization Costs

Argues: any portable and efficient parallel algorithm, must take into account all of these parameters.

Advantages:

- Analyzes all levels of architecture together
- Like Cache-Oblivious, but not oblivious

At depth d uses parameters: $\bigcup_i (p_i, g_i, L_i, m_i)$

- p_i : number of subcomponents (processors at leaf)
- g_i : communication bandwidth (e.g. I/O cost)
- L_i : synchronization cost
- m_i : memory/cache size

Bridging Model for Multi-Core Computing

Les Valiant [2010] Mulit-BSP
Many parameters:

- P : number of processors
- M : Memory/Cache Size
- B : Block Size/Cost
- L : Synchronization Costs

Argues: any portable and efficient parallel algorithm, must take into account all of these parameters.

Advantages:

- Analyzes all levels of architecture together
- Like Cache-Oblivious, but not oblivious

At depth d uses parameters: $\bigcup_i (p_i, g_i, L_i, m_i)$

- p_i : number of subcomponents (processors at leaf)
- g_i : communication bandwidth (e.g. I/O cost)
- L_i : synchronization cost
- m_i : memory/cache size

Matrix Multiplication, Fast Fourier Transform, Sorting

Two types of programmers

Two types of programmers

1. Wants to optimize the heck out of everything, tune all parameters

Two types of programmers

1. Wants to optimize the heck out of everything, tune all parameters
2. Wants to get something working, not willing to work too hard

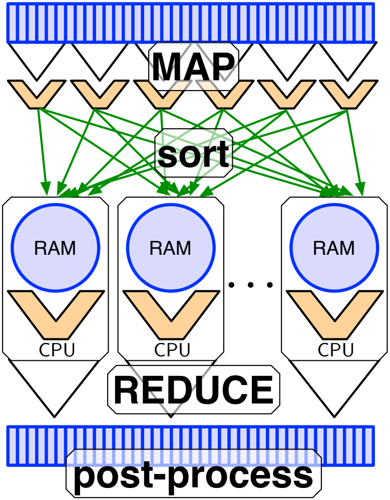
MapReduce

Each Processor has full hard drive, data items $<\text{KEY}, \text{VALUE}>$.
Parallelism Procedes in Rounds:

- Map: assigns items to processor by KEY.
- Reduce: processes all items using VALUE. Usually combines many items with same KEY.

Repeat M+R a constant number of times, often only one round.

- Optional post-processing step.

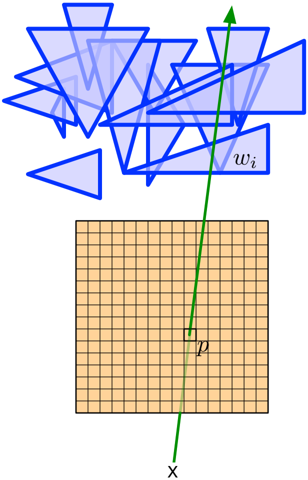


Pro: Robust (duplication) and simple. Can harness Locality
Con: Somewhat restrictive model

General Purpose GPU

Massive parallelism on your desktop.
Uses **G**raphics **P**rocessing **U**nit.
Designed for efficient video rasterizing.
Each *processor* corresponds to pixel p

- depth buffer:
  $D(p) = \min_i ||x - w_i||$
- color buffer: $C(p) = \sum_i \alpha_i \chi_i$
- ...



Pro: Fine grain, massive parallelism. Cheap.
Con: Somewhat restrictive model. Small memory.

… and Beyond

Google Sawzall / Dremel

- Compute statistics on massive distributed data.
- Separates local computation from aggregation.

… and Beyond

Google Sawzall / Dremel

- Compute statistics on massive distributed data.
- Separates local computation from aggregation.

Berkeley Spark: Processing in memory

- Keeps relevant information in memory.
- Faster on iterative algorithms (machine learning, SQL queries)

... and Beyond

Google Sawzall / Dremel

- Compute statistics on massive distributed data.
- Separates local computation from aggregation.

Berkeley Spark: Processing in memory

- Keeps relevant information in memory.
- Faster on iterative algorithms (machine learning, SQL queries)

Massive, Unorganized, Distributed Computing

- Bit-Torrent (distributed hash tables)
- SETI @ Home
- Twitter Storm / Facebook Casandra