Lazy Parallel Kronecker Algebra-operations on Heterogeneous Multicores

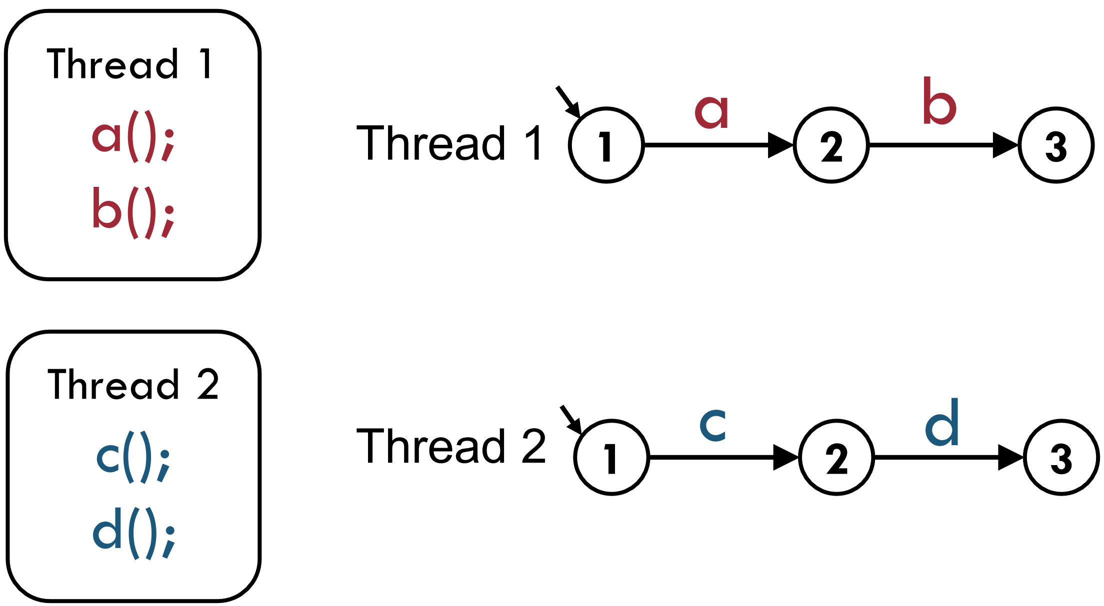
EuroPar 2017

Wasuwee Sodsong[1], Robert Mittermayr[2], Yoojin Park[1],
Bernd Burgstaller[1] and Johann Blieberger[2]

[1]Yonsei University, Seoul, Korea
[2]Vienna University of Technologies, Vienna, Austria

Motivation: Computing Thread Interleavings

- *Arbitrary* interleavings of threads such that…
  - the order on computation steps is taken from threads' program order



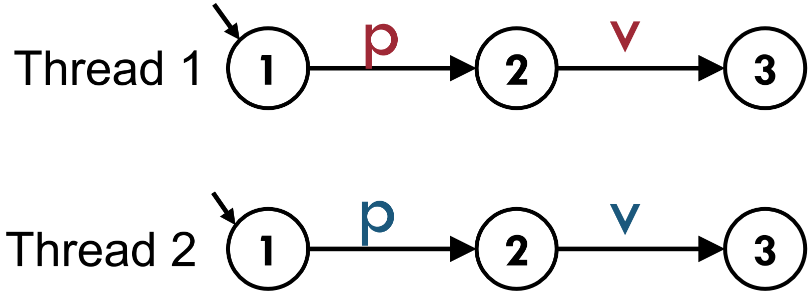
Interleavings
a · b · c · d
a · c · b · d
a · c · d · b
c · d · a · b
c · a · d · b
c · a · b · d

- Totality of all possible thread interleavings is well-suited for concurrent program analysis

Synchronization Primitives

- Semantics of synchronization primitives constrain possible interleavings
  - Not all interleavings are valid

Example: binary semaphore



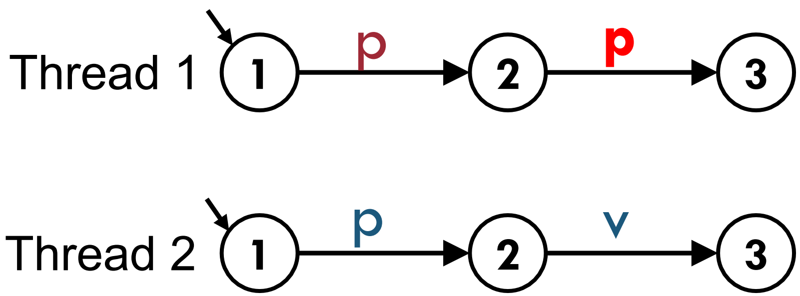
Interleavings

p · v · p · v ✓
p · p · v · v ✗
p · p · v · v ✗
p · v · p · v ✓
p · p · v · v ✗
p · p · v · v ✗

Synchronization Primitives

- Semantics of synchronization primitives allow additional interleavings that constitute deadlocks

Example: self-deadlock on binary semaphore

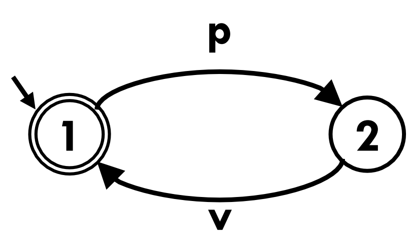


Interleavings

p

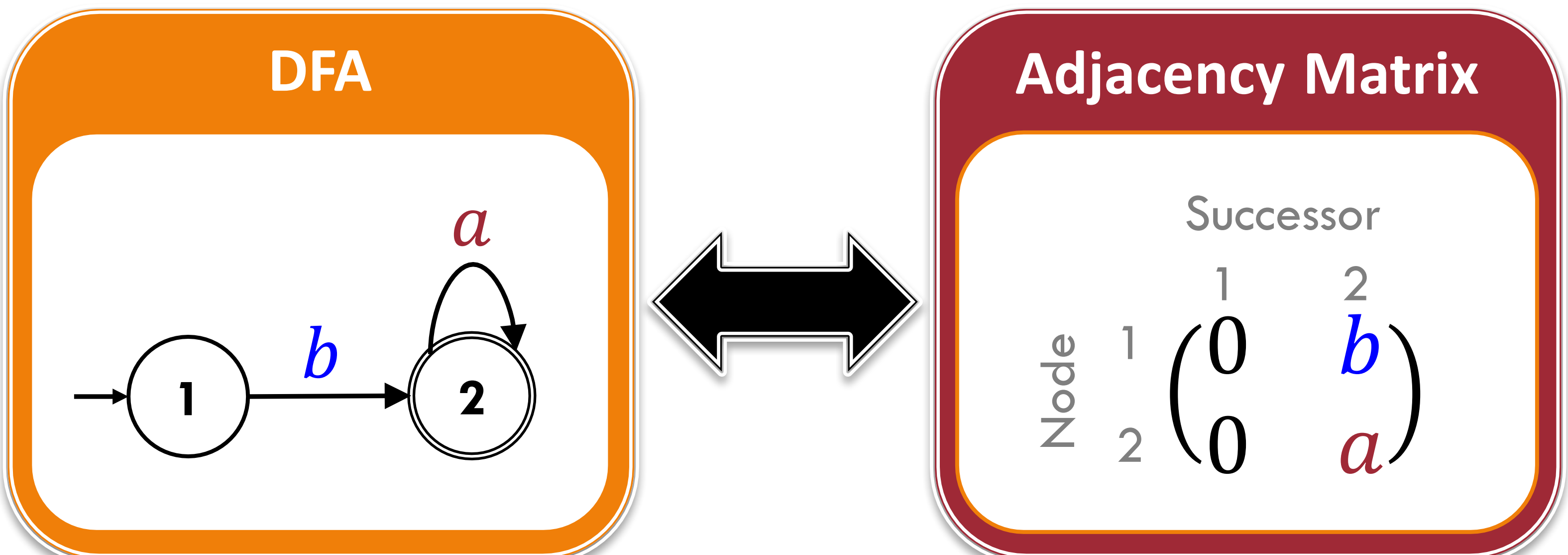
p · v · p

- Behavior of semaphores can be modelled by a deterministic finite automaton (DFA)



Encoding of Threads as Adjacency Matrices

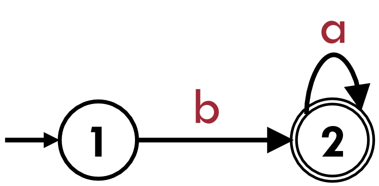
- DFA representation of a thread or a semaphore can be encoded as an adjacency matrix
  - Rows represent node IDs
  - Columns represent successor IDs



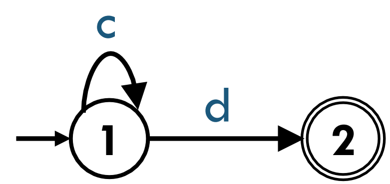
Kronecker Product: Lock-step Execution of Threads

- **Kronecker product:** Given an m-by-n matrix A and a p-by-q matrix B,

$$\mathrm{A} \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}$$

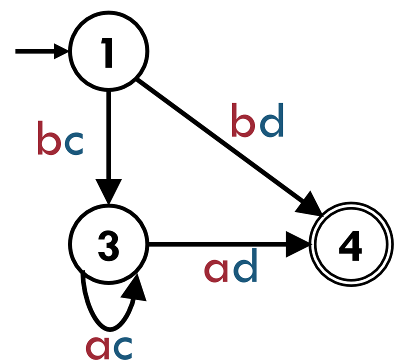


$$\mathrm{A} = \begin{pmatrix} 0 & b \\ 0 & a \end{pmatrix}$$



$$B = \begin{pmatrix} c & d \\ 0 & 0 \end{pmatrix}$$

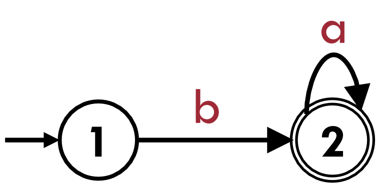
$$A \otimes B = \begin{pmatrix} 0 & 0 & bc & bd \\ 0 & 0 & 0 & 0 \\ 0 & 0 & ac & ad \\ 0 & 0 & 0 & 0 \end{pmatrix}$$



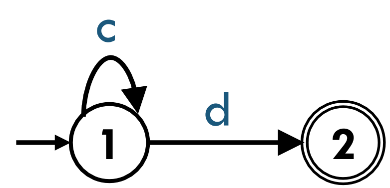
Kronecker Product: Lock-step Execution of Threads

- **Kronecker product:** Given an m-by-n matrix A and a p-by-q matrix B,

$$\mathrm{A} \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}$$

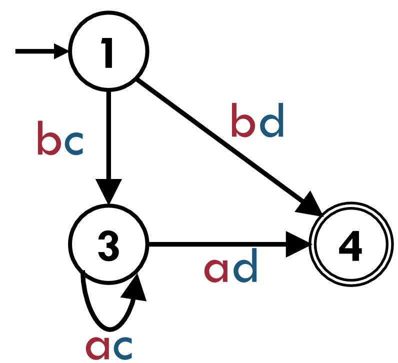


$$\mathrm{A} = \begin{pmatrix} 0 & b \\ 0 & a \end{pmatrix}$$



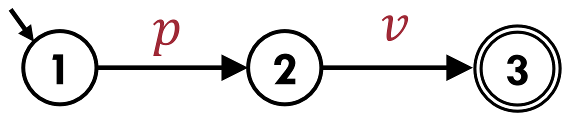
$$B = \begin{pmatrix} c & d \\ 0 & 0 \end{pmatrix}$$

- Executed threads A and B in lock-step
- $\otimes$ can be used to synchronize threads



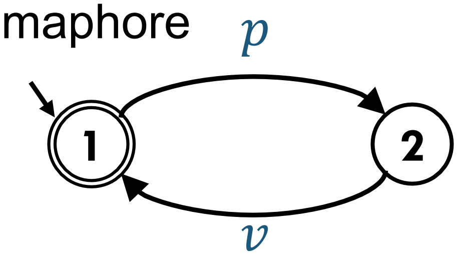
Synchronization: Lock-step Execution of Threads and Semaphores

Thread



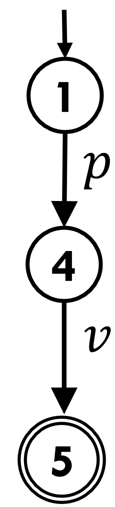
$$t = \begin{pmatrix} 0 & p & 0 \\ 0 & 0 & v \\ 0 & 0 & 0 \end{pmatrix}$$

Semaphore



$$s = \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix}$$

$$t \otimes s = \begin{pmatrix} 0 & 0 & 0 & pp & 0 & 0 \\ 0 & 0 & pv & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & vp \\ 0 & 0 & 0 & 0 & vv & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & p & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

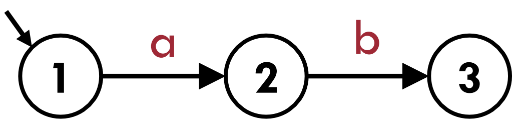


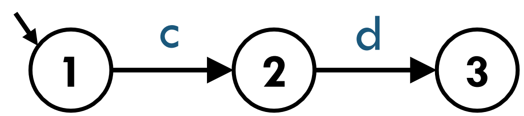
- For simplicity we write $x \cdot x = x$ and $x \cdot y = 0$

Kronecker Sum: Interleaved Execution of Threads

- **Kronecker sum:** Given a square matrix A of order m and B of order n,

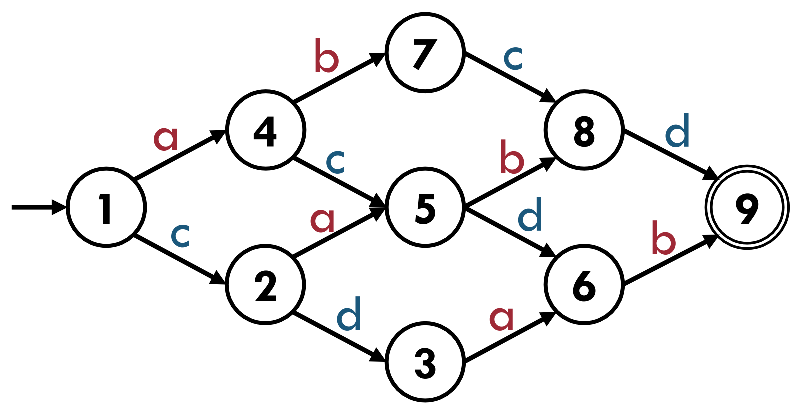
$$A \oplus B = A \otimes I_n + I_m \otimes B.$$





$$A = \begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & c & 0 \\ 0 & 0 & d \\ 0 & 0 & 0 \end{pmatrix}$$

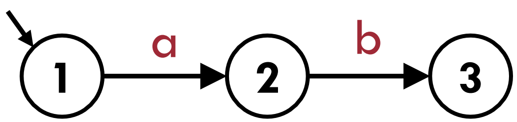
$$A \oplus B = \begin{pmatrix} 0 & c & 0 & a & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d & 0 & a & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & a & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c & 0 & b & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & d & 0 & b & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & b \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & c & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & d \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



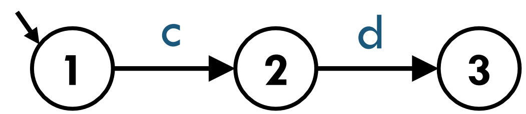
Kronecker Sum: Interleaved Execution of Threads

- **Kronecker sum:** Given a square matrix A of order m and B of order n,

$$\mathrm{A} \oplus B = A \otimes I_n + I_m \otimes B.$$

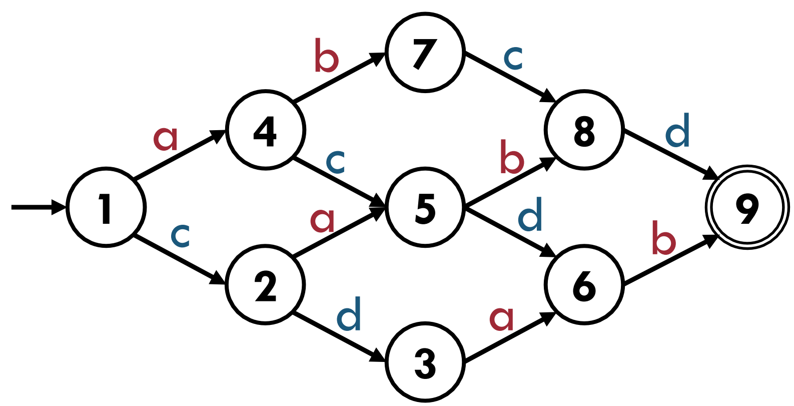


$$A = \begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix}$$



$$B = \begin{pmatrix} 0 & c & 0 \\ 0 & 0 & d \\ 0 & 0 & 0 \end{pmatrix}$$

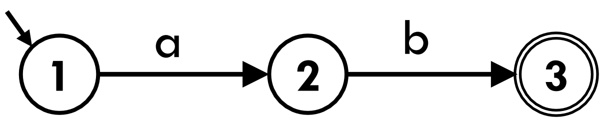
- All thread interleavings
- $\oplus$ can be used to model concurrency



Concurrent Threads and Semaphores

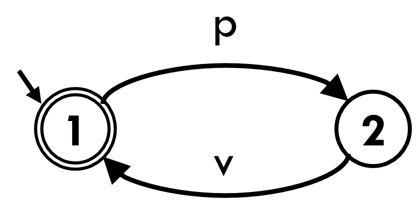
- Encode threads' control flow graphs and synchronization primitives as adjacency matrices.

Control flow graph of thread $t^{(i)}$



$$t^{(i)} = \begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix}$$

Semaphore $s^{(j)}$



$$s^{(j)} = \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix}$$

- $T = \bigoplus_{i=1}^{k} t^{(i)}$ models all interleavings of the threads

- $S = \bigoplus_{j=1}^{r} s^{(j)}$ models all interleavings of the semaphores

Synchronizing Threads and Semaphores

- Behaviors of overall systems can be modelled by

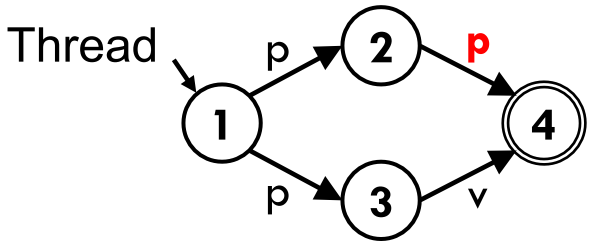
$$P = T_s \otimes S + T_v \oplus S$$

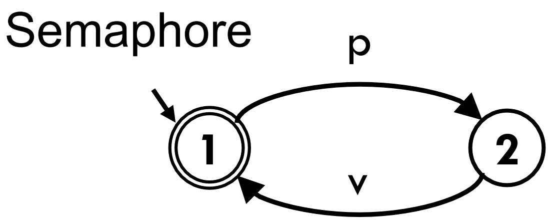
Execute threads and semaphores in lock steps

Interleaves semaphore calls and other statements

  - T_s contains only semaphore calls
  - T_v contains the other edge labels
  - $T = T_s + T_v$

Example: Overall System





Thread $t = \begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & \boldsymbol{p} \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & 0 \end{pmatrix}$

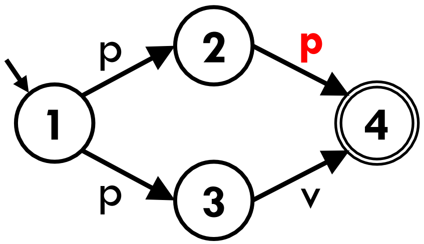
Semaphore $s = \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix}$

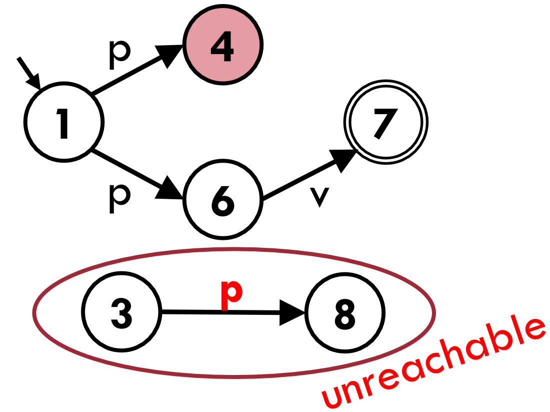
$$P = T_s \otimes S + T_v \oplus S$$

$$P = t \otimes s = \begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & \boldsymbol{p} \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix} =$$

Example: Overall System

Thread





$$\text{Thread } t = \begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & \boldsymbol{p} \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$P = T_s \otimes S + T_v \oplus S$$

$$P = t \otimes s = \begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & \boldsymbol{p} \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix} = \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & p & 0 & p & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boldsymbol{p} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}$$

Problems

- Combination of Kronecker algebra operations generates all possible (arbitrary) thread interleavings subject to semantics of synchronization primitives

- The number of interleavings increases exponentially in the number of threads
  - The state explosion problem
  - **But:** Not all interleavings need to be computed!

Contributions

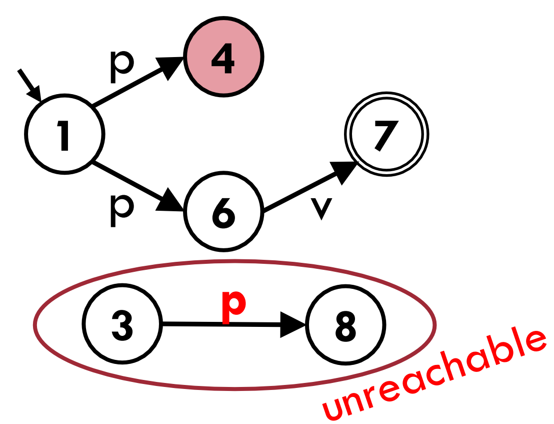
1. Two-step lazy evaluation scheme
   - Construct expression tree
   - Lazily evaluate *relevant* thread interleavings
2. Kronecker algebra operations optimized for multicore CPUs
3. Execution scheme that utilizes both the multicore CPU and the GPU
   - GPU: conducts lazy evaluation.
   - CPU: maintains the computed thread interleavings and coordinate the GPU-based evaluation process
4. Introduce a fast operation on a special case of Kronecker sums

Lazy Evaluation

Observations

- Adjacency matrices are sparse
- Not all nodes are reachable from the start node
- It is not necessary to compute the entire matrix

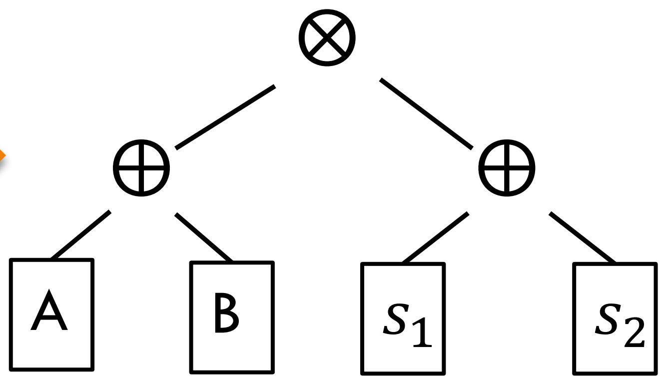
$$t \otimes s = \begin{array}{cc} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} & \\ \begin{pmatrix} 0 & 0 & 0 & \boldsymbol{p} & 0 & \boldsymbol{p} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boldsymbol{p} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \boldsymbol{v} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \end{array}$$



Lazy Evaluation

- We never compute an entire matrix
  - Instead, we compute only the non-zero elements of the reachable nodes

$$(A \oplus B) \otimes (s_1 \oplus s_2)$$



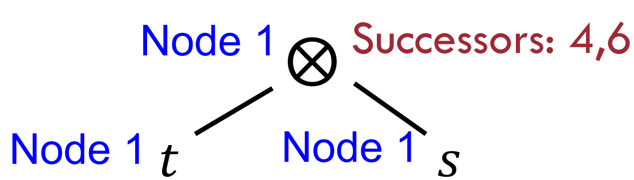
- **Step 1:** Construct an expression tree
  - Leaf nodes (operands) are stored as *sparse matrices*
  - Internal nodes (operators) are stored as *lazy matrices*
    - the algebra operator and pointers to the operands
  - No operator evaluation yet!

Lazy Evaluation

- **Step 2:** Lazily evaluate Kronecker algebra operations
  1. Find successors of the start node by evaluating the expression tree
  2. If a successor has not been processed before, insert it to a work-queue
  3. Repeatedly process nodes from the work-queue until it is empty

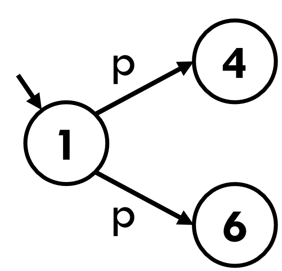
Example: self-deadlock thread

$$t \otimes s = \begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & p \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix} = \begin{array}{c|c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix} & \\ \begin{pmatrix} 0 & 0 & 0 & p & 0 & p & 0 & 0 \\ & & & & & & & \\ & & & & & & & \\ ? & ? & ? & ? & ? & ? & ? & ? \\ & & & & & & & \\ ? & ? & ? & ? & ? & ? & ? & ? \\ & & & & & & & \\ & & & & & & & \end{pmatrix} & \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{matrix} \end{array}$$



$$t = \begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & p \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad s = \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix}$$

work-queue
4
6



Processed nodes
1

Semaphore’s Kronecker Sum Optimization

- We can calculate a series of Kronecker sums of same-sized matrices in one step.
  - a node ID
  - number of Kronecker sum operations

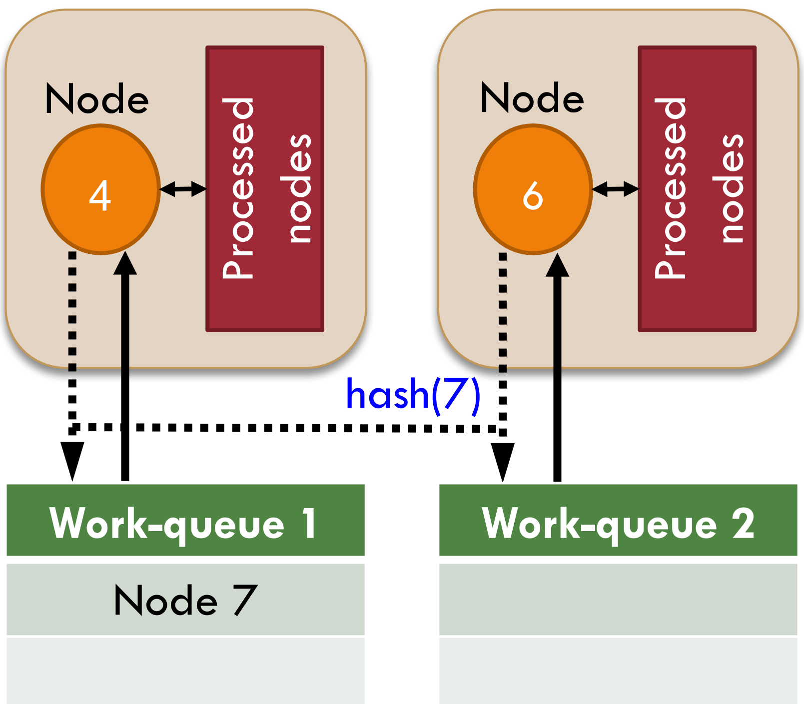
$$s_1 = \begin{bmatrix} 0 & p_1 \\ v_1 & 0 \end{bmatrix}$$

$$s_1 \oplus s_2 = \begin{bmatrix} 0 & p_2 & p_1 & 0 \\ v_2 & 0 & 0 & p_1 \\ v_1 & 0 & 0 & p_2 \\ 0 & v_1 & v_2 & 0 \end{bmatrix}$$

$$s_1 \oplus s_2 \oplus \boldsymbol{s_3} = \begin{bmatrix} 0 & p_3 & p_2 & 0 & p_1 & 0 & 0 & 0 \\ v_3 & 0 & 0 & p_2 & 0 & p_1 & 0 & 0 \\ v_2 & 0 & 0 & p_3 & 0 & 0 & p_1 & 0 \\ 0 & v_2 & v_3 & 0 & 0 & 0 & 0 & p_1 \\ v_1 & 0 & 0 & 0 & 0 & p_3 & p_2 & 0 \\ 0 & v_1 & 0 & 0 & v_3 & 0 & 0 & p_2 \\ 0 & 0 & v_1 & 0 & v_2 & 0 & 0 & p_3 \\ 0 & 0 & 0 & v_1 & 0 & v_2 & v_3 & 0 \end{bmatrix}$$

Kronecker Algebra on a Multicore CPU

- A thread finds all successors of a given node
- Employ a hash-function which hashes the node IDs of the successors
  - The hash-function guarantees one-to-one assignment of node IDs to worker threads
- Each worker thread maintains a local hash-table of processed nodes



$$t \otimes s = \begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & p \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & p \\ v & 0 \end{pmatrix} =$$

$$\begin{array}{cc} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} & \\ \begin{pmatrix} 0 & 0 & 0 & p & 0 & p & 0 & 0 \\ & & & & & & & \\ & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & v & 0 \\ ? & ? & ? & ? & ? & ? & ? & ? \\ & & & & & & & \end{pmatrix} & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \end{array}$$

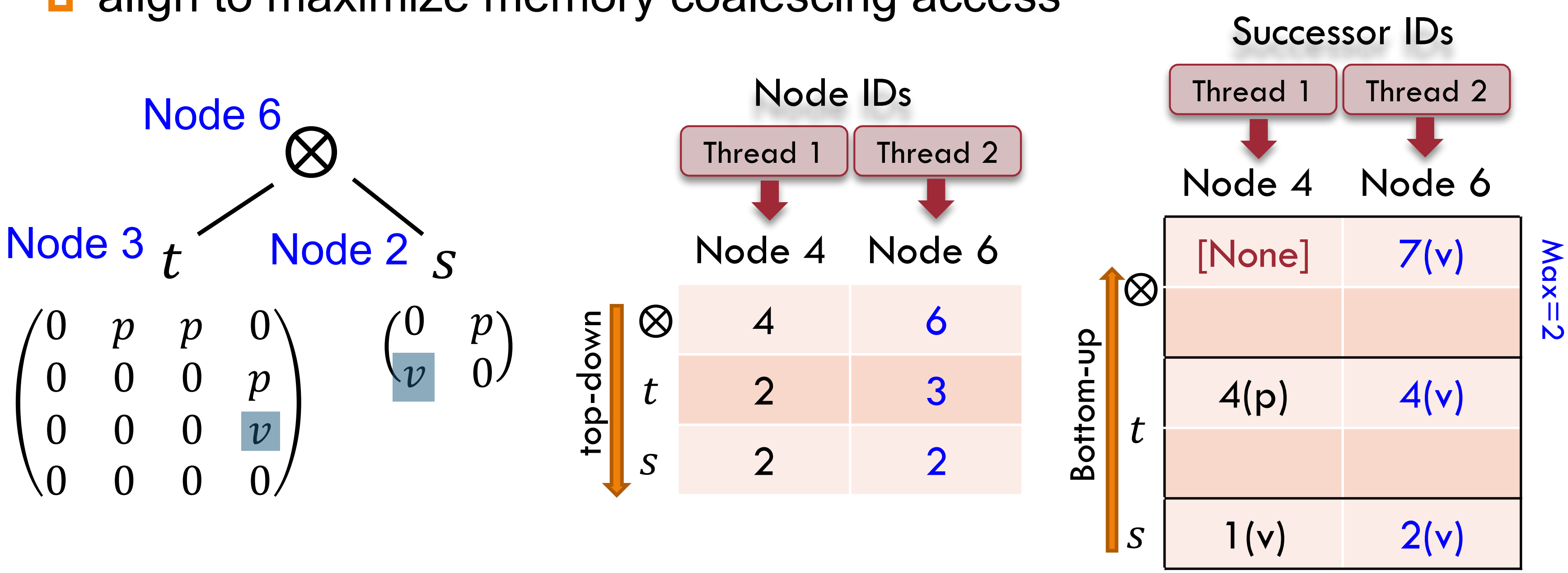
Kronecker Algebra using a CPU+GPU

- A GPU has relatively small memory compared to a CPU
  - unable to keep the continuously growing list of processed nodes

- Code-partitioning between the CPU and GPU according to the memory constraint
  - GPU lazily evaluates the Kronecker algebra operations
  - CPU maintains all computed thread interleavings

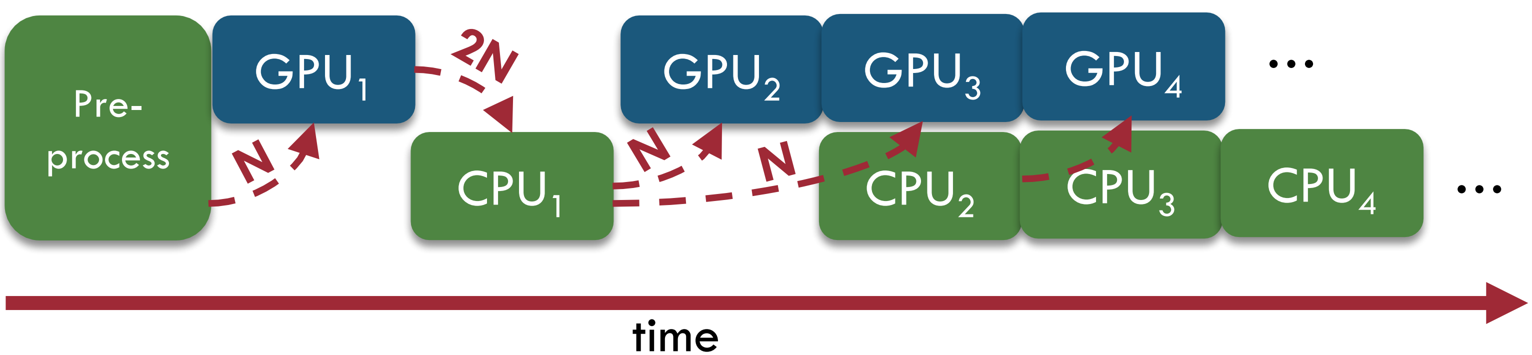
Lazy Evaluation on a GPU

- Execute lazy evaluation in 2 loops
  1. calculate node ID of all tree levels (top-down)
  2. retrieve successors (bottom-up)
- Intermediate results are stored in buffer
  - the size is pre-calculated from max. number of possible successors
  - align to maximize memory coalescing access



Pipelined Execution

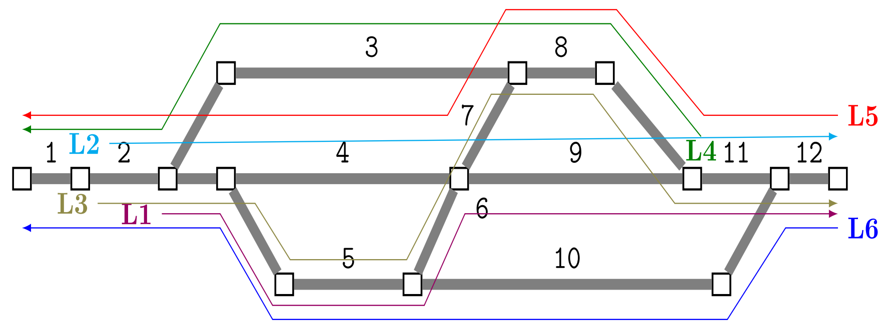
- GPU processes N nodes per iteration and may discover $> N$ new successors
- The first N new successors are computed in the next iteration
- The remaining successors are computed in the following iteration
- The GPU computation can overlap with the CPU computation of the previous iteration



Experimental Setups

Experiments:

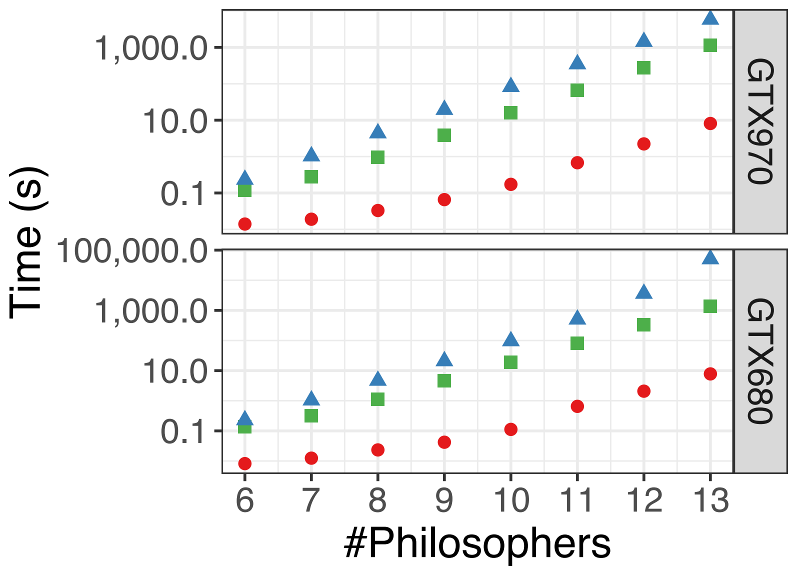
1. Dijkstra's Dining Philosophers
2. Linux kernel thread simulations
3. Railway system

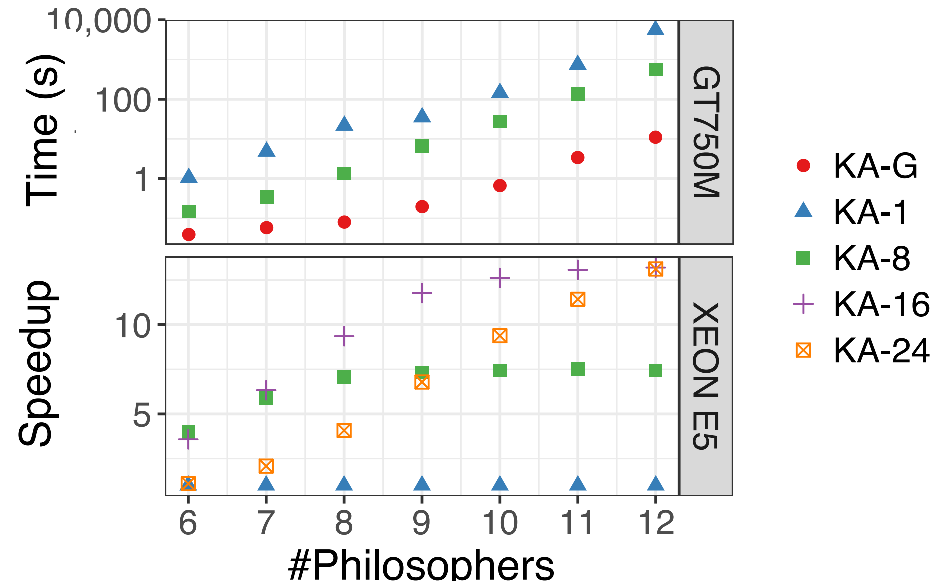


Hardware specifications:

Platform Name	GTX 970	GTX 680	GT 750M	Xeon E5
CPU model	Intel i7-6700	Intel i7-3770k	Intel i7-4850HQ	Xeon E5-2697
CPU frequency	3.4 GHz	3.5 GHz	2.3 GHz	1.8 GHz
GPU model	NVIDIA GTX 970	NVIDIA GTX 680	NVIDIA GT 750M	N/A
GPU core frequency	1329 MHz	1006 MHz	822 MHz	
No. of GPU cores	**1664**	**1536**	**384**	
Compute Capability	5.2	3.0	3.5	

Experimental Results: CPU only vs. CPU+GPU

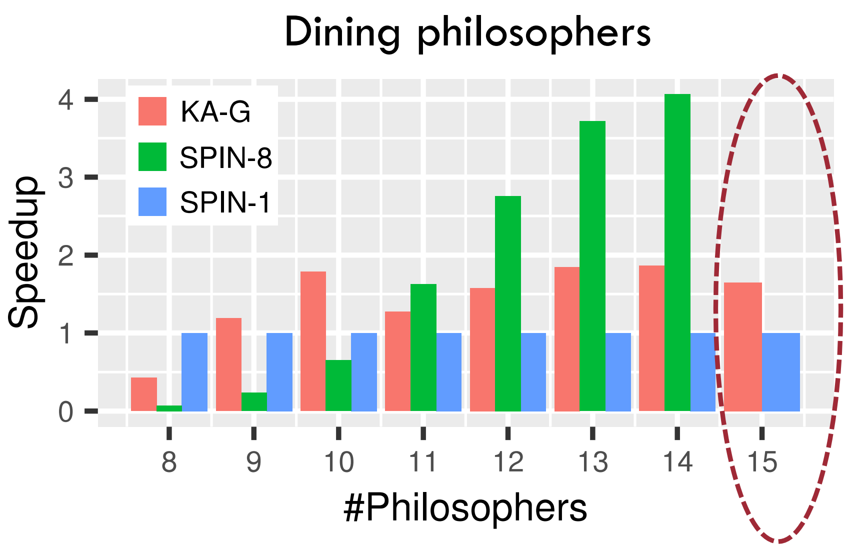


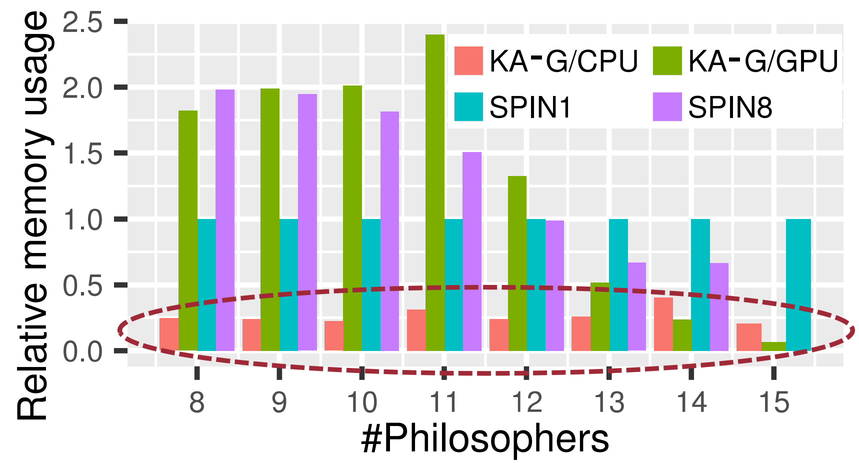


		3 kernel threads	5 kernel threads
GTX 970	KA-8	154 s	68024 s
	KA-G	0.30 s	14.64 s
GTX 680	KA-8	181 s	77107 s
	KA-G	0.29 s	14.14 s
GT 750M	KA-8	316 s	144321 s
	KA-G	0.52 s	69.10 s

- 12 Philosophers generate 1.6 million nodes
- 13 Philosophers generate 6.5 million nodes
- KA-G achieves up to 5453x speedup over the fastest multi-threaded CPU implementation

Experimental Results: SPIN





		GTX 970	GTX 680	GT 750M
Railway	SPIN-8	1.64 s	1.68 s	2.05 s
	KA-8	4.15 s	4.89 s	5.86 s
	KA-G	0.05 s	0.05 s	0.07 s

- KA-G is faster than single-threaded SPIN
- Multi-threaded SPIN is unable to handle > 14 philosophers
- KA consumes less CPU memory
  - can handle larger problems

Conclusions

- Two-step lazy evaluation scheme to mitigate the state explosion problem
- Multicore CPU implementation
- Multicore CPU + GPU implementation
  - GPU: conducts lazy evaluation
  - CPU: maintains the computed thread interleavings
- The CPU+GPU implementation is up to 5453x faster than our multicore CPU implementation
- Consumes up to 4.8x less memory than SPIN-1 and
  8.1x less memory than SPIN-8

Q&A

Thank you