PURDUE
UNIVERSITY

Scientific Computing in the Many-core Era: Scaling the Memory Wall

Alex Pothen
Purdue University
CSCAPES Institute
www.cs.purdue.edu/homes/apothen/

Assefaw Gebremedhin, Fredrik Manne, Umit Catlyurek, Erik Boman, Doruk Bozdag

Woudschoten Conference
Oct. 2010

cscapes

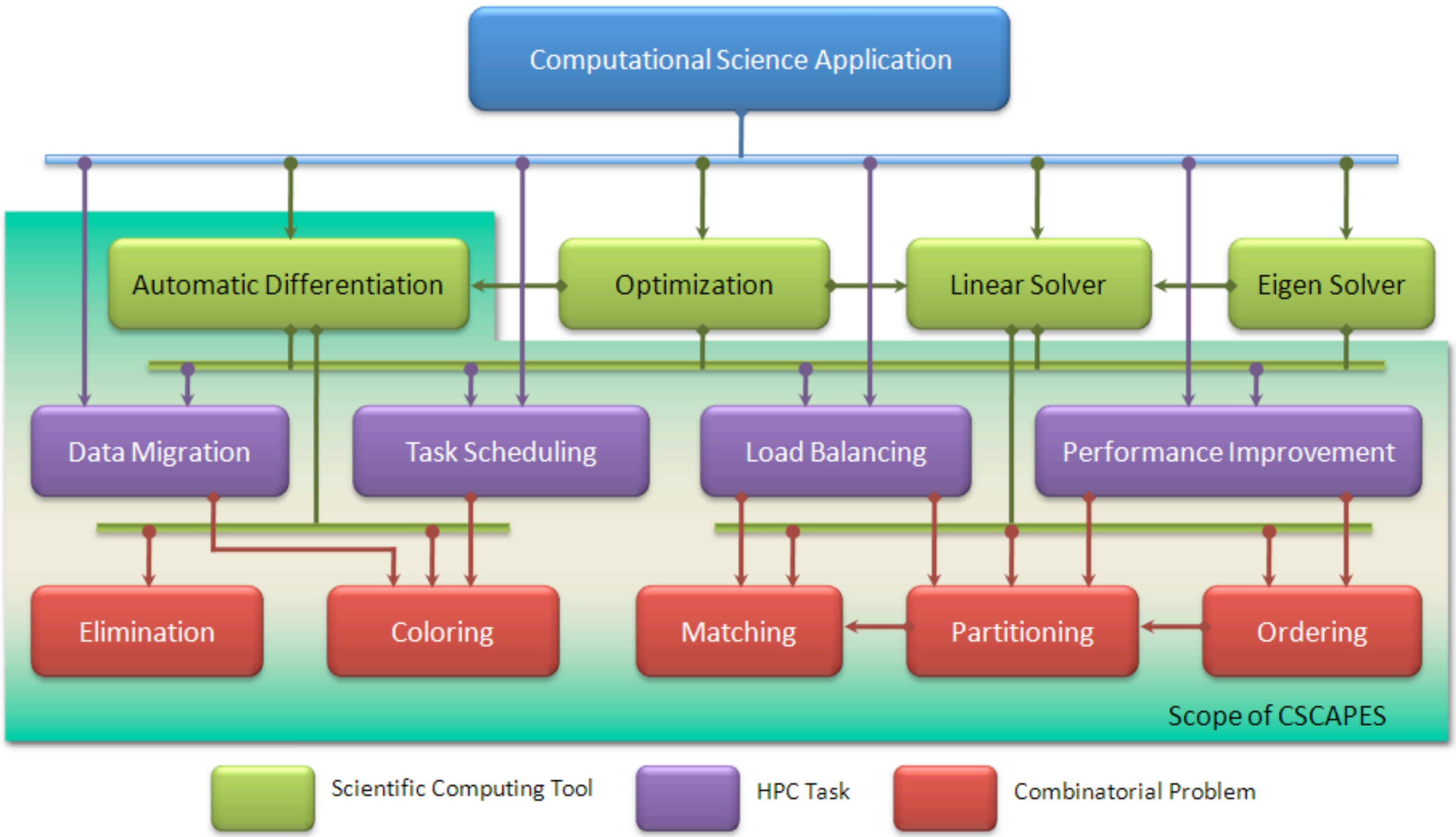
Outline

- Renaissance in Computer Architectures
- The Many-core and Multi-threaded World
  - Intel Nehalem
  - Sun Niagara
  - Cray XMT
- A case study on multithreaded graph coloring
  - An Iterative Coloring Algorithm
  - A Dataflow algorithm
- Distributed Memory Parallel Coloring
- Tentative Conclusions

Combinatorial Scientific Computing and Petascale Simulations (CSCAPES) Institute

- One of four DOE Scientific Discovery thru Advanced Computing (SciDAC) Institutes (2006-2012); only one in Appl. Math
  - Excellence in research, education and training
  - Collaborations with science projects in SciDAC
- Focus not on specific application, but on algorithms and software for combinatorial problems
- Participants from Purdue, Sandia, Argonne, Ohio State, Colorado State
- CSCAPES workshops with talks, tutorials on software, discussions on collaborations
- www.cscapes.org

CSCAPES Institute



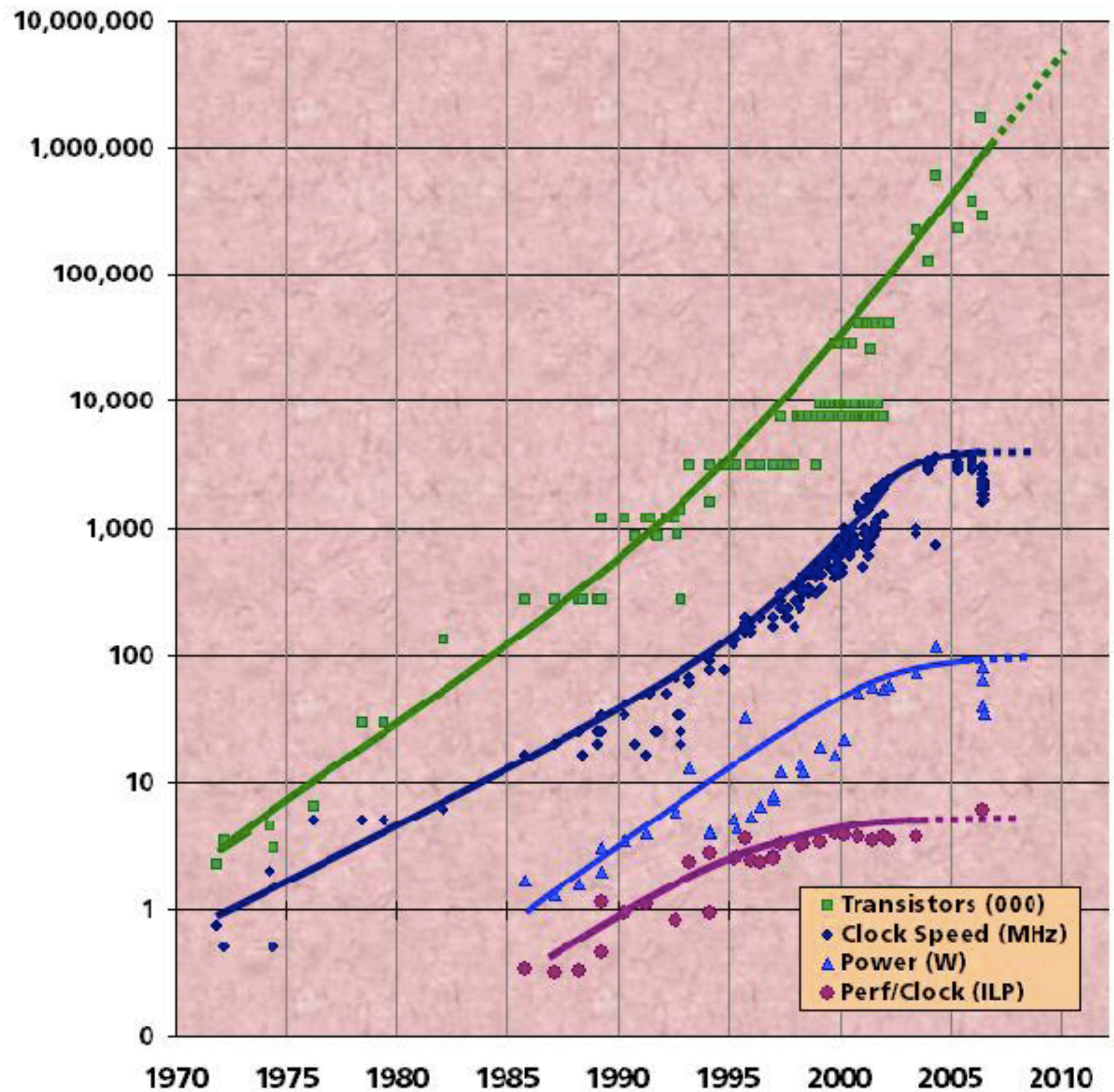
PURDUE UNIVERSITY

Advertisements!

- Journal of the ACM: New section on Scientific and High Performance Computing… venue for publishing work of excellent quality in these areas that have a Computer Science component
- SIAM Books, SIAM Monographs in Computational Science and Engineering: Potential book ideas? Please contact me!
- SIAM Workshop on Combinatorial Scientific Computing, CSC11, May 19-21, Darmstadt, www.siam.org/meetings/csc11/

cscapes

Moore's Law and Performance



A Renaissance in Architectures

- Good news
  - Moore's Law marches on
  - Real estate on a chip essentially free: Major paradigm change, and huge opportunity for innovation
- Bad news
  - Power limits improvement in clock speed
  - Memory accesses are the bottleneck for high-throughput computing
- Eventual consequences are unclear
- Current response: multi- and many-core processors
  - Computation/Communication ratio will get worse
  - Makes life harder for applications

PURDUE UNIVERSITY

Applications Complexity Increasing

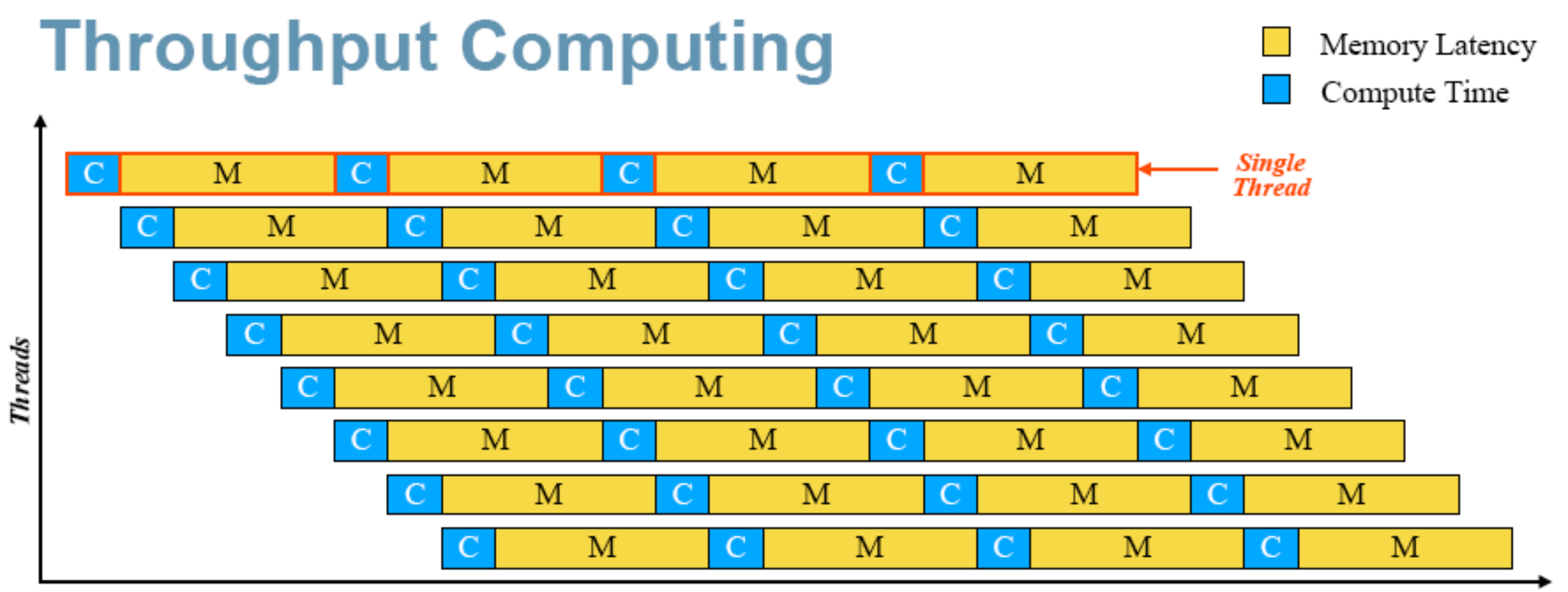
- Leading edge scientific applications increasingly include:
  - Adaptive, unstructured data structures
  - Complex, multiphysics simulations
  - Multiscale computations in space and time
  - Complex synchronizations (e.g., discrete events)
- Significant parallelization challenges on today's machines
  - Finite degree of coarse-grained parallelism
  - Load balancing and memory hierarchy optimization
- Dramatically harder on millions of cores
- Huge need for new algorithmic paradigms

cscapes

Architectural Wish List for Sparse Matrices/Graphs

- Low latency / high bandwidth
  - For small messages!
- Latency tolerance
- Light-weight synchronization mechanisms
- Global address space
  - No graph partitioning required
  - Avoid memory-consuming profusion of ghost-nodes
  - Correctness and performance easier
- One Solution: Multi-threaded computations

Multi-threaded Parallelism

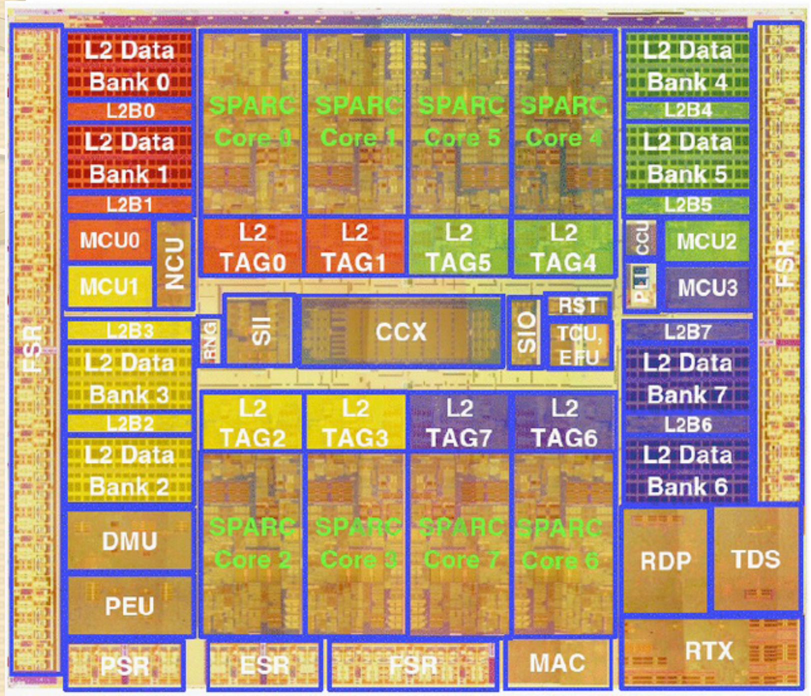


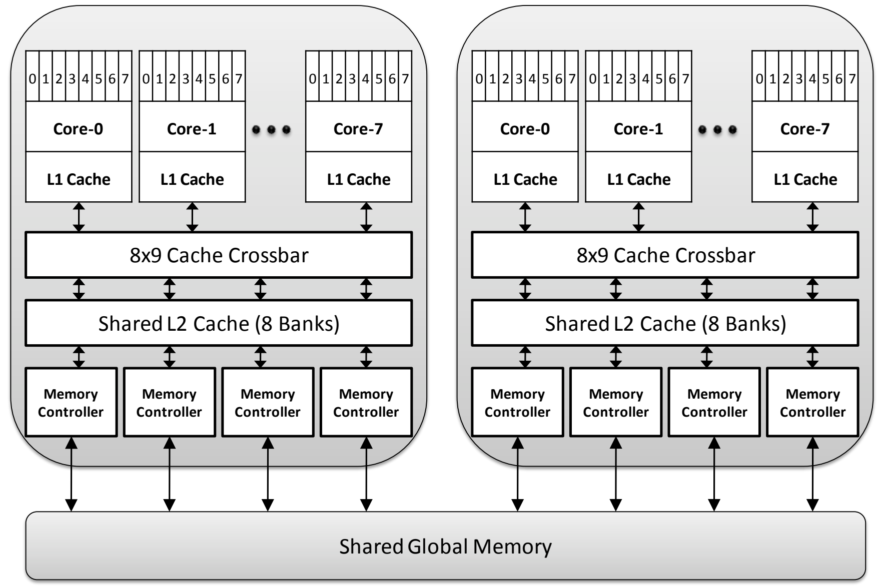
Time

Figure from Robert Golla, Sun

- Memory access times determine performance
- By issuing multiple threads, mask memory latency if a ready thread is available when a functional unit becomes free
- Interleaved vs. Simultaneous multithreading

Multi-core: Sun Niagara 2

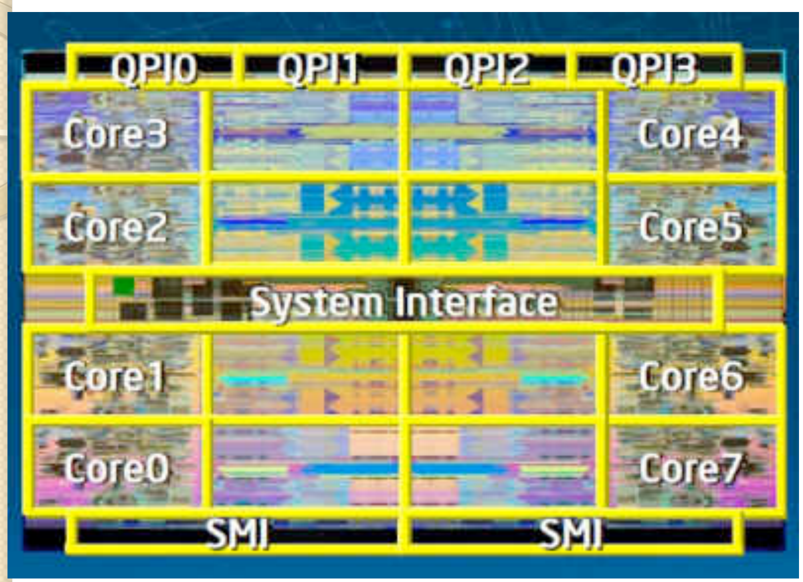


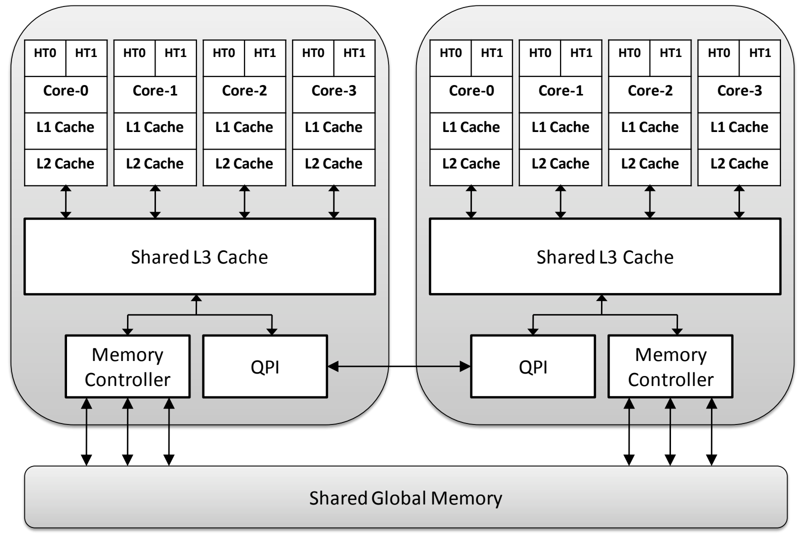


- Two 8-core sockets,
- 8 hw threads per core
- 1.2 GHz processors linked by 8 x 9 crossbar to L2 cache banks

- Simultaneous multithreading,
- Two threads from a core are executable in a cycle
- Shallow pipeline

Multicore: Intel Nehalem

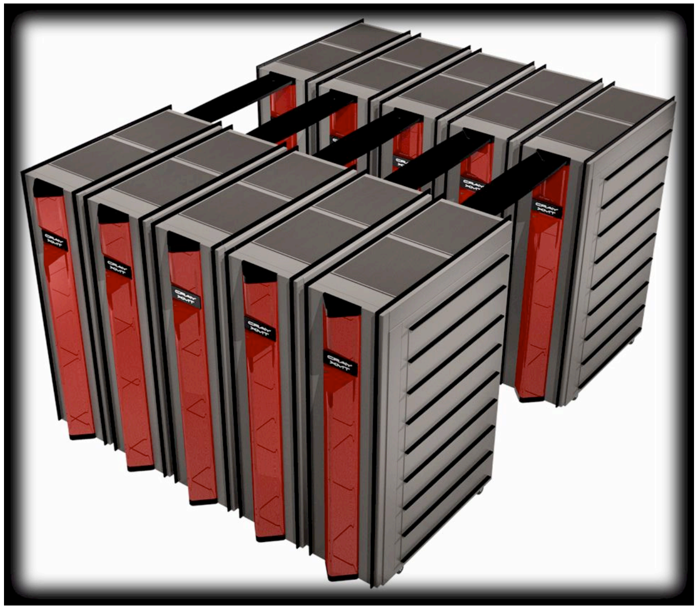


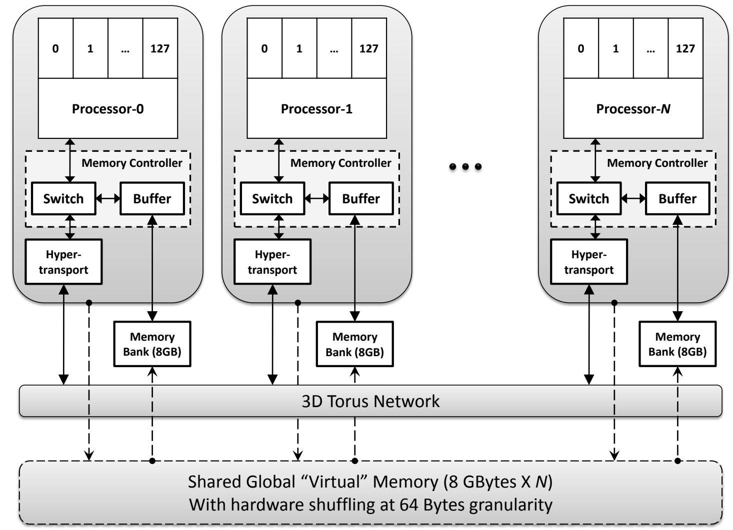


- Two quad-core chips, 2.5 GHz
- Two hyperthreads per core
- Off chip-data latency 106 cycles

- Advanced architectural features: Cache coherence protocol to reduce traffic, loop-stream detection, branch prediction, out-of-order execution
- Interleaved Multithreading

Massive Mutithreading: Cray XMT



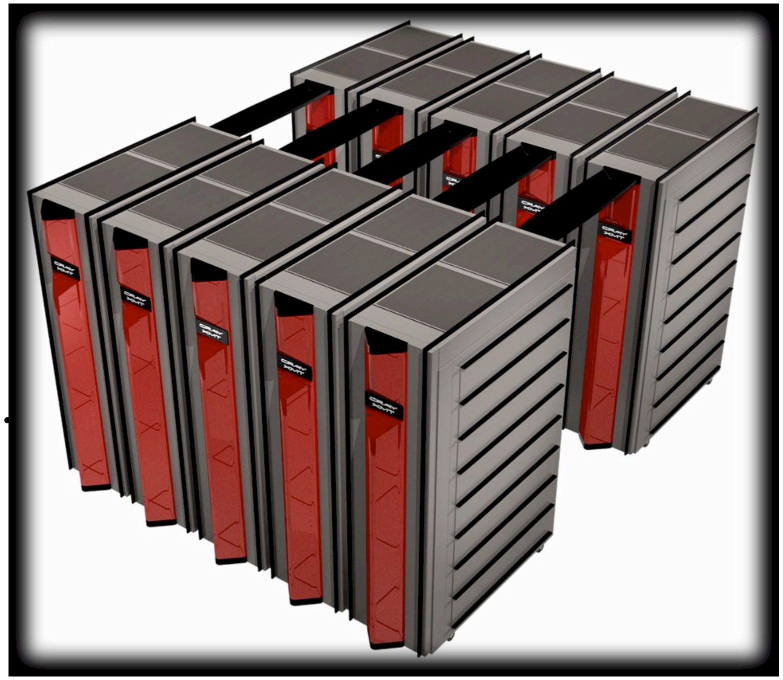


- 128 processors, 500 MHz
- 128 hw thread streams / proc.
- in each cycle a proc. issues one ready thread
- Deeply pipelined, M, A, C functional units

- Cache-less Globally shared memory
- Efficient hardware synchronziation via full/empty bits
- Data mapped randomly in 8 Byte blocks, no locality
- Average Memory latency 600 cycles

Massive Multithreading: Cray XMT

- Latency tolerance via massive multi-threading
  - Context switch between threads in a single clock cycle
  - Global address space, hashed to memory banks to reduce hot-spots
  - No cache or local memory, average latency 600 cycles
- Memory request doesn't stall processor
  - Other threads work while your request gets fulfilled
- Light-weight, word-level synchronization
- Notes:
  - 500 MHz clock
  - 128 Hardware thread streams / proc.



Multithreaded Algorithms for Graph Coloring

- We developed two kinds of multithreaded algorithms for graph coloring:
  - An *iterative*, coarse-grained method for generic shared-memory architectures
  - A *dataflow* algorithm designed for massively multithreaded architectures with hardware support for fine-grain synchronization, such as the Cray XMT
- Benchmarked the algorithms on three systems:
  - Cray XMT, Sun Niagara 2 and Intel Nehalem
- Excellent speedup observed on all three platforms

Coloring Algorithms

A Greedy coloring algorithm

- Distance-k, star, and acyclic coloring are NP-hard
- Approximating coloring to within $O(n^{1-e})$ is NP-hard too

GREEDY($G=(V,E)$)

Order the vertices in V

for i = 1 to $|V|$ **do**

Determine colors forbidden to v_i

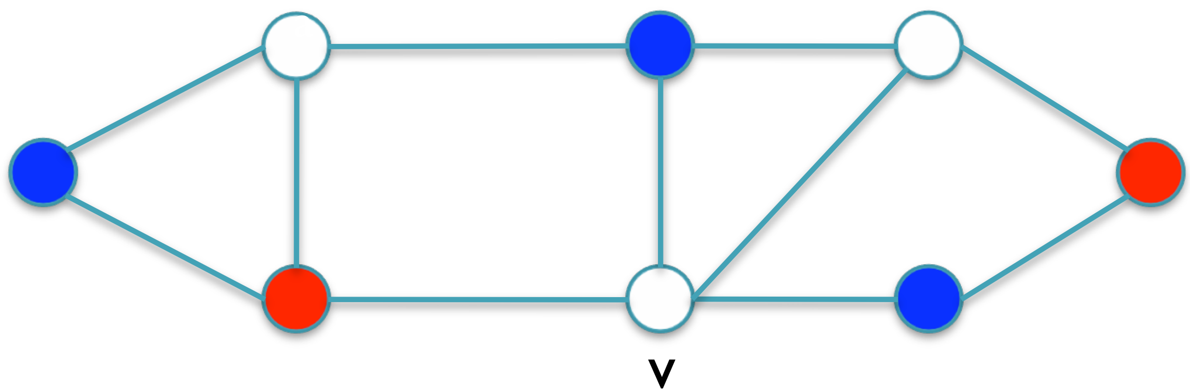
Assign v_i the smallest permissible color

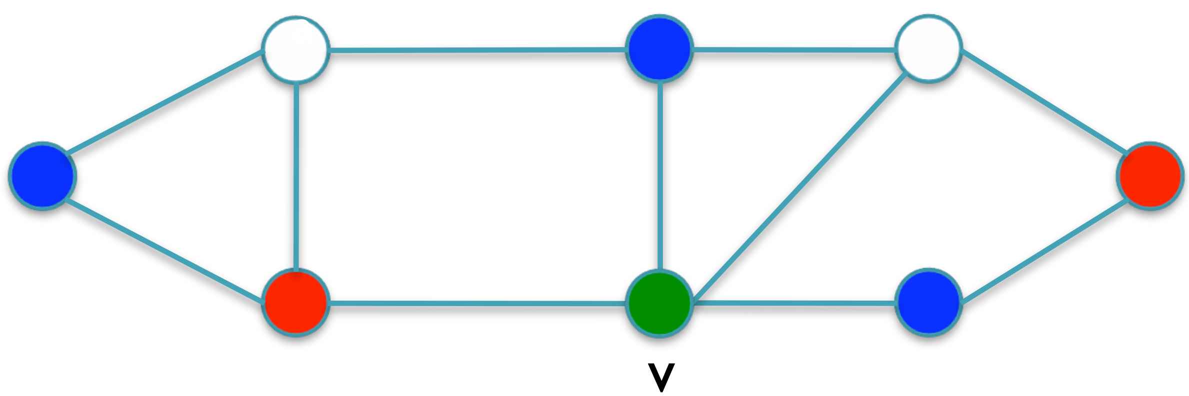
end-for

- A greedy heuristic usually gives a good, often optimal, solution
- The key is to find good orderings for coloring, and many have been developed

Ref: Gebremedhin, Tarafdar, Manne, Pothen, SIAM J. Sci. Compt. 29:1042–1072, 2007.

Greedy Coloring Algorithm



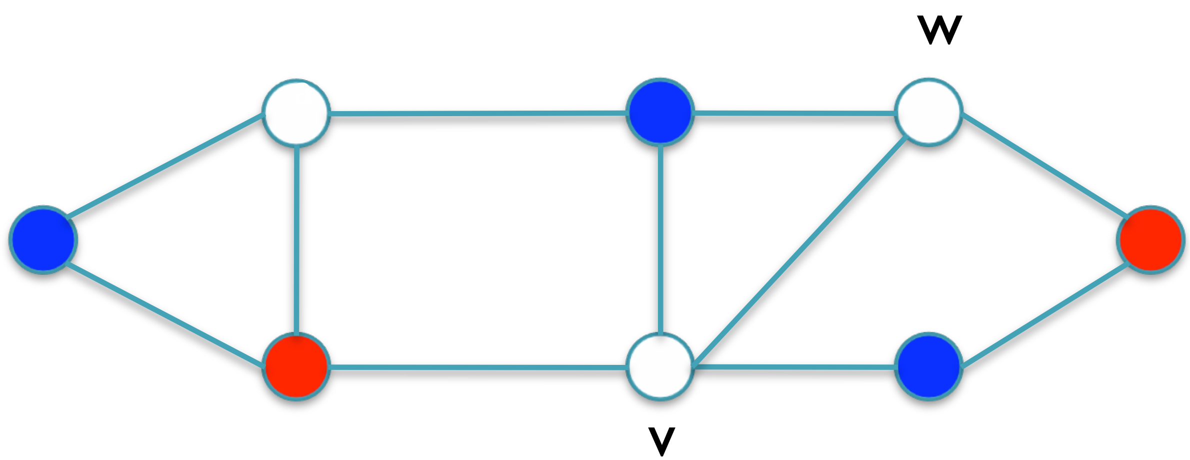


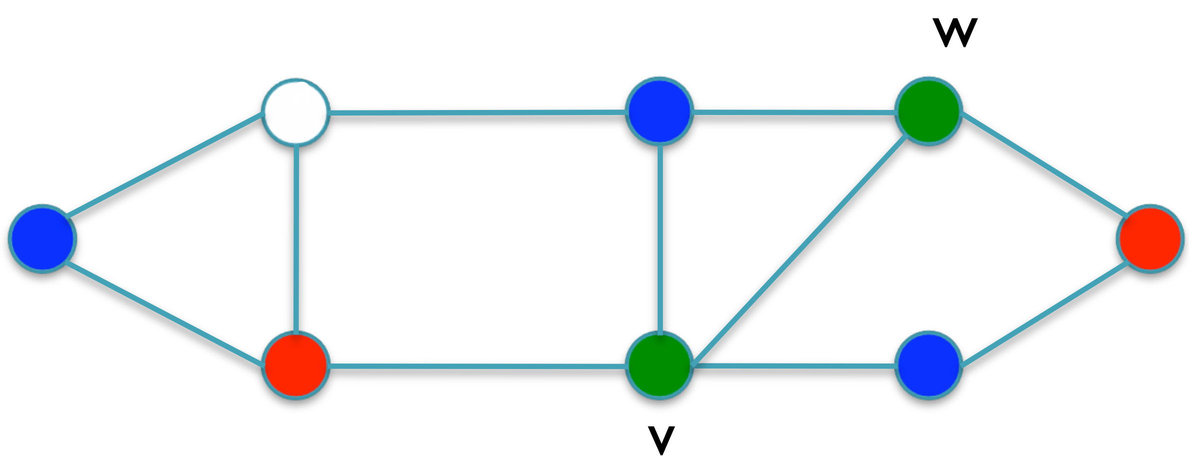
Many-core greedy coloring

- Given a graph, parallelize greedy coloring on many-core machines such that Speedup is attained, and Number of colors is roughly same as in serial
- Difficult task since greedy is inherently sequential, computation small relative to communication, and data accesses are irregular
- D1 coloring: Approaches based on Luby's parallel algorithm for maximal independent set had limited success
- Gebremedhin and Manne (2000) developed a parallel greedy coloring algorithm on shared memory machines
  - Uses speculative coloring to enhance concurrency, randomized partitioning to reduce conflicts, and serial conflict resolution
  - Number of conflicts bounded, so this approach yields an effective algorithm
  - Extended to distance-2 coloring by G, M and P (2002)
- We adapt this approach to implement the greedy algorithm for many-core computing

Parallel Coloring

Parallel Coloring: Speculation





Iterative Greedy Coloring: Parallel Algorithm

```
Proc IterativeGreedy (G = (V, E))
U is set of vertices to be colored, and R to be recolored
while U is not empty
        1. Speculatively color vertices
        for v ∈ U in parallel
            for each neighbor w of v
                Mark color[w] as forbidden to v
            Assign smallest available value to color[v]
        2. Detect conflicts and create recoloring list
        for v ∈ U in parallel
            for each neighbor w of v
                if color[w] = color[v]
                   add higher-numbered vertex to R
        U = R
end proc
```

Multithreaded: Data Flow

Multi-threaded: Data Flow

proc RecursiveDataflow $(G = (V, E))$
Set color[v] to zero, and bit to empty
for $v \in V$ **in parallel**
    Set state[v] using $\mathbf{int\_fetch\_add}$[state[$\mathbf{v}$], $\mathbf{1}$]
    if state[v] is 0 **then** PROCESS(v)
        First thread to process v
end proc

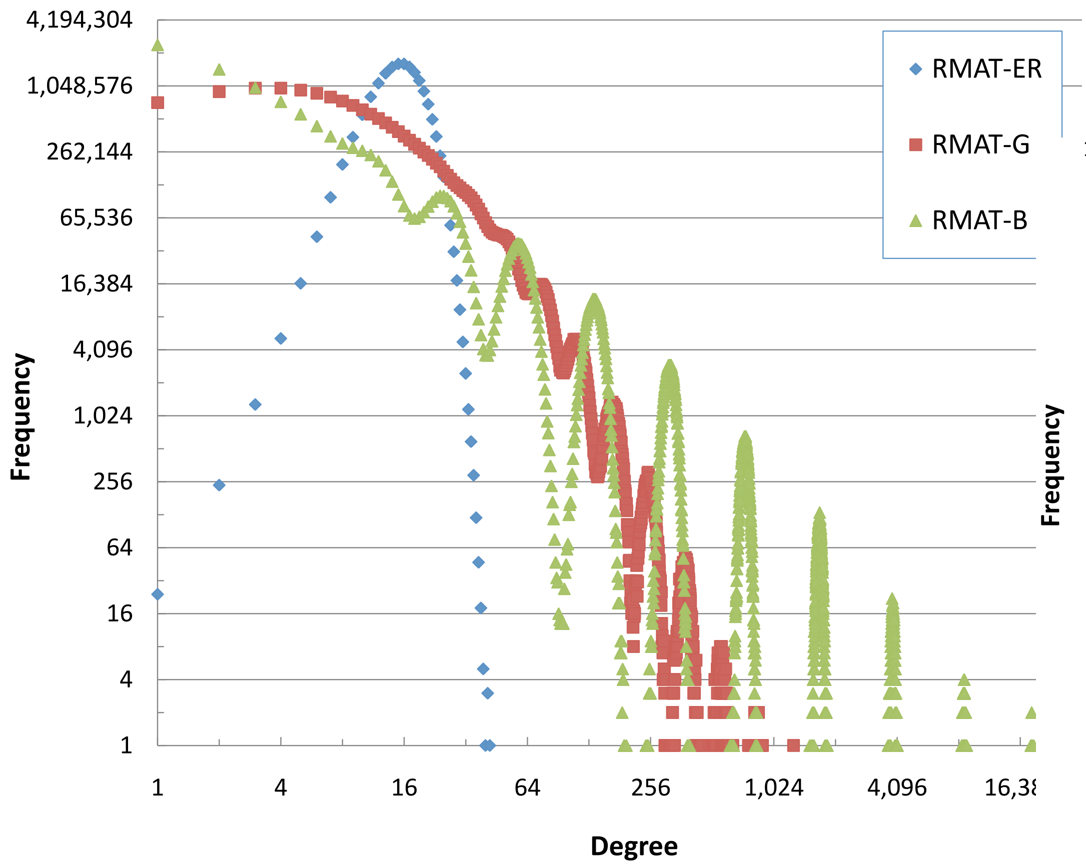
Multi-threaded: Data Flow

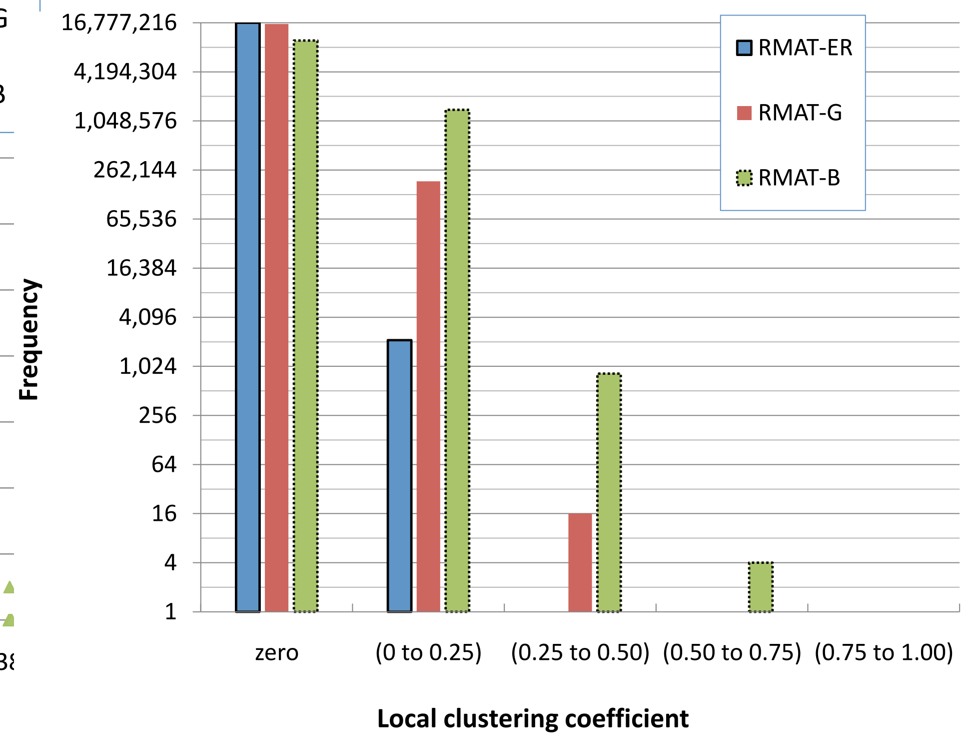
proc RecursiveDataflow $(G = (V, E))$
Set color$[v]$ to zero, and bit to empty
for $v \in V$ **in parallel**
    Set state$[v]$ using $\mathbf{int\_fetch\_add}$[state$[\mathbf{v}], \mathbf{1}$]
    if state$[v]$ is 0 **then** PROCESS(v)
        First thread to process v
end proc

proc PROCESS(v)
for each neighbor w with $w < v$
    Check state$[w]$ using $\mathbf{int\_fetch\_add}$[state$[w]$,1]
    if state$[w]$ is zero **then** PROCESS(w)
        First thread to process w
    end if
    $\mathbf{readff}$ color$[w]$
    Mark color$[w]$ as forbidden to v
Assign v the smallest available color
$\mathbf{writeef}$ color$[v]$
end proc

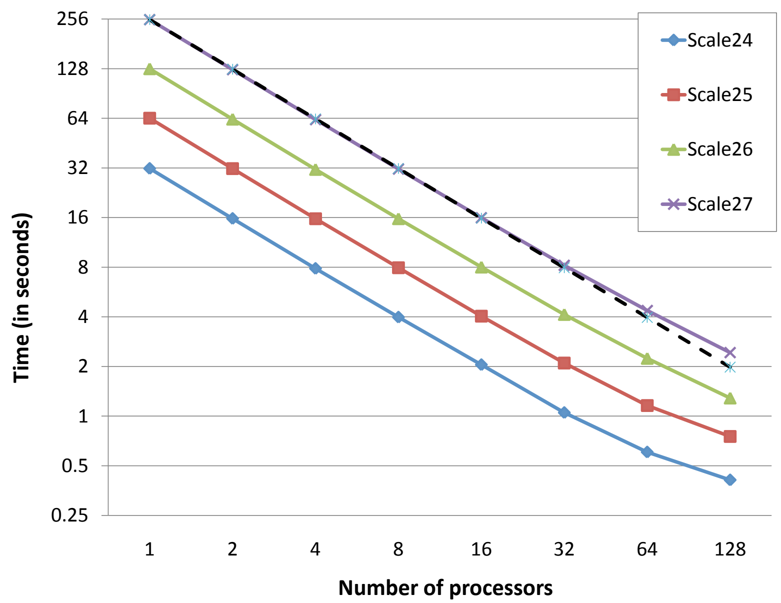
RMAT Graphs

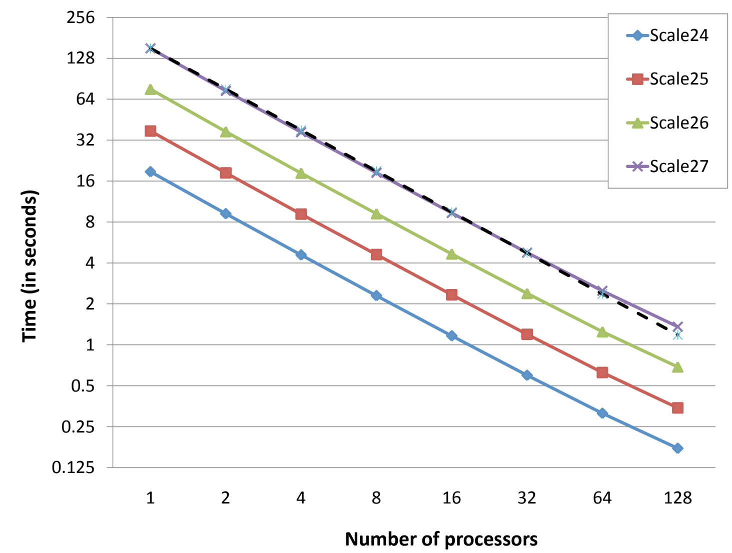
a	b
c	d





Experimental results





Iterative

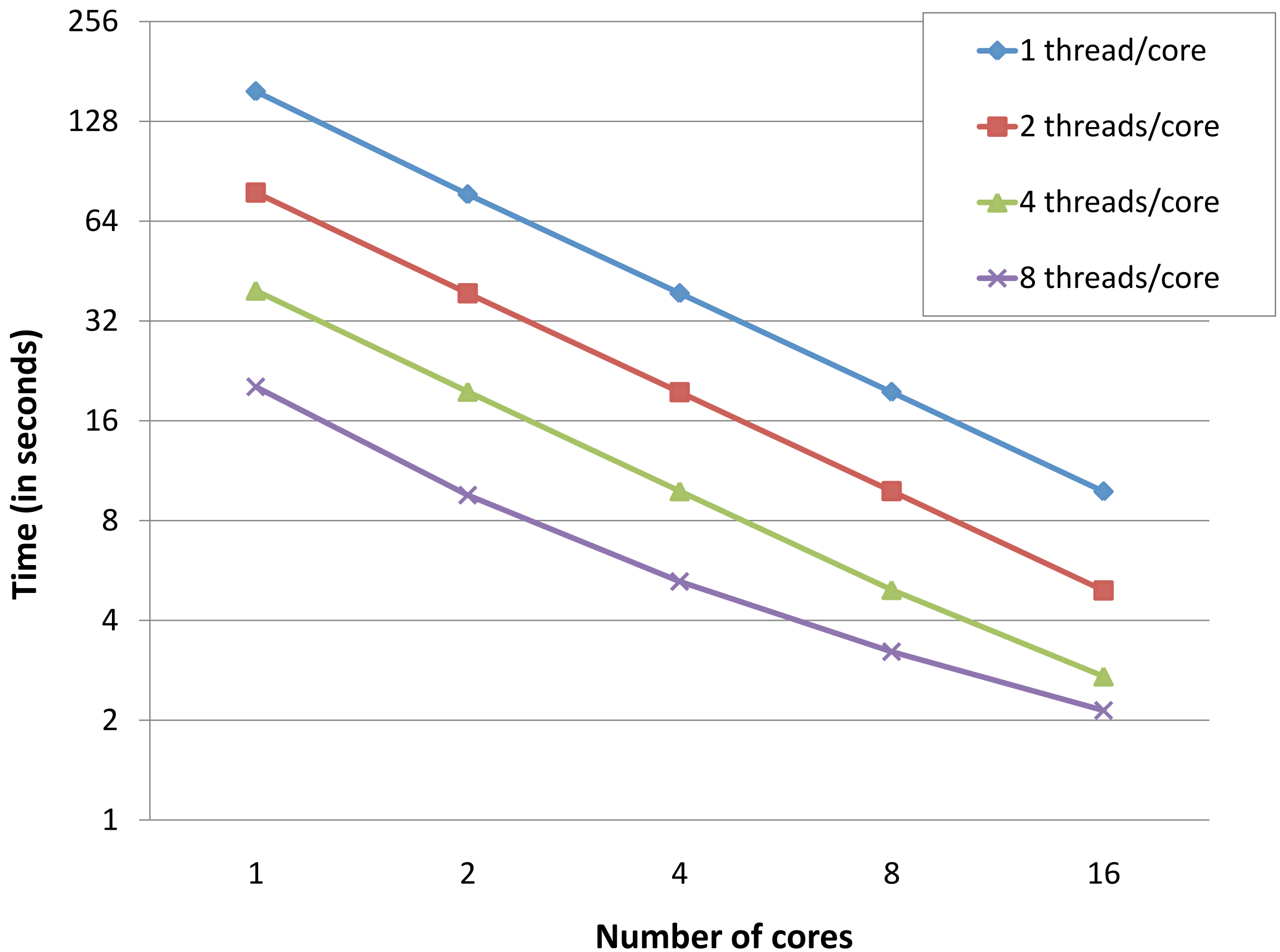
Dataflow

Cray XMT: RMAT-G with 2^{24}, …, 2^{27} vertices and 134M, …, 1B edges

Experimental results

Iterative

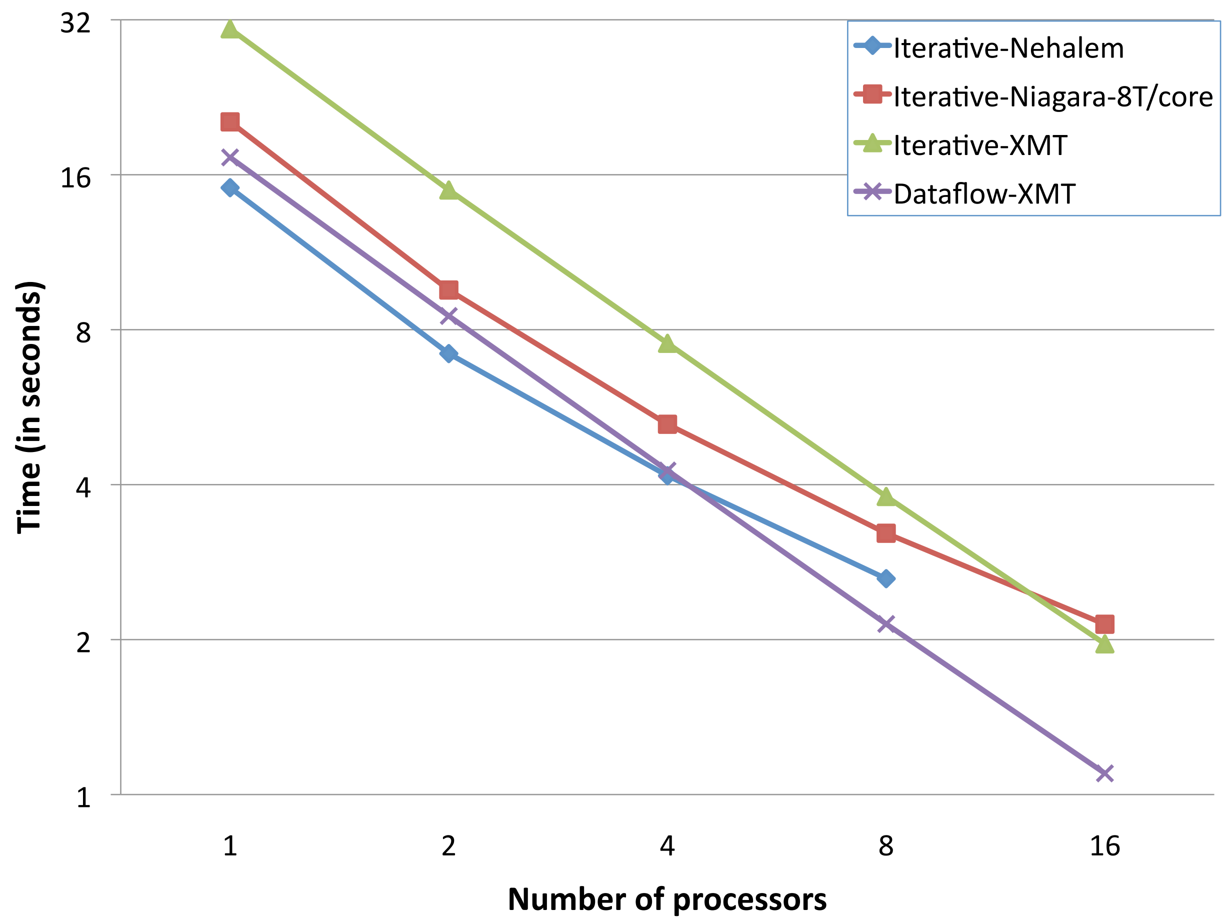
Niagara 2



Perf. With doubling threads on a core = Doubling cores!

Experimental results

RMAT-G with 2^{24} = 16M vertices and 134M edges



All Platforms

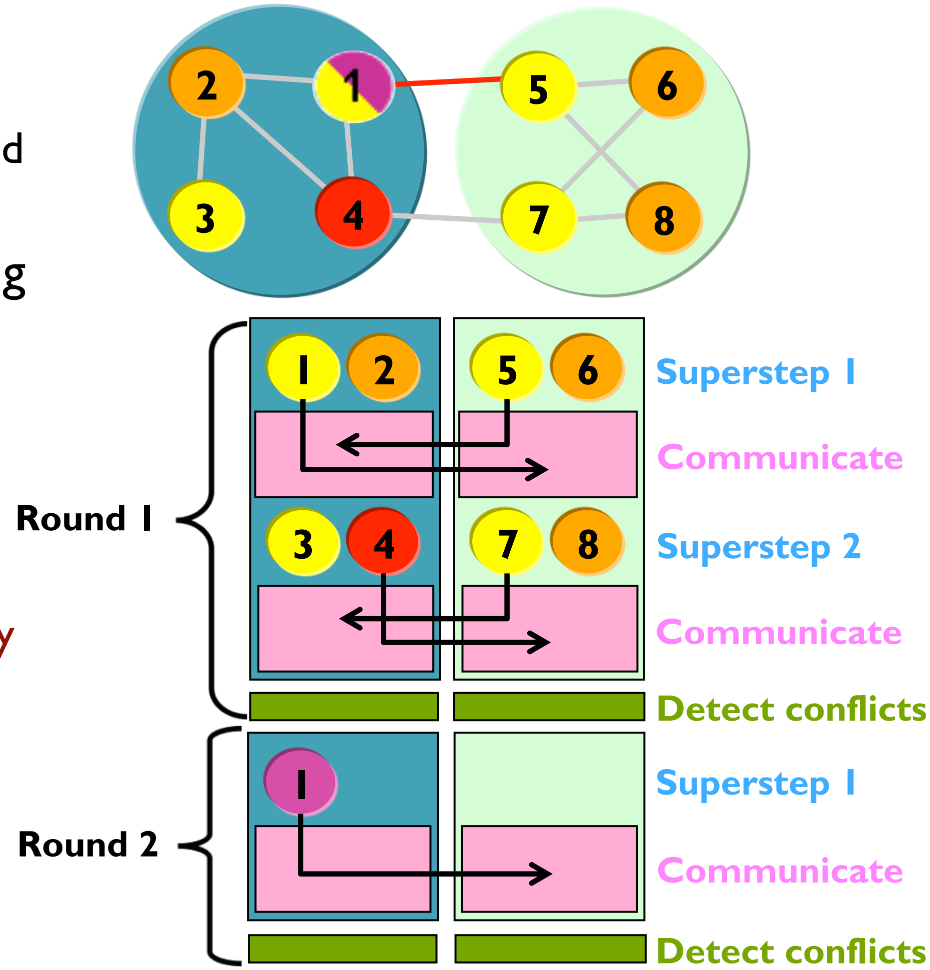
Distributed-Memory Parallel Machines

Parallelizing greedy coloring

- Goal: Given a distributed graph, parallelize greedy coloring such that
  - Speedup is attained
  - Number of colors used is roughly same as in serial
- Difficult task since greedy is inherently sequential, computation small relative to communication, and data accesses are irregular
- D1 coloring: approaches based on Luby's parallel algorithm for *maximal independent set* had very limited success
- D2 coloring: no practical parallel algorithms existed
- We developed a framework for effective parallelization of greedy coloring on distributed memory architectures
- Using the framework, we designed various specialized parallel algorithms for D1 and D2 coloring
  - First MPI implementations to yield speedup

Framework: Distributed-memory parallel Greedy Coloring

- Exploit features of initial data distribution
  - Distinguish between interior and boundary vertices
- Proceed in rounds, each having two phases:
  - Tentative coloring
  - Conflict detection
- Coloring phase organized in supersteps
  - A processor communicates only after coloring a subset of its assigned vertices
    ➔ infrequent, coarse-grain communication
- Randomization used in resolving conflicts



Specializations of the framework

PURDUE UNIVERSITY

Color selection strategies	• First Fit • Staggered First Fit
Coloring order	• Interior before boundary • Interior after boundary • Interior interleaved with boundary
Local vertex ordering	• Various degree-based techniques
Supersteps	• Synchronous • Asynchronous
Inter-processor communication	• Customized • Broadcast-based

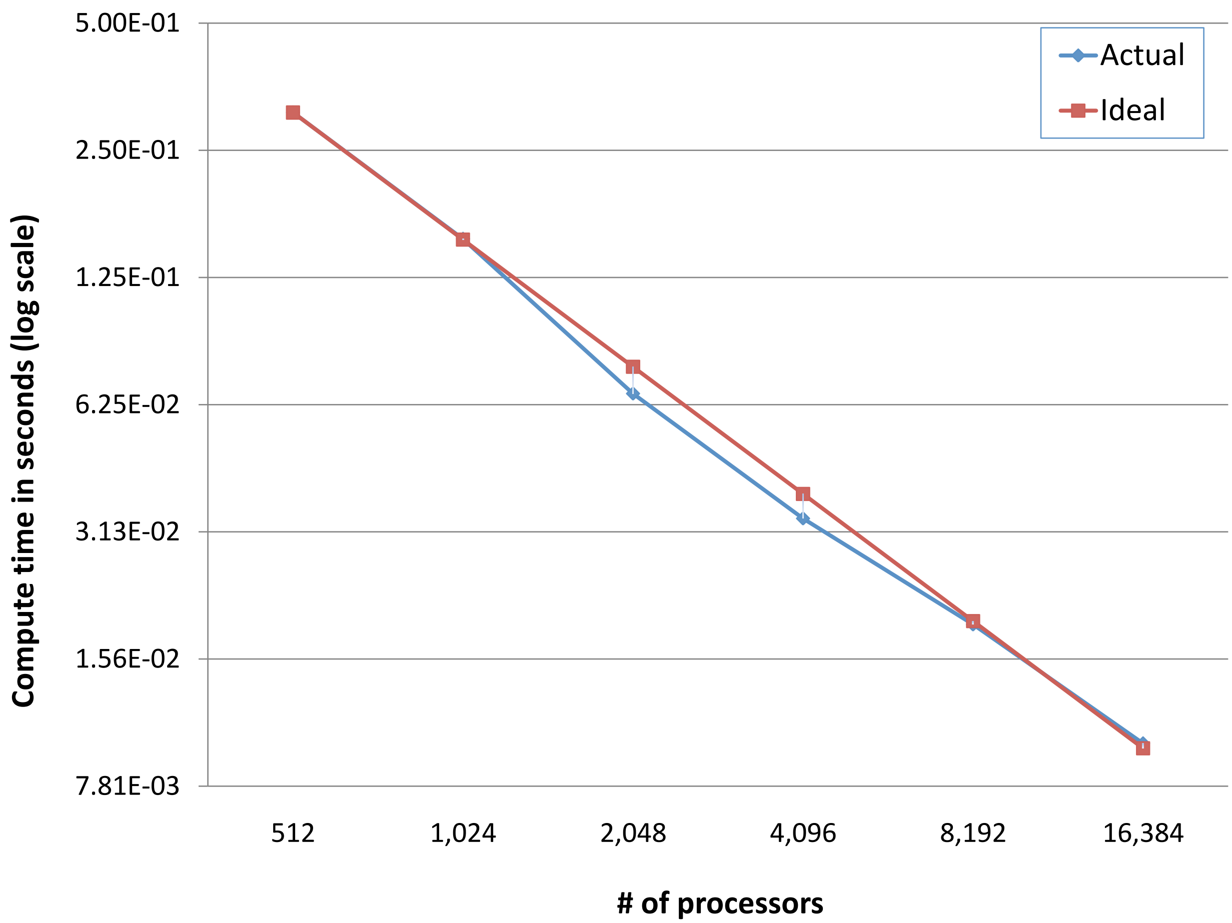
cscapes

Implementation and experimentation

- Using the framework (JPDC, 2008)
  - Designed specialized parallel algorithms for distance-1 coloring
  - Experimentally studied how to tune "parameters" according to
    - size, density, and distribution of input graph
    - number of processors
    - computational platform
- Extending the framework (SISC, under review)
  - Designed parallel algorithms for D2 and restricted star coloring (to support Hessian computation)
  - Designed parallel algorithms for D2 coloring of bipartite graphs (to support Jacobian computation)

  New Challenge: efficient mechanism for information exchange between processors hosting D2 neighboring vertices needs to be devised
- Software
  - MPI implementations of D1 and D2 coloring made available in Zoltan

D1-coloring, IBM Blue Gene/P



Our contributions: Parallel Coloring

- Global address-space machines
  - Parallel algorithms for distance-1 and distance-2 coloring on graphs for Jacobian and Hessian computations
  - Employs randomized graph partitioning, speculative coloring to improve scalability
- Distributed memory machines
  - Developed a parallelization framework for greedy coloring
  - Employs graph partitioning, speculative coloring, optimized communication granularity to achieve scalability on 16,000 processors IBM Blue Gene and Cray XT-5
  - Designed specialized parallel algorithms for D1 and D2 coloring
  - Deployed implementations via the Zoltan toolkit

Our contributions: Multithreaded Coloring

- Massive multithreading
  - Can tolerate memory latency for graphs/sparse matrices
  - Dataflow algorithms easier to implement than distributed memory versions
  - Thread concurrency ameliorates lack of caches, and lower clock speeds
  - Thread parallelism can be exploited at fine grain if supported by lightweight synchronization
  - Graph structure critically influences performance
- Many-core machines
  - Early days yet. Developed an iterative algorithm for greedy coloring that ports to many different machines.
  - X threads on one core can perform as well as 1 thread on X cores if simultaneous multithreading is supported
  - Decomposition into tasks at a finer grain than distributed-memory version, and need to relax synchronization to enhance concurrency in computational schedule
  - Will form nodes of Peta- and Exa-scale machines, so single node performance studies are needed

Coloring Algorithms: Our Contributions

- Serial algorithms and software
  - Jacobian computation via distance-2 coloring algorithms on bipartite graphs
  - Developed novel algorithms for acyclic, star, distance-k (k = 1,2) and other coloring problems; developed associated matrix recovery algorithms
  - Delivered implementations via the software package ColPack (released Oct. 2008)
  - Interfaced ColPack with the AD tool ADOL-C
- Application Highlights
  - Enabled Jacobian computation in Simulated Moving Beds
  - Enabled Hessian computation in optimizing electric power flow
- Parallel algorithms and software
  - Developed a parallelization framework for greedy coloring
  - Designed specialized parallel algorithms for D1 and D2 coloring
  - Deployed implementations via the Zoltan toolkit

Thanks

- Rob Bisseling, Erik Boman, Ümit Çatalürek, Karen Devine, Florin Dobrian, Assefaw Gebremedhin, Mahantesh Halappanavar, Paul Hovland, Gary Kumfert, Fredrik Manne, Ali Pınar, Sivan Toledo, Jean Utke

Further reading
www.cscapes.org

- Gebremedhin and Manne, Scalable parallel graph coloring algorithms, *Concurrency: Practice and Experience,* 12: 1131-1146, 2000.
- Gebremedhin, Manne and Pothen, Parallel distance-k coloring algorithms for numerical optimization, *Lecture Notes in Computer Science,* 2400: 912-921, 2002.
- Bozdag, Gebremedhin, Manne, Boman and Catalyurek. A framework for scalable greedy coloring on distributed-memory parallel computers. *J. Parallel Distrib. Comput.* 68(4):515-535, 2008.
- Catalyurek, Feo, Gebremedhin, Halappanavar and Pothen, Multi-threaded algorithms for graph coloring, *Submitted,* Aug. 2010.

Further reading
www.cscapes.org

- Gebremedhin, Manne and Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review 47(4):627—705, 2005.*
- Gebremedhin, Tarafdar, Manne and Pothen. New acyclic and star coloring algorithms with applications to computing Hessians. *SIAM J. Sci. Comput. 29:1042—1072, 2007.*
- Gebremedhin, Pothen and Walther. Exploiting sparsity in Jacobian computation via coloring and automatic differentiation: a case study in a Simulated Moving Bed process. *AD2008, LNCSE 64:339---349, 2008.*
- Gebremedhin, Pothen, Tarafdar and Walther. Efficient computation of sparse Hessians using coloring and Automatic Differentiation. *INFORMS Journal on Computing, 21:209---223, 2009.*
- Bozdag, Gebremedhin, Manne, Boman and Catalyurek. A framework for scalable greedy coloring on distributed-memory parallel computers. *J. Parallel Distrib. Comput. 68(4):515—535, 2008.*
- Gebremedhin, Nguyen, Patwary and Pothen. ColPack: Graph Coloring Software for Derivative Computation and Beyond, *Submitted, Oct. 2010.*

Applications Also Getting More Complex

- Leading edge scientific applications increasingly include:
  - Adaptive, unstructured data structures
  - Complex, multiphysics simulations
  - Multiscale computations in space and time
  - Complex synchronizations (e.g. discrete events)

- Significant parallelization challenges on today's machines
  - Finite degree of coarse-grained parallelism
  - Load balancing and memory hierarchy optimization

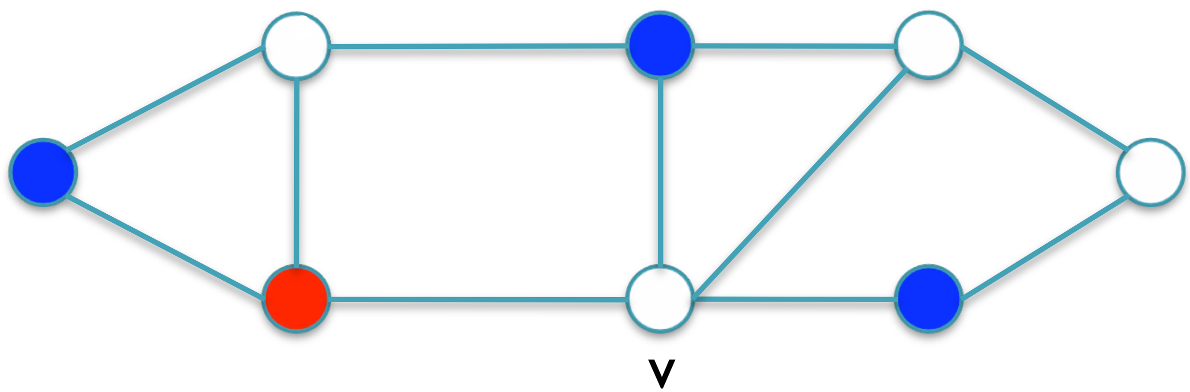
- Dramatically harder on millions of cores

- Huge need for new algorithmic ideas – CSC will be critical

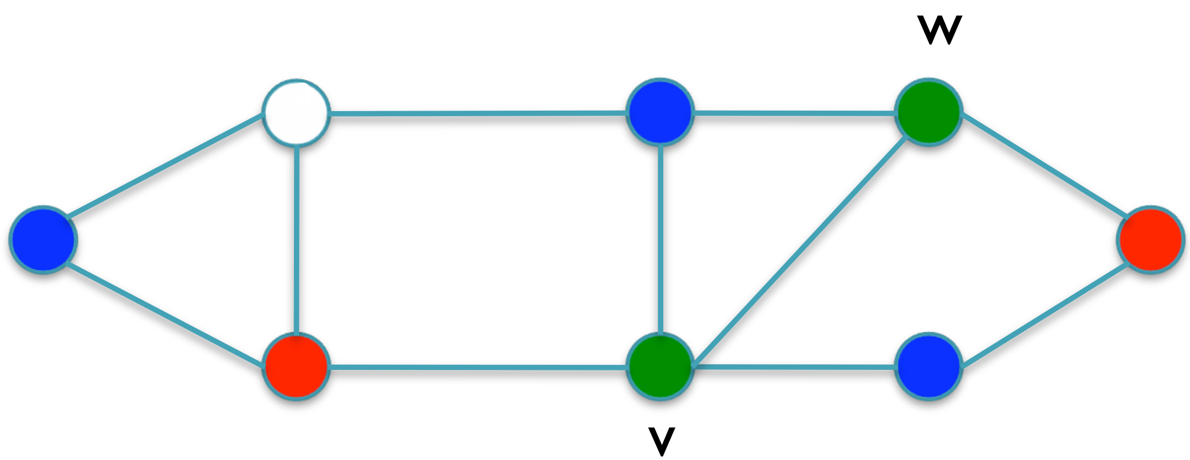
Architectural Challenges for Graph Algorithms

- Runtime is dominated by latency
  - Particularly true for data-centric applications
  - Random accesses to global address space
  - Perhaps many at once – fine-grained parallelism
- Essentially no computation to hide access time
- Access pattern is data dependent
  - Prefetching unlikely to help
  - Usually only want small part of cache line
- Potentially abysmal locality at all levels of memory hierarchy

PURDUE UNIVERSITY



cscapes


w
v