Performance Scalability on Embedded Many-Core Processors

Michael Champigny
Research Scientist
Advanced Computing Solutions
Mercury Computer Systems

2010 HPEC Workshop
September 15, 2010

© 2010 Mercury Computer Systems, Inc.

Outline

- Motivation
  - Single-chip parallelism and convergence
  - Variability challenges
- Dynamic scheduling
  - Task Parallelism
  - Load balancing
  - Work stealing
- Runtime design
  - Parallel runtime
- Scalability study
  - Data structure considerations

© 2010 Mercury Computer Systems, Inc.

Many-Core in Embedded HPC

- Large scale parallel chip multiprocessors are here
  - Power efficient
  - Small form factors
  - e.g., Tilera TILEPro64
- Convergence is inevitable for many workloads
  - Multi-board solutions became multi-socket solutions
  - …and multi-socket solutions will become single-socket solutions
  - e.g., ISR tasks will share a processor
- Software is a growing challenge
  - How do I scale my algorithms and applications?
  - …without rewriting them?
  - …and improve productivity?

© 2010 Mercury Computer Systems, Inc.

Sources of Variability

- Chip multiprocessors introduce variability to workloads
  - cc-NUMA
  - Memory hierarchies and block sizes
  - Asymmetries in processing elements due to
    - Thermal conditions
    - Process variation
    - Faults
- Workloads themselves are increasingly data-driven
  - Data dependencies lead to processor stalls
  - Complex state machines, branching, pointer chasing
- Convergence compounds the problem
  - Adversarial behavior of software components sharing resources

© 2010 Mercury Computer Systems, Inc.

Importance of Load Balancing

- Mapping algorithms to physical resources is painful
  - Requires significant analysis on a particular architecture
  - Doesn't translate well to different architectures
  - Mapping must be revisited as processing elements increase

- Static partitioning is no longer effective for many problems
  - Variability due to convergence and data-driven applications
  - Processing resources are not optimally utilized
    - e.g., Processor cores can become idle while work remains

- Load balancing must be performed dynamically
  - Language
  - Compiler
  - Runtime

© 2010 Mercury Computer Systems, Inc.

Task Parallelism & Cache-Oblivious Algorithms

- Load balancing requires small units of work to fill idle “gaps”
  - Fine-grained task parallelism
- Exposing all fine-grained parallelism at once is problematic
  - Excessive memory pressure
- *Cache-oblivious* algorithms have proven low cache complexity
  - Minimize number of memory transactions
  - Scale well unmodified on any cache-coherent parallel architecture
  - Based on divide-and-conquer method of algorithm design
    - Tasks only subdivided on demand when a processor idles
    - Tasks create subtasks recursively until a cutoff
    - Leaf tasks fit in private caches of all processors

© 2010 Mercury Computer Systems, Inc.

Scheduling Tasks on Many-Cores

- Runtime schedulers assign tasks to processing resources
  - Greedy: make decisions only when required (i.e., idle processor)
  - Ensure maximum utilization of available computes
  - Have knowledge of instantaneous system state
- Scheduler must be highly optimized for use by many threads
  - Limit sharing of data structures to ensure scalability
  - Any overhead in scheduler will impact algorithm performance
- *Work-stealing* based schedulers are provably efficient
  - Provide dynamic load balancing capability
  - Idle cores look for work to "steal" from other cores
  - Employ heuristics to improve locality and cache reuse

© 2010 Mercury Computer Systems, Inc.

Designing a Parallel Runtime for Many-Core

- Re-architected our dynamic scheduler for many-core
  - Chimera Parallel Programming Platform
  - Expose parallelism in C/C++ code incrementally using C++ compiler
  - Ported to several many-core architectures from different vendors

- Insights gained improved general performance scalability
  - Affinity-based work-stealing policy optimized for cc-NUMA
  - Virtual NUMA topology used to improve data locality
  - Core data structures adapt to current runtime conditions
  - Tasks are grouped into NUMA-friendly clusters to amortize steal cost
  - Dynamic load balancing across OpenCL and CUDA supported devices
  - No performance penalty for low numbers of cores (i.e., multi-core)

© 2010 Mercury Computer Systems, Inc.

Work-Stealing Scheduling Basics

- Cores operate on local tasks (i.e., work) until they run out
  - A core operating on local work is in the **work state**
  - When a core becomes idle it looks for work at a **victim** core
  - This operation is called **stealing** and the perpetrator is labeled a **thief**
  - This cycle is repeated until work is found or no more work exists
  - A thief looking for work is in the **idle state**
  - When all cores are idle the system reaches **quiescent state**

- Basic principles of optimizing a work-stealing scheduler
  - Keep cores in work state for as long as possible
    - This is good for locality as local work stays in private caches
  - Stealing is expensive so attempt to minimize it and to amortize cost
    - Stealing larger-grained work is preferable
  - Choose your victim wisely
    - Stealing from NUMA neighbor is preferable

© 2010 Mercury Computer Systems, Inc.

Work-Stealing Implications on Scheduler Design

- Work-stealing algorithm leads to many design decisions
  - What criteria to apply to choose a victim?
  - How to store pending work (i.e., tasks)?
  - What to do when system enters quiescent state?
  - How much work to steal?
  - Distribute work (i.e., load sharing)?
  - Periodically rebalance work?
  - Actively monitor/sample the runtime state?

© 2010 Mercury Computer Systems, Inc.

Example: Victim Selection Policy on Many-Core

- Victim selection policy
  - When a core becomes idle which core do I try to steal from?
- Several choice are available
  - Randomized order
  - Linear order
  - NUMA order
- We found NUMA ordering provided better scalability
- Benefits became more pronounced with larger numbers of cores

© 2010 Mercury Computer Systems, Inc.

Optimal Amount of Tasks to Steal

- When work is stolen how much do we take from the victim?
  - If we take too much
    - …victim will begin looking for work too soon
  - If we don't take enough
    - …thief begins looking for work too soon
- We conducted an empirical study to determine the best strategy
- Intuitively, stealing half the available work should be optimal

© 2010 Mercury Computer Systems, Inc.

Impact of Steal Amount Policy on Data Structures

- Steal a single task at a time
  - Implemented with any linear structure (i.e., dynamic array)
  - Allows for concurrent operation at both ends
    - …without locks in some cases
- Steal a block of tasks at a time
  - Implemented with a linear structure of blocks
    - Each block contains at most a fixed number of tasks
    - Can lead to load imbalance in some situations
      - If few tasks exist in system one core could own them all
- Steal a fraction of available tasks at a time
  - We picked 0.5 as the fraction to steal
  - Data structure is a more complex list of trees

© 2010 Mercury Computer Systems, Inc.

Empirical Study of Steal Amount on Many-Core

- Determine steal amount policy impact on performance scalability
  - Scalability defined as ratio of single core to *P* core latency

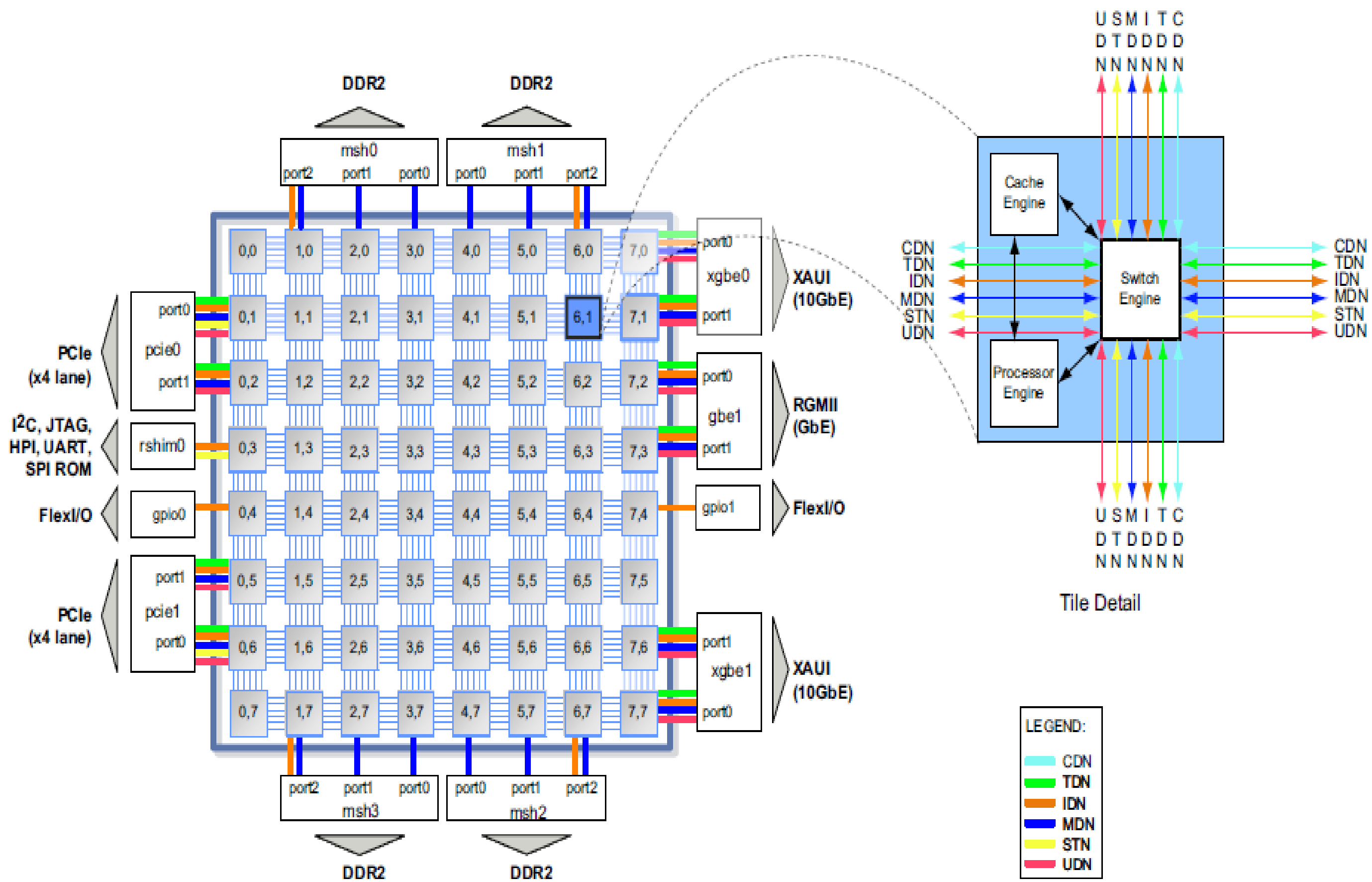
- Run experiment on existing many-core embedded processor
  - Tilera TILEPro64 using 56 cores
  - GNU compiler 4.4.3
  - SMP Linux 2.6.26

- Used Mercury Chimera as parallel runtime platform

- Modify existing industry standard benchmarks for task parallelism
  - Barcelona OpenMP Task Suite 1.1
  - MIT Cilk 5.4.6
  - Best-of-10 latency used for scalability calculation

© 2010 Mercury Computer Systems, Inc.

Tilera TILEPro64 Processor Architecture



Tilera TILEPro64 Processor Features

- Processing
  - 64 tiles arranged in 8 × 8 grid @ 23W
  - 866 MHz clock
  - 32-bit VLIW ISA with 64-bit instruction bundles (3 ops/cycle)
- Communication
  - iMesh 2D on-chip interconnect fabric
  - 1 cycle latency per tile-tile hop
- Memory
  - Dynamic Distributed Cache
    - Aggregates L2 caches into coherent 4 Mbytes L3 cache
    - 5.6 Mbytes combined on-chip cache

© 2010 Mercury Computer Systems, Inc.

Task Parallel Benchmarks

Benchmark	Source	Domain	Cutoff	Description
FFT	BOTS	Spectral	128	1M point, FFTW generated
Fibonacci	BOTS	Micro	10	Compute 45th number
Heat	Cilk	Solver	512	Diffusion, 16M point mesh
MatrixMult	Cilk	Dense Linear	16	512×512 square matrices
NQueens	BOTS	Search	3	13×13 chessboard
PartialLU	Cilk	Dense Linear	32	1M point matrix
SparseLU	BOTS	Sparse Linear	20	2K×2K sparse matrix
Sort	BOTS	Sort	2048, 20	20M 4-byte integers
StrassenMult	BOTS	Dense Linear	64, 3	1M point matrices

© 2010 Mercury Computer Systems, Inc.

Example: FFT Twiddle Factor Generator (Serial)

```
void fft_twiddle_gen (int i, int i1, COMPLEX* in,
 COMPLEX* out, COMPLEX* W, int n, int nW, int r, int m)
{
  if (i == (i1 - 1))
    fft_twiddle_gen1 (in+i, out+i, W, r, m, n, nW*i,
      nW*m);
  else {
     int i2 = (i + i1) / 2;
     fft_twiddle_gen (i, i2, in, out, W, n, nW, r, m);
     fft_twiddle_gen (i2, i1, in, out, W, n, nW, r, m);
  }
}
```

© 2010 Mercury Computer Systems, Inc.

Example: FFT Twiddle Factor Generator (OpenMP)

```
void fft_twiddle_gen (int i, int i1, COMPLEX* in,
 COMPLEX* out, COMPLEX* W, int n, int nW, int r, int m)
{
  if (i == (i1 - 1))
    fft_twiddle_gen1 (in+i, out+i, W, r, m, n, nW*i,
      nW*m);
  else {
     int i2 = (i + i1) / 2;
     #pragma omp task untied
     fft_twiddle_gen (i, i2, in, out, W, n, nW, r, m);
     #pragma omp task untied
     fft_twiddle_gen (i2, i1, in, out, W, n, nW, r, m);
     #pragma omp taskwait
  }
}
```

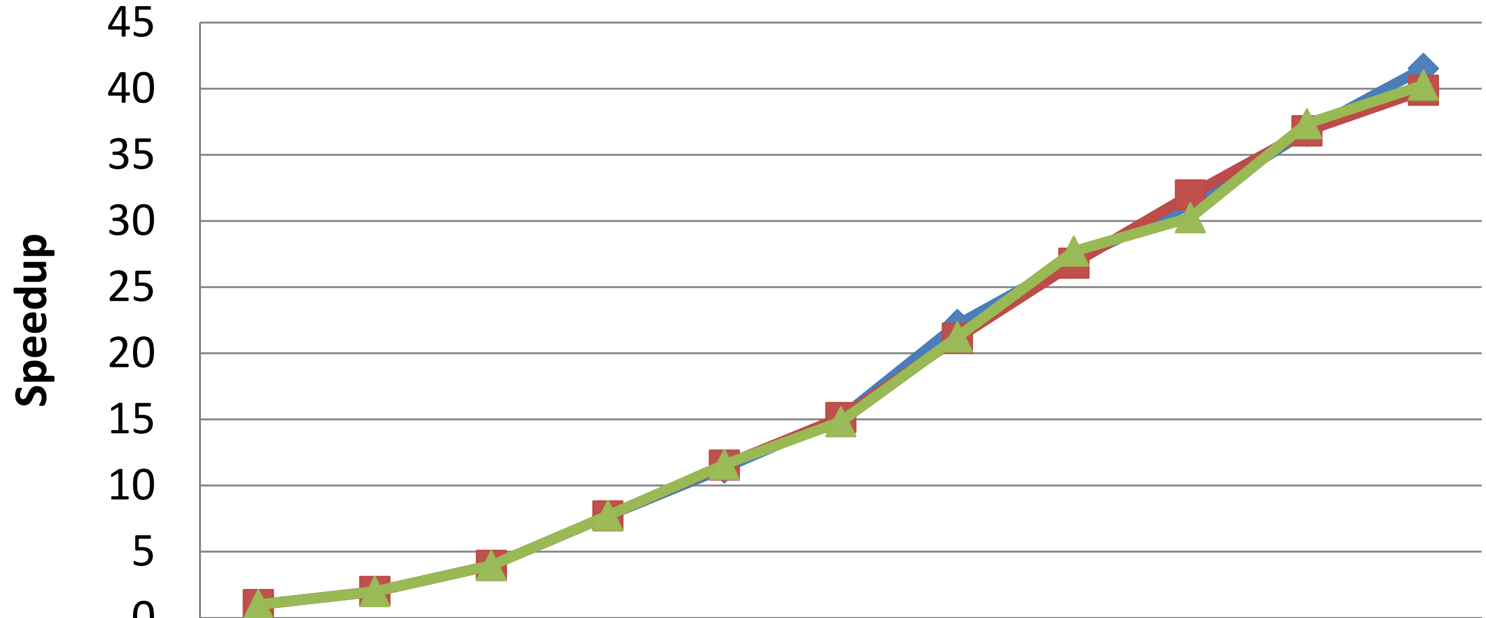
© 2010 Mercury Computer Systems, Inc.

Example: FFT Twiddle Factor Generator (Chimera)

```
void fft_twiddle_gen parallel (int i, int i1,
 COMPLEX* in, COMPLEX* out, COMPLEX* W, int n, int nW,
 int r, int m)
{
  if (i == (i1 - 1))
    fft_twiddle_gen1 (in+i, out+i, W, r, m, n, nW*i,nW*m);
  else join {
     int i2 = (i + i1) / 2;
     fork (fft_twiddle_gen, i, i2, in, out, W, n, nW,r,m);
     fork (fft_twiddle_gen, i2, i1, in, out, W, n,nW,r,m);
  }
}
```

© 2010 Mercury Computer Systems, Inc.

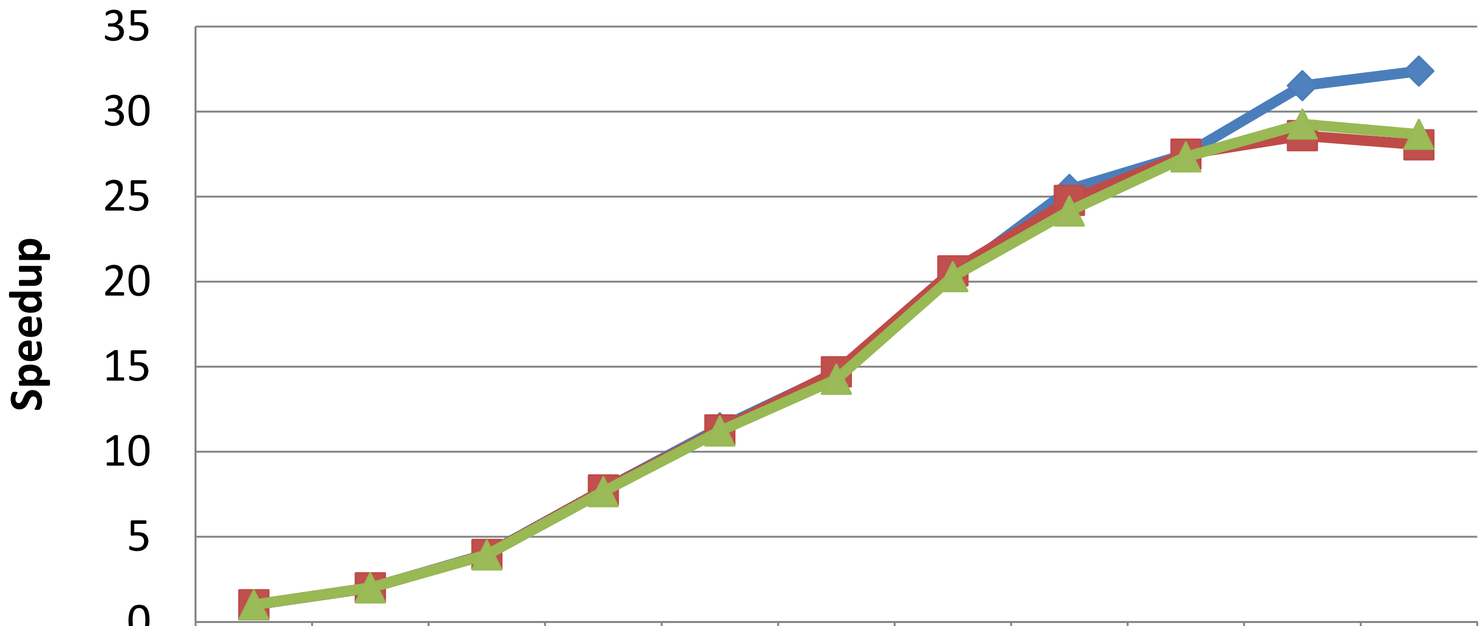
BOTS: Fibonacci



	1	2	4	8	12	16	24	32	36	48	56
Half	1	1.99	3.96	7.71	11.3	15.1	22.2	27	31.3	36.8	41.5
Block	1	1.99	3.96	7.71	11.6	15.1	21.1	26.8	32	36.8	39.9
Single	1	1.99	3.96	7.73	11.6	14.8	21.2	27.7	30.3	37.4	40.3

© 2010 Mercury Computer Systems, Inc.

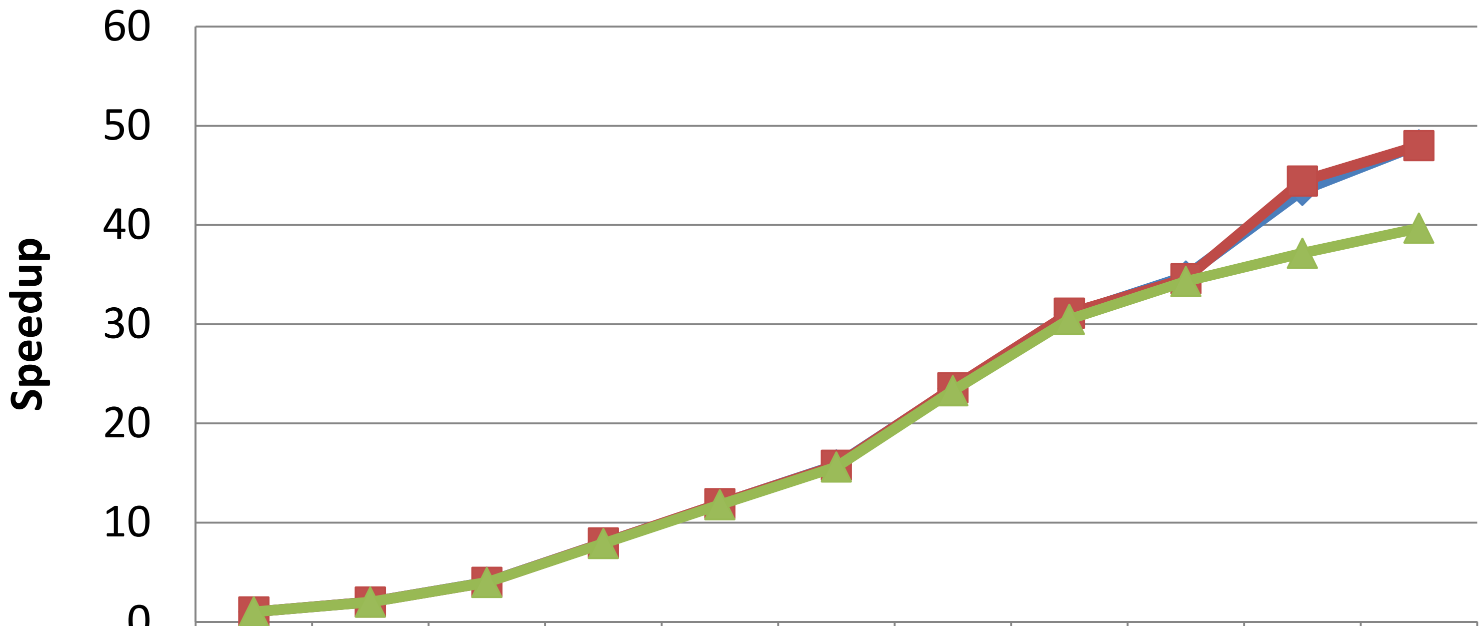
BOTS: Fast Fourier Transform



	1	2	4	8	12	16	24	32	36	48	56
Half	1	2	3.98	7.76	11.4	14.7	20.3	25.4	27.5	31.5	32.4
Block	1	2	3.96	7.75	11.3	14.7	20.6	24.8	27.5	28.6	28.1
Single	1	2.01	3.94	7.66	11.2	14.3	20.3	24.2	27.3	29.3	28.7

© 2010 Mercury Computer Systems, Inc.

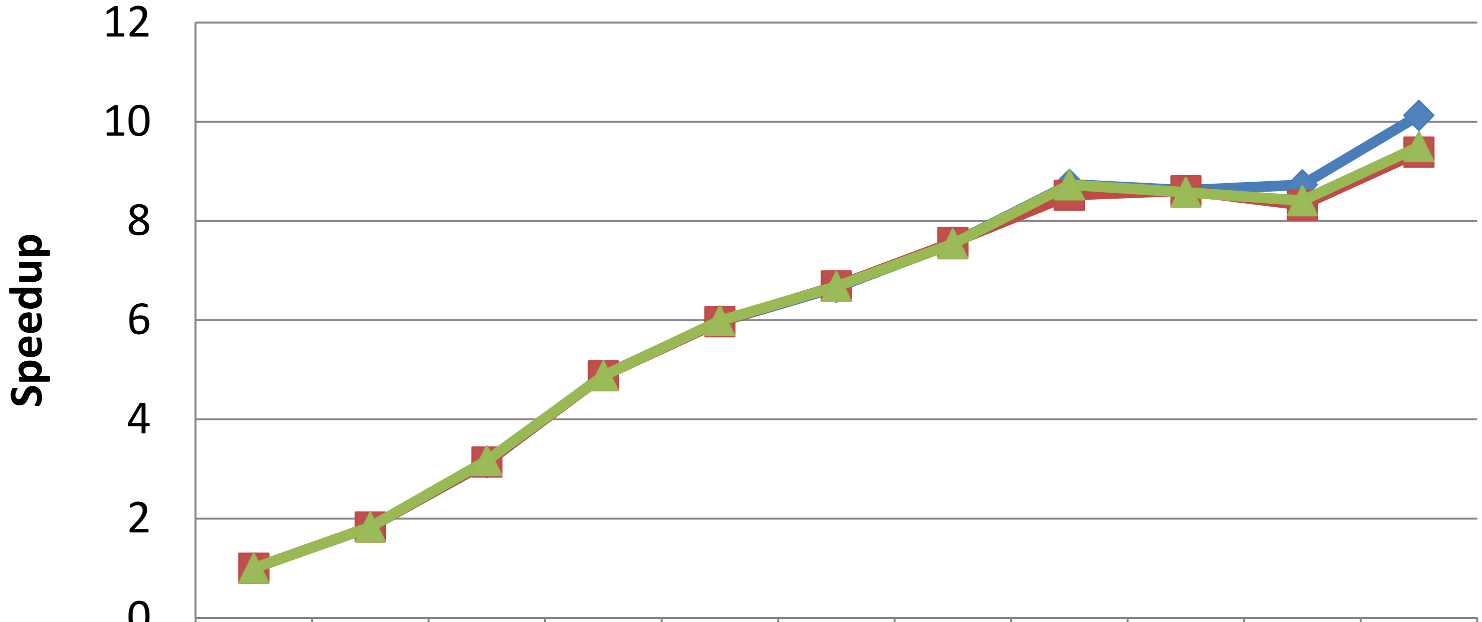
Cilk: Matrix-Matrix Multiply



	1	2	4	8	12	16	24	32	36	48	56
Half	1	2	3.99	7.95	11.9	15.8	23.5	31	34.9	43.5	48.1
Block	1	2	3.98	7.95	11.9	15.8	23.6	31.1	34.6	44.4	48
Single	1	1.99	3.97	7.89	11.8	15.6	23.3	30.5	34.3	37.2	39.7

© 2010 Mercury Computer Systems, Inc.

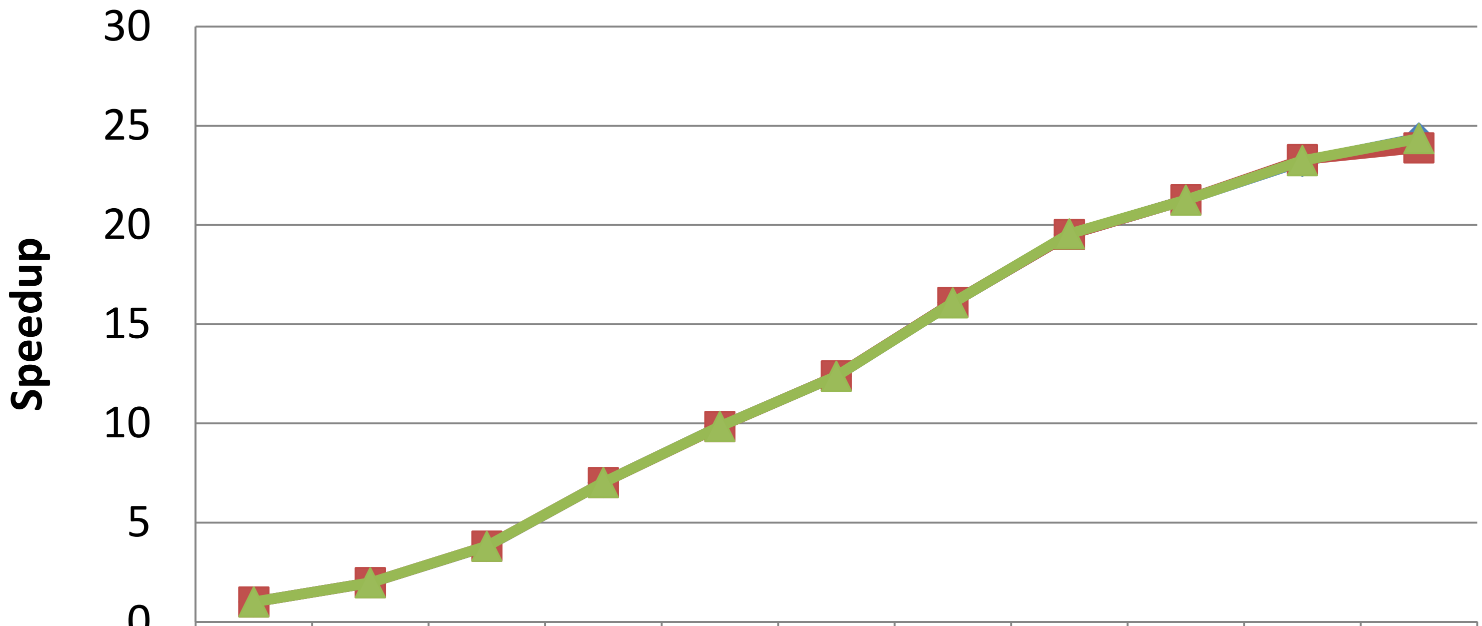
BOTS: Strassen Matrix-Matrix Multiply



	1	2	4	8	12	16	24	32	36	48	56
Half	1	1.83	3.13	4.89	5.97	6.66	7.55	8.74	8.62	8.73	10.1
Block	1	1.83	3.14	4.88	5.97	6.69	7.57	8.52	8.61	8.32	9.39
Single	1	1.83	3.17	4.88	5.99	6.68	7.54	8.73	8.58	8.41	9.5

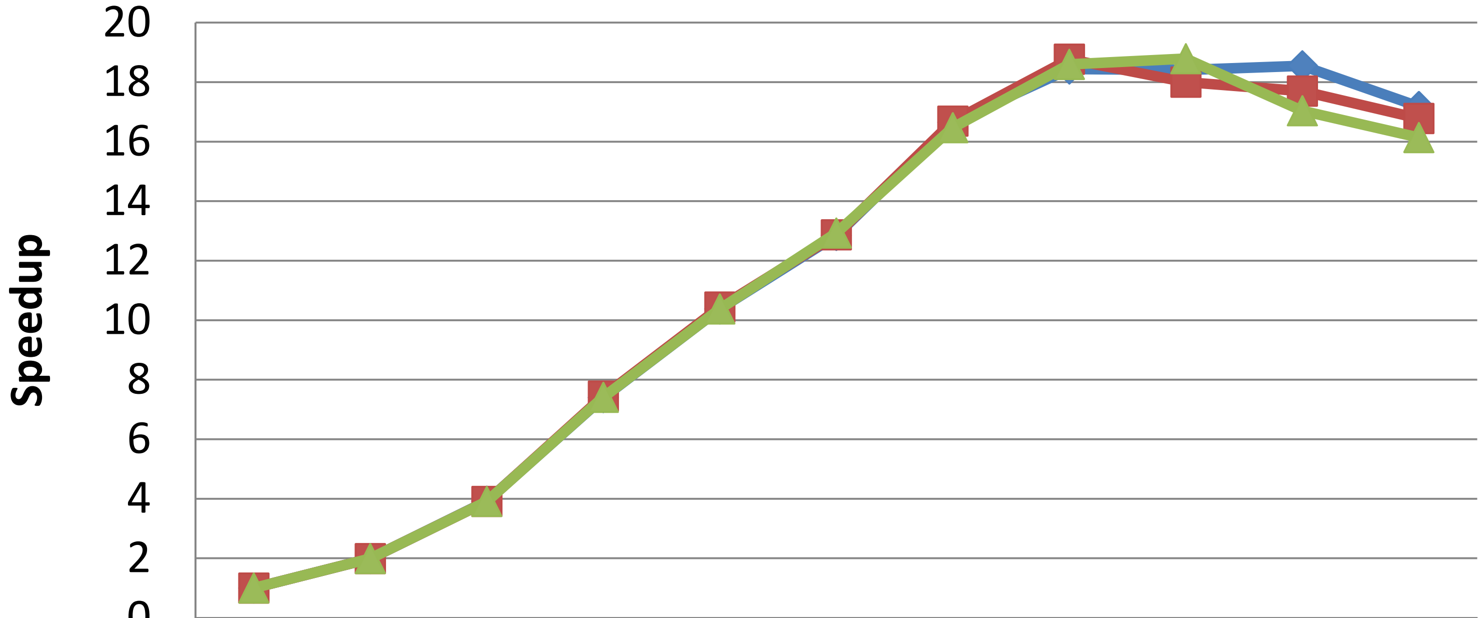
© 2010 Mercury Computer Systems, Inc.

BOTS: Sparse LU Factorization



	1	2	4	8	12	16	24	32	36	48	56
Half	1	1.97	3.8	7.02	9.85	12.4	16.1	19.5	21.3	23.2	24.4
Block	1	1.97	3.81	7.02	9.84	12.4	16.1	19.5	21.3	23.3	23.9
Single	1	1.97	3.81	7.02	9.85	12.4	16.1	19.6	21.3	23.3	24.4

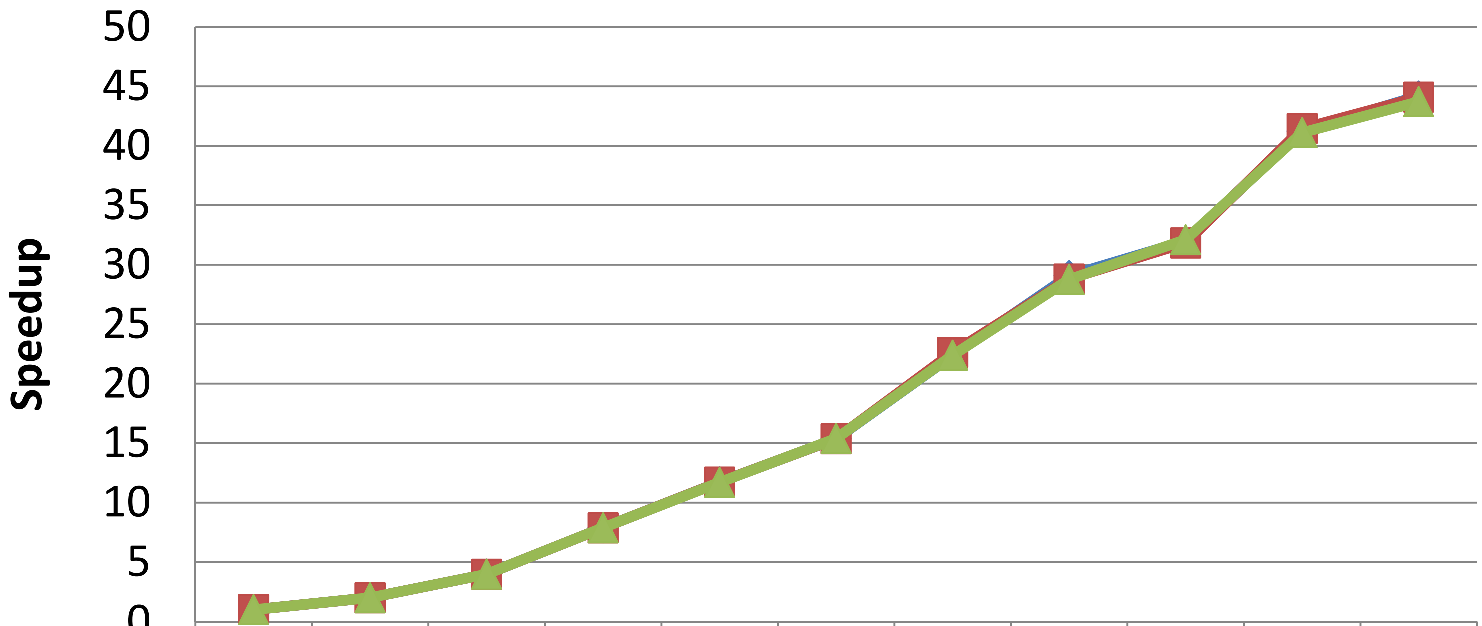
Cilk: Partial Pivoting LU Decomposition



	1	2	4	8	12	16	24	32	36	48	56
Half	1	1.99	3.92	7.4	10.4	12.9	16.7	18.4	18.4	18.6	17.2
Block	1	1.99	3.91	7.45	10.4	12.9	16.7	18.8	18	17.7	16.8
Single	1	1.99	3.91	7.41	10.4	12.9	16.5	18.6	18.8	17	16.1

© 2010 Mercury Computer Systems, Inc.

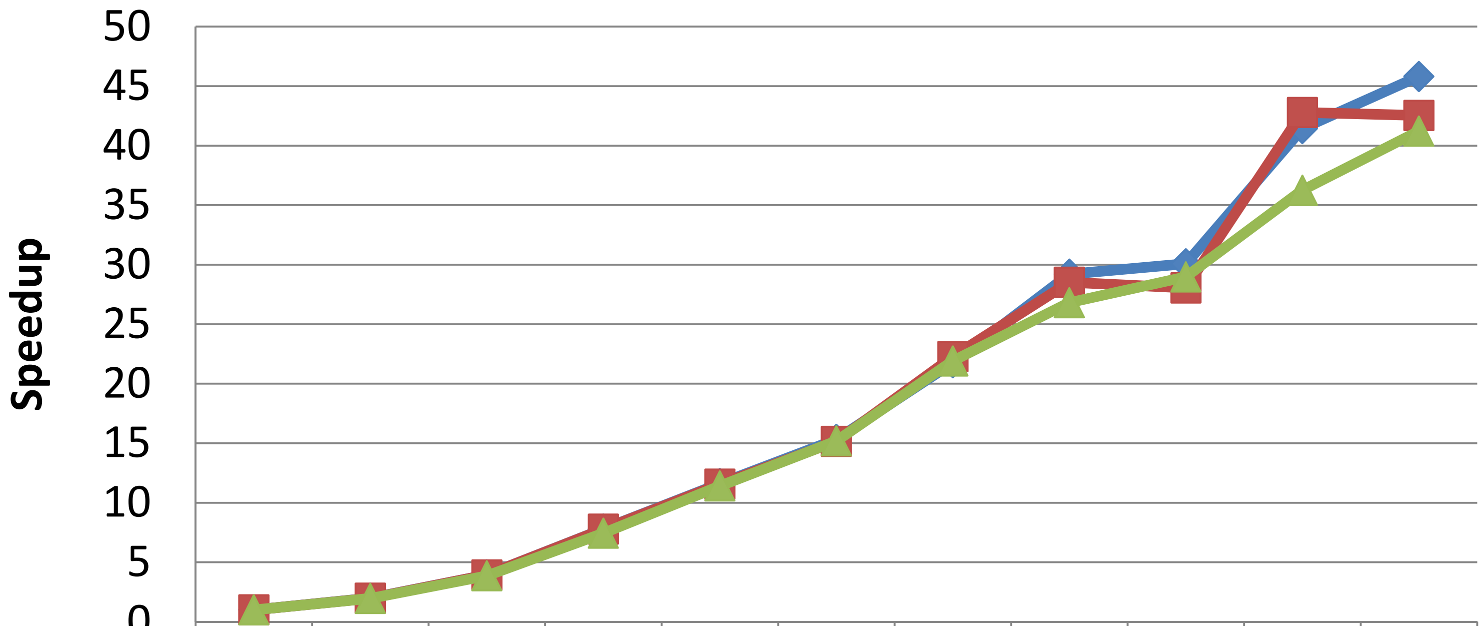
Cilk: Heat



	1	2	4	8	12	16	24	32	36	48	56
Half	1	2	3.98	7.88	11.7	15.4	22.4	29.1	32	41.2	44.2
Block	1	2	3.98	7.88	11.7	15.4	22.6	28.8	31.8	41.5	44.1
Single	1	2	3.98	7.88	11.7	15.4	22.4	28.8	32.1	41.1	43.7

© 2010 Mercury Computer Systems, Inc.

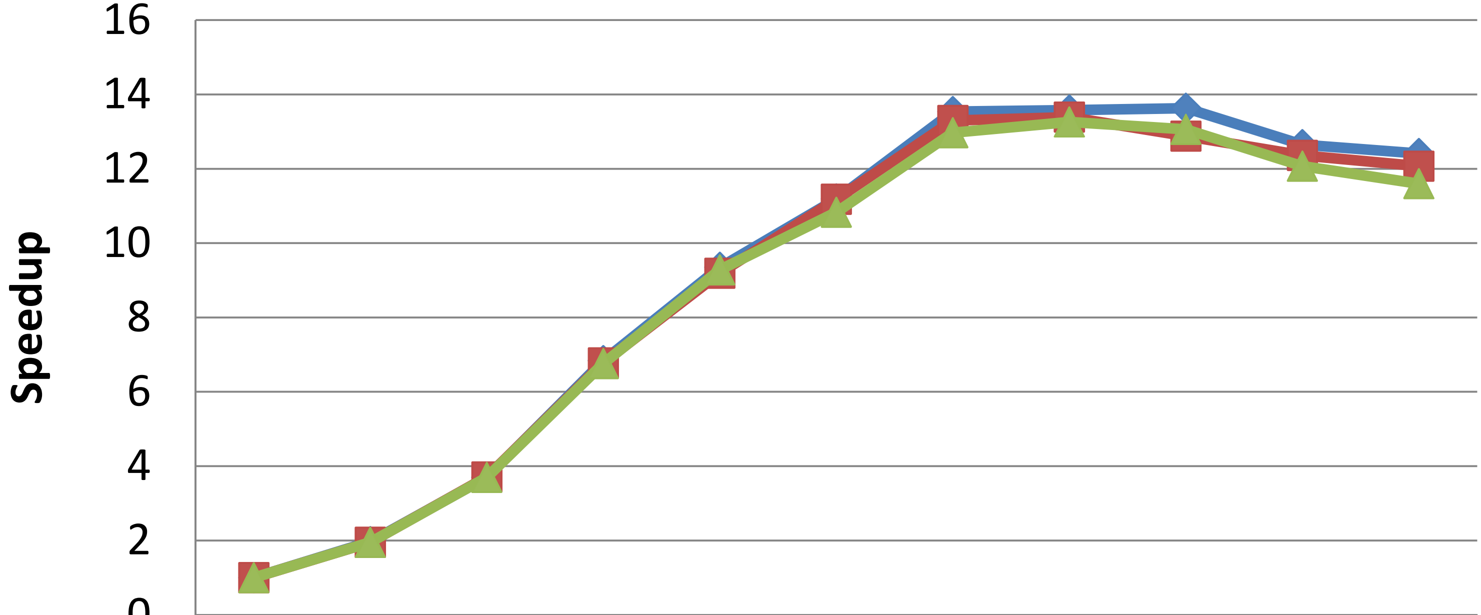
BOTS: N-Queens



	1	2	4	8	12	16	24	32	36	48	56
Half	1	1.99	3.94	7.79	11.6	15.3	21.8	29.2	30.1	41.4	45.8
Block	1	1.98	3.94	7.78	11.6	15.2	22.3	28.5	28.1	42.8	42.5
Single	1	1.96	3.86	7.44	11.4	15.2	21.9	26.8	29	36.3	41.2

© 2010 Mercury Computer Systems, Inc.

BOTS: Sort



	1	2	4	8	12	16	24	32	36	48	56
Half	1	1.97	3.7	6.84	9.35	11.2	13.5	13.6	13.6	12.7	12.4
Block	1	1.95	3.71	6.78	9.19	11.2	13.3	13.4	12.9	12.4	12.1
Single	1	1.95	3.7	6.76	9.28	10.8	13	13.3	13.1	12.1	11.6

© 2010 Mercury Computer Systems, Inc.

Conclusions

- Popular choice of stealing a single task at a time is suboptimal
  - Choosing a fraction of available tasks led to improved scalability
- Popular choice of randomized victim selection is suboptimal
  - We found NUMA ordering improved scalability slightly
- Cache-oblivious algorithms are a good fit for many-core platforms
  - Many implementations available in literature
  - Scale well across a wide range of processors
- …but research continues and questions remain
  - What about 1000s of cores?
  - How far can we scale algorithms on cc-NUMA architectures?

© 2010 Mercury Computer Systems, Inc.

Questions?

Michael Champigny

mchampig@mc.com

Thank you!

© 2010 Mercury Computer Systems, Inc.