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
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?
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
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
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
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
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)
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
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?
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
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
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
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
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
## Task Parallel Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Domain</th>
<th>Cutoff</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>BOTS</td>
<td>Spectral</td>
<td>128</td>
<td>1M point, FFTW generated</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>BOTS</td>
<td>Micro</td>
<td>10</td>
<td>Compute 45(^{th}) number</td>
</tr>
<tr>
<td>Heat</td>
<td>Cilk</td>
<td>Solver</td>
<td>512</td>
<td>Diffusion, 16M point mesh</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>Cilk</td>
<td>Dense Linear</td>
<td>16</td>
<td>512×512 square matrices</td>
</tr>
<tr>
<td>NQueens</td>
<td>BOTS</td>
<td>Search</td>
<td>3</td>
<td>13×13 chessboard</td>
</tr>
<tr>
<td>PartialLU</td>
<td>Cilk</td>
<td>Dense Linear</td>
<td>32</td>
<td>1M point matrix</td>
</tr>
<tr>
<td>SparseLU</td>
<td>BOTS</td>
<td>Sparse Linear</td>
<td>20</td>
<td>2K×2K sparse matrix</td>
</tr>
<tr>
<td>Sort</td>
<td>BOTS</td>
<td>Sort</td>
<td>2048, 20</td>
<td>20M 4-byte integers</td>
</tr>
<tr>
<td>StrassenMult</td>
<td>BOTS</td>
<td>Dense Linear</td>
<td>64, 3</td>
<td>1M point matrices</td>
</tr>
</tbody>
</table>
Example: FFT Twiddle Factor Generator (Serial)

```c
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);
    }
}
```
Example: FFT Twiddle Factor Generator (OpenMP)

```c
void fft_twiddle_gen (int i, int il, COMPLEX* in,
    COMPLEX* out, COMPLEX* W, int n, int nW, int r, int m)
{
    if (i == (il - 1))
        fft_twiddle_gen1 (in+i, out+i, W, r, m, n, nW*i, 
            nW*m);
    else {
        int i2 = (i + il) / 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, il, in, out, W, n, nW, r, m);
        #pragma omp taskwait
    }
}
```
Example: FFT Twiddle Factor Generator (Chimera)

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

<table>
<thead>
<tr>
<th>Speedup</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>36</th>
<th>48</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Half</strong></td>
<td>1.99</td>
<td>3.96</td>
<td>7.71</td>
<td>11.3</td>
<td>15.1</td>
<td>22.2</td>
<td>27.0</td>
<td>31.3</td>
<td>36.8</td>
<td>41.5</td>
<td></td>
</tr>
<tr>
<td><strong>Block</strong></td>
<td>1.99</td>
<td>3.96</td>
<td>7.71</td>
<td>11.6</td>
<td>15.1</td>
<td>21.1</td>
<td>26.8</td>
<td>32.0</td>
<td>36.8</td>
<td>39.9</td>
<td></td>
</tr>
<tr>
<td><strong>Single</strong></td>
<td>1.99</td>
<td>3.96</td>
<td>7.73</td>
<td>11.6</td>
<td>14.8</td>
<td>21.2</td>
<td>27.7</td>
<td>30.3</td>
<td>37.4</td>
<td>40.3</td>
<td></td>
</tr>
</tbody>
</table>
BOTS: Fast Fourier Transform

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>36</th>
<th>48</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>1</td>
<td>2</td>
<td>3.98</td>
<td>7.76</td>
<td>11.4</td>
<td>14.7</td>
<td>20.3</td>
<td>25.4</td>
<td>27.5</td>
<td>31.5</td>
<td>32.4</td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>2</td>
<td>3.96</td>
<td>7.75</td>
<td>11.3</td>
<td>14.7</td>
<td>20.6</td>
<td>24.8</td>
<td>27.5</td>
<td>28.6</td>
<td>28.1</td>
</tr>
<tr>
<td>Single</td>
<td>1</td>
<td>2</td>
<td>2.01</td>
<td>3.94</td>
<td>7.66</td>
<td>11.2</td>
<td>14.3</td>
<td>20.3</td>
<td>24.2</td>
<td>27.3</td>
<td>29.3</td>
</tr>
</tbody>
</table>
Cilk: Matrix-Matrix Multiply

<table>
<thead>
<tr>
<th>Speedup</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>36</th>
<th>48</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>1</td>
<td>2</td>
<td>3.99</td>
<td>7.95</td>
<td>11.9</td>
<td>15.8</td>
<td>23.5</td>
<td>31</td>
<td>34.9</td>
<td>43.5</td>
<td>48.1</td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>2</td>
<td>3.98</td>
<td>7.95</td>
<td>11.9</td>
<td>15.8</td>
<td>23.6</td>
<td>31.1</td>
<td>34.6</td>
<td>44.4</td>
<td>48</td>
</tr>
<tr>
<td>Single</td>
<td>1</td>
<td>1.99</td>
<td>3.97</td>
<td>7.89</td>
<td>11.8</td>
<td>15.6</td>
<td>23.3</td>
<td>30.5</td>
<td>34.3</td>
<td>37.2</td>
<td>39.7</td>
</tr>
</tbody>
</table>
BOTS: Strassen Matrix-Matrix Multiply

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.83</td>
</tr>
<tr>
<td>2</td>
<td>3.13</td>
</tr>
<tr>
<td>4</td>
<td>4.89</td>
</tr>
<tr>
<td>8</td>
<td>5.99</td>
</tr>
<tr>
<td>12</td>
<td>6.68</td>
</tr>
<tr>
<td>16</td>
<td>7.54</td>
</tr>
<tr>
<td>24</td>
<td>8.74</td>
</tr>
<tr>
<td>32</td>
<td>8.62</td>
</tr>
<tr>
<td>36</td>
<td>8.73</td>
</tr>
<tr>
<td>48</td>
<td>8.32</td>
</tr>
<tr>
<td>56</td>
<td>9.39</td>
</tr>
</tbody>
</table>

- **Half**: 1.83, 3.13, 4.89, 5.97, 6.66, 7.55, 8.74, 8.62, 8.73, 10.1
- **Block**: 1.83, 3.14, 4.88, 5.97, 6.69, 7.57, 8.52, 8.61, 8.32, 9.39
- **Single**: 1.83, 3.17, 4.88, 5.99, 6.68, 7.54, 8.73, 8.58, 8.41, 9.5
BOTS: Sparse LU Factorization

<table>
<thead>
<tr>
<th></th>
<th>Half</th>
<th>Block</th>
<th>Single</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.97</td>
<td>1.97</td>
<td>1.97</td>
</tr>
<tr>
<td>2</td>
<td>3.8</td>
<td>3.81</td>
<td>3.81</td>
</tr>
<tr>
<td>4</td>
<td>7.02</td>
<td>7.02</td>
<td>7.02</td>
</tr>
<tr>
<td>8</td>
<td>9.85</td>
<td>9.84</td>
<td>9.85</td>
</tr>
<tr>
<td>12</td>
<td>12.4</td>
<td>12.4</td>
<td>12.4</td>
</tr>
<tr>
<td>16</td>
<td>16.1</td>
<td>16.1</td>
<td>16.1</td>
</tr>
<tr>
<td>24</td>
<td>19.5</td>
<td>19.5</td>
<td>19.6</td>
</tr>
<tr>
<td>32</td>
<td>21.3</td>
<td>21.3</td>
<td>21.3</td>
</tr>
<tr>
<td>36</td>
<td>23.2</td>
<td>23.3</td>
<td>23.3</td>
</tr>
<tr>
<td>48</td>
<td>24.4</td>
<td>23.9</td>
<td>24.4</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Cilk: Partial Pivoting LU Decomposition

![Graph showing speedup for different configurations.](image)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>36</th>
<th>48</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>1</td>
<td>1.99</td>
<td>3.92</td>
<td>7.4</td>
<td>10.4</td>
<td>12.9</td>
<td>16.7</td>
<td>18.4</td>
<td>18.4</td>
<td>18.6</td>
<td>17.2</td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>1.99</td>
<td>3.91</td>
<td>7.45</td>
<td>10.4</td>
<td>12.9</td>
<td>16.7</td>
<td>18.8</td>
<td>18</td>
<td>17.7</td>
<td>16.8</td>
</tr>
<tr>
<td>Single</td>
<td>1</td>
<td>1.99</td>
<td>3.91</td>
<td>7.41</td>
<td>10.4</td>
<td>12.9</td>
<td>16.5</td>
<td>18.6</td>
<td>18.8</td>
<td>17</td>
<td>16.1</td>
</tr>
</tbody>
</table>
Cilk: Heat

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>36</th>
<th>48</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>1</td>
<td>2</td>
<td>3.98</td>
<td>7.88</td>
<td>11.7</td>
<td>15.4</td>
<td>22.4</td>
<td>29.1</td>
<td>32</td>
<td>41.2</td>
<td>44.2</td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>2</td>
<td>3.98</td>
<td>7.88</td>
<td>11.7</td>
<td>15.4</td>
<td>22.6</td>
<td>28.8</td>
<td>31.8</td>
<td>41.5</td>
<td>44.1</td>
</tr>
<tr>
<td>Single</td>
<td>1</td>
<td>2</td>
<td>3.98</td>
<td>7.88</td>
<td>11.7</td>
<td>15.4</td>
<td>22.4</td>
<td>28.8</td>
<td>32.1</td>
<td>41.1</td>
<td>43.7</td>
</tr>
</tbody>
</table>

© 2010 Mercury Computer Systems, Inc.
BOTS: N-Queens

<table>
<thead>
<tr>
<th>Speedup</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>36</th>
<th>48</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>1</td>
<td>1.99</td>
<td>3.94</td>
<td>7.79</td>
<td>11.6</td>
<td>15.3</td>
<td>21.8</td>
<td>29.2</td>
<td>30.1</td>
<td>41.4</td>
<td>45.8</td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>1.98</td>
<td>3.94</td>
<td>7.78</td>
<td>11.6</td>
<td>15.2</td>
<td>22.3</td>
<td>28.5</td>
<td>28.1</td>
<td>42.8</td>
<td>42.5</td>
</tr>
<tr>
<td>Single</td>
<td>1</td>
<td>1.96</td>
<td>3.86</td>
<td>7.44</td>
<td>11.4</td>
<td>15.2</td>
<td>21.9</td>
<td>26.8</td>
<td>29.0</td>
<td>36.3</td>
<td>41.2</td>
</tr>
</tbody>
</table>

© 2010 Mercury Computer Systems, Inc.
BOTS: Sort

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>36</th>
<th>48</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>1.97</td>
<td>3.7</td>
<td>6.84</td>
<td>9.35</td>
<td>11.2</td>
<td>13.5</td>
<td>13.6</td>
<td>13.6</td>
<td>12.7</td>
<td>12.4</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>1.95</td>
<td>3.71</td>
<td>6.78</td>
<td>9.19</td>
<td>11.2</td>
<td>13.3</td>
<td>13.4</td>
<td>12.9</td>
<td>12.4</td>
<td>12.1</td>
<td></td>
</tr>
<tr>
<td>Single</td>
<td>1.95</td>
<td>3.7</td>
<td>6.76</td>
<td>9.28</td>
<td>10.8</td>
<td>13</td>
<td>13.3</td>
<td>13.1</td>
<td>12.1</td>
<td>11.6</td>
<td></td>
</tr>
</tbody>
</table>
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?
Questions?

Michael Champigny
mchampig@mc.com

Thank you!