Programming Environments for Multigrain Parallelization

Programming Environments for Multigrain Parallelization

Dimitris Nikolopoulos
Assistant Professor
Department of Computer Science
The College of William & Mary
In a nutshell

- State-of-the-art
  - Deep machines
  - Multiple forms of parallelism
  - Balanced hardware
  - Unbalanced software

- Proposal:
  - Integrated management of multilevel parallelism
  - Explicitly parallel programming models
  - Compiler/runtime support for
    - fine-grain multithreading
    - nested data placement and layout optimizations
    - novel architectural features (multithreading, SIMD, speculation)
This talk

• A proposed programming model for deep systems
  – Balancing programming complexity with performance
  – Compiler/runtime support for managing (not discovering) parallelism and data

• Preliminary work
  – locality optimizations for shared cache memories
• Multigrain parallelization (state-of-the-art)
• GAS programming notation
• Compiler/runtime components
• Preliminary work
  – Data-centric thread formation for shared caches
  – Bandwidth-driven thread scheduling
• Future plans
Hybrid Programming Models

• First idea: hybrid programming models
  – Combination of MPI and one of many shared-memory models (OpenMP, Pthreads, DSM,...)

• Pros
  – Well-known parallel programming paradigms
  – Easy (?) to parallelize at multiple levels
  – Available software components
Hybrid Programming Models

- **Cons**
  - Uncooperative libraries (thread safety, threading semantics, memory management)
  - Ad-hoc parallelization process
    - Usually start with efficient MPI and use OpenMP whenever possible
    - Unfortunately, the MPI/OpenMP ratio is critical (64x1, 32x2, 16x4 ?)
  - Contradicting performance results
    - IBM SP cluster: probably bad
    - Pentium clusters: mostly bad sometimes good
    - Origin2000: mostly good sometimes bad
Flatten Parallelism

• MPI for everything
  – MPI over distributed memory
  – MPI over shared memory
  – Backward compatibility
  – Unbalanced code

• MPI for distributed-memory communication, threads for shared-memory communication
  – Hide deeper levels of parallelism in the communication libraries
  – Fine-grain parallelism only for communication instead of communication and computation
Unification

- Unified management of multiple levels of parallelism
  - Processes, user-level threads, kernel-level threads, hardware threads
  - Share-nothing, shared-memory, on-chip shared caches
- One means to express parallelism at all levels
- One means to express data distribution at all levels
Global Address Space

- Abstraction of shared memory
  - But software knows always were data is placed
- Built around the thread and loop constructs
  - Threads and loops can be nested
- Difficulties
  - Managing communication
  - Data placement and data layout
  - Managing threads at many layers
  - Exploit deep on-chip parallelism
Global Address Space

• Earlier work
  – HPF
  – ZPL, NESL, other data-parallel languages
  – UPC
  – Other Threaded-C languages (Split-C, EARTH-C)

• Limitations
  – One level of parallelism (except NESL, EARTH-C)
  – If multiple levels of parallelism, limited to data (NESL) or computation (EARTH-C), not both
  – Limited exploitation of architectural innovations
An Explicit Multigrain Parallel Programming Model

- Multiple levels of parallelism mapped to multiple levels of physical parallel execution contexts
  - Vertical exploitation of parallelism
  - Separation of the abstraction from the execution vehicle
- Utilization of deep parallelism (hardware threads) for communication and computation
- Multiple levels of implicit data distribution
  - Multilevel caches, on-chip DRAM, off-chip DRAM, ...
  - Explicit data distribution for backward compatibility
- Effective use of new processor features
Notation for Parallelism

- Pin-point parallelism
  - Threads (not necessarily functions, can be basic blocks)
  - Loops
- Describe parallelism in the compiler as a hierarchy
  - No automatic parallelization, hierarchy specified by the user
- A greedy programming strategy
  - Programmer specifies all the parallelism at all levels
  - Programmer specifies distributable data
  - Compiler/RTS decides how much parallelism should be extracted
  - Fully dynamic model, subject to resource constraints
Simple motivating example

for {
  ...
}

for {
  ...
}

for {
  ...
}
Simple motivating example
Vertical exploitation of parallelism

• Top-down approach
  – Optimize first level (e.g. MPI)
  – After optimizing MPI optimize second level (e.g. user-level threads)

• Vertical approach
  – Assume $n$ levels with capacity $P_i$, $i = 1, \ldots, n$
  – Estimate the speedup at each level as a function of the execution contexts used $f_i(T_i)$, $T_i \leq P_i$
  – Estimate multiplicative speedup $f_1(T_1)f_2(T_2)\ldots f_n(T_n)$
  – Maximize multiplicative speedup subject to constraints $T_i \leq P_i$
Data notations

\[ v<T>; \quad \text{// A simple templated parallel vector e.g. } v<\text{Double}> \quad \text{// } = \quad [1.0 \quad ... \quad 100.0]; \]

\[ v<v<T>>; \quad \text{// A collection of a templated parallel vectors e.g. } \]
\[ \quad \text{// } v<v<\text{Double}>> = [[1.0 \quad ... \quad 50.0]...[50.0...100.0]] \]
\[ \quad \text{// for two processors, which is derived from the} \]
\[ \quad \text{// compiler/runtime system} \]
\[ v<v<v<T>>>; \quad \text{// A collection of collections of templated parallel} \]
\[ \quad \text{vectors } v \]
\[ s = [100]; \quad \text{// A data segment descriptor for a uniprocessor} \]
\[ s = [50 \ 50]; \quad \text{// A data segment descriptor for a dual SMP} \]
\[ s = [[25 \ 25] \ [25 \ 25]]; \quad \text{// A data segment descriptor for two dual SMPs} \]
Runtime Data Management

• Implicit approach
  – A small set of predefined data distributions (block/cyclic)
  – Runtime support for tuning data distributions at execution time
  – Based on previous work on runtime data distribution on ccNUMA

• Novel features
  – Data distribution combined with data layout optimizations
  – Flattening/explosion of data types
  – Profile-guided optimization for irregular data structures
Data-centric thread formation

• Conceptually
  – Organize data in blocks that fit levels of the memory hierarchy (L1, L2, L3, DRAM)
  – Associate blocks with threads (affinity relationship, no hardware parallelism assumed here)
  – Group threads based on block affinity
  – Match with hardware resources
  – Runtime support: dynamic splitting and coarsening (flattening/floating parallelism)
Managing architectural features

- Explicit parallelism exposed to the programming model
  - SIMD
  - Hardware threads
- Prefetching controlled by the compiler
- Speculative execution controlled by the compiler/hardware
- Open problems:
  - Inner processor parallelism vs. outer parallelism
  - When and how to use speculative execution
Scheduling Issues

- Static and dynamic scheduling for computation is manageable
- Dynamic scheduling of computation and communication is difficult
  - How many threads for fine-grain computation?
  - How many threads for communication, where to schedule them?
  - Constrained optimization by multiple parameters (CPU, bandwidth, cache space, possibly others)
  - Currently investigating runtime scheduling methods based on observed hardware metrics
Preliminary Results - Tiling on SMTs

• Data-centric thread formation
• Problem: multithreaded processors use shared caches
• Share caches may help parallelization:
  – Communication/synchronization is accelerated
• Shared caches may hurt parallelization:
  – If code is properly tiled threads will conflict in the cache
  – Overall performance is poor
SMT execution engine
First solution: smaller tiles and copy

- Partition the cache between two tiles
- Pin the tiles in a fixed set of cache locations
- How?
  - Copy tiles to buffers
- Setup a buffer per thread
- Align buffers so that buffers from different threads do not conflict
Second solution: block data layouts

Row-major array layout

<table>
<thead>
<tr>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
</tr>
</tbody>
</table>

Block array layout

<table>
<thead>
<tr>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>43</td>
<td>44</td>
</tr>
</tbody>
</table>
Using block data layouts

- Select best block size and organize array in blocks
- Store tiles contiguously in memory
- Align tiles so that:
  - Tiles of the same thread conflict (!)
  - Tiles of different threads do not conflict
Scheduling and Runtime Support

- Automatic detection of cache contention
  - Using hardware counters
  - Adaptivity to execution contexts: runs with 1 thread/processor or multiple threads/processor

- Interleaving for virtual cache partitioning, blocking for non-partitioned caches
  - Adaptivity to cache organization

- Multilevel cache management
  - Multilevel tiling
Results

• blocked matrix multiply (3 levels)
• sor 5-point stencil
• Intel C compiler (7.0)
• Manual parallelization (Linux threads, OpenMP under construction)
• 4-CPU Xeon MP 1.4 GHz, 512 Kb L3 cache, 256 Kb L2 cache, 8 Kb L1 cache
Results: matrix multiplication

<table>
<thead>
<tr>
<th>matrix size (n)</th>
<th>execution time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>2200</td>
<td>200</td>
</tr>
<tr>
<td>2400</td>
<td>400</td>
</tr>
<tr>
<td>2600</td>
<td>600</td>
</tr>
<tr>
<td>2800</td>
<td>800</td>
</tr>
<tr>
<td>3000</td>
<td>1000</td>
</tr>
</tbody>
</table>

- 1 thread
- 4 threads on 2 SMTs
- 4 threads on 2 SMTs, dynamic tiling
- 8 threads on 4 SMTs
- 8 threads on 4 SMTs, dynamic tiling
Results matrix multiplication

matrix multiply

L1 cache misses

4 threads on 2 SMTs, fixed tiles
4 threads on 2 SMTs, dynamic tiling
8 threads on 4 SMTs, fixed tiles
8 threads on 4 SMTs dynamic tiling

matrix size (n)

0.00E+00 5.00E+05 1.00E+06 1.50E+06 2.00E+06 2.50E+06 3.00E+06 3.50E+06 4.00E+06 4.50E+06 5.00E+06

2000 2200 2400 2600 2800 3000
Results sor stencil

- 1 thread
- 4 threads on 2 SMTs
- 4 threads on 2 SMTs, dynamic tiling
- 8 threads on 4 SMTs
- 8 threads on 4 SMTs, dynamic tiling

execution time (seconds)

matrix size (n)
Impact of bandwidth saturation

- One app running on an idle bus using two SMTs
- One app using two SMTs running with two bandwidth-friendly communication threads
- One app using two SMTs running with a bandwidth-unfriendly communication threads
Goal

• Devise runtime scheduling algorithms that
  – Do not oversubscribe the memory bandwidth
  – Meet the other scheduling objectives

• Methodology
  – Move from a cache-centric to a bandwidth-centric scheduling algorithm
  – Use existing time-sharing schedulers
A scheduling algorithm

• A gang-scheduling like policy
  – Threads of the application are co-scheduled in rounds

• Each batch of threads is selected so that the bandwidth is not over-subscribed

• How do we measure bandwidth?
  – On-line measurement using hardware counters
  – 1 measurement per time quantum (100 ms.)
  – Weighted average of measurements for discarding obsolete history
Last quantum

- Fitness metric:
  - Available bandwidth per unallocated thread
  - Select thread that **utilizes best** the available bandwidth

\[
ABTR_{\text{thread}} = \frac{SBTR_{\text{max}} - SBTR_{\text{expected}}}{\text{unallocated} \_ \text{threads}}
\]

\[
\text{fitness} = \frac{1000}{1 + |ABTR_{\text{thread}} - BTR_{\text{thread}}|}
\]
Results with synthetic workloads

2 Apps (2 Threads each) + 2 BBMA + 2 nBBMA

- Radiosity
- Volrend
- Barnes
- Water-nsq
- BT
- FMM
- LU CB
- SP
- MG
- CG
- Raytrace

Avg. Turnaround Time Improvement (%)
Conclusions

• Multigrain parallelization is necessary because of the multigrain nature of parallelism in deep computing systems

• A mixture of common programming notations (threads and data) suffices to express multilevel parallelism

• More powerful compiler/runtime support is required, even if we stay with MPI

• Modern microprocessors need to be utilized for HPC and it’s not everything up to the compiler to do it