Hierarchical Locality and Parallel Programming in the Extreme Scale Era

Tarek El-Ghazawi
The George Washington University

University of Southern California
September 29, 2016
Overview

◆ Fundamental Challenges for Extreme Computing
◆ Locality and Hierarchical Locality
◆ Programming Models
◆ Hardware Support for Productive Locality Exploitation- Address Remapping
◆ Hierarchical Locality Exploitation
◆ Concluding Remarks
Top Ten Challenges for Exascale: Areas where Research and advances are needed!

1. Energy Efficiency ✓
2. Interconnect Technology ✓
3. Memory Technology ✓
4. Scalable System Software ✓
5. Programming Systems ✓
6. Data Management ✓
7. Exascale Algorithms
data movement and/or programming related ✓
8. Algorithms for Discovery, Design & Decision
9. Resilience and Correctness
10. Scientific Productivity ✓
Technological Challenges: Combined Bandwidth and Energy Challenges for Exascale

Locality and data movement matter a lot, cost (energy and time) rapidly increases with distance.

Locality and data movement are critical even at short distance, more so at far distances.

[Source: ASCAC 14]
**Technological Challenges**: (2) Bandwidth

Growing manycore bandwidth requirements

Widening gap between available I/O and compute capability

- Interconnect is not keeping up with the growth in compute capability
  - Many apps require 1 Byte/FLOP off-chip, not possible in 10 TFLOPs chips and beyond
  - Intel Knights Landing: 500 GB/s => 1/6 Byte/FLOP
  - Huge bandwidth density (GB/s/μm) needed on-chip due to large #cores in small area

Overview

- Fundamental Challenges for Extreme Computing
- **Locality and Hierarchical Locality**
- Programming Models
- Hardware Support for Productive Locality Exploitation - Address Remapping
- Hierarchical Locality Exploitation
- Concluding Remarks
Architectural Challenges: Architectures are becoming Deeply Hierarchical in Extreme Scale – Chips and Systems
Architectural Challenges: Architectures are becoming Deeply Hierarchical in Extreme Scale – Chips and Systems
Architectural Challenges: Architectures are becoming Deeply Hierarchical in Extreme Scale – Chips and Systems
Architectural Challenges: Architectures are becoming Deeply Hierarchical in Extreme Scale – Chips and Systems
Architectural Challenges: Architectures are becoming Deeply Hierarchical in Extreme Scale – Chips and Systems

Cray XC40

Cabinet Level

Chassis Level

System Level
Architectural Challenges: Architectures are becoming Deeply Hierarchical in Extreme Scale – Chips and Systems

Cray XC40

Tile64
Overview

- Fundamental Challenges for Extreme Computing
- Locality and Hierarchical Locality
- Programming Models
- Hardware Support for Productive Locality Exploitation - Address Remapping
- Hierarchical Locality Exploitation
- Concluding Remarks
Where are Programming Models from That?

What is a programming model?
- An abstract virtual machine
- A view of data and execution
- The logical interface between architecture and applications

Why Programming Models?
- Decouple applications and architectures
  - Write applications that run effectively across architectures
  - Design new architectures that can effectively support legacy applications

Programming Model Design Considerations
- Expose modern architectural features to exploit machine power and improve performance
- Maintain Ease of Use
- Two previous points mean increase productivity!
Current Programming Models and Locality Awareness

- **Message Passing**
  - Località Awareness
  - Two-Sided Communication
  - Example MPI

- **Shared Memory**
  - Località Awareness
  - One-Sided Communication
  - Example OpenMP

- **Partitioned Global Address Space**
  - Località Awareness
  - One-Sided Communication
  - Examples UPC and Chapel
PGAS Languages Include UPC, Chapel and X10

UPC Language Specifications
V1.0

Tarek A. El-Ghazawi
George Washington University
tarek@gwu.edu

William W. Carlson Jesse M Draper
IDA Center for Computing Sciences
wwc@super.org jdraper@super.org

February 2001
Overview

- Fundamental Challenges for Extreme Computing
- Locality and Hierarchical Locality
- Programming Models
- **Hardware Support for Productive Locality Exploitation** - Address Remapping
- Hierarchical Locality Exploitation
- Concluding Remarks
Memory Accesses in UPC - Shared Address Translation Overheads

- **Measurement of the address space overheads**
- **Set of micro-benchmarks measuring the different aspects separately:**
  - Network Time
  - Address Translation
  - Address Incrementation
  - Memory Access

<table>
<thead>
<tr>
<th>Type of access</th>
<th>Time (ns)</th>
<th>% time in memory access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>1.42</td>
<td>0%</td>
</tr>
<tr>
<td>Local</td>
<td>4.53</td>
<td>10%</td>
</tr>
<tr>
<td>Remote</td>
<td>4.53</td>
<td>20%</td>
</tr>
<tr>
<td>Private</td>
<td>1.42</td>
<td>20%</td>
</tr>
<tr>
<td>Local</td>
<td>4.53</td>
<td>30%</td>
</tr>
<tr>
<td>Remote</td>
<td>4.25 MB/s</td>
<td>40%</td>
</tr>
</tbody>
</table>

Thread 0
- Private 0: 1.42
- Local: 4.53
- Remote: 4.53

Thread 1
- Private 0: 1.42
- Local: 4.53
- Remote: 4.53

Thread (Threads - 1)
- Private 0
- Private 1
- Private (Threads - 1)

Shared
Memory Access Costs in Chapel

- Tested shared address access costs in Chapel:
  - Used Chapel Syntax to test
    - Local part of a distributed object, un-optimized: Accessing local data without saying local
    - Local Optimized – local part hand-optimized by saying “local”
    - Local and Non-Distributed
  - Compiler optimization -> 2x faster
  - Both compiler and hand optimization -> 70x faster
  - Compiler optimization affects remote accesses as well
  - Both UPC and Chapel require “unproductive!” hand tuning to improve local shared accesses
Fast Address Translation for PGAS

Software solutions
- Hand tweaking – Non-productive
- Compiler optimizations - reduced arithmetic for some straightforward cases
- Look up tables, full and reduced- Take memory! ICPP05
- TLB's ....

Hardware solutions
- Create hardware that understands how to traverse the PGAS memory model and support basic costly needs
- Avail it through instructions and leverage them by the compiler

Some work for UPC, little for Chapel
Hardware Support for PGAS

Example Operations for Support in Hardware

- Shared address incrementing
- Load/store to/from a PGAS shared address
  - Address translation support: convert a shared address to a system virtual address used to perform the access
- Locality tests for remote data
  - Can be used to tell whether to call the network subroutines, by e.g. testing the affinity field in a work sharing construct

Availed as ISA extension

- New instructions used directly by compiler
- Current hardware support and instructions only support address mapping
- Future support for remote data accesses and various types of synchronizations are of interest
Hardware/Software Co-Design Platform in a Nutshell

- First prototype in FPGAs, supports small core count and apps
- Second is primarily software, supports bigger core counts and codes

<table>
<thead>
<tr>
<th>Benchmarking Kernels</th>
<th>UPC Code Out of the Box</th>
<th>Benchmarking Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BUPC</strong></td>
<td>New Instructions Inserted into Code Gen</td>
<td><strong>BUPC</strong></td>
</tr>
<tr>
<td><strong>GasNet</strong></td>
<td>Ported on top of Gem5</td>
<td><strong>GasNet</strong></td>
</tr>
<tr>
<td><strong>Gem5</strong></td>
<td>A Runtime System that recognizes and enforces the developed mapping</td>
<td><strong>Leon3 Cores</strong></td>
</tr>
<tr>
<td><strong>Workstation Cluster - Future</strong></td>
<td>Extended with proposed PGAS hardware support for shared addressing</td>
<td><strong>Virtex-6 FPGA</strong></td>
</tr>
</tbody>
</table>

Simulator Prototype | FPGA Prototype

Tarek El-Ghazawi, GWU
arrayA[10] = 5;

Thread 0

Thread 1

Thread 2

Thread 3

0 1 2 3
16 17 18 19

4 5 6 7
20 21 22 23

8 9 10 11
24 25 26 27

12 13 14 15
28 29 30 31

pgas_inc_{x}

pgas_st_{x}

Address Incrementation

Address Translation/Store

0xffff01203f14

Shared Pointer Representation

Regular pointer representation
Early Results - NPB Kernels with HW Support
Gem5 Alpha 21264
Overview

- Fundamental Challenges for Extreme Computing
- Locality and Hierarchical Locality
- Programming Models
- Hardware Support for Productive Locality
  - Exploitation- Address Remapping
- Hierarchical Locality Exploitation
- Concluding Remarks
Possible Solutions for Hierarchical Locality Exploitation

- Rewrite your code with low-level tricks to target the underlying hierarchical architecture?
  - Great performance, but not productive & non-portable

- Extend programming models with hierarchical syntax and semantics and ask programmers to worry about all of those hardware details? (make them hierarchical-locality-aware!)
  - Portable but not productive
Productive Division of Responsibilities: The Programmer and the System

- **Programmer**
  - Use a locality-aware programming paradigm such as MPI or a PGAS language
  - Let programmer worry about the first-order locality, thread-data affinity

- **System**
  - Understand your system hierarchy, costs associated with data movements across levels
  - Understand the program characteristics
  - Derive locality exploitation on level-by-level basis via Hierarchical Thread Grouping/partitioning
Motivations and Early Investigations

- Proper placement will
  - Avoid unnecessary data movement by exploiting locality
  - Utilize the shared memory and caches in the neighborhood
  - Utilize the best interconnect for the underlying communication
  - Yield a rising benefit as the size of your system increases! A must for exascale!!

Synthetic benchmark showing the gain of proper with varying number of threads and percentage of remote communication
Motivations and Early Investigations

The response of each level to communication varies according to message sizes

- Closer is not always faster

- Know and characterize your architecture!!
PHLAME Methodology
(Parallel Hierarchical Abstraction Model of Execution)

1. Characterize the machine message costs at each level to generate PHLAME description File (PDF)
2. Profile the application communication
3. Build a placement layout for the threads based on the above
4. Run the application with the layout built in the previous step
Characterizing the target machine

- **Message cost:** total time for message to be delivered

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.516956</td>
<td>0.665469</td>
<td>1.209482</td>
<td>1.986097</td>
<td>3.606203</td>
<td>7.593014</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.688468</td>
<td>1.038422</td>
<td>1.54703</td>
<td>2.772387</td>
<td>5.138746</td>
<td>10.86957</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.687853</td>
<td>1.033378</td>
<td>1.543448</td>
<td>2.770083</td>
<td>5.128205</td>
<td>10.85776</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.706414</td>
<td>1.05042</td>
<td>1.548707</td>
<td>2.77855</td>
<td>5.128205</td>
<td>11.02536</td>
<td></td>
</tr>
</tbody>
</table>

Example: time per message (ns) machine communication characterization
Characterizing the application communication

- Instrument the application code
  - generate the communication activity matrices

- The message sizes range is partitioned into bins
  - Each bin corresponds to a sub range, example: 1→64, 64→128, ...

- There are two communication activity matrices for each bin
  - Average size
  - Number of messages
Calculating Level Costs

- Placement decisions require a measurement of how threads fit together.

- Repeat for each level $l$:
  - For each pair of threads $(i, j)$, where $i \neq j$, calculate the cost of their communication.

$$LevelCost_{lij} = \sum_{b=1}^{B} (AvgMsgSize_{ijb} \times NumMsgs_{ijb} \times LevelBinCost_{lb})$$

Where $B$ is the number of bins.
Hierarchical Thread Fitness Measure

- The fit measure shows how two threads benefit or lose if scheduled on a given level.

- The fit measure is based on the difference of message costs at each level.

\[
FIT \ Measure(i, j, L) = \sum (Worse - Level) - \sum (Level - Better)
\]
Hierarchical Thread Fitness Measure

- The fit measure shows how two threads benefit or lose if scheduled on a given level.

- The fit measure is based on the difference of message costs at each level.

\[ \text{FIT Measure}(i, j, L) = \sum (\text{Worse} - \text{Level}) - \sum (\text{Level} - \text{Better}) \]
Hierarchical Thread Fitness Measure

- The fit measure shows how two threads benefit or lose if scheduled on a given level.

- The fit measure is based on the difference of message costs at each level.

\[
FIT\ Measure(l, i, j) = \sum (Worse - Level) - \sum (Level - Better)
\]

Gain of placement at current level, given current level is better

loss due to placement at current level, given current level is worse
Hierarchical Thread Fitness Measure

- The fit measure shows how two threads benefit or lose if scheduled on a given level.

- The fit measure is based on the difference of message costs at each level.

Gain of placement at current level, given current level is better.

Loss due to placement at current level, given current level is worse.

\[
FIT\ Measure(i, j, L) = \sum (Worse - Level) - \sum (Level - Better)
\]
Hierarchical Thread Fitness Measure

- The fit measure shows how two threads benefit or lose if scheduled on a given level.

- The fit measure is based on the difference of message costs at each level.

\[ FIT \ Measure (i, j, L) = \sum (Worse - Level) - \sum (Level - Better) \]
Mapping to Graph Theory

- The application communication pattern can be mapped into a graph
  - Vertices represent the threads
  - Edges represent interactions between threads
  - HTF at each level are edge weights
    - Multiple weights per edge

\[
W_{ij_1}, W_{ij_2}, W_{ij_3}, \ldots W_{ij_L}
\]
Hierarchical Graph Partitioning

- **Algorithms can be**
  - **Bottom Up**
    - Form partitions at lower levels first and recursively group them at higher levels
  - **Top Down**
    - Form partitions at upper levels first and recursively break them at lower levels

**Abstract Machine:**
- Level 1: Width = 4 (# of locales)
- MaxLocaleSize = 4 (# of cores in each locale)
Testbed

- Cray XE6m/XK7m
  - 24 cores per node
    - Two 12-core AMD Magny Cours
  - Gemini Interconnect
    - 2D Torus

- UPC NPB Benchmark from GWU
  - IS – Class C
  - FT – Class C
  - CG – Class C
  - MG – Class C
  - EP – Class C

- Heat Diffusion
Profiling the application communication – Implementation

- TAU is selected to profile UPC and MPI programs
  - Generates activity matrix for each bin

- Bins are not supported in TAU profiles

- Modifications were made to TAU backend and frontends to support bins
Customizing GASNet

- The clustering algorithm usually assigns unequal number of threads to different nodes.

- The Cray Application Level Placement Scheduler (ALPS) does not support this feature.

- A modified GASNet Geminie Conduit was used to trick the system to achieve the non-uniform thread count per node:
  - Dummy processes are launched.
  - Environment variables control how the runtime pick the correct number of processes on each node.
Experimental Results

- FT – all-to-all communication

![Graph showing relative communication overhead and gain for different thread numbers and algorithms.](image)

- (Legend: Clustering, Splitting, Splitting - Non Restricted, PHAST)
Experimental Results

MPI

- FT – all-to-all communication

- Clustering
- Splitting
- Splitting - Non Restricted
- PHAST

![Graph showing experimental results](image-url)
Experimental Results

UPC

- CG – Irregular memory access and communication

![Graph showing experimental results](image)

- Clustering
- Splitting
- Splitting - Non Restricted
- PHAST

![Another graph showing relative communication overhead](image)
Experimental Results

MPI

- CG – Irregular memory access and communication

- Clustering
- Splitting
- Splitting - Non Restricted
- PHAST

Gain (%)

Number of Threads

Relative Communication Overhead

Number of Threads

Tarek El-Ghazawi, GWU
CG – Non Restricted Explanation

Node 0
Remote

Node 1
Remote

Tarek El-Ghazawi, GWU
Overview

- Fundamental Challenges for Extreme Computing
- Locality and Hierarchical Locality
- Programming Models
- Hardware Support for Productive Locality Exploitation - Address Remapping
- Hierarchical Locality Exploitation
- Concluding Remarks
Concluding Remarks

- Due to energy and bandwidth constraints, data movements are becoming too expensive.
- Locality exploitation is an obvious target.
- Extreme scale architectures are becoming deeply hierarchical, giving rise to hierarchical locality.
- Hierarchical locality exploitation must be done productively, leaving programmers with the necessary minimum work to do.
- We can expect some programming paradigms to provide explicit solutions.
- Locality-aware programming, hardware support, and run-time systems can play a bigger role while keeping programmers' productivity high.
Publications


Follow up work in Hierarchical Locality Exploitation

- Use thread data-affinity from locality-aware program as a starting point into a hierarchical locality exploitation system (PHLAME or FLAME: Parallel Hierarchical Abstraction Model of Execution)
- Examine best graph partitioning methods
- Decentralize algorithms, and build in fast predictions to handle the Exascale
- Consider dynamic solutions
- Consider unprofiled cases and collecting intelligence on runs for later use and optimizations
- Consider data dependent cases
- Consider dynamic parallelism cases
- Investigate hardware support