Programming the X-Stack: 
Challenges and Opportunities

Barbara Chapman 
University of Houston
Fall Creek Falls Conference, 
Gatlinburg 
September 16, 2011
High Performance Computing and Tools Group
http://www.cs.uh.edu/~hpctools

Petascale is a Global Reality

- K computer
  - 68,544 SPARC64 VIIIfx processors, Tofu interconnect, Linux-based enhanced OS, produced by Fujitsu
- Tianhe-1A
  - 7,168 Fermi GPUs and 14,336 CPUs; it would require more than 50,000 CPUs and twice as much floor space to deliver the same performance using CPUs alone.
- Jaguar
  - 224,256 x86-based AMD Opteron processor cores. Each compute node features two Opterons with 12 cores and 16GB of shared memory
- Nebulae
  - Nvidia Tesla 4640 GPUs, Intel X5650-based 9280 CPUs
- Tsubame
  - 4200 GPUs

Many More Cores

- Biggest change is within the node
  - Some full-featured cores
  - Many low-power cores
  - Technology rapidly evolving, will be integrated
  - Easiest way to get power efficiency and high performance
  - Specialized cores
  - Global memory
  - Low amount of memory per core
  - Coherency domains, networks on chip

Programming Models? Today’s Scenario

```c
// Run one OpenMP thread per device per MPI node
#pragma omp parallel num_threads(devCount) if ([initDevice()])
{
    // Block and grid dimensions
dim3 dimBlock(12,12);
kernal<<<1,dimBlock>>>(...);
cudaThreadExit();
} else
{
    printf("Device error on %s\n", processor_name);
}
MPI_Finalize();
return 0;
```

Top 10 Energy-efficient Supercomputers (June 2011)

<table>
<thead>
<tr>
<th>#</th>
<th>Cray/SGI</th>
<th>56%</th>
<th>Cray/SGI</th>
<th>56%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>K computer</td>
<td>72%</td>
<td>K computer</td>
<td>72%</td>
</tr>
<tr>
<td>2</td>
<td>Tianhe-1A</td>
<td>68%</td>
<td>Tianhe-1A</td>
<td>68%</td>
</tr>
<tr>
<td>3</td>
<td>Jaguar</td>
<td>56%</td>
<td>Jaguar</td>
<td>56%</td>
</tr>
<tr>
<td>4</td>
<td>Nebulae</td>
<td>36%</td>
<td>Nebulae</td>
<td>36%</td>
</tr>
<tr>
<td>5</td>
<td>Tsubame</td>
<td>28%</td>
<td>Tsubame</td>
<td>28%</td>
</tr>
</tbody>
</table>

Program/Data Storage

System Connectivity

Peripherals

Serial Interfaces

Display Subsystems

Camera I/F

POWER SGX™ Graphics (3515/3530 only)

C64x+™ DSP and video accelerators (3525/3530 only)

L3/L4 Interconnect

AR Corre™
Exascale Programming Models

- Programming models is biggest “worry factor” for application developers
- MPI-everywhere model no longer viable
  - Hybrid MPI+OpenMP already in use in HPC
- Need to explore new approaches, adapt existing APIs
- Exascale models and their implementation must take account of:
  - Scale of parallelism, levels of parallelism
  - Heterogeneous cores
  - Potential coherency domains
  - Need to reduce power consumption
  - Resource allocation and management
  - Legacy code, libraries; interoperability
  - Support for resilience, verification

IESP Programming Models Agenda

- Alternative R&D strategies
  - Uniform vs. hybrid programming models
  - MPI 7.0, OpenMP 5.0, or revolutionary approaches
  - Domain specific vs. general PMs
- Recommended research agenda
  - Explore enhancements to existing models
  - Revolutionary approach with interoperability to existing models

Delivering The Programming Model

- Between nodes, MPI with enhancements might work
  - Needs more help for fault tolerance and improved scalability
- Within nodes, too many models, no complete solution
  - OpenMP, PGAS, CUDA and OpenCL all potential starting points
  - In layered system, migrate code to most appropriate level, develop at most suitable level
  - Incremental path for existing codes
- Timing of programming model delivery is critical
  - Must be in place when machines arrive
  - Needed earlier for development of new codes

IESP Programming Models

Proposed timeline

- Interoperability among existing programming models
- System-wide high-level programming model
- Fault-tolerant MPi
- Interoperability among existing programming models

A Layered Programming Approach

- Efficient, Deterministic, Declarative, Restrictive Expressiveness-based language (DRL, maybe?)
- Parallel Programming Languages (OpenMP, PGAS, APOGAS)
- Low-level APIs (MPI, pthreads, OpenCL, Verilog)
- Machine code, Assembly
Programming Model Major Requirements

- Portable expression of scalable parallelism
  - Across exascale platforms and intermediate systems
- Uniformity
  - One model across variety of resources
  - Across node, across machine?
- Locality
  - For performance and power
  - At all levels of hierarchy
- Asynchrony
  - Minimize delays
  - But trade-off with locality

Evolutionary approach enhances/adapts current APIs (esp. MPI, OpenMP, PGAS and APGAS languages)

OpenMP Evolution Toward Exascale

- OpenMP language committee is actively working toward the expression of locality and heterogeneity
  - And to improve task model to enhance asynchrony
- How to identify code that should run on a certain kind of core?
- How to share data between host cores and other devices?
- How to minimize data motion?
- How to support diversity of cores?

Work is already beginning to enhance variety of models and/or extend their range of applicability

OpenMP 4.0 Attempts To Target Range of Acceleration Configurations

- Dedicated hardware for specific function(s)
  - Attached to a master processor
  - Multiple types or levels of parallelism
  - Process level, thread level, ILP/SIMD
- May not support a full C/C++ or Fortran compiler
  - May lack stack or interrupts, may limit control flow, types

OpenMP Locality Research

- Coordinate data layout, work
  - Collection of locations represent execution environment
  - Map data, threads to a location; distribute data across locations
  - Align computations with data’s location, or map them
  - Location inherited unless task explicitly migrated

Research on Locality in OpenMP

- Implementation in OpenUH compiler shows good performance
  - Eliminates unnecessary thread migrations;
  - Maximizes locality of data operations;
  - Affinity support for heterogeneous systems.

Enabling Asynchronous Computation

- Directed acyclic graph (DAG):
  where each node represents a task and the edges represents inter-task dependencies
- A task can begin execution only if all its predecessors have completed execution
- Should user express this directly?
- Compiler can generate tasks and graph (at least partially) to enhance performance
- What is “right” size of task?
OpenMP Research: Enhancing Tasks for Asynchrony

- Implicitly create DAG by specifying data input and output related to tasks
  - A task that produces data may need to wait for any previous child task that reads or writes the same locations
  - A task that consumes data may need to wait for any previous child task that writes the same locations
- Task weights (priorities)
- Groups of tasks and synchronization on groups

Asynchronous OpenMP Execution


OpenUH “All Task” Execution Model

- Replace fork-join with data flow execution model
  - Reduce synchronization, increase concurrency
  - Compiler transforms OpenMP code to a collection of tasks and a task graph.
  - Task graph represents dependences between tasks
  - Array region analysis; data read/write requests for the data the task needs.
- Map tasks to compute resources through task characterization using cost modeling
- Enable locality-aware scheduling and load-balancing

Feedback Optimizations

- Light-weight performance data collection
- Dynamic optimization
- Interoperate with external tools and schedulers
- Need very low-level interfaces to facilitate its implementation

Runtime Is Critical for Performance

- Compiler’s runtime support must
  - Adapt workload and data to environment
  - Respond to changes caused by application characteristics, power, faults, system noise
  - Dynamically and continuously
  - Provide feedback on application behavior
  - Uniform support for multiple programming models on heterogeneous platforms
  - Facilitate interoperability, (dynamic) mapping

Compiler’s Runtime Must Adapt
Interactions Across System Stack

- More interactions needed to share information
  - To support application development and tuning
  - Increase execution efficiency
  - Runtime needs application metadata, compiler analyses, architectural information, smart monitoring
  - Application developer needs feedback
  - Selective information-gathering in dynamic optimizer

Summary

- Projected exascale hardware requires us to rethink programming model and its execution support
  - Intra-node concurrency is fine-grained, heterogeneous
  - Memory is scarce and power is expensive
  - Data locality has never been so important
  - Opportunity for new programming models
  - Runtime continues to grow in importance for ensuring good performance
  - Need to migrate large apps: Where are the tools?

Increase Locality, Reduce Power

```c
void foo(double A[], double B[], double C[], int nrows, int ncols)
{
#pragma omp data_region acc_copyout(C), host_shared(A,B)
    #pragma omp acc_region
    for (int i=0; i < nrows; ++i)
        for (int j=0; j < ncols; j += NLANES)
            for (int k=0; k < NLANES; ++k) {
                int index = (i * ncols) + j + k;
            } // end accelerator region
    print2d(A,nrows,ncols);
    print2d(B,nrows,ncols);
    Transpose(C); // calls function w/another accelerator construct
} // end data_region

void Transpose(double X[], int nrows, int ncols) {
    #pragma omp acc_region acc_copy(X), acc_present(X)
    { … }
}
```
Lessons Learned from Prior Work

- Only a tight integration of application-provided meta-data and architecture description can let the runtime system take appropriate decisions
- Good analysis of thread affinity and data locality
- Task reordering
- H/W aware selective information gathering

Runtime needs to be integrated with OS