Challenges in Using Hierarchical Programming Approaches for Colossal Scale Systems

William Gropp
wgropp.cs.illinois.edu
Towards Exascale Architectures

DOE Sierra
- Power 9 with 4 NVIDIA Volta GPU
- 4320 nodes
- DOE Summit similar, but
  - 6 NVIDIA GPUs/node
  - 4608 nodes

Fugaku
- Fujitsu A64FX (includes Vector Extensions)
- 158,976 (+) nodes

DOE Frontier
- AMD with 4 AMD GPU
- Being deployed, 100+ racks

DOE Aurora
- Intel SR with 6 Intel Ponte Vecchio GPUs
- Being deployed, >9K nodes
Hardware Implications For Programs

• Heterogeneity in many ways
  • Processor – complex compute modes with scalar and vector
  • Many (but not all) include separate accelerators (GPUs and others)
  • Memory – Cache was bad enough; now HBM, other
  • I/O – Burst buffers (usually violating POSIX semantics), on node, central, remote (cloud)

• For algorithm developer and programmer, the issue is Performance Heterogeneity
  • Whether the implementation uses more than one chip(let) isn’t the issue – can you see performance impact of the different elements?
  • Even vectorization counts as performance heterogeneity in this view
    • Compilers still not great at vectorizing code, and often algorithmic changes needed to take full advantage of vectorization

• Impacts algorithm choice and program realization
Algorithm Considerations

• Start with the choice of mathematical model/numerical method
  • E.g., higher-order approximations for finite difference/element/volume trade floating point operations, data motion, and data size
  • Higher level choices can provide better locality
    • E.g., nonlinear Schwarz, with “local” nonlinear solves

• Performance models needed to guide algorithm design/choice
  • Model does not need to be precise – just good enough to guide
  • This is fortunate, as highly accurate performance models are very difficult to create and validate

• One Example: Node-aware algorithms
  • Performance model captures basic system hierarchy at node level
  • Avoid redundant data copies; optimize data motion for HW characteristics
  • Suggests a different approach for process topology mapping…
MPI On Multicore Nodes

• MPI Everywhere (single core/single thread MPI processes) still common
  • Easy to think about
  • We have good performance models (or do we?)

• In reality, there are issues
  • Memory per core declining
    • Need to avoid large regions for data copies, e.g., halo cells
    • MPI implementations could share internal table, data structures
      • May only be important for extreme scale systems
  • MPI Everywhere implicitly assume uniform communication cost model
    • Limits algorithms explored, communication optimizations used

• Even here, there is much to do for
  • Algorithm designers
  • Application implementers
  • MPI implementation developers

• One example: Can we use the single core performance model for MPI?
  • $T = s + r n$
  • Widely used and effective for designing parallel algorithms
Rates Per MPI Process

- Ping-pong between 2 nodes using 1-16 cores on each node
- Top is BG/Q, bottom Cray XE6
- “Classic” model predicts a single curve – rates independent of the number of communicating processes
Rates Per MPI Process: 128 cores

- Increasing core count makes the situation more complex
- Note roughly similar behavior for first 32 processes
  - 1 process / core
  - 64 cores/socket
- As before, classic model predicts a single curve – rate depends only on length, independent of number of communicating processes
Why this Behavior?

- The $T = s + r n$ model predicts the *same* performance independent of the number of communicating processes.
  - What is going on?
  - How should we model the time for communication?
A Slightly Better Model

• For \( k \) processes sending messages, the sustained rate is
  \[ \text{min}(R_{\text{NIC-NIC}}, k R_{\text{CORE-NIC}}) \]

• Thus
  \[ T = s + k \frac{n}{\text{min}(R_{\text{NIC-NIC}}, k R_{\text{CORE-NIC}})} \]

• Note if \( R_{\text{NIC-NIC}} \) is very large (very fast network), this reduces to
  \[ T = s + \frac{k n}{R_{\text{CORE-NIC}}} = s + \frac{n}{R_{\text{CORE-NIC}}} \]

• This model is approximate; additional terms needed to capture effect of
  shared data paths in node, contention for shared resources, etc.

• But this new term is by far the dominant one

• This is the max-rate model (for performance limited by the maximum
  available bandwidth)
  \[ \text{Logp model has a similar limitation and needs a similar modification} \]
Comparison on Cray XE6

Measured Data

Max-Rate Model

Performance Model to Algorithm

- Performance measurements of halo exchange show poor communication performance
  - Bandwidth per process low relative to “ping pong” measurements
  - Easy target – blame contention in the network
- But common default mapping of processes to nodes leads to more off-node communication
  - The max rate model predicts reduced performance once $R_{\text{NIC-NIC}}$ limit reached
- We can use this to create a better, and simpler, implementation of MPI_Cart_create
Building A Better MPI_Cart_create

- Hypothesis: A better process mapping within a node will provide significant benefits
  - Ignore the internode network topology
    - Vendors have argued that their network is fast enough that process mapping isn’t necessary
    - They may be (almost) right – once data enters the network

- Idea for Cartesian Process Topologies
  - Identify nodes (see MPI_Comm_split_type)
  - Map processes within a node to minimize internode communication
    - Trading intranode for internode communication
      [https://dl.acm.org/citation.cfm?id=3236377](https://dl.acm.org/citation.cfm?id=3236377)
    - *Using Node and Socket Information to Implement MPI Cartesian Topologies*, Parallel Computing, 2019
      [https://doi.org/10.1016/j.parco.2019.01.001](https://doi.org/10.1016/j.parco.2019.01.001)
Increasing Core Count Makes Proper Mapping More Important

• Cartesian mapping on Delta
  • CPU nodes have 2 AMD Milan x 64 cores each (GPU nodes have 1 AMD Milan and 4 A100 or A40 NVIDEA GPUs)
  • Slingshot network (mostly – NIC update coming)
  • Performance in B/s (higher is better)
• Default mapping provides poor performance
  • Cart is MPI_Cart_create – also MPI_COMM_WORLD
  • Nodec uses node-awareness, inspired by max-rate model
  • Nodech extends to socket (3-level)
How Important is Network Topology?

- No answer yet, but…
- 432 nodes, 3D halo exchange on Blue Waters
  - Requested a cube of nodes, used non-standard routines to implement mapping for network topology
- Part of study into scalable Krylov methods (looking to avoid the blocking MPI_Allreduce)
- Nodectrical version provides most of the benefit with no need for network topology information
- Some (nontrivial) further benefit possible by taking network topology into account
- But the largest contribution comes from node-awareness
- Thanks to Paul Eller for these results
Node Aware Algorithms

• Can use max rate model to design better algorithms for SpMV, related operations
  • May need to add additional terms – as in queue search overhead
  • May have unexpected results – e.g., MPI Direct from GPU not always best on multicore nodes (upcoming paper with Lockhart, Bienz, and Olson)

• Work with Amanda Bienz, Shelby Lockhart, Luke Olson
Realizing Algorithms for Hierarchical Systems

- Programming systems are used to implement algorithms
- Systems implement one or more programming models
  - MPI supports several programming models, including message passing, RDMA, and BSP
  - MPI designed to work with multi-threaded programming systems, e.g. OpenMP
- There is no one solution; different needs require different approaches
  - Seems obvious, but “MPI is bad” is all too common, and an example of fallacious argument
  - One set of needs is high vs. low level
  - Realization for performance (see hard-to-describe performance models) remains difficult
- Composing programming models requires attention to their interaction
  - In ”MPI+X”, the “+” is the hard part, and many of the challenges relate to maintaining performance in presence of contention for resources
- Performance for complex nodes hard to realize
  - Even matrix transpose difficult – due to heterogeneity in memory access (cache etc.)
  - Provide a high-level programming approach - permit algorithm flexibility
  - Provide a low-level programming approach – collaborate with compiler to explore code/data structure transformations
  - Work within existing languages to leverage software ecosystem
A High Level Approach

• Start with Python
  • High level language with strong software ecosystem
  • Integrate with code transformation/generation tools to create high-performance versions

• Alternative to creating a new Domain Specific Language

• Center for Exascale-Enabled Scramjet Design
  • Ceesd.Illinois.edu
  • Coupled hypersonic fluid flow with combustion and material interaction
  • Target is DOE Exascale systems – nodes with multiple accelerators
  • Changing nodes – P9+NVIDIA to AMD+AMD (and Intel+Intel if ANL included)
MIRGE Overview

**Computational Scientist Interface**

**CONTROL Layer** (Python)

while \( t < t_{\text{final}} \):
  \( p = \text{eos}(u) \)
  \( d = \text{diff}(u, p) \)
  \( f = \text{flux}(u, p) \)
  \( b = \text{bcs}(u, p) \)
  \( s = \text{surf}(u, p) \)
  \( r = \text{rhs}(d, f, b, s) \)
  \( u_{\text{next}} = \text{adv}(u, r) \)

**Kernels**

- EOS
- DG diff
- DG flux
- Particles
- Volume + Flux + BCs
- Time-Advance

**Material Models**
- Black Box
- WARP3D
- PUMA
- SPARITA

**Intermediate Representation** (DAG of kernels)

**Execution / Computation**

**Transform Engines**

- Adapter
- ROSE
- Adapter
- PIPS
- Adapter
- Loopy

**Code Generation**

- Cache

- **CUDA**
- **OpenMP**
- **OpenCL**

**Device + Scheduler**

- **PTX**
- **C**
- **SPIR-V**

- **Cache**

- **eager eval.**
Early Performance Results

- Abstractions visible to app:
  - numpy-like array, nested containers thereof
  - Indirection layer for array implementation

- Intermediate representations
  - Array DFG ("pytato") via lazy eval, lowered to
  - Imperative, polyhedral ("loopy")
  - OpenCL (for execution)

- Transformations
  - On Array DFG: Decide on materialization, fold redundant expressions
  - On loop IR: Loop fusion, array contraction, tile and prefetch
    - Organizational unit for tile/prefetch: "Fused einsum"

- Numerical method is DG-FEM
  - High level expression of method in NumPy
  - Compilation step translates to OpenCL
  - "Lazy" evaluation enables loop fusion, others

- Performance measured on Nvidia Titan V GPU
- Work of Kaushik Kulkarni and Andreas Klöckner
Practical Low-level Performance

• Processors have very complex performance behavior; extremely difficult to accurately predict performance or even order different alternatives
  • Without accurate, affordable performance model, no a priori decision can be made on which code (transformations) to use
• In practice, often need to consider alternatives
  • While compiler can do this in principle, rare and often impractical in practice
• How can you harness the power of code transformation and autotuning systems?
Locus

- Source code is annotated to define code regions
- Optimization file notation orchestrates the use of the optimization tools on the code regions defined
- Interface provides operations on the source code to invoke optimizations through:
  - Adding pragmas
  - Adding labels
  - Replacing code regions
- These operations are used by the interface to plug-in optimization tools
- Most tools are source-to-source
  - tools must understand output of previous tools
- Joint work with Thiago Teixeira and David Padua, “Managing Code Transformations for Better Performance Portability”, IJHPCA, 2019
  
  https://doi.org/10.1177%2F1094342019865606
Matrix Multiply Example

* #pragma @LOCUS loop=matmul
  
  for(i=0; i<M; i++)
    for(j=0; j<N; j++)
      for(k=0; k<K; k++)
        C[i][j] = beta*C[i][j] + alpha*A[i][k] * B[k][j];
Locus Generated Code
(for specific platform/size)

- `#pragma @LOCUS loop=matmul`
  ```c
  for(i_t = 0; i_t <= 7; i_t += 1)
  for(k_t = 0; k_t <= 3; k_t += 1)
  for(j_t = 0; j_t <= 1; j_t += 1)
    for(i_t_t = 8 * i_t; i_t_t <= ((8 * i_t) + 7); i_t_t += 1)
    for(k_t_t = 256 * k_t; k_t_t <= ((256 * k_t) + 255); k_t_t += 1)
    for(j_t_t = 32 * j_t; j_t_t <= ((32 * j_t) + 31); j_t_t += 1)
    for(i = 64 * i_t_t; i <= ((64 * i_t_t) + 63); i += 1)
      for(k = 4 * k_t_t; k <= ((4 * k_t_t) + 3); k += 1)
        for(j = 64 * j_t_t; j <= ((64 * j_t_t) + 63); j += 1)
          C[i][j] = beta*C[i][j] + alpha*A[i][k]*B[k][j];
  ```
DGEMM by Matrix Size

DGEMM on IBM Power

- XLC base
- Locus+XLC 2lt
- Locus+XLC 3lt
- GCC base
- Locus+GCC 2lt
- Locus+GCC 3lt

Matrices shape (n*n)

2048 4096 8192

Speedup (over XLC base)

DGEMM on Intel x86

- ICC base
- Locus+ICC 2lt
- Locus+ICC 3lt
- Intel MKL

Matrices Shape (n*n)

2048 4096

Speedup (over ICC base)
For most processors and regular (e.g., vectorizable) computations
  - Memory bandwidth for a chip is much larger than needed by a single core
  - Share of memory bandwidth for a core (with all cores accessing memory) is much smaller than needed to avoid waiting on memory

Performance tests on a single core can be very misleading
  - Example follows
  - Can use simple MPI tools to explore dependence on using one to all cores
    - See baseenv package
  - Ask this question when you review papers 😊
Stencil Sweeps

- Common operation for PDE solvers
  - Structured are often “matrix free”
  - Unstructured and structured mesh stencils have low "computational intensity" – number of floating-point operations per bytes moved
- Conventional wisdom is that cache blocking and similar optimizations are ineffective
  - For example, “Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors” argues this, and provides experimental data to support it
- But the analysis and experiments are usually based on one core per chip/socket
  - And the number of cores has grown substantially since 2007
  - What if every core is executing a stencil sweep?
void heat3d(double A[2][N+2][N+2][N+2]) {
    int i, j, t, k;
    #pragma @LOCUS loop=heat3d
    for(t = 0; t < T-1; t++) {
        for(i = 1; i < N+1; i++) {
            for(j = 1; j < N+1; j++) {
                for(k = 1; k < N+1; k++) {
                    A[(t+1)%2][i][j][k] = 0.125 * (A[t%2][i+1][j][k] - 2.0 * A[t%2][i][j][k] + A[t%2][i-1][j][k]) + 0.125 * (A[t%2][i][j+1][k] - 2.0 * A[t%2][i][j][k] + A[t%2][i][j-1][k]) + 0.125 * (A[t%2][i][j][k-1] - 2.0 * A[t%2][i][j][k] + A[t%2][i][j][k+1]) + A[t%2][i][j][k];
                }
            }
        }
    }
}

3D Heat on IBM Power

3D Heat on Intel x86
Summary

• Need to enable algorithm development
  • Great to see so many talks at this meeting embracing the need to match algorithms to real hardware – and take advantage of specialization
  • As others have stated, need the “right” performance model to drive algorithm design
  • Max-rate model can guide some node-aware algorithms

• Build on software ecosystem to realize algorithms
  • Need to consider high and low level needs – and address separately but compatibly

• Need to embrace composition of programming systems, address “+”