Obtaining Parallelism on Multicore and GPU Architectures in a Painless Manner

2010 Post-Convention Workshop
High Performance Implementation of Geophysical Applications

October 21, 2010

Michael Wolf, Mike Heroux, Chris Baker (ORNL)
Extreme-scale Algorithms and Software Institute (EASI)
EASI

• Work is part of Extreme-scale Algorithms and Software Institute (EASI)
  – DOE joint math/cs institute
  – Focused on closing the architecture-application performance gap
• Work primarily with Mike Heroux, Chris Baker (ORNL)
• Additional contributors
  – Erik Boman (SNL)
  – Carter Edwards (SNL)
  – Alan Williams (SNL)
Trilinos Framework

• Object-oriented software framework to enable the solution of large-scale, complex multi-physics engineering and scientific problems
  – Open source, implemented in C++

• Current work on new capabilities
  – Templated C++ code
    • Ordinal, scalar types
    • Node type
  – Better parallel abstraction
    • Abstract inter-node communication
    • Generic shared memory parallel node
    • Template meta-programming for write-once, run-anywhere kernel support
Shift in High Performance Computing (HPC)

- HPC shift in architectures (programming models?)
- CPUs increasingly multicore
  - Clock rates have peaked
  - Processors are becoming more NUMA
- Impact of accelerators/GPUs
  - #2 (Nebulae), #3 (Roadrunner) on Top500 list
  - Will play a role in or at least impact future supercomputers
- Complications
  - More diverse set of promising architectures
  - Heterogeneous architectures
    (e.g., multicore CPUs + GPUs)
Challenges in High Performance Computing (HPC)

• HPC shift in architectures (programming models?)
  – CPUs increasingly multicore
  – Impact of accelerators/GPUs
  – Heterogenous architectures

• Complications
  – More diverse set of promising architectures
  – Heterogeneous architectures

• Challenges
  – Obtaining good performance with our numerical kernels on many different architectures (w/o rewriting code)
  – Modifying current MPI-only codes
Obtaining good performance with our kernels on many different architectures
API for Shared Memory Nodes

- Goal: minimize effort needed to write scientific codes for a variety of architectures without sacrificing performance
  - Focus on shared memory node (multicore/GPU)
  - Abstract communication layer (e.g., MPI) between nodes
  - Our focus: multicore/GPU support in Trilinos distributed linear algebra library, Tpetra
API for Shared Memory Nodes

• Find the correct level for programming the node architecture
  – Too low: code numerical kernel for each node
    • Too much work to move to a new platform

    Num. Implementations
    $m$ kernels * $n$ nodes = $mn$

  – Too high: code once for all nodes
    • Difficult to exploit hardware features
    • API is too big and always growing

• Somewhere in the middle (Trilinos package Kokkos):
  – Implement small set of parallel constructs (parallel for, parallel reduce) on each architecture
  – Write kernels in terms of constructs

    Num. Implementations
    $m$ kernels + $c$ constructs * $n$ nodes = $m + cn$

Trilinos: $c=2$
**Kokkos Compute Model**

- Trilinos package with API for programming to a generic parallel node
  - Goal: allow code, once written, to run on any parallel node, regardless of architecture
- Kokkos compute model
  - User describes kernels for parallel execution on a node
  - Kokkos provides common parallel work constructs
    - Parallel for loop, parallel reduction
- Different nodes for different architectures
  - Intel Thread Building Blocks
    - **TBBNode**
    - **TPINode**
    - Pthread based
  - CUDA (via Thrust)
    - **CUDANode**
    - **SerialNode**
- Support new platforms by implementing new node classes
  - Same user code
Kokkos Compute Model

• Kokkos node provides generic parallel constructs:
  – `Node::parallel_for()` and `Node::parallel_reduce()`
  – Currently implemented for several node architectures
    • TBBNode, TPINode, CUDANode, SerialNode

• User develops kernels in terms of these parallel constructs

• Template meta-programming does the rest
  – Produces kernels tailored for the specific architecture
Kokkos: axpy() with Parallel For

template <class WDP>
void Node::parallel_for(int beg, int end, WDP workdata);

struct AxpyOp {
    const double *x;
    double *y;
    double a;
    void execute(int i)
    {
        y[i] = a*x[i] + y[i];
    }
};

Kokkos

void exampleFn(double *x, double *y, double a) {
    AxpyOp op1;
    op1.y = y;
    op1.x = x;
    op1.a = a;
    node->parallel_for<AxpyOp>(0,n,op1);
}

WDP
## Shared Memory Timings for Simple Iterations

<table>
<thead>
<tr>
<th>Node</th>
<th>Power method (mflop/s)</th>
<th>CG iteration (mflop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SerialNode</td>
<td>101</td>
<td>330</td>
</tr>
<tr>
<td>TPINode(1)</td>
<td>116</td>
<td>375</td>
</tr>
<tr>
<td>TPINode(2)</td>
<td>229</td>
<td>735</td>
</tr>
<tr>
<td>TPINode(4)</td>
<td>453</td>
<td>1,477</td>
</tr>
<tr>
<td>TPINode(8)</td>
<td>618</td>
<td>2,020</td>
</tr>
<tr>
<td>TPINode(16)</td>
<td>667</td>
<td>2,203</td>
</tr>
<tr>
<td>CUDANode</td>
<td>2,584</td>
<td>8,178</td>
</tr>
</tbody>
</table>

- **Physical node:**
  - One NVIDIA Tesla C1060
  - Four 2.3 GHz AMD Quad-core CPUs

- Power method: one SpMV op, three vector operations
- Conjugate gradient: one SpMV op, five vector operations
- Matrix is a simple 3-point discrete Laplacian with 1M rows
- Wrote kernels once in terms of constructs
  - Got different architecture implementations for “free”
Modifying Current MPI-Only Codes
(Bimodal MPI and MPI+Threads Programming)
Multithreading can improve some numerical kernels
- E.g., domain decomposition preconditioning with incomplete factorizations

For flat MPI, inflation in iteration count due to number of subdomains

By introducing multithreaded triangular solves on each node
- Solve triangular system on larger subdomains
- Reduce number of subdomains (MPI tasks), mitigate iteration inflation

Motivation: Why Not Flat MPI?

Strong scaling of Charon on TLCC (P. Lin, J. Shadid 2009)
Bimodal MPI and MPI+Threads Programming

- Parallel machine with \( p = m \times n \) processors:
  - \( m \) = number of nodes
  - \( n \) = number of shared memory cores per node

- Two typical ways to program
  - Way 1: \( p \) MPI processes (flat MPI)
    - Massive software investment in this programming model
  - Way 2: \( m \) MPI processes with \( n \) threads per MPI process
    - Requires ubiquitous change when starting from “way 1”
• Two typical ways to program
  – Way 1: \( p \) MPI processes (flat MPI)
  – Way 2: \( m \) MPI processes with \( n \) threads per MPI process

• Third way (bimodal MPI and hybrid MPI+threads)
  – “Way 1” in some parts of the execution (the app)
  – “Way 2” in others (the solver)

• Challenges for bimodal programming model
  – Utilizing all cores (in Way 1 mode)
  – Threads on node need access to data from all MPI tasks on node

• Solution: MPI shared memory allocation
**MPI Shared Memory Allocation**

Idea:

- Shared memory alloc/free functions:
  - MPI_Comm_alloc_mem
  - MPI_Comm_free_mem
- Architecture-aware communicators:
  - MPI_COMM_NODE – ranks on node
  - MPI_COMM_SOCKET – UMA ranks
  - MPI_COMM_NETWORK – inter node
- Status:
  - Available in current development branch of OpenMPI
  - Under development in MPICH
  - Demonstrated usage with threaded triangular solve
  - Proposed to MPI-3 Forum

```c
int n = ...;
double* values;
MPI_Comm_alloc_mem(
    MPI_COMM_NODE, // comm (SOCKET works too)
    n * sizeof(double), // size in bytes
    MPI_INFO_NULL, // placeholder for now
    &values); // Pointer to shared array (out)

// At this point:
// - All ranks on a node/socket have pointer to a shared buffer.
// - Can continue in MPI mode (using shared memory algorithms)
// - or can quiet all but one rank:
int rank;
MPI_Comm_rank(MPI_COMM_NODE, &rank);

// Start threaded code segment, only on rank 0 of the node
if (rank==0)
{
...
}
MPI_Comm_free_mem(MPI_COMM_NODE, values);
```

Collaborators: B. Barrett, R. Brightwell - SNL; Vallee, Koenig - ORNL
Simple MPI Program

double *x = new double[n];
double *y = new double[n];

MPIkernel1(x,y);
MPIkernel2(x,y);

delete [] x;
delete [] y;

• Simple MPI application
  – Two distributed memory/MPI kernels

• Want to replace an MPI kernel with more efficient hybrid MPI/threaded kernel
  – Threading on multicore node
### Simple Bimodal MPI + Hybrid Program

```c
double *x = new double[n];
double *y = new double[n];
MPIkernel1(x,y);
MPIkernel2(x,y);
delete [] x;
delete [] y;
```

```c
MPI_Comm_size(MPI_COMM_NODE, &nodeSize);
MPI_Comm_rank(MPI_COMM_NODE, &nodeRank);
double *x, *y;
MPIComm_alloc_mem(MPI_COMM_NODE,n*nodeSize*sizeof(double),
    .MPI_INFO_NULL, &x);
MPIComm_alloc_mem(MPI_COMM_NODE,n*nodeSize*sizeof(double),
    .MPI_INFO_NULL, &y);
MPIkernel1(&(x[nodeRank * n]),&(y[nodeRank * n]));
if(nodeRank==0)
{
    .hybridKernel2(x,y);
}
MPIComm_free_mem(MPI_COMM_NODE, &x);
MPIComm_free_mem(MPI_COMM_NODE, &y);
```

- Very minor changes to code
  - MPIKernel1 does not change
- Hybrid MPI/Threaded kernel runs on rank 0 of each node
  - Threading on multicore node
Iterative Approach to Hybrid Parallelism

- Many sections of parallel applications scale extremely well using flat MPI
- Approach allows introduction of multithreaded kernels in iterative fashion
  - “Tune” how multithreaded an application is
- Focus on parts of application that don’t scale with flat MPI
Iterative Approach to Hybrid Parallelism

MPI_Comm_size(MPI_COMM_NODE, &nodeSize);
MPI_Comm_rank(MPI_COMM_NODE, &nodeRank);

double *x, *y;

MPI_Comm_alloc_mem(MPI_COMM_NODE,n*nodeSize*sizeof(double),
 . MPI_INFO_NULL, &x);
MPI_Comm_alloc_mem(MPI_COMM_NODE,n*nodeSize*sizeof(double),
 . MPI_INFO_NULL, &y);

MPIKernel1(&((x[nodeRank * n]),&(y[nodeRank * n])));

if(nodeRank==0)
{
 . hybridKernel2(x,y);
}

MPI_Comm_free_mem(MPI_COMM_NODE, &x);
MPI_Comm_free_mem(MPI_COMM_NODE, &y);

• Can use 1 hybrid kernel
Iterative Approach to Hybrid Parallelism

MPI_Comm_size(MPI_COMM_NODE, &nodeSize);
MPI_Comm_rank(MPI_COMM_NODE, &nodeRank);

double *x, *y;

MPI_Comm Alloc_mem(MPI_COMM_NODE,n*nodeSize*sizeof(double),
.    MPI_INFO_NULL, &x);
MPI_Comm_alloc_mem(MPI_COMM_NODE,n*nodeSize*sizeof(double),
.    MPI_INFO_NULL, &y);

if(nodeRank==0)
{
    .    hybridKernel1(x,y);
    .    hybridKernel2(x,y);
}

MPI_Comm_free_mem(MPI_COMM_NODE, &x);
MPI_Comm_free_mem(MPI_COMM_NODE, &y);

• Or use 2 hybrid kernels
\[ \begin{align*}
    r_0 &= b - Ax_0 \\
    z_0 &= M^{-1}r_0 \\
    p_0 &= z_0 \\
    \text{for } (k = 0; \ k < \text{maxit}, \ ||r_k|| < \text{tol}) \{ \\
    &\quad \alpha_k = \frac{r_k^T z_k}{p_k^T Ap_k} \\
    &\quad x_{k+1} = x_k + \alpha_k p_k \\
    &\quad r_{k+1} = r_k - \alpha_k Ap_k \\
    &\quad z_{k+1} = M^{-1}r_{k+1} \\
    &\quad \beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k} \\
    &\quad p_{k+1} = z_{k+1} + \beta_k p_k
    \} \]

Mantevo miniapp: HPCPCG

Use multithreading for preconditioning
PCG Algorithm

\[ r_0 = b - Ax_0 \]
\[ z_0 = M^{-1}r_0 \]
\[ p_0 = z_0 \]
\[ \text{for } (k = 0; k < \text{maxit}, \| r_k \| < \text{tol}) \]
\{ 
\[ \alpha_k = \frac{r_k^T z_k}{p_k^T Ap_k} \]
\[ x_{k+1} = x_k + \alpha_k p_k \]
\[ r_{k+1} = r_k - \alpha_k Ap_k \]
\[ z_{k+1} = M^{-1}r_{k+1} \]
\[ \beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k} \]
\[ p_{k+1} = z_{k+1} + \beta_k p_k \]
\}
PCG Algorithm – MPI part

\[ r_0 = b - Ax_0 \]
\[ z_0 = M^{-1}r_0 \]
\[ p_0 = z_0 \]

for \( (k = 0; k < \text{maxit}, \|r_k\| < \text{tol}) \)

\[
\begin{align*}
\alpha_k &= \frac{r_k^T z_k}{p_k^T Ap_k} \\
x_{k+1} &= x_k + \alpha_k p_k \\
r_{k+1} &= r_k - \alpha_k Ap_k \\
\beta_k &= \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k} \\
p_{k+1} &= z_{k+1} + \beta_k p_k
\end{align*}
\]

Flat MPI operations
Multithreaded block preconditioning to reduce number of subdomains
Preliminary PCG Results

Itera&ons

Flat MPI PCG  Threaded Preconditioning

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Threads=1</th>
<th>Threads=2</th>
<th>Threads=4</th>
<th>Threads=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD16</td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
</tr>
<tr>
<td>FD32</td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
</tr>
</tbody>
</table>

Runtime relative to flat MPI PCG

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Threads=2</th>
<th>Threads=4</th>
<th>Threads=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD16</td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
</tr>
<tr>
<td>FD32</td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
<td><img src="Image" alt="" /></td>
</tr>
</tbody>
</table>
Summary: Kokkos Package in Trilinos

• Goal: To help obtain good performance of numerical kernels on different architectures (w/o rewriting code)

• API for programming generic shared-memory nodes
  – Allows write-once, run-anywhere portability
  – Support new nodes by writing parallel constructs for new node

• Nodes implemented support
  – Intel TBB, Pthreads, CUDA-capable GPUs (via Thrust), serial

• For more info about Kokkos, Trilinos:
  – http://trilinos.sandia.gov/
Summary: Bimodal MPI and MPI+Threads Programming

• How to modify current MPI-only codes
  – Massive investment in MPI-only software

• MPI shared memory allocation will be important
  – Allows seamless combination of traditional MPI programming with multithreaded or hybrid kernels

• Iterative approach to multithreading

• Work-in-progress: PCG implementation using MPI shared memory extensions
  – Effective in reducing iterations
  – Runtime did not decrease much
  – Need more scalable multithreaded triangular solver algorithm