SAND2010-7399C

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)

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

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 enabled the solution of large-scale, complex multi-physics engineering and scientific problems
  - Open source, implemented in object-oriented C++
- Stage 2 Trilinos under development
- Templated C++ code
  - Ordinal, scalar types
  - Node type
- Abstract inter-node communication
- Generic shared memory parallel node
  - Template meta-programming aiming for write-once, run-anywhere kernel support

Programming Today for Tomorrow's Machines

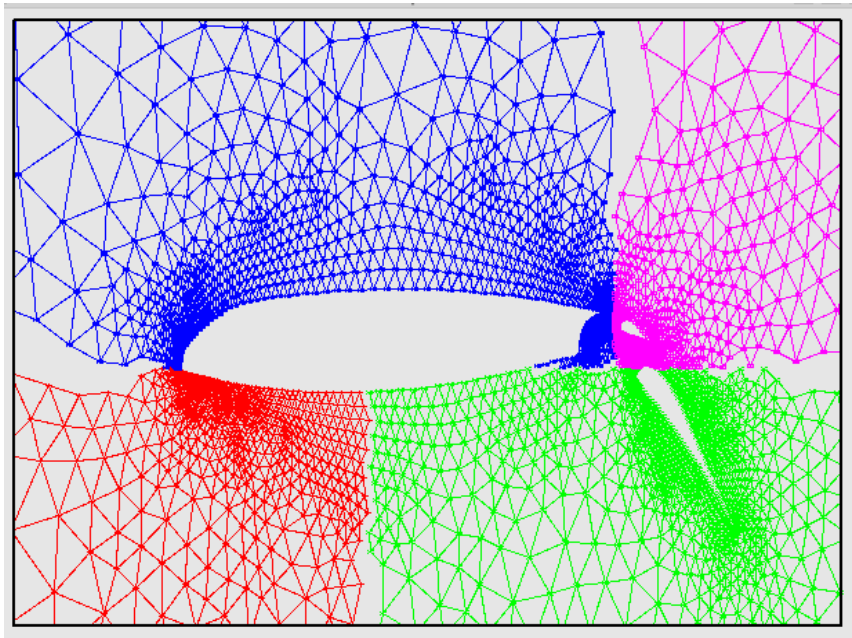
- Parallel Programming in the small:
  - Focus: writing sequential code fragments.
  - Programmer skills:
    - 10%: Pattern/framework experts (domain-aware).
    - 90%: Domain experts (pattern-aware)
- Languages needed are already here.
  - Exception: Large-scale data-intensive graph?

FE/FV/FD Parallel Programming Today

```
for ((i,j,k) in points/elements on subdomain) {
  compute coefficients for point (i,j,k)
  inject into global matrix
}
```

Notes:

- User in charge of:
  - Writing physics code
  - Iteration space traversal
  - Storage association
- Pattern/framework/runtime in charge of:
  - SPMD execution

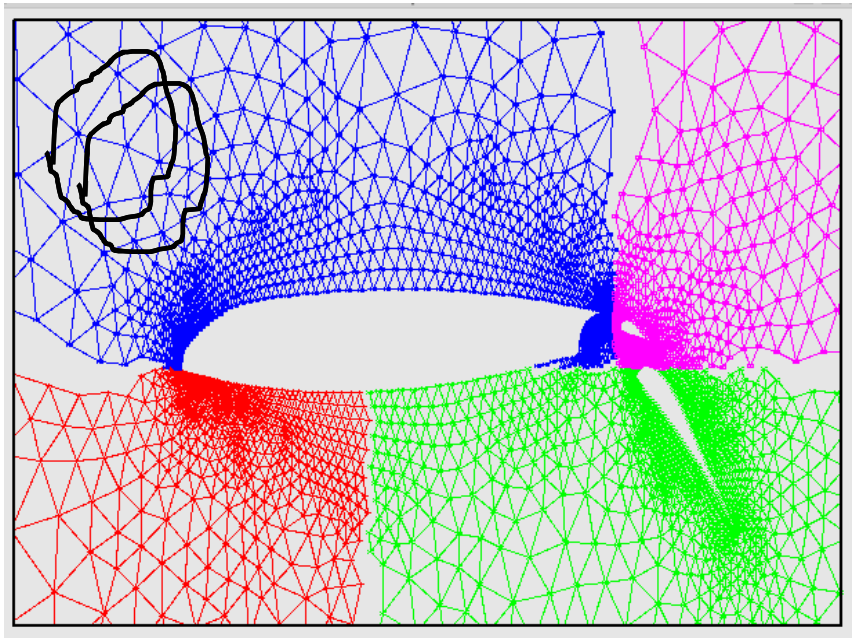


FE/FV/FD Parallel Programming Tomorrow

```
pipeline <i,j,k> {
  filter(addPhysicsLayer1<i,j,k)>);
    ...
  filter(addPhysicsLayern<i,j,k>);
   filter(injectIntoGlobalMatrix<i,j,k>);
}
```

Notes:

- User in charge of:
  - Writing physics code (filter)
  - Registering filter with framework
- Pattern/framework/runtime in charge of:
  - SPMD execution
  - Iteration space traversal
    - Sensitive to temporal locality
  - Filter execution scheduling
  - Storage association
- Better assignment of responsibility (in general)



Challenges in High Performance Computing (HPC)

- HPC shift in architectures (programming models?)
- CPUs increasingly multicore
  - Flatlining of clock rates
  - 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
- Heterogenous architectures
  - e.g., multicore CPUs + GPUs
- Challenges
  - Obtaining good performance with our kernels on many different architectures (w/o rewriting code)
  - Modifying current MPI-only codes

Obtaining good performance with our kernels on many different architectures

An API for Shared Memory Nodes

- Goal: minimize effort needed to write scientific codes for a variety of architectures
  - Our focus: multicore/GPU support in our distributed linear algebra library, Tpetra
- Find the correct level for programming the node:
  - Too low: code kernel for each node
    - Too much work to move to a new platform
  - Too high: code once for all nodes.
    - Difficult to exploit hardware features
    - API is too big and always growing
- Kokkos: somewhere in the middle:
  - Implement small set of parallel constructs on each architecture
  - Write kernels in terms of constructs

Num. Implementations
m kernels * n nodes = mn

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

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
  - Description of kernels for parallel execution on a node
  - Provides common parallel work constructs
    - Parallel for loop, parallel reduction
- Different nodes for different architectures

• TBBNode	• TPINode
• 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()`
- User develops kernels for parallel constructs
- Template meta-programming does the rest:
  - `TBBNode< ComputePotentials<3D,LJ> >::parallel_for`

- Parallel for:

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

- Work-data pair (WDP) struct provides:
  - loop body via WDP::execute(int i)
- Semantics: execute(i) will be called exactly once for all i in [beg,end)

Kokkos: axpy() with Parallel For

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

```
template <class T>
struct AxpyOp {
  const T * x;
  T * y;
  T alpha, beta;
  void execute(int i)
  {  y[i] = alpha*x[i] + beta*y[i]; }
};
```

```
double *x,*y;
…
AxpyOp op1;
op1.y = y;
op1.x = x;
…
node->parallel_for< AxpyOp<double> >(0,n,op1);
```

Kokkos Linear Algebra Library

- Subpackage of Kokkos provides set of data structures and kernels for local SMP linear algebra objects
  - Coded using the Kokkos Parallel Node API
- Tpetra (global) objects consist of abstract inter-node communicator and corresponding (local) Kokkos object:

```
T Tpetra::Vector<T,N>::dot(Tpetra::Vector<T,N> v)
{
   T lcl = lclVec_->dot( v.lclVec_ );
   return comm_->reduceAll(SUM, lcl);
}
```

- Implementing new Node ports Tpetra without changes.

Shared Memory Timings for Simple Iterations

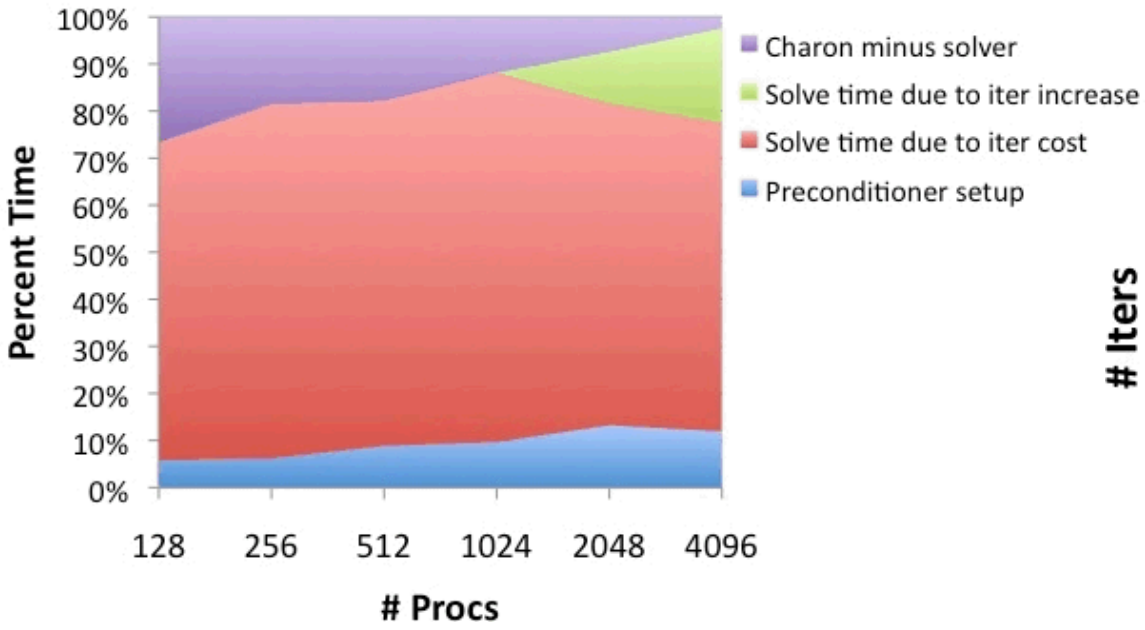
Node	Power method (mflop/s)	CG iteration (mflop/s)
SerialNode	**101**	**330**
TPINode(1)	116	375
TPINode(2)	229	735
TPINode(4)	453	1,477
TPINode(8)	618	2,020
TPINode(16)	**667**	**2,203**
GPUNode	**2,584**	**8,178**

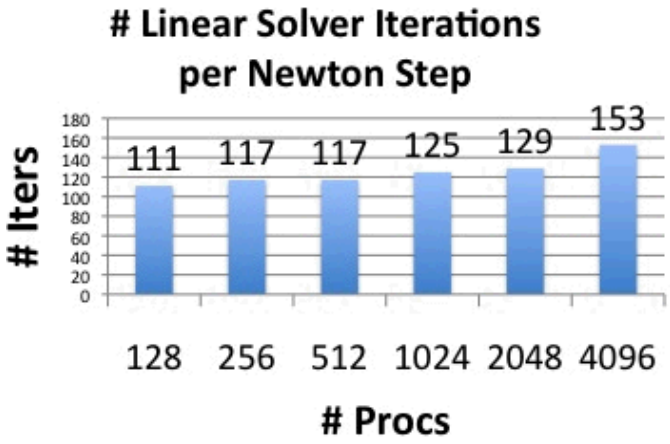
- Physical node:
  - One NVIDIA Tesla C1060
  - Four 2.3 GHz AMD Quad-core CPUs

- 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
- TPINode: pthreads-based node
- GPUNode: Thrust-based CUDA node

Modifying Current MPI-Only Codes (Bimodal MPI and MPI+Threads Programming)

Motivation: Why Not MPI-Only?

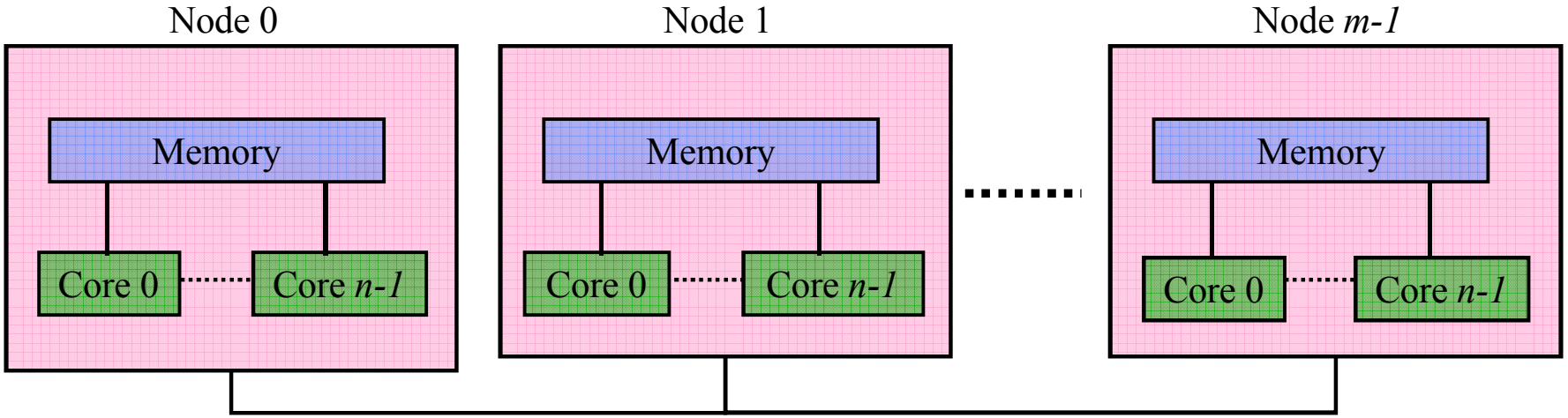




Strong scaling of Charon on TLCC (P. Lin, J. Shadid 2009)

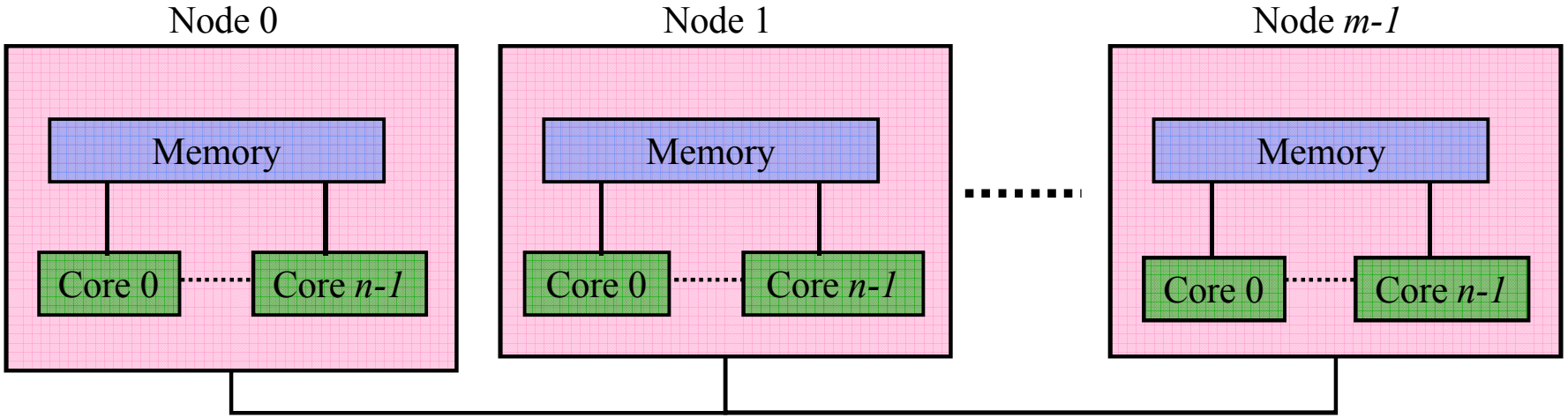
- Multithreading can improve some numerical kernels
  - E.g., domain decomposition preconditioning with incomplete factorizations
- Inflation in iteration count due to number of subdomains
- By introducing multithreaded triangular solves on each node
  - Solve triangular system on larger subdomains
  - Reduce number of subdomains (MPI tasks)

MPI + Hybrid MPI/Multithreaded Programming



- Parallel machine with $p = m * n$ 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
- Third way
  - "Way 1" in some parts of the execution (the app)
  - "Way 2" in others (the solver)

MPI + Hybrid MPI/Multithreaded Programming



- 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)
  - Interfacing between two modes
- Solution: MPI shared memory allocation

MPI Shared Memory Allocation

Idea:

- Shared memory alloc/free functions:
  - MPI_Comm_alloc_mem
  - MPI_Comm_free_mem
- Predefined 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

```
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[4];
double *y = new double[4];

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
  - Threading on multicore node

Simple MPI + Hybrid Program

```
double *x = new double[4];
double *y = new double[4];

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

delete [] x;
delete [] y;
```

```
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);
```

- 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
- Can 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

PCG Algorithm

$$r_0 = b - Ax_0$$
$$z_0 = M^{-1} r_0$$
$$p_0 = z_0$$
$$\text{for } (k = 0;\ k < maxit,\ ||r_k|| < tol)$$
$$\{$$
$$\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$$
$$x_{k+1} = x_k + \alpha_k p_k$$
$$r_{k+1} = r_k - \alpha_k A p_k$$
$$z_{k+1} = M^{-1} r_{k+1}$$
$$\beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$$
$$p_{k+1} = z_{k+1} + \beta_k p_k$$
$$\}$$

Used symmetric Gauss-Seidel as preconditioner (2 triangular solves)

PCG Algorithm

$$r_0 = b - Ax_0$$
$$z_0 = M^{-1} r_0$$
$$p_0 = z_0$$
$$\text{for } (k = 0;\ k < maxit,\ ||r_k|| < tol)$$
$$\{$$
$$\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$$
$$x_{k+1} = x_k + \alpha_k p_k$$
$$r_{k+1} = r_k - \alpha_k A p_k$$
$$z_{k+1} = M^{-1} r_{k+1}$$
$$\beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$$
$$p_{k+1} = z_{k+1} + \beta_k p_k$$
$$\}$$

Shared memory variables

PCG Algorithm – MPI part

$$r_0 = b - Ax_0$$
$$z_0 = M^{-1} r_0$$
$$p_0 = z_0$$
$$\text{for } (k = 0;\ k < maxit,\ ||r_k|| < tol)$$
$$\{$$
$$\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$$
$$x_{k+1} = x_k + \alpha_k p_k$$
$$r_{k+1} = r_k - \alpha_k A p_k$$
$$z_{k+1} = M^{-1} r_{k+1}$$
$$\beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$$
$$p_{k+1} = z_{k+1} + \beta_k p_k$$
$$\}$$

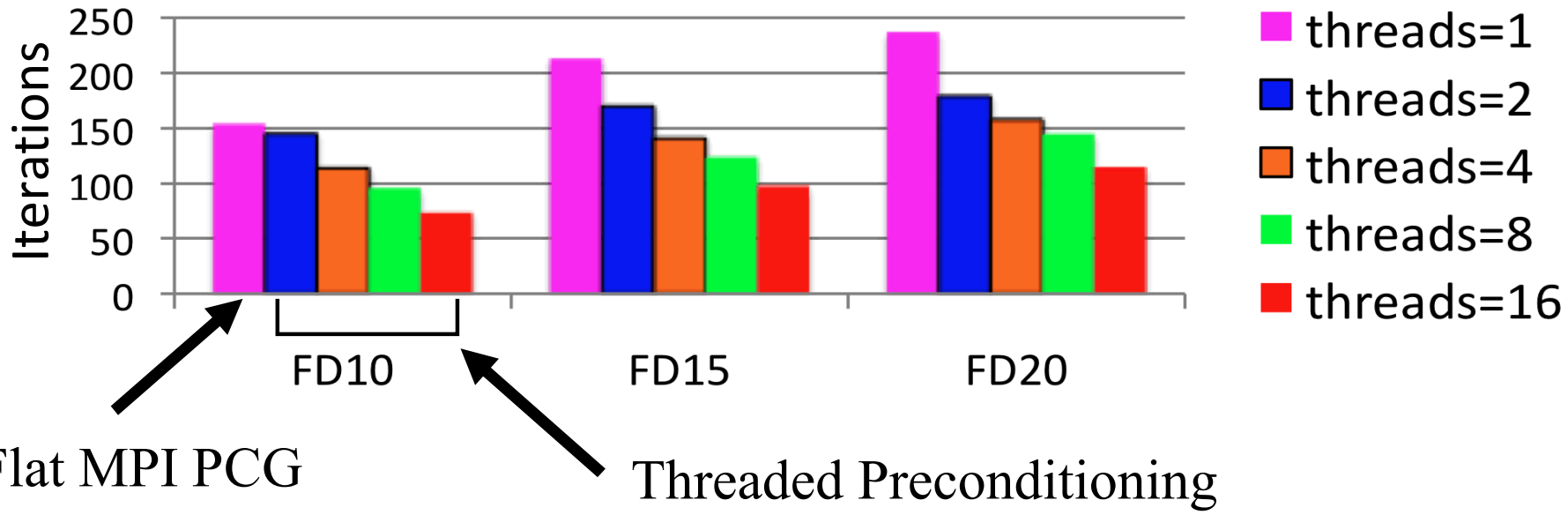
Flat MPI operations

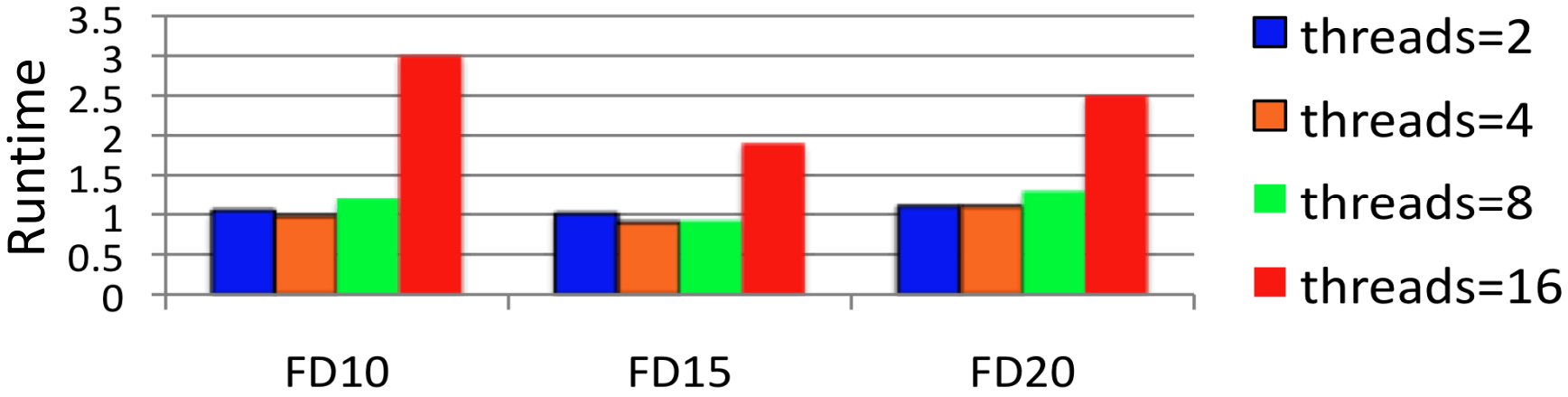
PCG Algorithm – Threaded Part

$$r_0 = b - Ax_0$$
$$z_0 = M^{-1} r_0$$
$$p_0 = z_0$$
$$\text{for } (k = 0;\ k < maxit,\ ||r_k|| < tol)$$
$$\{$$
$$\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$$
$$x_{k+1} = x_k + \alpha_k p_k$$
$$r_{k+1} = r_k - \alpha_k A p_k$$
$$z_{k+1} = M^{-1} r_{k+1}$$
$$\beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$$
$$p_{k+1} = z_{k+1} + \beta_k p_k$$
$$\}$$

Multithreaded block preconditioning to reduce number of subdomains

Preliminary PCG Results





Runtime relative to flat MPI PCG

Summary: Kokkos Package in Trilinos

- API for programming generic shared-memory nodes
- C++ template meta-programming approach
  - Allows write-once, run-anywhere portability
  - Support new nodes by writing parallel constructs for node
- Nodes implemented support
  - Intel TBB, Pthreads, CUDA-capable GPUs (via Thrust)
- Provide shared-memory linear algebra objects and kernels
  - Currently used by the Tpetra distributed linear algebra library
- For more info about Kokkos, Trilinos:
  - http://trilinos.sandia.gov/

Summary: MPI + Hybrid MPI/threaded Programming

- MPI shared memory allocation useful
  - Allows seamless combination of traditional MPI programming with multithreaded or hybrid kernels
- Iterative approach to multithreading
- Implemented PCG using MPI shared memory extensions and level set method
  - Effective in reducing iterations
  - Runtime did not decrease (work in progress)
- Better triangular solver algorithms needed for matrices with small average level size

Extra

Limitations/Issues

- Constructs limited
  - Much more limited than TBB
- Data partitioning to simplistic
  - May be problematic for stencils
  - Handles nested parallel loops poorly

Kokkos Compute Model

- Have to find the correct level for programming the node:
  - Too low: code kernel for each node
    - Too much work to move to a new platform.
    - Duplicated effort: `dot()` is a lot like `norm1()`
  - Too high: code once for all nodes.
    - Difficult to exploit hardware features.
    - API is too big and always growing.
    - A programming language without a compiler.
- Somewhere in the middle:
  - Parallel reduction is the intersection of `dot()` and `norm1()`
  - Parallel for loop is the intersection of `axpy()` and mat-vec
  - We need a way of fusing kernels with these basic constructs.

m kernels * n nodes = m * n

Too many implementations!

m kernels + k ops * n nodes = m + k * n

Too many instructions!

m kernels + 2 constructs * n nodes = m + 2 * n

Kokkos Compute Model

- Template meta-programming
  - Dispatch model (function + partitionable data)
- Node provides generic parallel constructs:
  - `Node::parallel_for()` and `Node::parallel_reduce()`
- User develops kernels for parallel constructs
- Template meta-programming does the rest:
  - `TBBNode< ComputePotentials<3D,LJ> >::parallel_for`
- Parallel for:

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

- Work-data pair (WDP) struct provides:
  - loop body via WDP::execute(int i)
- Semantics: execute(i) will be called exactly once for all i in [beg,end)
- Calls may occur in parallel and in any order.

Kokkos Compute Model

- Template meta-programming is the answer.
  - This is the same approach that Intel TBB and Thrust take.
- Node provides generic parallel constructs:
  - `Node::parallel_for()` and `Node::parallel_reduce()`
- User fills the holes in the generic constructs via custom kernels.

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

Work-data pair (WDP) struct provides:

- **loop body via** `WDP::execute(int i)`

Semantics: `execute(i)` **will be called exactly once for all i in [beg, end)**

Calls may occur in parallel and in any order.

```
template <class WDP>
WDP::ReductionType
Node::parallel_reduce(int beg, int end,
                      WDP workdata    );
```

Work-data pair (WDP) struct provides:

- **reduction type** `WDP::ReductionType`
- **element generation via** `WDP::generate(int i)`
- **identity element via** `WDP::identity()`
- **reduction via** `WDP::reduce(...)`

Semantics: `generate(i)` **will be called exactly once for all i in [beg, end)**

Calls may occur in parallel and in any order.

Example Kernels: axpy() and dot()

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

```
template <class WDP>
WDP::ReductionType
Node::parallel_reduce(int beg, int end,
                      WDP workdata    );
```

```
template <class T>
struct AxpyOp {
  const T * x;
  T * y;
  T alpha, beta;
  void execute(int i)
  { y[i] = alpha*x[i] + beta*y[i]; }
};
```

```
template <class T>
struct DotOp {
  typedef T ReductionType;
  const T * x, * y;
  T identity()       { return (T)0;      }
  T generate(int i)  { return x[i]*y[i]; }
  T reduce(T x, T y) { return x + y;     }
};
```

```
AxpyOp<double> op;
op.x = ...;  op.alpha = ...;
op.y = ...;  op.beta  = ...;
node.parallel_for< AxpyOp<double> >
                 (0, length, op);
```

```
DotOp<float> op;
op.x = ...;  op.y = ...;
float dot;
dot = node.parallel_reduce< DotOp<float> >
                          (0, length, op);
```