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

Sandia National Laboratories

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

Sandia National Laboratories

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);
```

Kokkos

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

Data needed for operation

Serial op for index i

WDP

```
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);
}
```

Sandia National Laboratories

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**
CUDANode	**2,584**	**8,178**

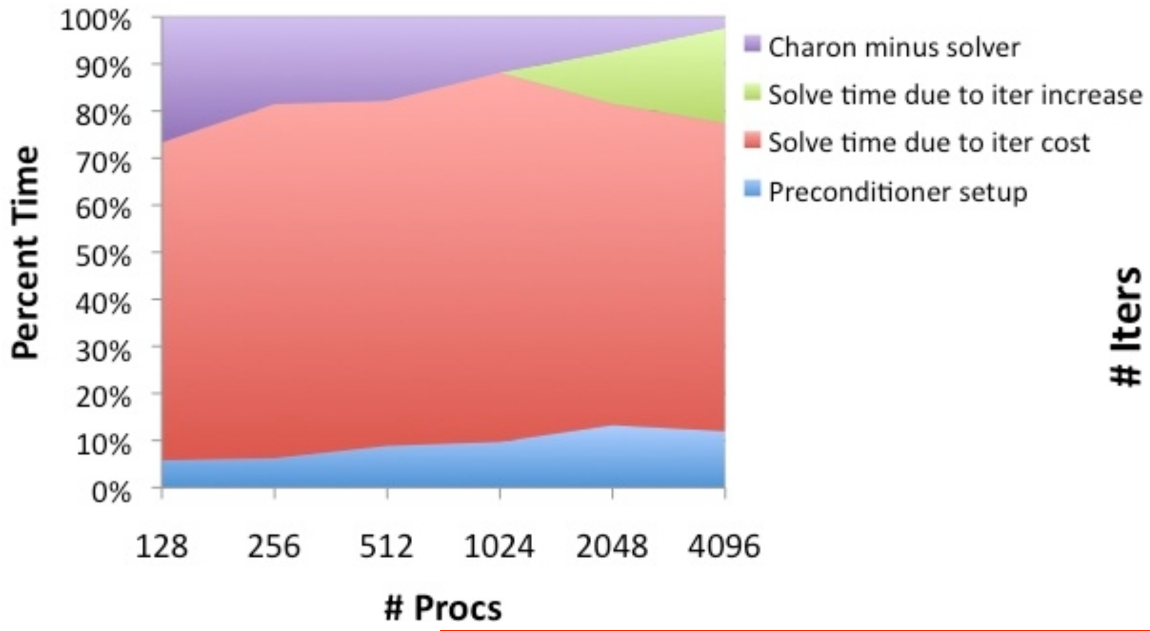
- Physical node:
  - One NVIDIA Tesla C1060
  - Four 2.3 GHz AMD Quad-core CPUs

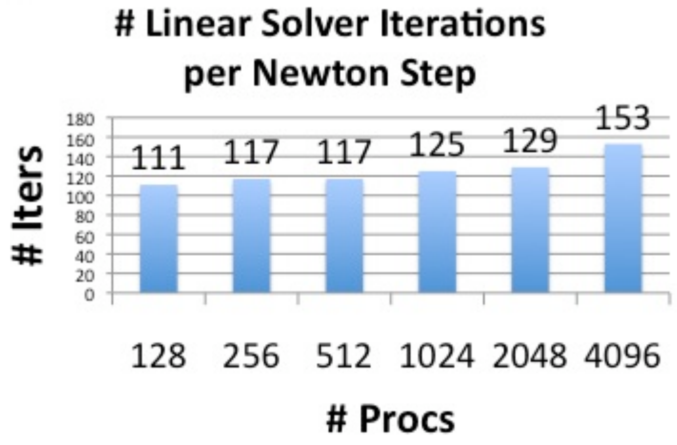
- Power method: one SpMV op, three vector operations
- Conjugate gradient: one SpMV op, five vector operations
- Matrix is a simple 3-point discrete Laplacian with 1M rows
- Wrote kernels once in terms of constructs
  - Got different architecture implementations for "free"

Sandia National Laboratories

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

Motivation: Why Not Flat MPI?

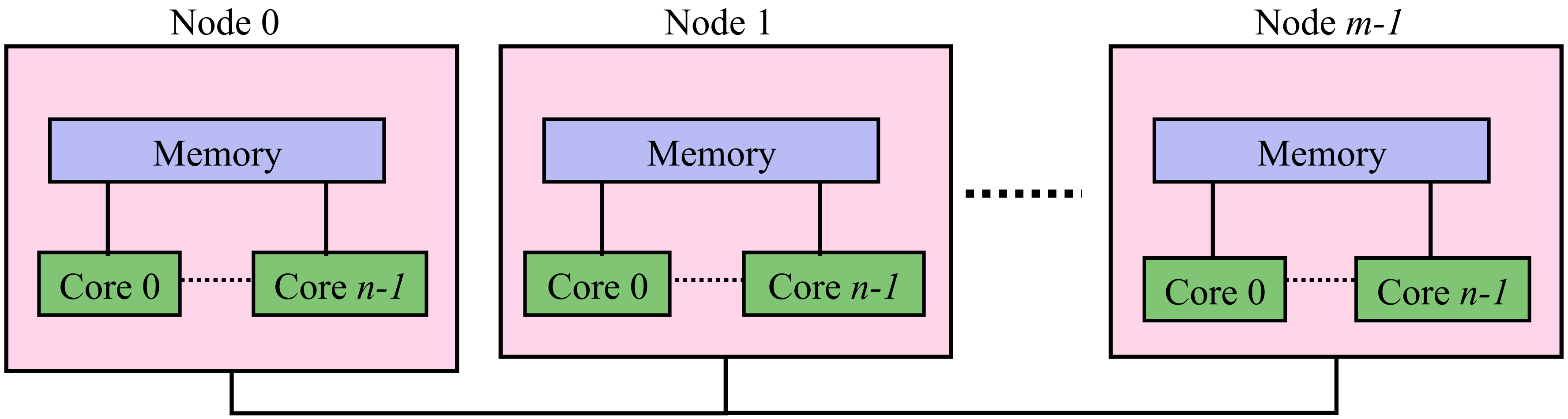




Strong scaling of Charon on TLCC (P. Lin, J. Shadid 2009)

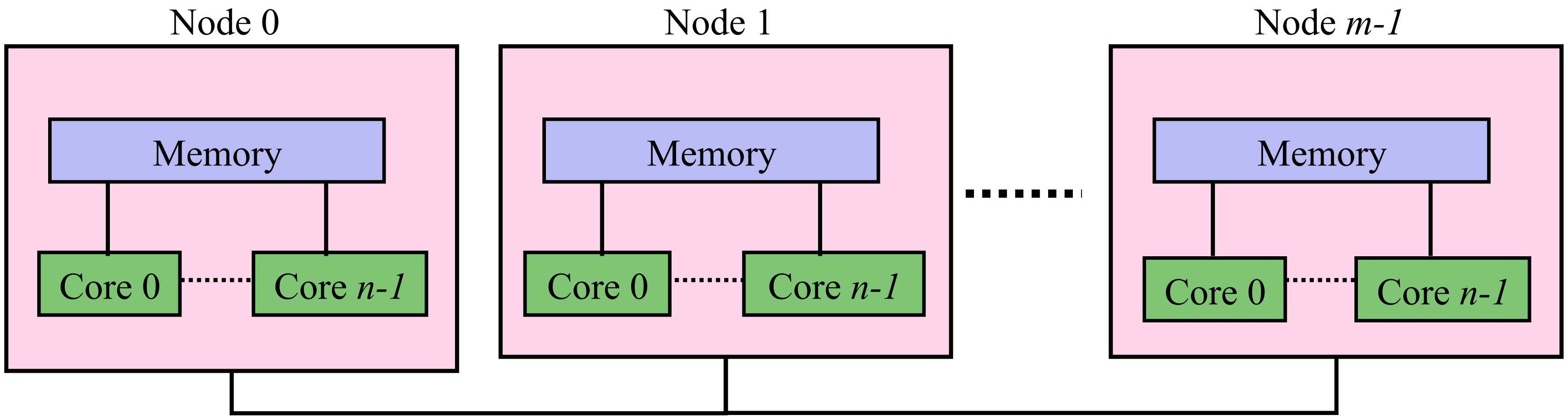
- Multithreading can improve some numerical kernels
  - E.g., domain decomposition preconditioning with incomplete factorizations
- For flat MPI, inflation in iteration count due to number of subdomains
- By introducing multithreaded triangular solves on each node
  - Solve triangular system on larger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

Bimodal MPI and MPI+Threads 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
    - Requires ubiquitous change when starting from “way 1”

Bimodal MPI and MPI+Threads 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)
  - 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

```
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

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

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
- 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

Work in Progress: PCG Algorithm

Mantevo miniapp: HPCPCG

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

Use multithreading for precondtioning

PCG Algorithm

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

Shared memory variables

PCG Algorithm – MPI part

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

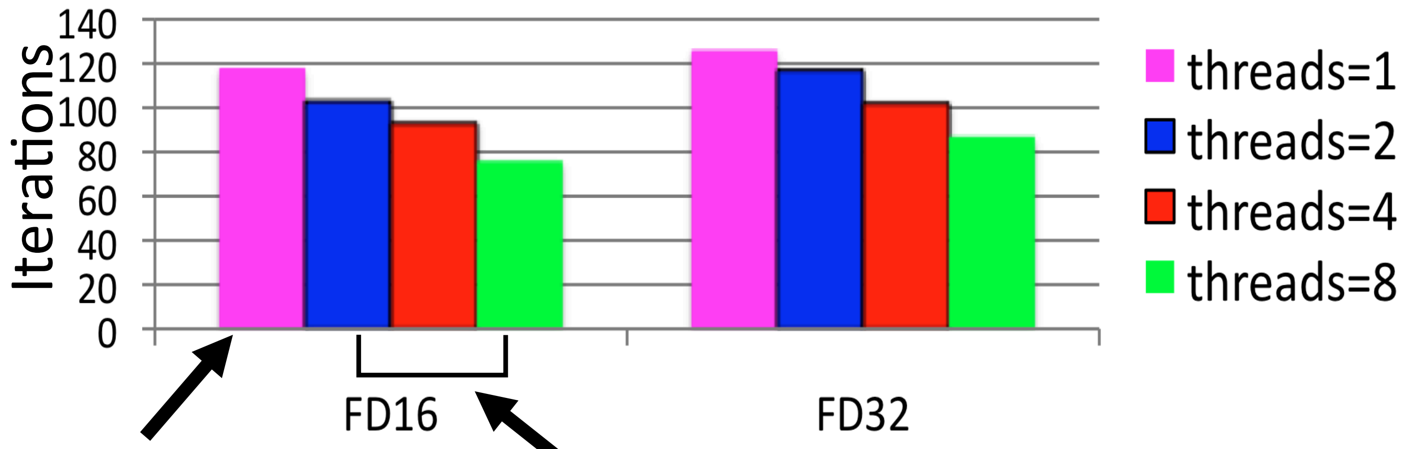
Flat MPI operations

PCG Algorithm – Threaded Part

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

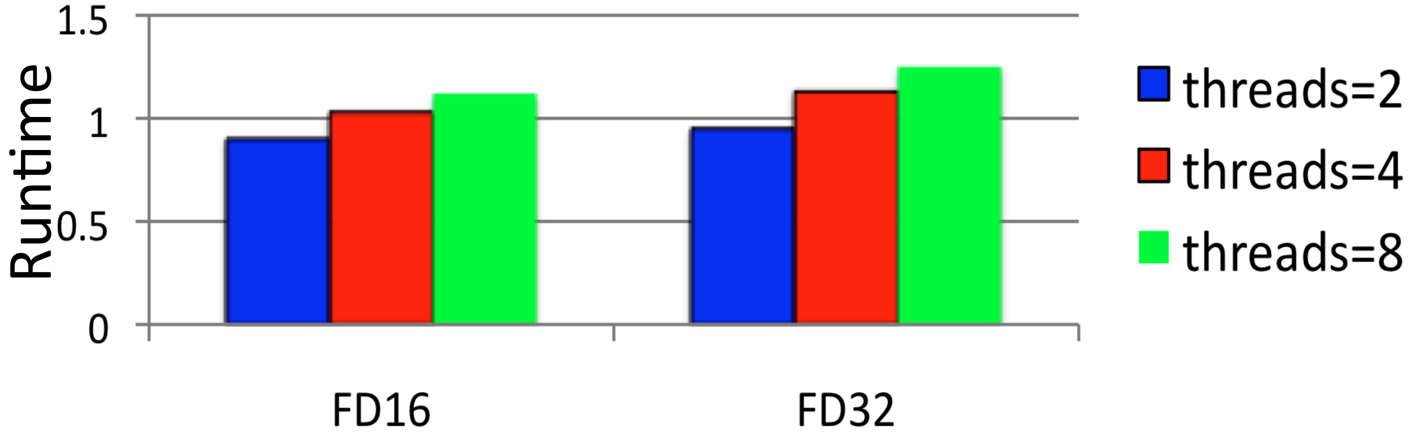
Multithreaded block preconditioning to reduce number of subdomains

Preliminary PCG Results



Flat MPI PCG

Threaded Preconditioning



Runtime relative to flat MPI PCG

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