



# **Building the Next Generation of Parallel Applications and Libraries**

Michael A. Heroux  
Scalable Algorithms Department  
Sandia National Laboratories

## Collaborators:

SNL Staff: [B.]R. Barrett, E. Boman, R. Brightwell, H.C. Edwards, A. Williams

SNL Postdocs: M. Hoemmen, S. Rajamanickam

MIT Lincoln Lab: M. Wolf

ORNL staff: Chris Baker

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





# Building Next Generation Applications & Libraries: 15 Strategies to Consider

---

1. Prepare for disruptive change
2. Design to the new scalability parameters: thread count and vector lengths
3. Encapsulate all parallelizable functionality into stateless (sequential) functions
4. Organize for vectorization
5. Decide on struct of arrays or array of structs
6. Prefer computation to data storage
7. Consider lower precision storage or computation, or both
8. Consider higher precision storage or computation, or both
9. Exploit data regularity
10. Separate physics indexing from storage indexing
11. Separate definition of array contents from filling of array data structures
12. Create library interfaces, even if you only call your own libraries
13. Look for untapped resources of parallelism
14. Make template meta-programming your friend
15. Build resilience into your software



# Three Parallel Computing Design Points

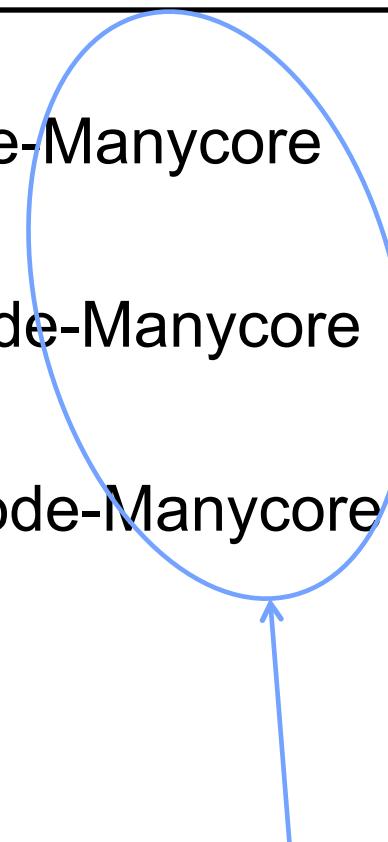
---

- Terascale Laptop:
- Petascale Deskside:
- Exascale Center:

Uninode-Manycore

Multinode-Manycore

Manynode-Manycore



---

# Emerging Architecture Programming Challenges: Overview

Herbert Stein, chairman of the Council of Economic Advisers under Nixon and Ford.

## Factoring 1K to 1B-Way Parallelism

---

- Why 1K to 1B?
  - Clock rate:  $O(1\text{GHz}) \rightarrow O(10^9)$  ops/sec sequential
  - Terascale:  $10^{12}$  ops/sec  $\rightarrow O(10^3)$  simultaneous ops
    - 1K parallel intra-node.
  - Petascale:  $10^{15}$  ops/sec  $\rightarrow O(10^6)$  simultaneous ops
    - 1K-10K parallel intra-node.
    - 100-1K parallel inter-node.
  - Exascale:  $10^{18}$  ops/sec  $\rightarrow O(10^9)$  simultaneous ops
    - 1K-10K parallel intra-node.
    - 100K-1M parallel inter-node.
- Current nodes:
  - SPARC64™ Vllfx: **128GF** (at 2.2GHz). “K” machine
  - NVIDIA Fermi: **500GF** (at 1.1GHz). Tianhe-1A.
  - Sunway BlueLight MPP:
    - 1PF with 8,700 ShenWei SW1600 proc  $\sim 115\text{GF/proc}$



# Data Movement: Locality

---

- Locality always important:
  - Caches: CPU
  - L1\$ vs L2\$ vs DRAM: Order of magnitude latency.
- Newer concern:
  - NUMA affinity.
  - Initial data placement important (unless FLOP rich).
  - Example:
    - 4-socket AMD with dual six-core per socket (48 cores).
    - BW of owner-compute: 120 GB/s.
    - BW of neighbor-compute: 30 GB/s.
- GPUs: Different but related concerns.



## Memory Size

---

- Current “healthy” memory/core:
  - 512 MB/core (e.g. MD computations).
  - 2 GB/core (e.g. Implicit CFD).
- Future:
  - 512 MB/core “luxurious”.



# Resilience

---

- Individual component reliability:
  - Tuned for “acceptable” failure rate.
- Aggregate reliability:
  - Function of all components not failing.
  - May decline.
- Size of data sets may limit usage of standard checkpoint/restart.



## Summary of Algorithms Challenge

---

- Realize node parallelism of O(1K-10K).
- Do so
  - Within a more complicated memory system and
  - With reduced relative memory capacity and
  - With decreasing reliability.



# New Trends and Responses

---

- Increasing data parallelism:
  - Design for vectorization and increasing vector lengths.
  - SIMD a bit more general, but fits under here.
- Increasing core count:
  - Expose task level parallelism.
  - Express task using DAG or similar constructs.
- Reduced memory size:
  - Express algorithms as multi-precision.
  - Compute data vs. store.
  - Data compression techniques.
- Memory architecture complexity:
  - Localize allocation/initialization.
  - Favor algorithms with higher compute/communication ratio.
- Resilience: Distinguish what must be reliably computed.



# Designing for Trends

---

- Long-term success must include design for change.
- Algorithms we develop today must adapt to future changes.
- Lesson from Distributed Memory (SPMD):
  - What was the trend? Increasing processor count.
  - Domain decomposition algs matched trend.
    - Design algorithm for  $p$  domains.
    - Design software for expanded modeling within a domain.



# Summary of Emerging Architecture Challenges & Responses

---

- Concurrency:
  - The new scalability parameter, includes vectorization.
  - Sample effort: **ShyLU**
- Data motion:
  - Don't do it, but if you must, don't do much.
  - Sample effort: **CA-GMRES**
- Data generation/use:
  - Generate and retain only that which is necessary.
  - Exploit underlying data patterns.
  - Example: **Functor-based data initialization**
- Resilience:
  - Formulate resilient algorithms.
  - Distinguish between high-reliability vs. bulk.
  - Example: **FTGMRES**.

# Prepare for Disruptive Change (And some ways to learn from the past)



# Lessons Learned from MPI and SPMD

---

- SPMD pattern (implemented via MPI) is unmitigated success.
- Approach for future applications:
  - Study why SPMD/MPI was successful.
  - Study new parallel landscape.
  - Try to cultivate a similar approach.

---

## *MPI Impresssions*

# MPI: It Hurts So Good

## • Observations

- “assembly language” of parallel
- lowest common denominator
  - portable across architectures
- upfront effort required
  - system
  - environment

## • Current Status

So What Would Life Be Like Without MPI?

$F(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ F(n-1) + F(n-2) & n>1 \end{cases}$

```
long fib_serial(long n)
{
    if (n < 2) return n;
    return fib_serial(n-1) + fib_serial(n-2);
}
```

```
long fib_parallel(long n)
{
    long x, y;
    if (n < 2) return n;
    #pragmaomp task default(shared,x,n)
    { x = fib_parallel(n-1);
    y = fib_parallel(n-2);
    #pragmaomp taskwait
    return (x+y);
}
```

Tim Stitts, CSCS  
SOS14 Talk  
March 2010

Dan Reed, Microsoft

Workshop on the Road Map for the  
Revitalization of High End  
Computing  
June 16-18, 2003

Peter S. Pachence

Looking Forward to a New Age of Large-Scale Parallel Programming and the Demise of MPI  
...hopes and dreams of an HPC educator



“ MPI is often considered the “portable assembly language” of parallel computing, ...”

Brad Chamberlain, Cray, 2000.



# 3D Stencil in NAS MG

HPCS

```

subroutine comm3(n1,n2,n3,kk)
use caf_intrinsic
implicit none
double precision u(n1,n2,n3)
include 'cafnpb.h'
include 'globalis.h'

integer axis, dir, n1, n2, n3, k, ierr
double precision u(n1, n2, n3)

integer i3, i2, i1, buff_len,buff_id
buff_id = 2 * dir
buff_len = 0

if( axis .eq. 1 ) then
  if( dir .eq. -1 ) then
    do i3=2,n3-1
      do i2=1,n1
        buff_len = buff_len + 1
        buff(buff_len, buff_id) = u( i1, i2,i3 )
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  else if( dir .eq. +1 ) then
    do i3=2,n3-1
      do i2=2,n2-1
        buff_len = buff_len + 1
        buff(buff_len, buff_id) = u( i1, i2,i3 )
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  endif
endif

do axis = 1, 3
  call sync_all()
enddo
call zero3(u,n1,n2,n3)
endif
return
end

subroutine give3( axis, dir, u, n1, n2, n3, k )
use caf_intrinsics
implicit none
double precision u(n1, n2, n3)

integer axis, dir, n1, n2, n3, k, ierr
double precision u(n1, n2, n3)

integer i3, i2, i1, buff_len,buff_id
buff_id = 2 * dir
buff_len = 0

if( axis .eq. 1 ) then
  if( dir .eq. -1 ) then
    do i3=2,n3-1
      do i2=1,n2-1
        buff_len = buff_len + 1
        buff(buff_len, buff_id) = u( i1, i2,i3 )
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  else if( dir .eq. +1 ) then
    do i3=2,n3-1
      do i2=2,n2-1
        buff_len = buff_len + 1
        buff(buff_len, buff_id) = u( i1, i2,i3 )
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  endif
endif

if( axis .eq. 2 ) then
  if( dir .eq. -1 ) then
    do i3=2,n3-1
      do i2=1,n1
        buff_len = buff_len + 1
        buff(buff_len, buff_id) = u( i1, i2,i3 )
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  else if( dir .eq. +1 ) then
    do i3=2,n3-1
      do i2=1,n1
        buff_len = buff_len + 1
        buff(buff_len, buff_id) = u( i1, i2,i3 )
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  endif
endif

if( axis .eq. 3 ) then
  if( dir .eq. -1 ) then
    do i3=1,n2
      do i2=1,n1
        buff_len = buff_len + 1
        buff(buff_len, buff_id) = u( i1, i2,i3 )
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  else if( dir .eq. +1 ) then
    do i3=1,n2
      do i2=1,n1
        buff_len = buff_len + 1
        buff(buff_len, buff_id) = u( i1, i2,i3 )
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  endif
endif
endif

```

```

      u(n1,i2,i3) = buff(indx, buff_id)
    enddo
  endif

  else if( dir .eq. +1 ) then
    do i3=2,n3-1
      do i2=2,n2-1
        indx = indx + 1
        u(i1,i2,i3) = buff(indx, buff_id)
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  endif

  else if( dir .eq. -1 ) then
    do i3=2,n3-1
      do i2=1,n1
        indx = indx + 1
        u(i1,i2,i3) = buff(indx, buff_id)
      enddo
    enddo

    buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
    > buff(1:buff_len,buff_id)
  endif

  else if( dir .eq. 2 ) then
    if( dir .eq. -1 ) then
      do i3=2,n3-1
        do i2=1,n1
          indx = indx + 1
          u(i1,i2,i3) = buff(indx, buff_id)
        enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
      > buff(1:buff_len,buff_id)
    else if( dir .eq. +1 ) then
      do i3=2,n3-1
        do i2=1,n1
          indx = indx + 1
          u(i1,i2,i3) = buff(indx, buff_id)
        enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
      > buff(1:buff_len,buff_id)
    endif
endif

  else if( dir .eq. 3 ) then
    if( dir .eq. -1 ) then
      do i3=1,n2
        do i2=1,n1
          indx = indx + 1
          u(i1,i2,i3) = buff(indx, buff_id)
        enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
      > buff(1:buff_len,buff_id)
    else if( dir .eq. +1 ) then
      do i3=1,n2
        do i2=1,n1
          indx = indx + 1
          u(i1,i2,i3) = buff(indx, buff_id)
        enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
      > buff(1:buff_len,buff_id)
    endif
endif
endif

```

```

do i=1,nm2
  buff(i,buff_id) = 0.0D0
enddo

dir = +1

buff_id = 3 + dir
buff_len = nm2

do i=1,nm2
  buff(i,buff_id) = 0.0D0
enddo

dir = +1

buff_id = 2 + dir
buff_len = 0

if( axis .eq. 1 ) then
  do i3=2,n3-1
    do i2=1,n1
      indx = indx + 1
      u(i1,i2,i3) = buff(indx, buff_id)
    enddo
  enddo

  if( axis .eq. 2 ) then
    do i3=2,n3-1
      do i2=1,n1
        indx = indx + 1
        u(i1,i2,i3) = buff(indx, buff_id)
      enddo
    enddo

    if( axis .eq. 3 ) then
      do i3=2,n3-1
        do i2=1,n1
          indx = indx + 1
          u(i1,i2,i3) = buff(indx, buff_id)
        enddo
      enddo
    endif
  endif
endif

if( axis .eq. 2 ) then
  do i3=2,n3-1
    do i2=1,n1
      indx = indx + 1
      u(i1,i2,i3) = buff(indx, buff_id)
    enddo
  enddo

  if( axis .eq. 3 ) then
    do i3=2,n3-1
      do i2=1,n1
        indx = indx + 1
        u(i1,i2,i3) = buff(indx, buff_id)
      enddo
    enddo
  endif
endif

if( axis .eq. 3 ) then
  do i3=2,n3-1
    do i2=1,n1
      indx = indx + 1
      u(i1,i2,i3) = buff(indx, buff_id)
    enddo
  enddo
endif

dir = +1

buff_id = 3 + dir
indx = 0

if( axis .eq. 1 ) then
  do i3=2,n3-1
    do i2=1,n2-1
      do i1=1,n1
        indx = indx + 1
        u(i1,i2,i3) = buff(indx, buff_id)
      enddo
    enddo
  enddo

  if( axis .eq. 2 ) then
    do i3=2,n3-1
      do i2=1,n2-1
        do i1=1,n1
          indx = indx + 1
          u(i1,i2,i3) = buff(indx, buff_id)
        enddo
      enddo
    enddo

    if( axis .eq. 3 ) then
      do i3=2,n3-1
        do i2=1,n2-1
          do i1=1,n1
            indx = indx + 1
            u(i1,i2,i3) = buff(indx, buff_id)
          enddo
        enddo
      enddo
    endif
  endif
endif

if( axis .eq. 2 ) then
  do i3=2,n3-1
    do i2=1,n2-1
      do i1=1,n1
        indx = indx + 1
        u(i1,i2,i3) = buff(indx, buff_id)
      enddo
    enddo
  enddo

  if( axis .eq. 3 ) then
    do i3=2,n3-1
      do i2=1,n2-1
        do i1=1,n1
          indx = indx + 1
          u(i1,i2,i3) = buff(indx, buff_id)
        enddo
      enddo
    enddo
  endif
endif

if( axis .eq. 3 ) then
  do i3=2,n3-1
    do i2=1,n2-1
      do i1=1,n1
        indx = indx + 1
        u(i1,i2,i3) = buff(indx, buff_id)
      enddo
    enddo
  enddo
endif

```

---

## *MPI Reality*

## dft\_fill\_wjdc.c

```
/* (C) 2007 W.G. Chapman, A. Dominik, S. Jain, and J. Werthim
   This code is part of the WJDC-DFT package. It is distributed under the
   terms of the GNU General Public License (GPL). See the file LICENSE
   for more information. */

#include "dft.h"
#include "dft_wjdc.h"

void dft_fill_wjdc(dft_t *dft, const dft_wjdc_t *wjdc)
{
    /* ... */
}
```

```
/* (C) 2007 W.G. Chapman, A. Dominik, S. Jain, and J. Werthim
   This code is part of the WJDC-DFT package. It is distributed under the
   terms of the GNU General Public License (GPL). See the file LICENSE
   for more information. */

#include "dft.h"
#include "dft_wjdc.h"

void dft_fill_wjdc(dft_t *dft, const dft_wjdc_t *wjdc)
{
    /* ... */
}
```

```
/* (C) 2007 W.G. Chapman, A. Dominik, S. Jain, and J. Werthim
   This code is part of the WJDC-DFT package. It is distributed under the
   terms of the GNU General Public License (GPL). See the file LICENSE
   for more information. */

#include "dft.h"
#include "dft_wjdc.h"

void dft_fill_wjdc(dft_t *dft, const dft_wjdc_t *wjdc)
{
    /* ... */
}
```

```
/* (C) 2007 W.G. Chapman, A. Dominik, S. Jain, and J. Werthim
   This code is part of the WJDC-DFT package. It is distributed under the
   terms of the GNU General Public License (GPL). See the file LICENSE
   for more information. */

#include "dft.h"
#include "dft_wjdc.h"

void dft_fill_wjdc(dft_t *dft, const dft_wjdc_t *wjdc)
{
    /* ... */
}
```

# Tramonto WJDC Functional

- New functional.
- Bonded systems.
- 552 lines C code.

WJDC-DFT (Werthim, Jain, Dominik, and Chapman) theory for bonded systems. (S. Jain, A. Dominik, and W.G. Chapman. *Modified interfacial statistical associating fluid theory: A perturbation density functional theory for inhomogeneous complex fluids.* *J. Chem. Phys.*, 127:244904, 2007.) Models stoichiometry constraints inherent to bonded systems.

How much MPI-specific code?

dft\_fill\_wjdc.c  
MPI-specific  
code

# MFIX

## Source term for pressure correction

```
source_pp_g.f

       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
       !-----*
```

```
      END IF
      ! CHIM & SAT end (run wie)
      ! Calculate convection-diffusion fluxes through each of the faces
      !-----*
      !-----*
      !-----*
      !-----*
      !-----*
      !-----*
      !-----*
```

```
      !-----*
      !-----*
      !-----*
      !-----*
      !-----*
      !-----*
      !-----*
      !-----*
```

- MPI-callable, OpenMP-enabled.
- 340 Fortran lines.
- No MPI-specific code.
- Ubiquitous OpenMP markup (red regions).



# Reasons for MPI Success?

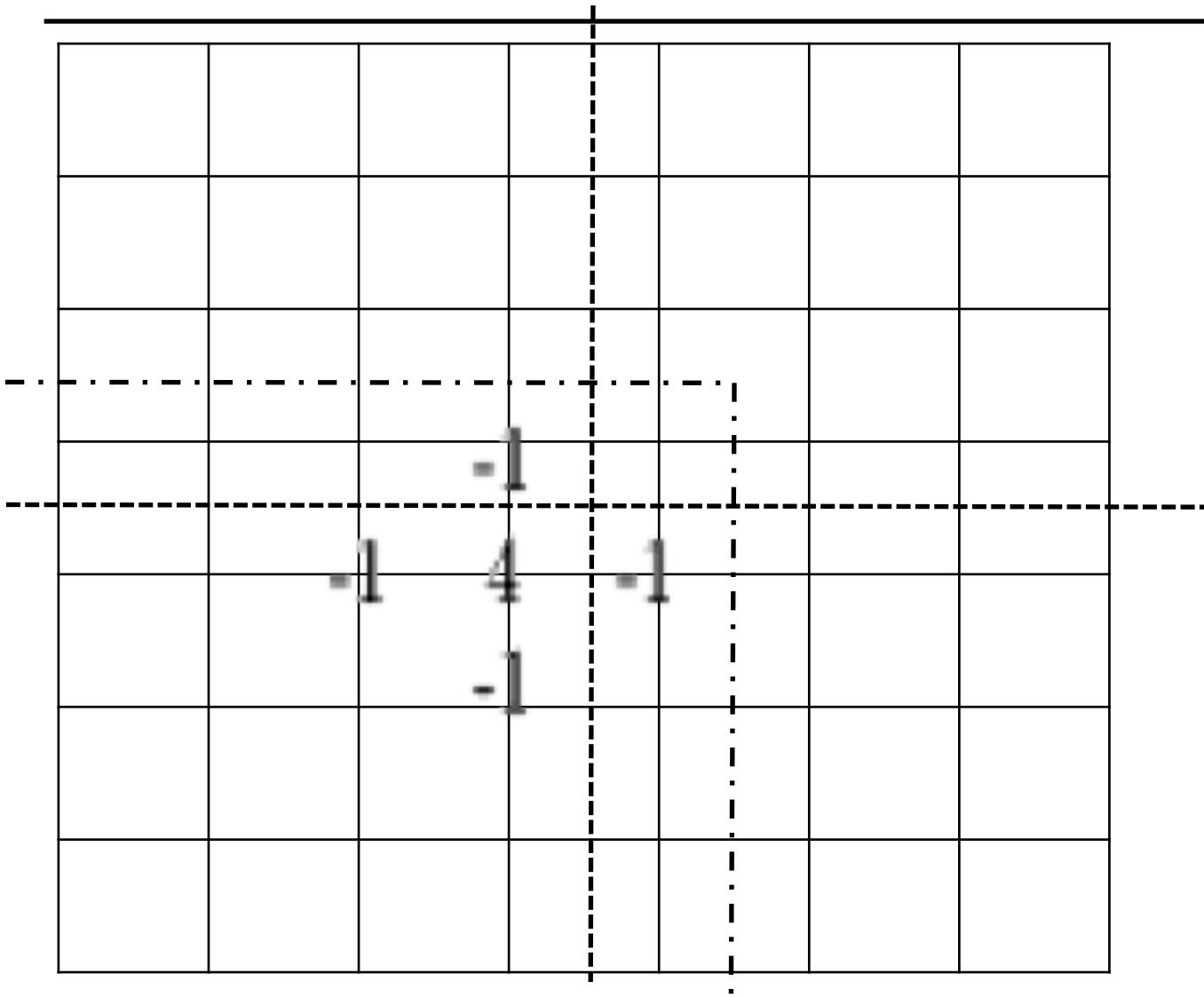
---

- Portability? Yes.
  - Standardized? Yes.
  - Momentum? Yes.
  - Separation of many Parallel & Algorithms concerns? Big Yes.
- 
- Once framework in place:
    - Sophisticated physics added as sequential code.
    - Ratio of science experts vs. parallel experts: 10:1.
  - Key goal for new parallel apps: Preserve this ratio

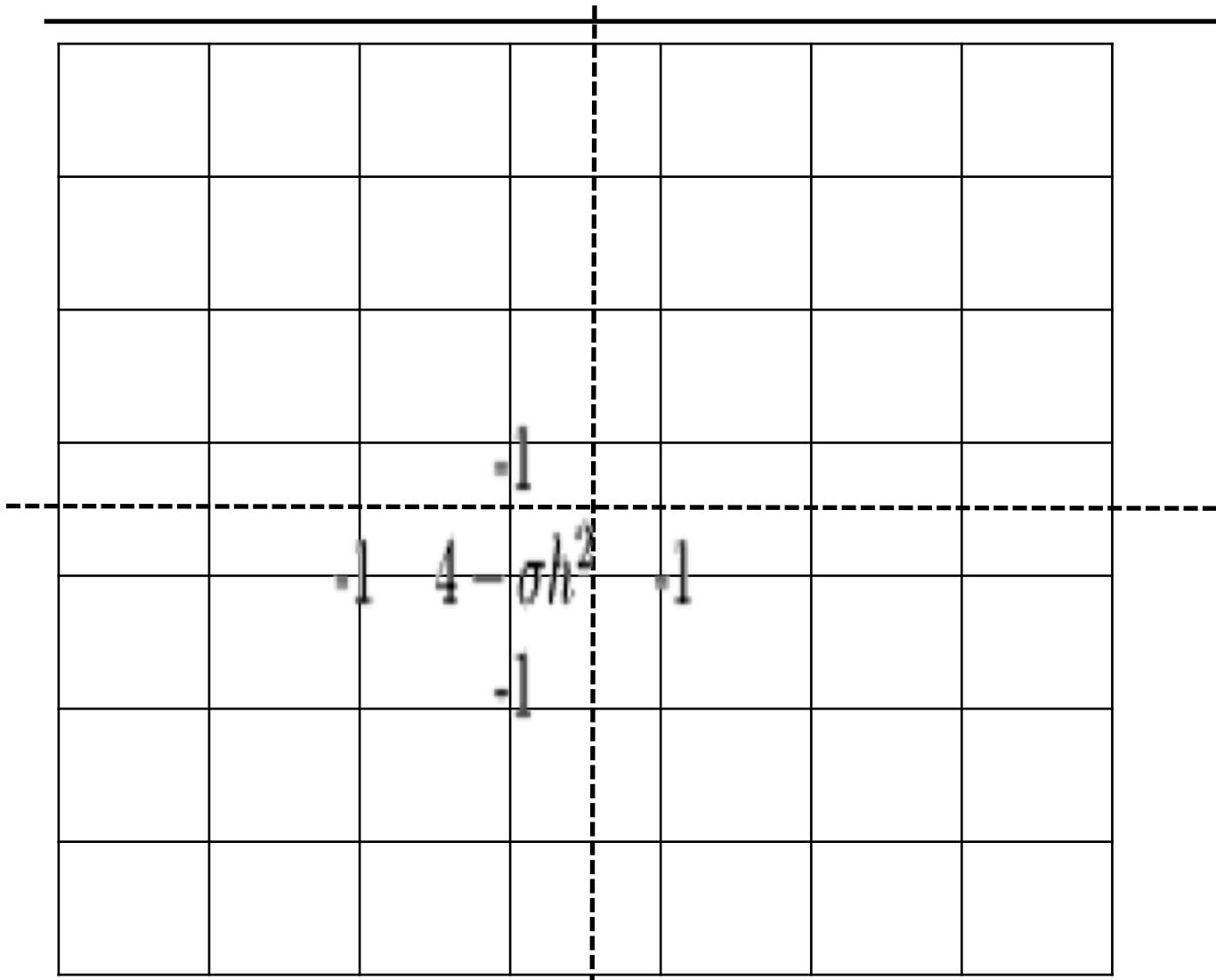
---

# *Single Program Multiple Data (SPMD) 101*

# 2D PDE on Regular Grid (Standard Laplace)

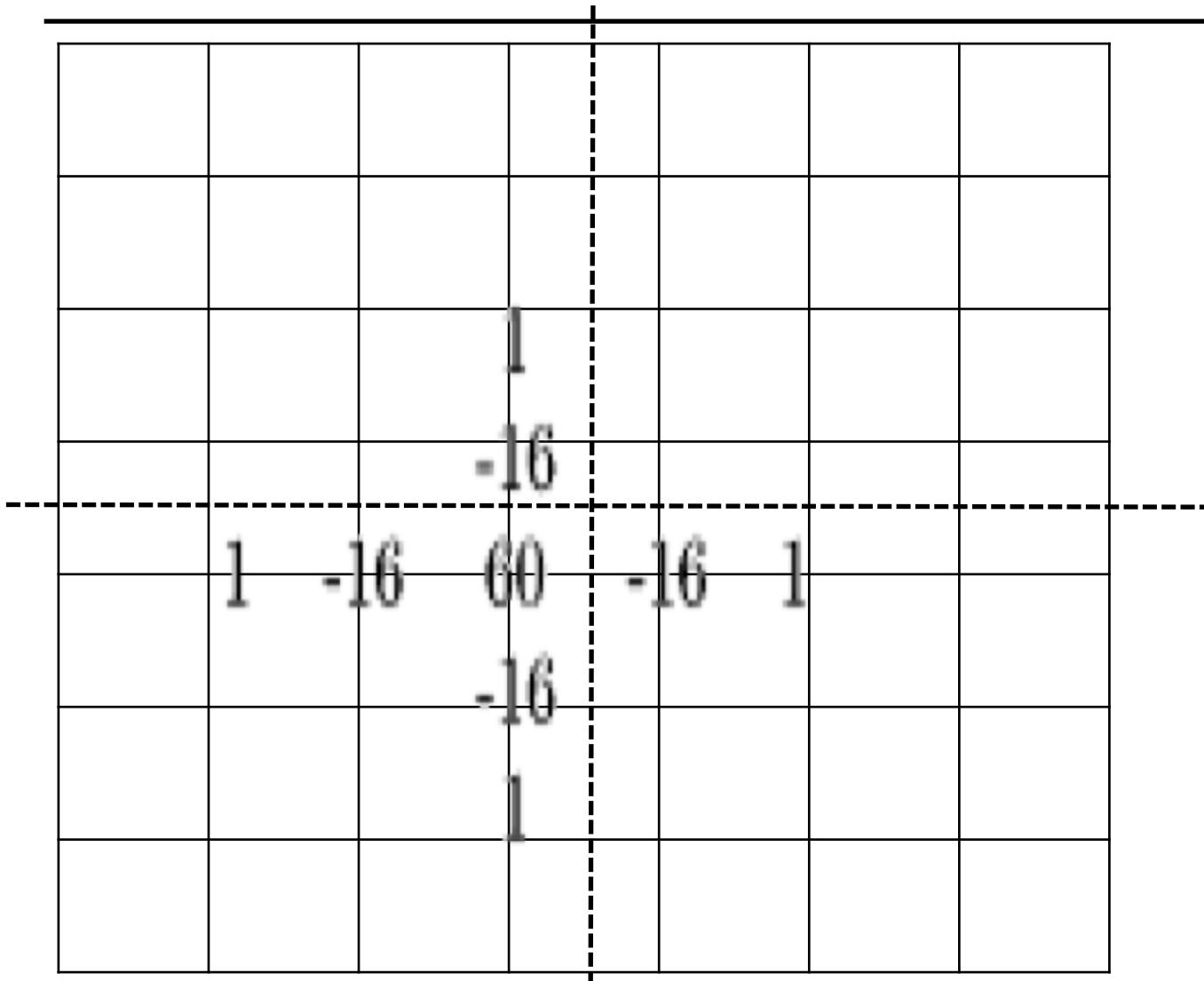


# 2D PDE on Regular Grid (Helmholtz)



$$-\nabla u - \sigma u = f \quad (\sigma \geq 0)$$

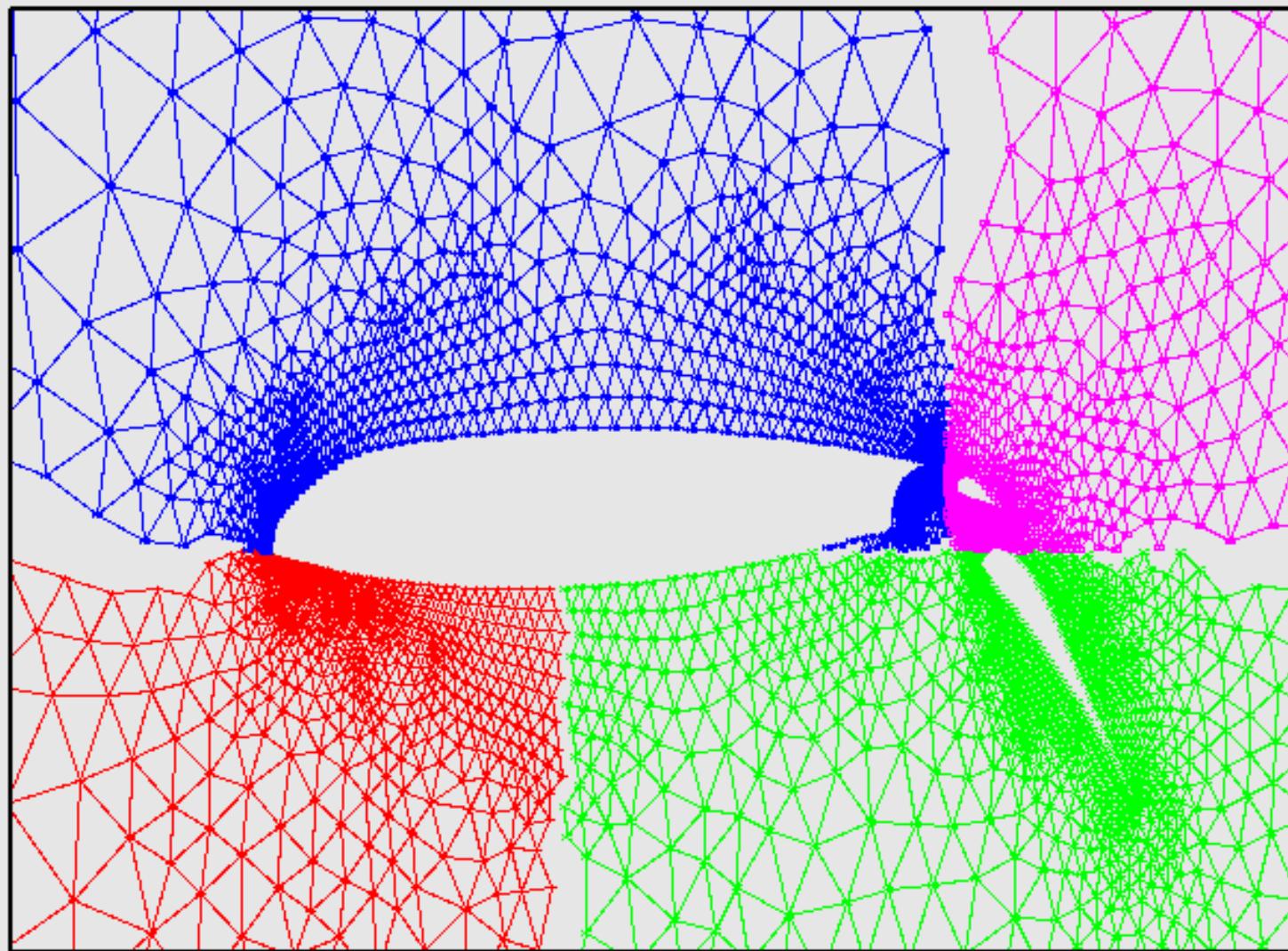
# 2D PDE on Regular Grid (4<sup>th</sup> Order Laplace)





# More General Mesh and Partitioning

---





# SPMD Patterns for Domain Decomposition

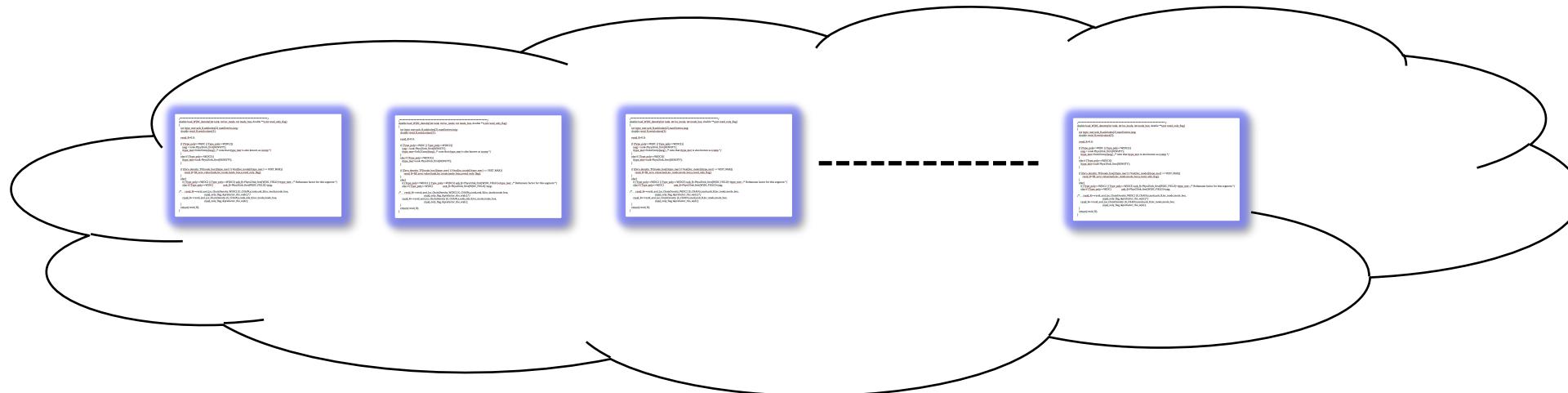
---

- Halo Exchange:
  - Conceptual.
  - Needed for any partitioning, halo layers.
  - MPI is simply portability layer.
  - Could be replace by PGAS, one-sided, ...
- Collectives:
  - Dot products, norms.
- All other programming:
  - Sequential!!!



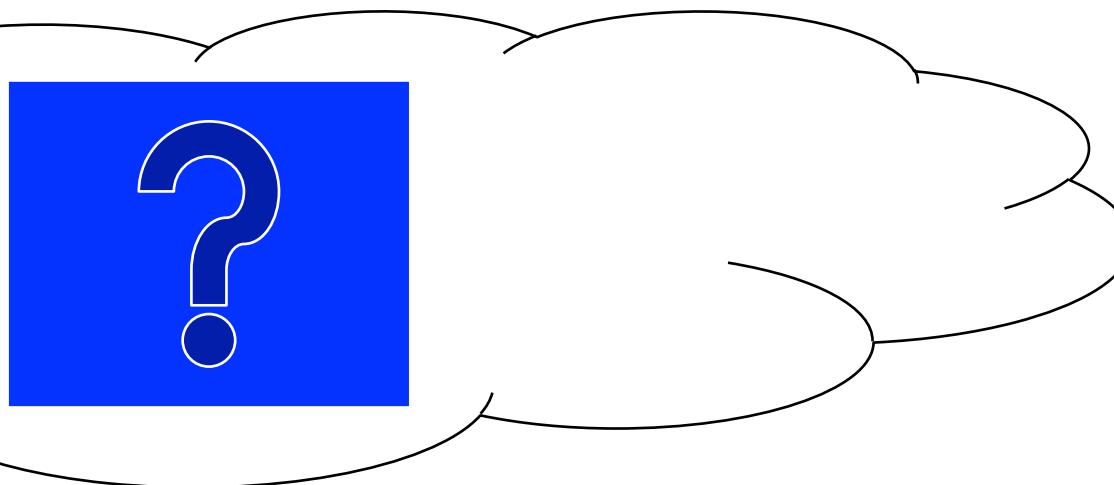
# Computational Domain Expert Writing MPI Code

---



# Computational Domain Expert Writing Future Parallel Code

---



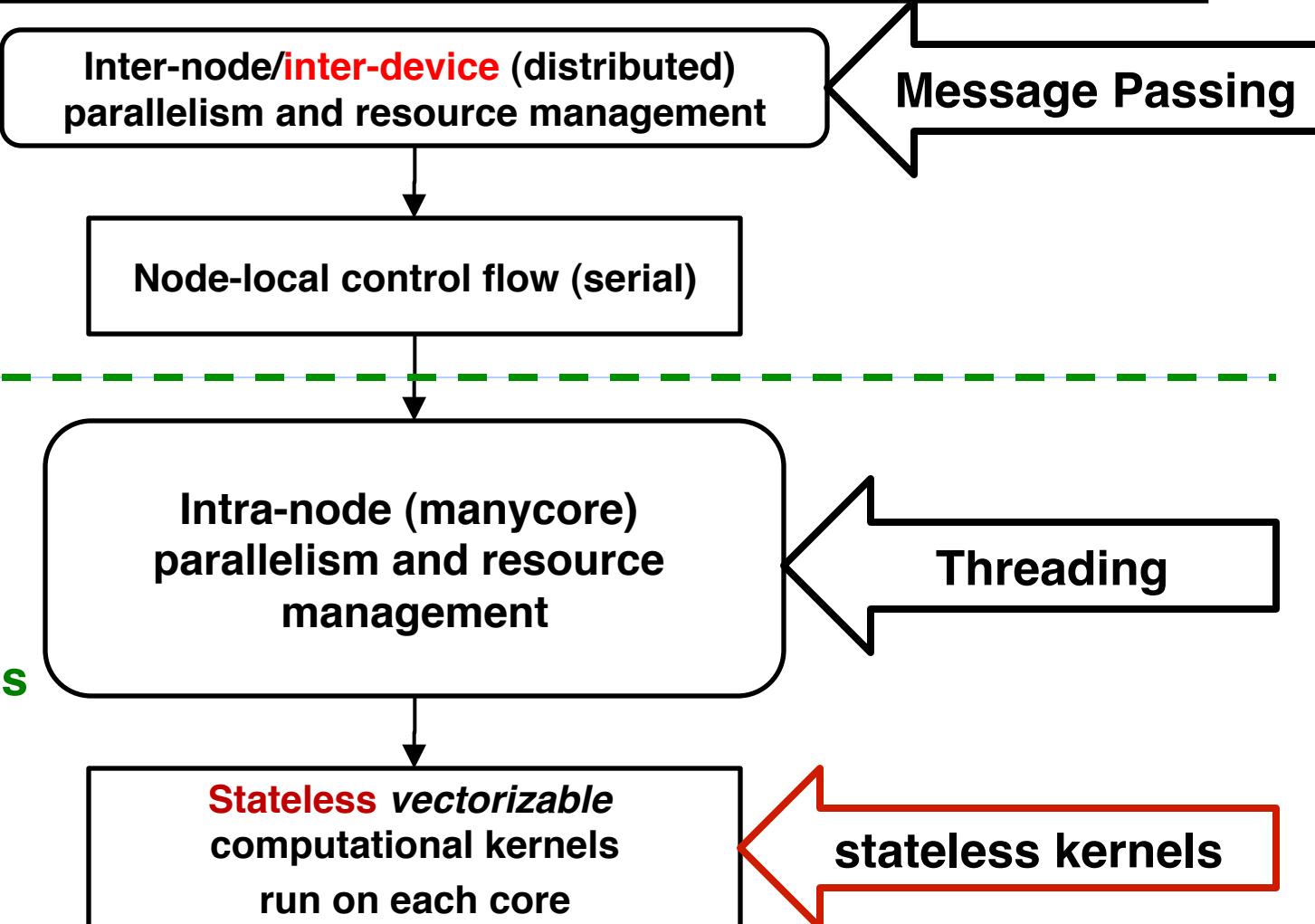
---

## *Evolving Parallel Programming Model*

# Parallel Programming Model: Multi-level/Multi-device

network of  
computational  
nodes

computational  
node with  
manycore CPUs  
and / or  
GPGPU



Adapted from slide of H. Carter Edwards



# Domain Scientist's Parallel Palette

---

- Today: MPI-only (SPMD) apps
  - Single parallel construct.
  - Simultaneous execution.
  - Parallelism of even the messiest serial code.
- Next-generation PDE and related applications:
  - Internode:
    - MPI, yes, or something like it.
    - Composed with intranode.
  - Intranode:
    - Much richer palette.
    - More care required from programmer.
- What are the constructs in our new palette?



# Obvious Constructs/Concerns

---

- Parallel for:

```
forall (i, j) in domain {...}
```

- No loop-carried dependence.
- Rich loops.
- Use of shared memory for temporal reuse, efficient device data transfers.

- Parallel reduce:

```
forall (i, j) in domain {  
    xnew(i, j) = ...;  
    delx+= abs(xnew(i, j) - xold(i, j));
```

```
}
```

- Couple with other computations.
- Concern for reproducibility.

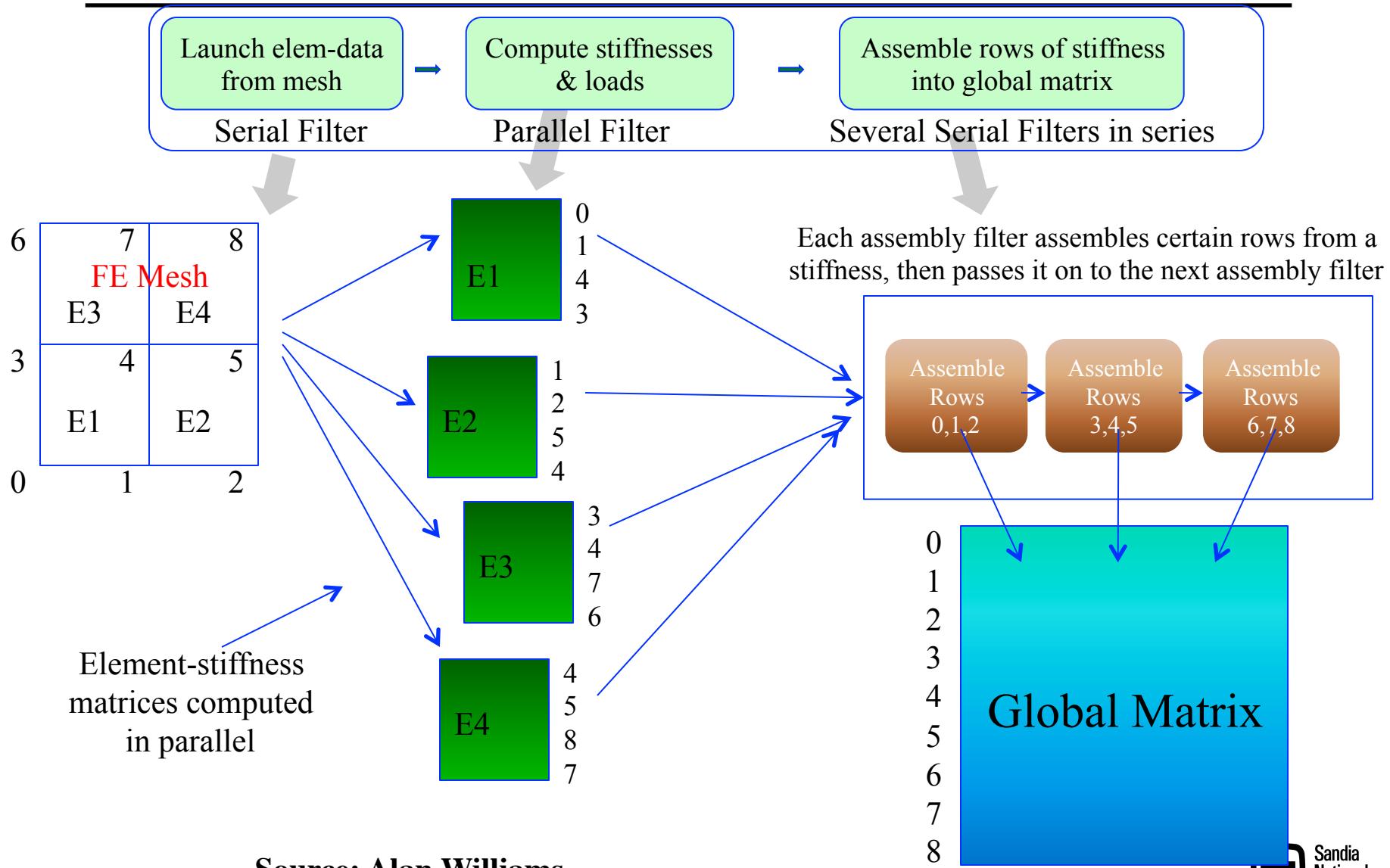


## Other construct: Pipeline

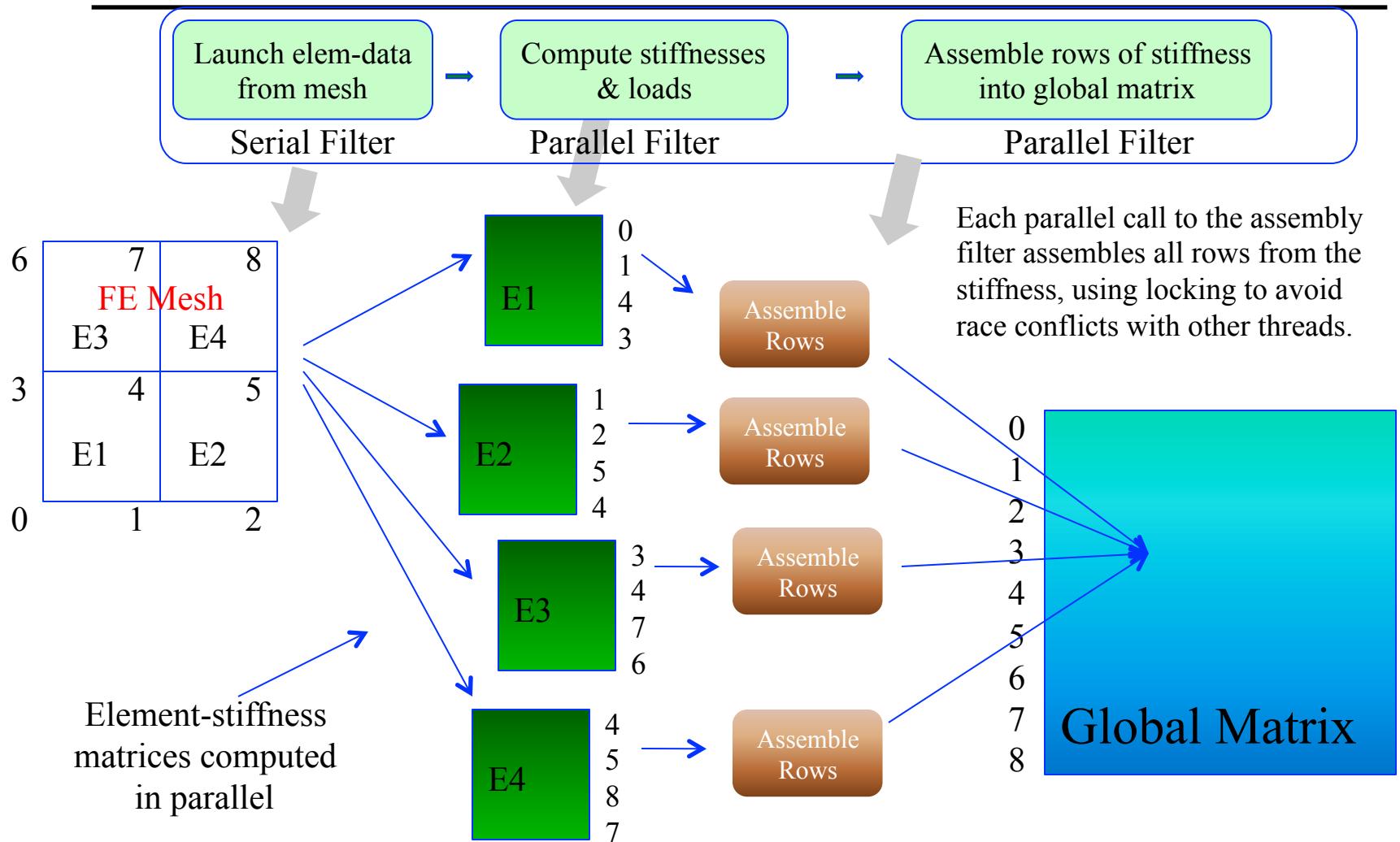
---

- Sequence of filters.
- Each filter is:
  - Sequential (grab element ID, enter global assembly) or
  - Parallel (fill element stiffness matrix).
- Filters executed in sequence.
- Programmer's concern:
  - Determine (conceptually): Can filter execute in parallel?
  - Write filter (serial code).
  - Register it with the pipeline.
- Extensible:
  - New physics feature.
  - New filter added to pipeline.

# TBB Pipeline for FE assembly



# Alternative TBB Pipeline for FE assembly





# Base-line FE Assembly Timings

---

Problem size:  $80 \times 80 \times 80 == 512000$  elements, 531441 matrix-rows  
The finite-element assembly performs 4096000 matrix-row sum-into operations  
(8 per element) and 4096000 vector-entry sum-into operations.

MPI-only, no threads. Linux dual quad-core workstation.

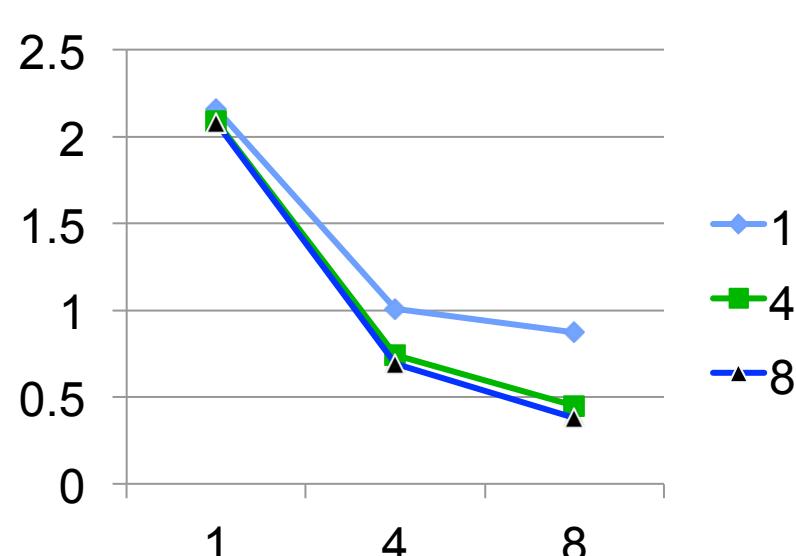
Num-procs	Assembly-time Intel 11.1	Assembly-time GCC 4.4.4
1	1.80s	1.95s
4	0.45s	0.50s
8	0.24s	0.28s

# FE Assembly Timings

Problem size:  $80 \times 80 \times 80 == 512000$  elements, 531441 matrix-rows

The finite-element assembly performs 4096000 matrix-row sum-into operations (8 per element) and 4096000 vector-entry sum-into operations.

No MPI, only threads. Linux dual quad-core workstation.



Num-threads	Elem-group-size	Matrix-conflicts	Vector-conflicts	Assembly-time
1	1	0	0	2.16s
1	4	0	0	2.09s
1	8	0	0	2.08s
4	1	95917	959	1.01s
4	4	7938	25	0.74s
4	8	3180	4	0.69s
8	1	64536	1306	0.87s
8	4	5892	49	0.45s
8	8	1618	1	0.38s

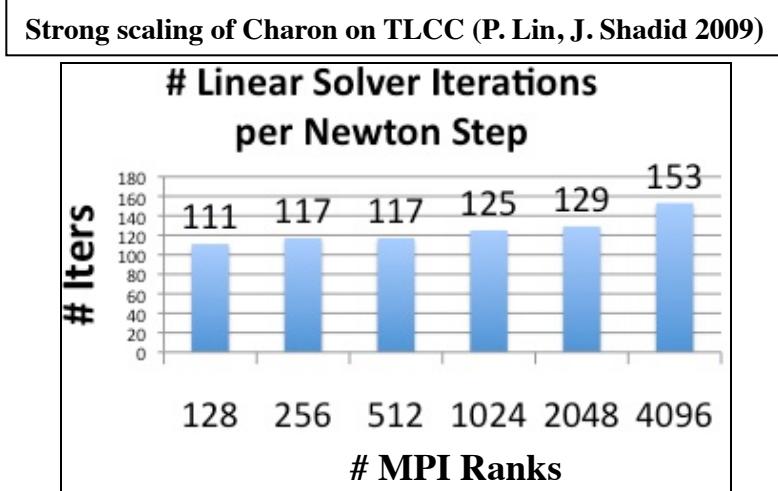
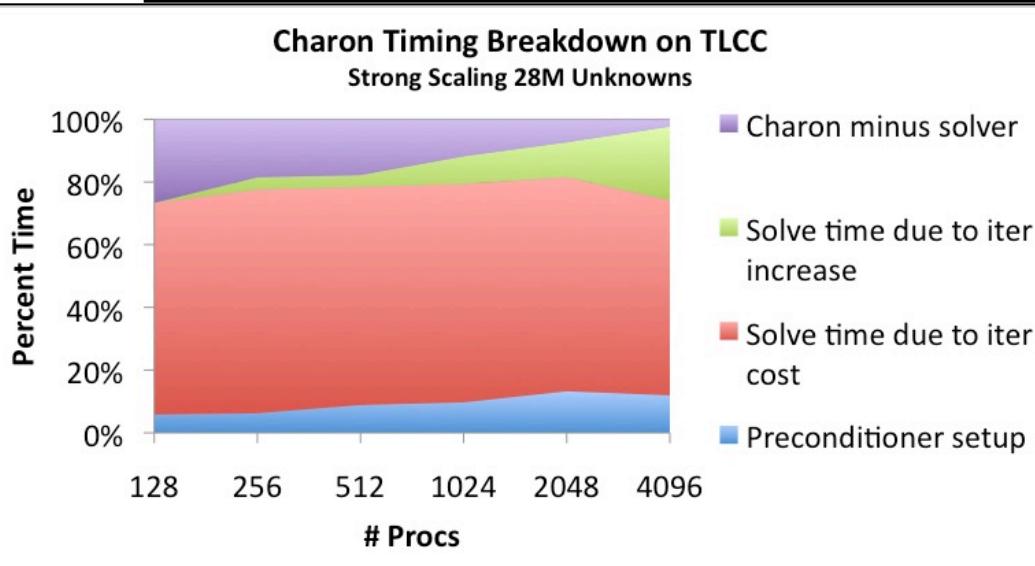


## Other construct: Thread team

---

- Multiple threads.
- Fast barrier.
- Shared, fast access memory pool.
- Example: Nvidia SM
- X86 more vague, emerging more clearly in future.

# Preconditioners for Scalable Multicore Systems



- Observe: Iteration count increases with number of subdomains.
- With scalable threaded smoothers (LU, ILU, Gauss-Seidel):
  - Solve with fewer, larger subdomains.
  - Better kernel scaling (threads vs. MPI processes).
  - Better convergence, More robust.
- Exascale Potential: Tiled, pipelined implementation.
- **Three efforts:**
  - Level-scheduled triangular sweeps (ILU solve, Gauss-Seidel).
  - **Decomposition by partitioning**
  - Multithreaded direct **factorization**

MPI Tasks	Threads	Iterations
4096	1	153
2048	2	129
1024	4	125
512	8	117
256	16	117
128	32	111



# Thread Team Advantages

---

- Qualitatively better algorithm:
  - Threaded triangular solve scales.
  - Fewer MPI ranks means fewer iterations, better robustness.
- Exploits:
  - Shared data.
  - Fast barrier.
  - Data-driven parallelism.



# Summary: FE/FV/FD and parallel node patterns

---

- Parallel for, reduce, pipeline:
  - Sufficient for vast majority of node level computation.
  - Supports:
    - Complex modeling expression.
    - Vanilla parallelism.
  - Must be “stencil-aware” for temporal locality.
- Thread team:
  - Complicated.
  - Requires deeper parallel algorithm knowledge.
  - Useful in solvers.



# Patterns & Frameworks in Other Contexts

---

- MapReduce:
  - Plug-n-Play data processing framework - 80% Google cycles.
- Pregel: Graph framework (other 20%).
- Ratio of domain to parallel computing experts: 1000s:1.

---

# *Programming Today for Tomorrow's Machines*



# 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?



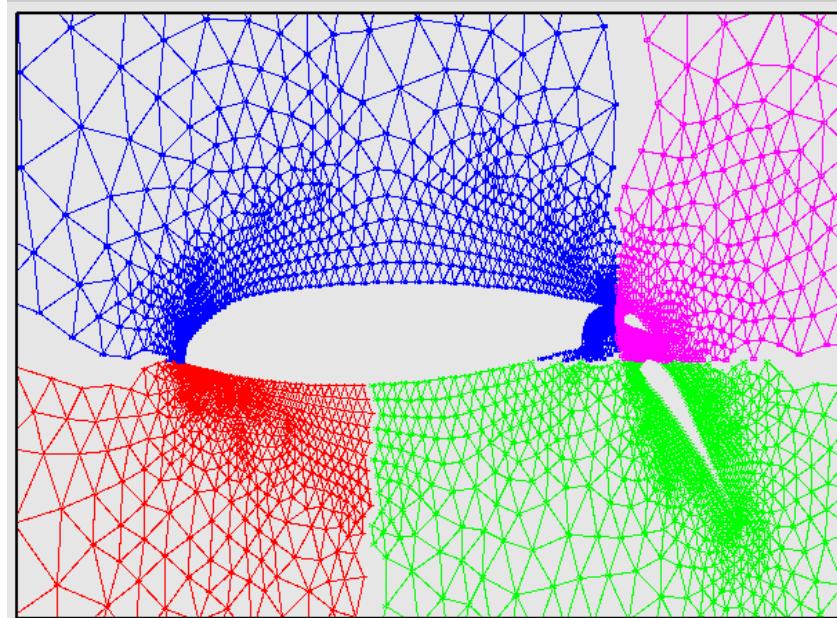
# FE/FV/FD Parallel Programming Today

---

```
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.

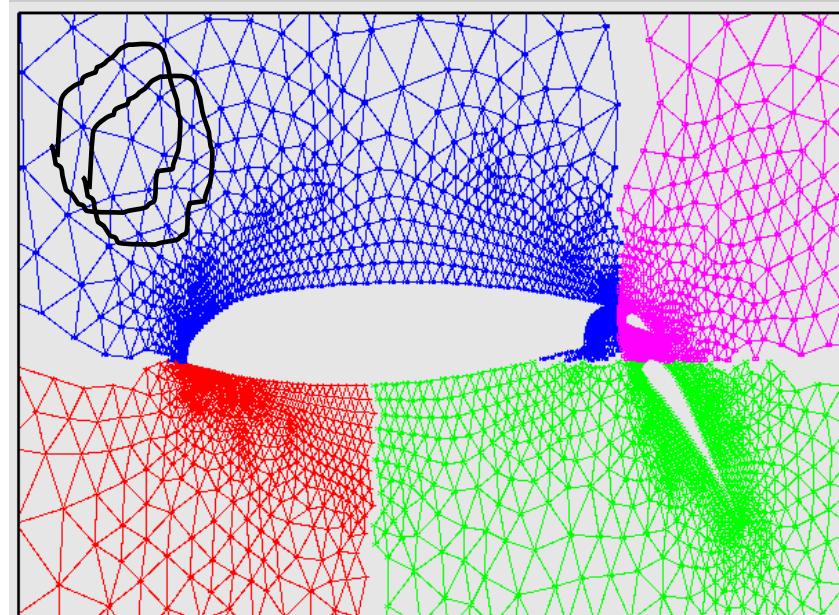


# FE/FV/FD Parallel Programming Tomorrow

```
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).





# Preparing for Disruptive Change

---

- Getting onto the new commodity curves requires ubiquitous change:
  - Scalability in thread count, vectorization.
  - Design for sequential physics expression.
- Not unlike move from serial/vector to MPI:
  - ID parallel patterns.
  - Build scalable framework with minimal physics.
  - Mine previous application for functionality.
- Don't be tempted by incremental approaches:
  - Quick payoff now, but ultimately limiting.

---

# Portable Multi/Manycore Programming Trilinos/Kokkos Node API



# Generic Node Parallel Programming via C++ Template Metaprogramming

---

- Goal: Don't repeat yourself (DRY).
- Every parallel programming environment supports basic patterns: `parallel_for`, `parallel_reduce`.
  - OpenMP:

```
#pragma omp parallel for
for (i=0; i<n; ++i) {y[i] += alpha*x[i];}
```
  - Intel TBB:

```
parallel_for(blocked_range<int>(0, n, 100), loopRangeFn(...));
```
  - CUDA:

```
loopBodyFn<<< nBlocks, blockSize >>> (...);
```
- How can we write code once for all these (and future) environments?

# Tpetra and Kokkos

- **Tpetra** is an implementation of the Petra Object Model.
  - Design is similar to Epetra, with appropriate deviation.
  - Fundamental differences:
    - heavily exploits templates
    - utilizes hybrid (distributed + **shared**) parallelism via Kokkos Node API
- **Kokkos** is an API for shared-memory parallel nodes
  - Provides parallel\_for and parallel\_reduce skeletons.
  - Support shared memory APIs:
    - ThreadPool Interface (TPI; Carter Edwards's pthreads Trilinos package)
    - Intel Threading Building Blocks (TBB)
    - NVIDIA CUDA-capable GPUs (via Thrust)
    - *OpenMP (implemented by Radu Popescu/EPFL)*



# Generic Shared Memory Node

---

- Abstract inter-node comm provides DMP support.
- Need some way to **portably** handle SMP support.
- Goal: allow code, once written, to be run on **any parallel node**, regardless of architecture.
- **Difficulty #1:** Many different **memory architectures**
  - Node may have multiple, disjoint memory spaces.
  - Optimal performance may require special memory placement.
- **Difficulty #2:** **Kernels** must be tailored to architecture
  - Implementation of optimal kernel will vary between archs
  - No universal binary → need for separate compilation paths
- Practical goal: Cover 80% kernels with generic code.



# Kokkos Node API

---

- Kokkos provides two main components:
  - Kokkos memory model addresses Difficulty #1
    - Allocation, deallocation and efficient access of memory
    - compute buffer: special memory used for parallel computation
    - New: Local Store Pointer and Buffer with size.
  - Kokkos compute model addresses Difficulty #2
    - Description of kernels for parallel execution on a node
    - Provides stubs for common parallel work constructs
    - Currently, parallel for loop and parallel reduce
- Code is developed around a polymorphic Node object.
- Supporting a new platform requires only the implementation of a new node type.



# Kokkos Memory Model

---

- A generic node model must at least:
  - support the scenario involving **distinct device memory**
  - allow **efficient** memory access under traditional scenarios
- Nodes provide the following memory routines:

```
ArrayRCP<T> Node::allocBuffer<T>(size_t sz);  
void          Node::copyToBuffer<T>(    T * src,  
                                         ArrayRCP<T> dest);  
void          Node::copyFromBuffer<T>(ArrayRCP<T> src,  
                                         T * dest);  
ArrayRCP<T> Node::viewBuffer<T>(ArrayRCP<T> buff);  
void          Node::readyBuffer<T>(ArrayRCP<T> buff);
```



# Kokkos Compute Model

---

- How to make shared-memory programming generic:
  - 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.
- Template meta-programming is **the answer**.
  - This is the same approach that Intel TBB and Thrust take.
  - Has the effect of requiring that Tpetra objects be templated on Node type.
- Node provides generic parallel constructs, user fills in the rest:

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

Work-data pair (WDP) struct provides:

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

```
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(i)`
- reduction via `WDP::reduce(x, y)`

# Example Kernels: `axpy()` and `dot()`

```
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]; }
};
```

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

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

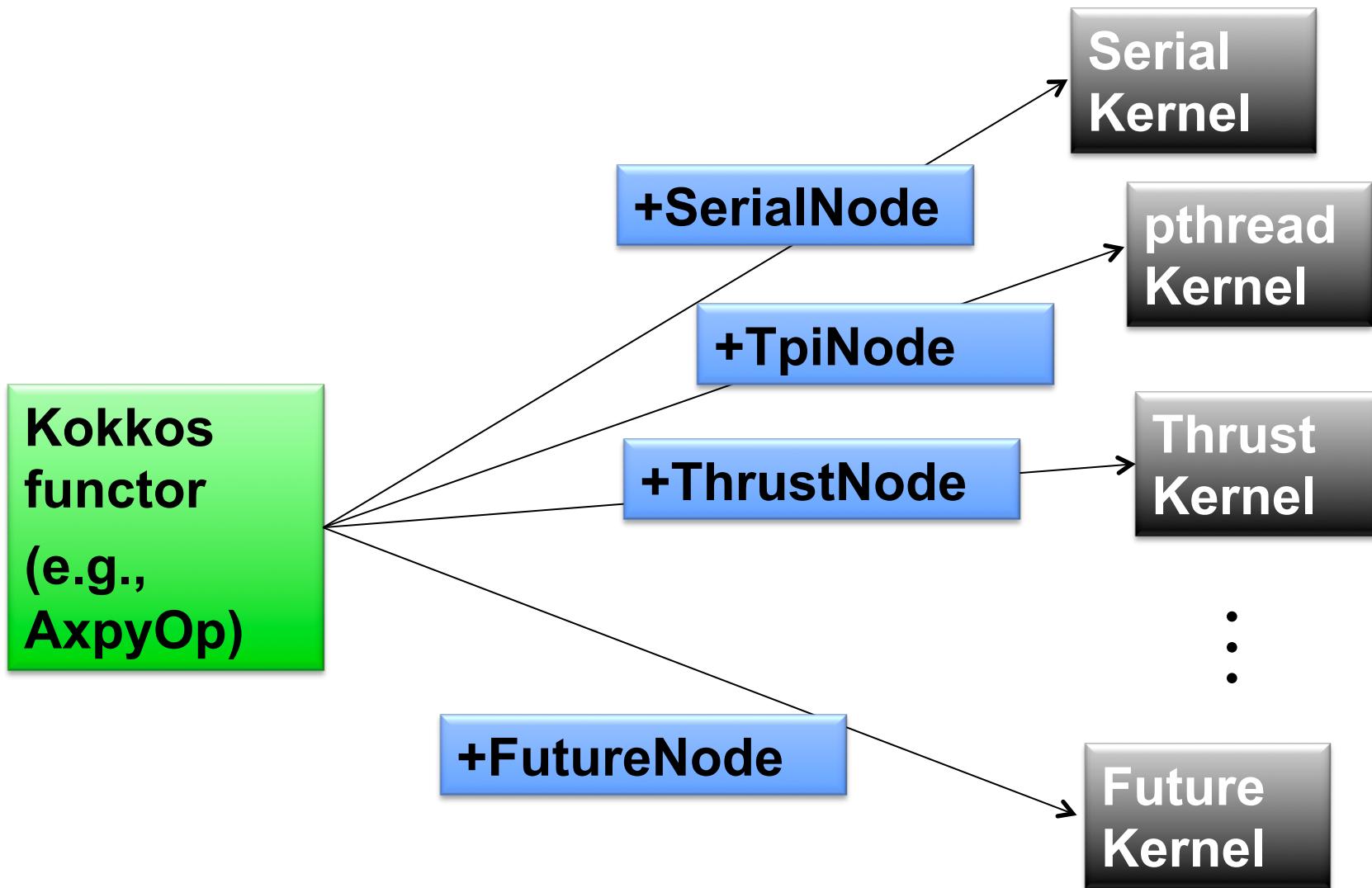
```
template <class T>
struct DotOp {
    typedef T ReductionType;
    const T * x, * y;
    T identity()      { return (T)0;      }
    T generate(int i) { return x[i]*y[i]; }
    T reduce(T x, T y) { return x + y;    }
};
```

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



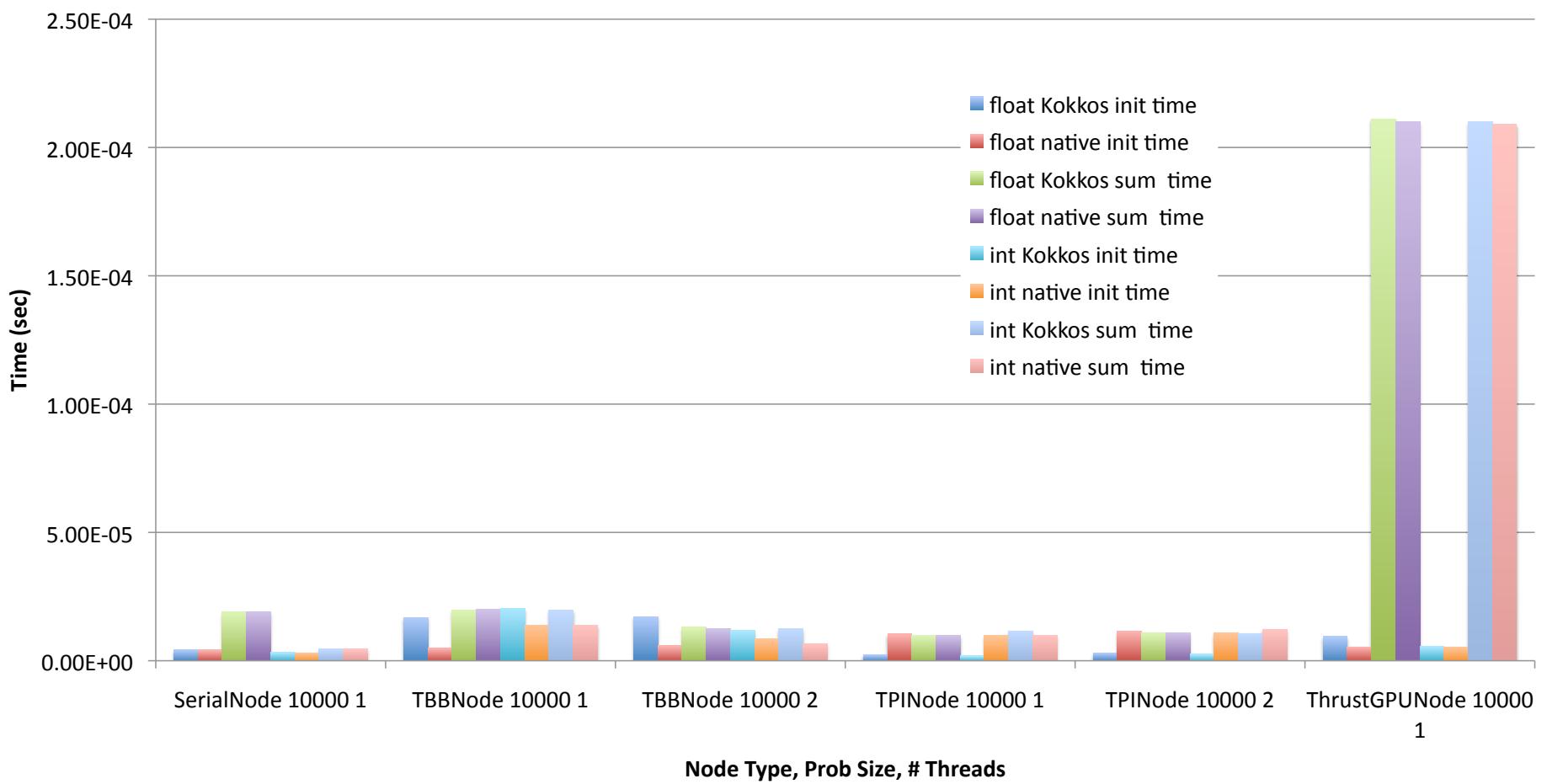
# Compile-time Polymorphism

---

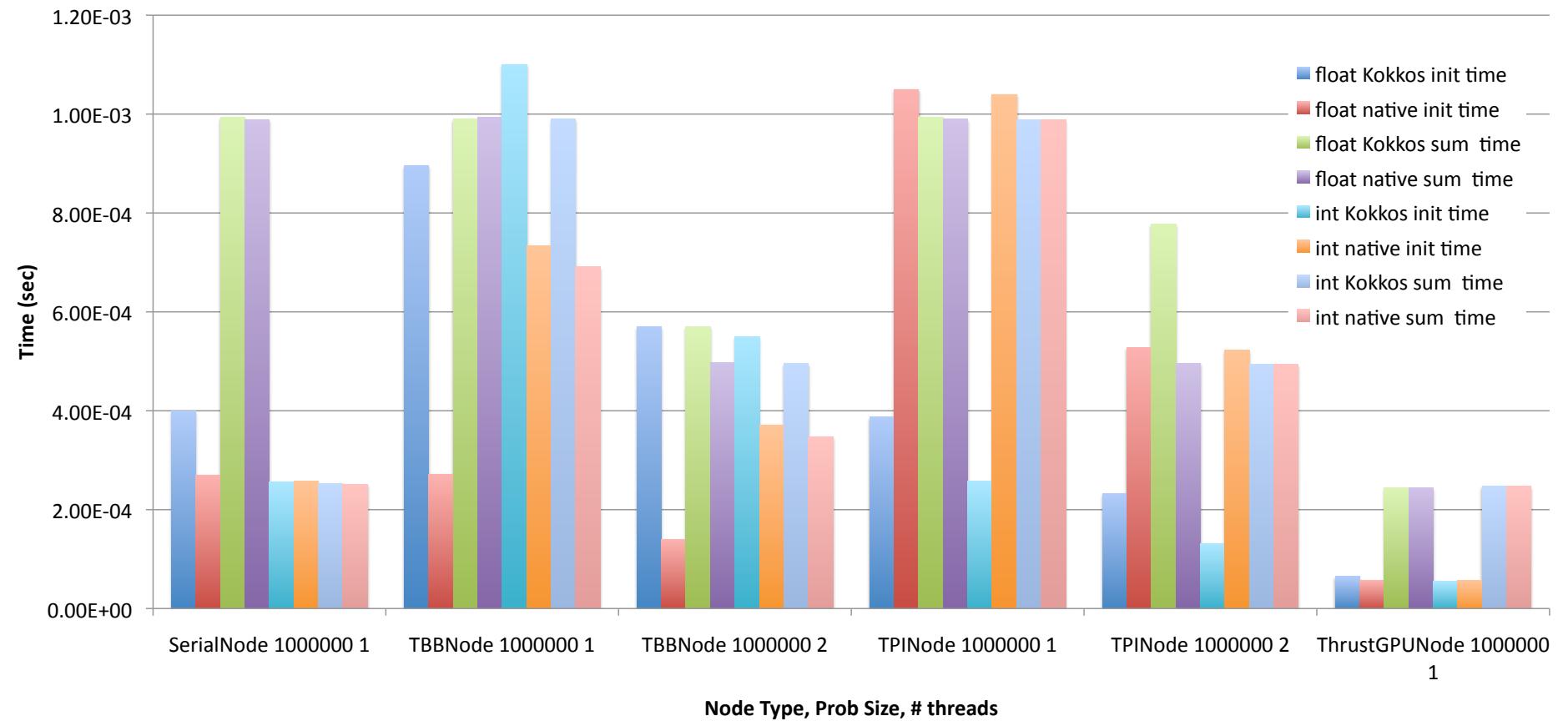


## Kokkos Node API vs Native Implementation

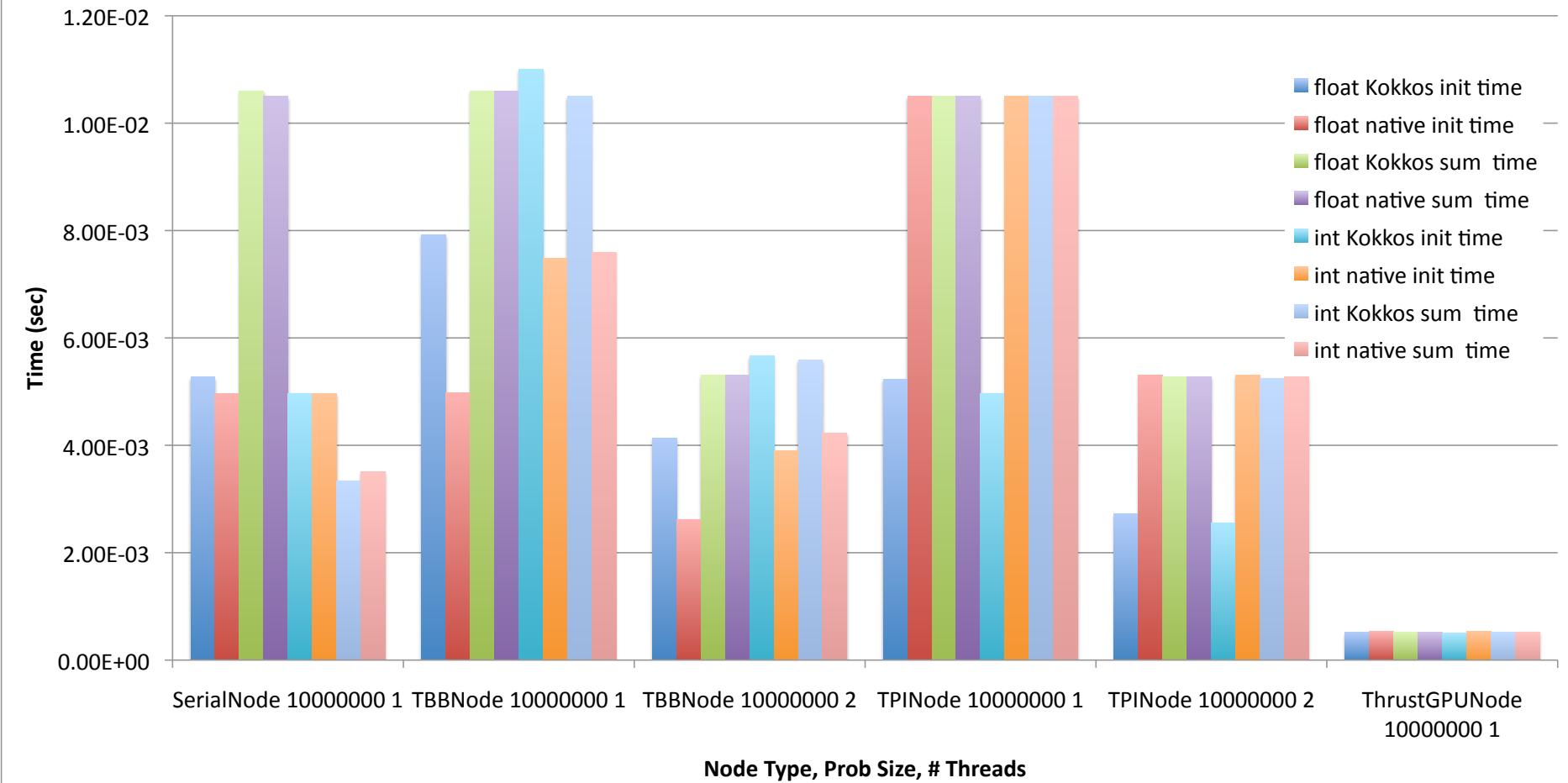
Axpy, len=10K, float, int data



## Kokkos Node API vs Native Implementation Apxy, len=1M



## Kokkos Node API vs Native Implementation Axpy, len=10M, float, int data



# What's the Big Deal about Vector-Vector Operations?

## Examples from OOQP (Gertz, Wright)

$$y_i \leftarrow y_i + \alpha x_i z_i \quad , i = 1 \dots n$$

$$y_i \leftarrow \begin{cases} y^{\min} - y_i & \text{if } y_i < y^{\min} \\ y^{\max} - y_i & \text{if } y_i > y^{\max} \\ 0 & \text{if } y^{\min} \leq y_i \leq y^{\max} \end{cases}, i = 1 \dots n$$

$$y_i \leftarrow y_i / x_i \quad , i = 1 \dots n$$

$$\alpha \leftarrow \{\max \alpha : x + \alpha d \geq \beta\}$$

## Example from TRICE (Dennis, Heinkenschloss, Vicente)

$$d_i \leftarrow \begin{cases} (b - u)_i^{1/2} & \text{if } w_i < 0 \text{ and } b_i < +\infty \\ 1 & \text{if } w_i < 0 \text{ and } b_i = +\infty \\ (u - a)_i^{1/2} & \text{if } w_i \geq 0 \text{ and } a_i > -\infty \\ 1 & \text{if } w_i \geq 0 \text{ and } a_i = -\infty \end{cases}, i = 1 \dots n$$

## Example from IPOPT (Waechter)

$$x_i \leftarrow \begin{cases} \left( x_i^L + \frac{(x_i^U - x_i^L)}{2} \right) & \text{if } \ddot{x}_i^L > \ddot{x}_i^U \\ \ddot{x}_i^L & \text{if } x_i < \ddot{x}_i^L \\ \ddot{x}_i^U & \text{if } x_i > \ddot{x}_i^U \end{cases}, i = 1 \dots n$$

where:

$$\ddot{x}_i^L = \min(x_i^L + \eta(x_i^U - x_i^L)x_i^L + \delta)$$

$$\ddot{x}_i^U = \max(x_i^L - \eta(x_i^U - x_i^L)x_i^U - \delta)$$

Many different and unusual vector operations are needed by interior point methods for optimization!

Currently in MOOCHO :  
 > 40 vector operations!

# Tpetra RTI Components

- Set of stand-alone non-member methods:
  - `unary_transform<UOP>(Vector &v, UOP op)`
  - `binary_transform<BOP>(Vector &v1, const Vector &v2, BOP op)`
  - `reduce<G>(const Vector &v1, const Vector &v2, G op_glob)`
  - `binary_pre_transform_reduce<G>( Vector &v1, const Vector &v2, G op_glob)`
- These are non-member methods of `Tpetra::RTI` which are loosely coupled with `Tpetra::MultiVector` and `Tpetra::Vector`.
- `Tpetra::RTI` also provides Operator-wrappers:
  - `class KernelOp<..., Kernel > : Tpetra::Operator<...>`
  - `class BinaryOp<..., BinaryOp> : Tpetra::Operator<...>`

# Tpetra RTI Example



## Future Node API Trends

---

- TBB provides very rich pattern-based API.
  - It, or something very much like it, will provide environment for sophisticated parallel patterns.
- Simple patterns: FutureNode may simply be OpenMP.
  - OpenMP handles parallel\_for, parallel\_reduce fairly well.
  - Deficiencies being addressed.
  - Some evidence it can beat CUDA.
- OpenCL practically unusable?
  - Functionally portable.
  - Performance not.
  - Breaks the DRY principle.

---

## *Additional Benefits of Templates*

# Multiprecision possibilities

- Tpetra is a templated version of the Petra distributed linear algebra model in Trilinos.
  - Objects are templated on the underlying data types:

```
MultiVector<scalar=double, local_ordinal=int,  
           global_ordinal=local_ordinal> ...
```

```
CrsMatrix<scalar=double, local_ordinal=int,  
           global_ordinal=local_ordinal> ...
```

- Examples:

```
MultiVector<double, int, long int> v;  
CrsMatrix<float> A;
```

Speedup of float over double  
in Belos linear solver.

float	double	speedup
18 s	26 s	1.42x

Scalar	float	double	double-double	quad-double
Solve time (s)	2.6	5.3	29.9	76.5
Accuracy	$10^{-6}$	$10^{-12}$	$10^{-24}$	$10^{-48}$

Arbitrary precision solves  
using Tpetra and Belos  
linear solver package

# FP Accuracy Analysis: FloatShadowDouble Datatype

```
class FloatShadowDouble {  
  
public:  
    FloatShadowDouble( ) {  
        f = 0.0f;  
        d = 0.0; }  
    FloatShadowDouble( const FloatShadowDouble & fd) {  
        f = fd.f;  
        d = fd.d; }  
    ...  
    inline FloatShadowDouble operator+= (const FloatShadowDouble & fd ) {  
        f += fd.f;  
        d += fd.d;  
        return *this; }  
    ...  
    inline std::ostream& operator<<(std::ostream& os, const FloatShadowDouble& fd) {  
        os << fd.f << "f " << fd.d << "d"; return os;}
```

- Templates enable new analysis capabilities
- Example: Float with “shadow” double.

# FloatShadowDouble

Sample usage:

```
#include "FloatShadowDouble.hpp"
Tpetra::Vector<FloatShadowDouble> x, y;
Tpetra::CrsMatrix<FloatShadowDouble> A;
A.apply(x, y); // Single precision, but double results also computed, available
```

Initial Residual =	455.194f	455.194d
Iteration = 15	Residual = 5.07328f	5.07618d
Iteration = 30	Residual = 0.00147022f	0.00138466d
Iteration = 45	Residual = 5.14891e-06f	2.09624e-06d
Iteration = 60	Residual = 4.03386e-09f	7.91927e-10d

```
#ifndef TPETRA_POWER_METHOD_HPP
#define TPETRA_POWER_METHOD_HPP

#include <Tpetra_Operator.hpp>
#include <Tpetra_Vector.hpp>
#include <Teuchos_ScalarTraits.hpp>

namespace TpetraExamples {

/** \brief Simple power iteration eigensolver for a Tpetra::Operator.
 */
template <class Scalar, class Ordinal>
Scalar powerMethod(const Teuchos::RCP<const Tpetra::Operator<Scalar,Ordinal>> &A,
                   int niters, typename Teuchos::ScalarTraits<Scalar>::magnitudeType tolerance,
                   bool verbose)
{
    typedef typename Teuchos::ScalarTraits<Scalar>::magnitudeType Magnitude;
    typedef Tpetra::Vector<Scalar,Ordinal> Vector;

    if ( A->getRangeMap() != A->getDomainMap() ) {
        throw std::runtime_error("TpetraExamples::powerMethod(): operator must have domain and range maps that are equivalent.");
    }
}
```

```

// create three vectors, fill z with random numbers
Teuchos::RCP<Vector> z, q, r;
q = Tpetra::createVector<Scalar>(A->getRangeMap());
r = Tpetra::createVector<Scalar>(A->getRangeMap());
z = Tpetra::createVector<Scalar>(A->getRangeMap());
z->randomize();
//
Scalar lambda = 0.0;
Teuchos::ScalarTraits<Scalar>::magnitudeType normz, residual = 0.0;
// power iteration
for (int iter = 0; iter < niters; ++iter) {
    normz = z->norm2();                      // Compute 2-norm of z
    q->scale(1.0/normz, *z);                  // Set q = z / normz
    A->apply(*q, *z);                        // Compute z = A*q
    lambda = q->dot(*z);                     // Approximate maximum eigenvalue: lambda = dot(q,z)
    if ( iter % 100 == 0 || iter + 1 == niters ) {
        r->update(1.0, *z, -lambda, *q, 0.0); // Compute A*q - lambda*q
        residual = Teuchos::ScalarTraits<Scalar>::magnitude(r->norm2() / lambda);
        if (verbose) {
            std::cout << "Iter = " << iter << " Lambda = " << lambda
                << " Residual of A*q - lambda*q = " << residual << std::endl; }
    }
    if (residual < tolerance) { break; }
}
return lambda;
}
} // end of namespace TpetraExamples

```

---

## *Placement and Migration*



# Placement and Migration

---

- MPI:
  - Data/work placement clear.
  - Migration explicit.
- Threading:
  - It's a mess (IMHO).
  - Some platforms good.
  - Many not.
  - Default is bad (but getting better).
  - Some issues are intrinsic.



## Data Placement on NUMA

---

- Memory Intensive computations: Page placement has huge impact.
- Most systems: First touch (except LWKs).
- Application data objects:
  - Phase 1: Construction phase, e.g., finite element assembly.
  - Phase 2: Use phase, e.g., linear solve.
- Problem: First touch difficult to control in phase 1.
- Idea: Page migration.
  - Not new: SGI Origin. Many old papers on topic.

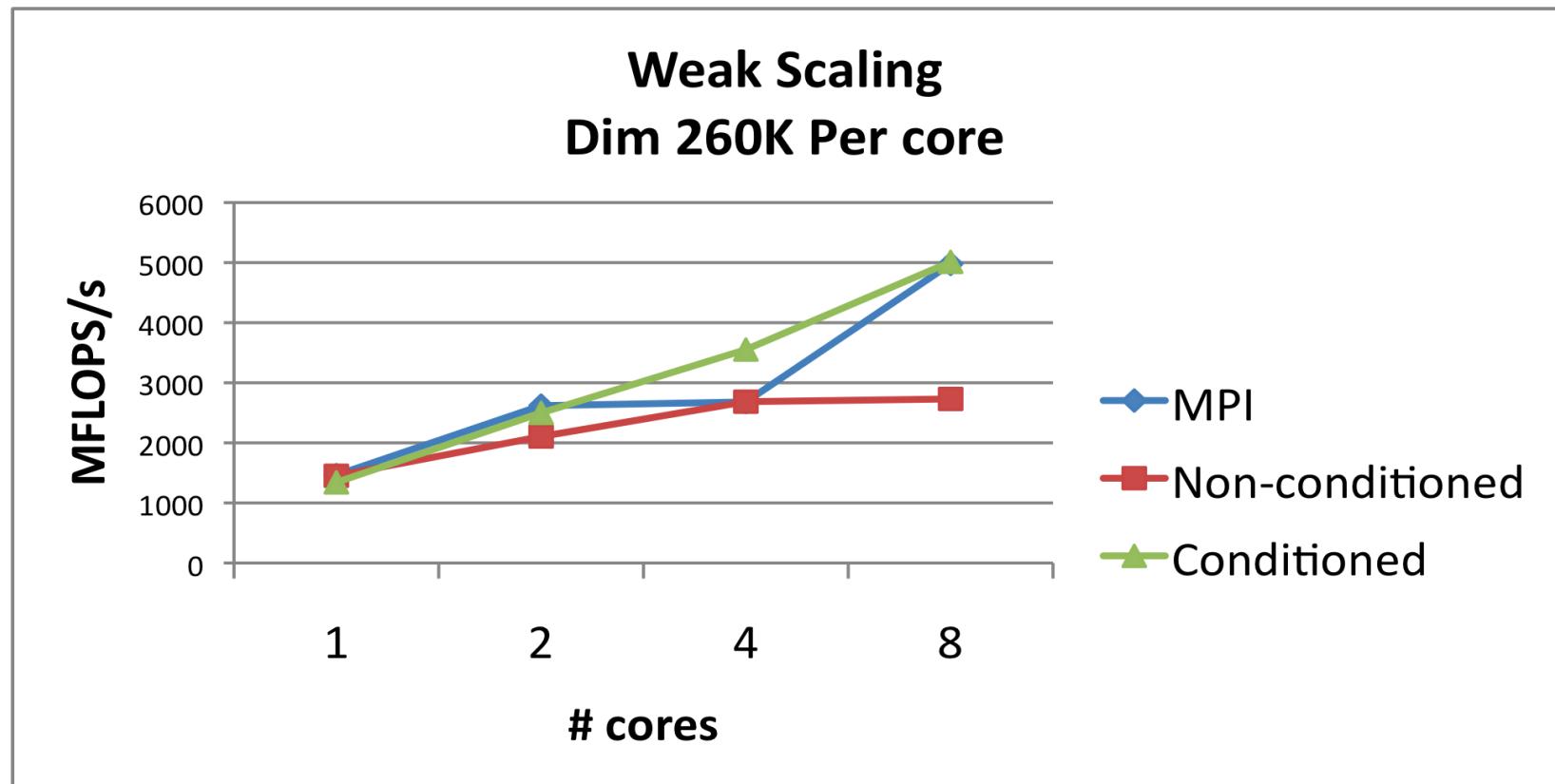


## Data placement experiments

---

- MiniApp: HPCCG (Mantevo Project)
- Construct sparse linear system, solve with CG.
- Two modes:
  - Data placed by assembly, not migrated for NUMA
  - Data migrated using parallel access pattern of CG.
- Results on dual socket quad-core Nehalem system.

# Weak Scaling Problem



- MPI and conditioned data approach comparable.
- Non-conditioned very poor scaling.



## Page Placement summary

---

- MPI+OpenMP (or any threading approach) is best overall.
- But:
  - Data placement is big issue.
  - Hard to control.
  - Insufficient runtime support.
- Current work:
  - Migrate on next-touch (MONT).
  - Considered in OpenMP (next version).
  - Also being studied in Kitten (Kevin Pedretti).
- Note: This phenomenon especially damaging to OpenMP common usage.

---

*Resilient Algorithms:*  
**A little reliability, please.**



## My Luxury in Life (wrt FT/Resilience)

---

The privilege to think of a computer as a  
*reliable, digital* machine.

“At 8 nm process technology, it will be harder  
to tell a 1 from a 0.”

(W. Camp)



## Users' View of the System Now

---

- “All nodes up and running.”
- Certainly nodes fail, but invisible to user.
- No need for me to be concerned.
- Someone else’s problem.



## Users' View of the System Future

---

- Nodes in one of four states.
  1. Dead.
  2. Dying (perhaps producing faulty results).
  3. Reviving.
  4. Running properly:
    - a) Fully reliable or...
    - b) Maybe still producing an occasional bad result.



## Hard Error Futures

---

- C/R will continue as dominant approach:
  - Global state to global file system OK for small systems.
  - Large systems: State control will be localized, use SSD.
- Checkpoint-less restart:
  - Requires full vertical HW/SW stack co-operation.
  - Very challenging.
  - Stratified research efforts not effective.



# Soft Error Futures

---

- Soft error handling: A legitimate algorithms issue.
- Programming model, runtime environment play role.



## Consider GMRES as an example of how soft errors affect correctness

---

- Basic Steps
  - 1) Compute Krylov subspace (preconditioned sparse matrix-vector multiplies)
  - 2) Compute orthonormal basis for Krylov subspace (matrix factorization)
  - 3) Compute vector yielding minimum residual in subspace (linear least squares)
  - 4) Map to next iterate in the full space
  - 5) Repeat until residual is sufficiently small
- More examples in Bronevetsky & Supinski, 2008



## Why GMRES?

---

- Many apps are implicit.
- Most popular (nonsymmetric) linear solver is preconditioned GMRES.
- Only small subset of calculations need to be reliable.
  - GMRES is iterative, but also direct.

# Every calculation matters

## Soft Error Resilience

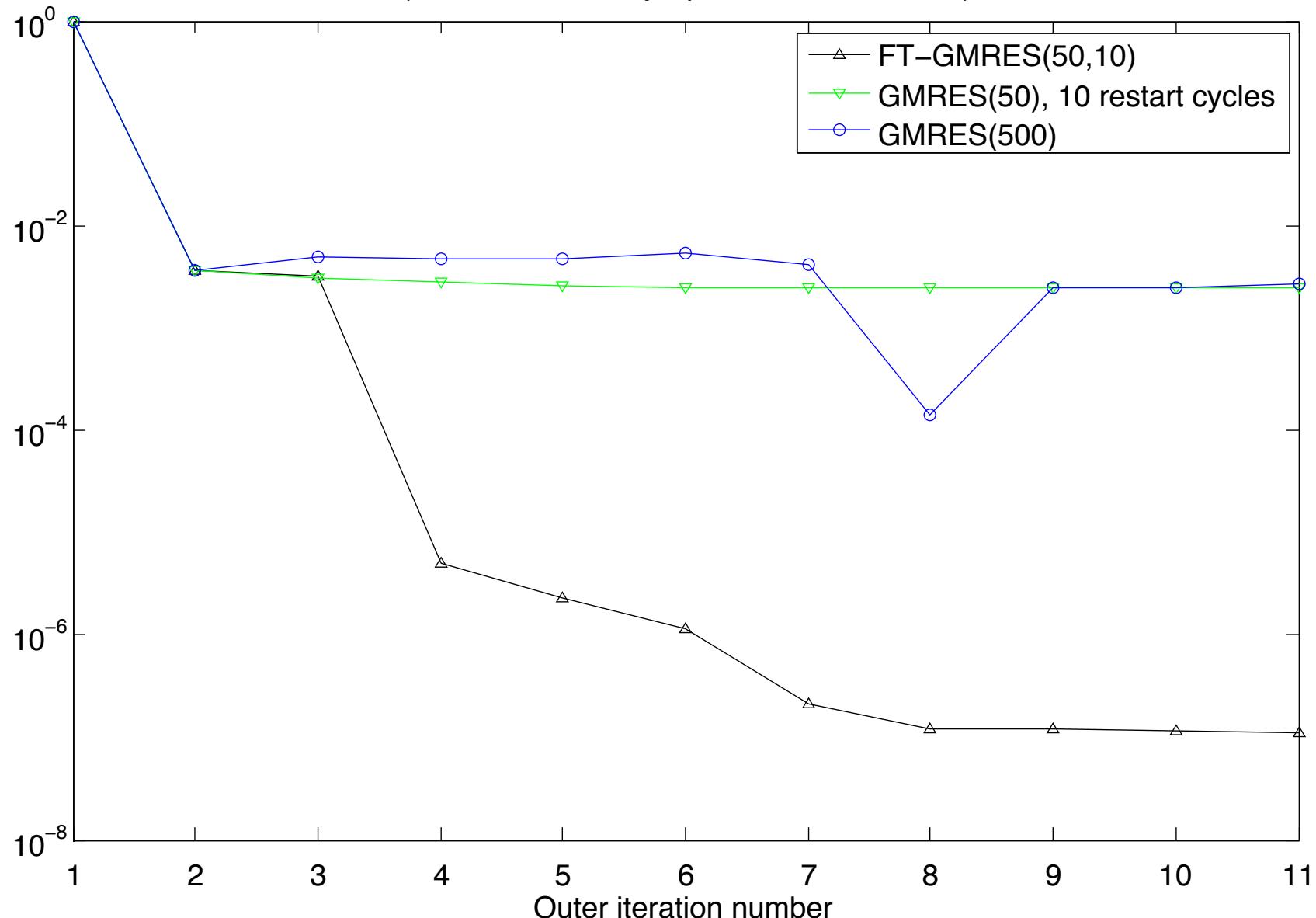
Description	Iters	FLOPS	Recursive Residual Error	Solution Error
All Correct Calcs	35	343M	4.6e-15	1.0e-6
Iter=2, $y[1] += 1.0$ SpMV incorrect Ortho subspace	35	343M	6.7e-15	3.7e+3
$Q[1][1] += 1.0$ Non-ortho subspace	N/C	N/A	7.7e-02	5.9e+5

- Small PDE Problem: ILUT/GMRES
- Correct result: 35 Iters, 343M FLOPS
- 2 examples of a **single** bad op.
- Solvers:
  - 50-90% of total app operations.
  - Soft errors most likely in solver.
- Need new algorithms for soft errors:
  - Well-conditioned wrt errors.
  - Decay proportional to number of errors.
  - Minimal impact when no errors.

- New Programming Model Elements:
  - SW-enabled, highly reliable:
    - Data storage, paths.
    - Compute regions.
- Idea: *New algorithms with minimal usage of high reliability.*
- First new algorithm: FT-GMRES.
  - Resilient to soft errors.
  - Outer solve: Highly Reliable
  - Inner solve: “bulk” reliability.
- General approach applies to many algorithms.

# FTGMRES Results

Fault-Tolerant GMRES, restarted GMRES, and nonrestarted GMRES  
(deterministic faulty SpMVs in inner solves)





## Quiz (True or False)

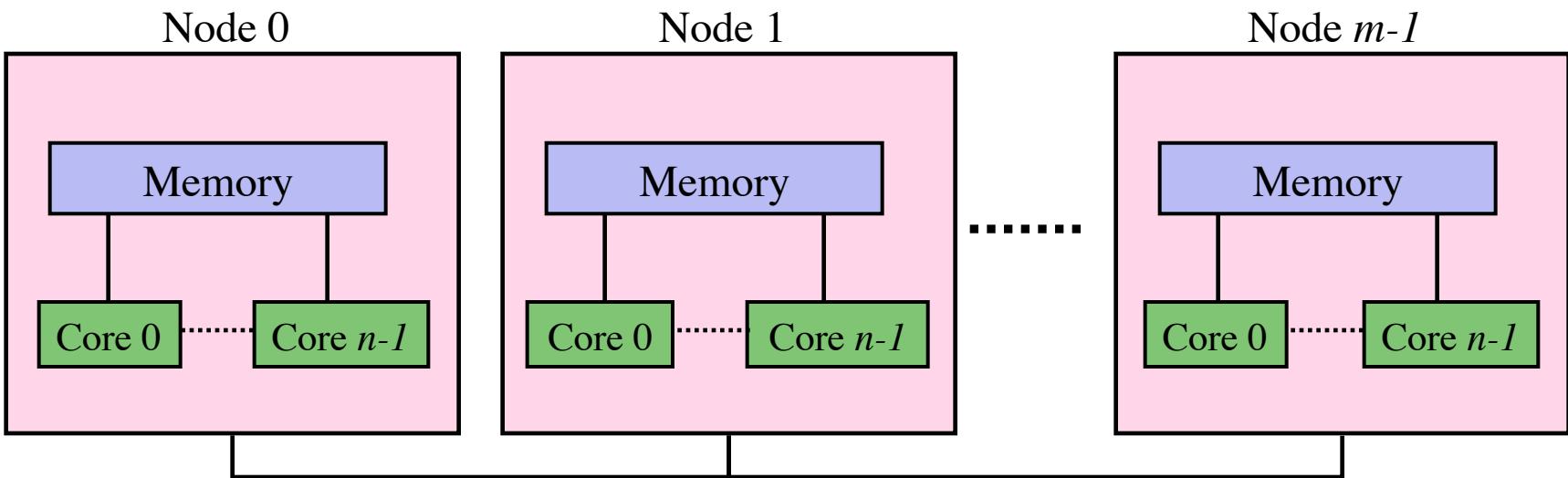
---

5. DRY is not possible across CPUs and GPUs.
6. Extended precision is too expensive to be useful.
7. Resilience will be built into algorithms.

---

*Bi-Modal: MPI-only and MPI+[X|Y|Z]*

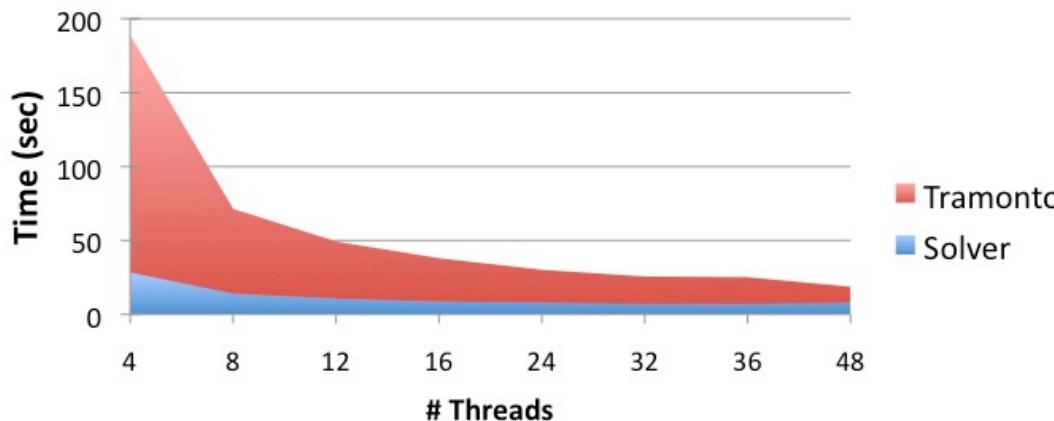
# Parallel Machine Block Diagram



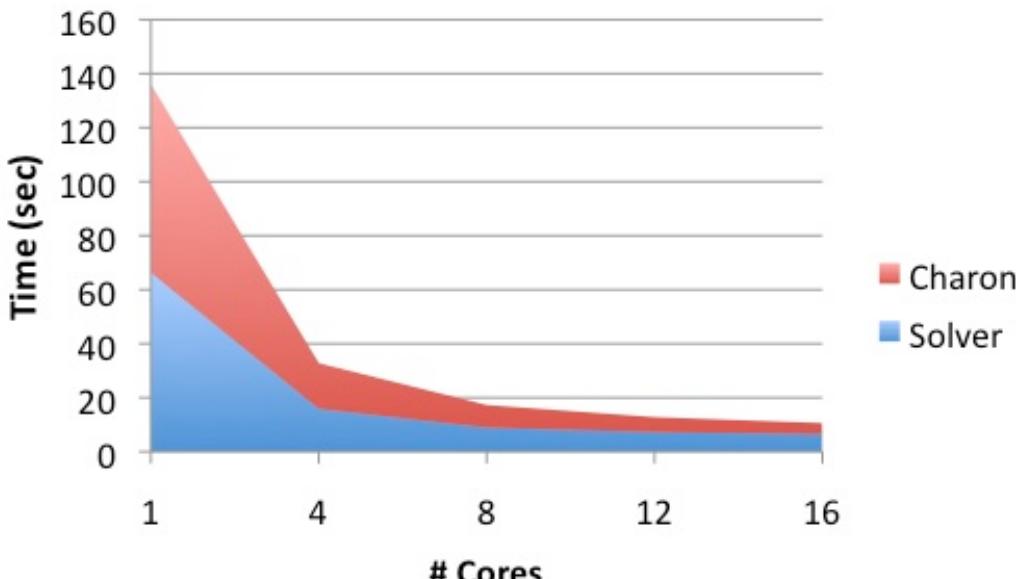
- Parallel machine with  $p = m * n$  processors:
  - $m$  = number of nodes.
  - $n$  = number of shared memory processors per node.
- Two ways to program:
  - Way 1:  $p$  MPI processes.
  - Way 2:  $m$  MPI processes with  $n$  threads per MPI process.
- New third way:
  - “Way 1” in some parts of the execution (the app).
  - “Way 2” in others (the solver).

# Multicore Scaling: App vs. Solver

Tramonto vs. Solver Time on Niagara2:  
4-48 Threads



Charon vs Solver Time: 1-16 Cores



## Application:

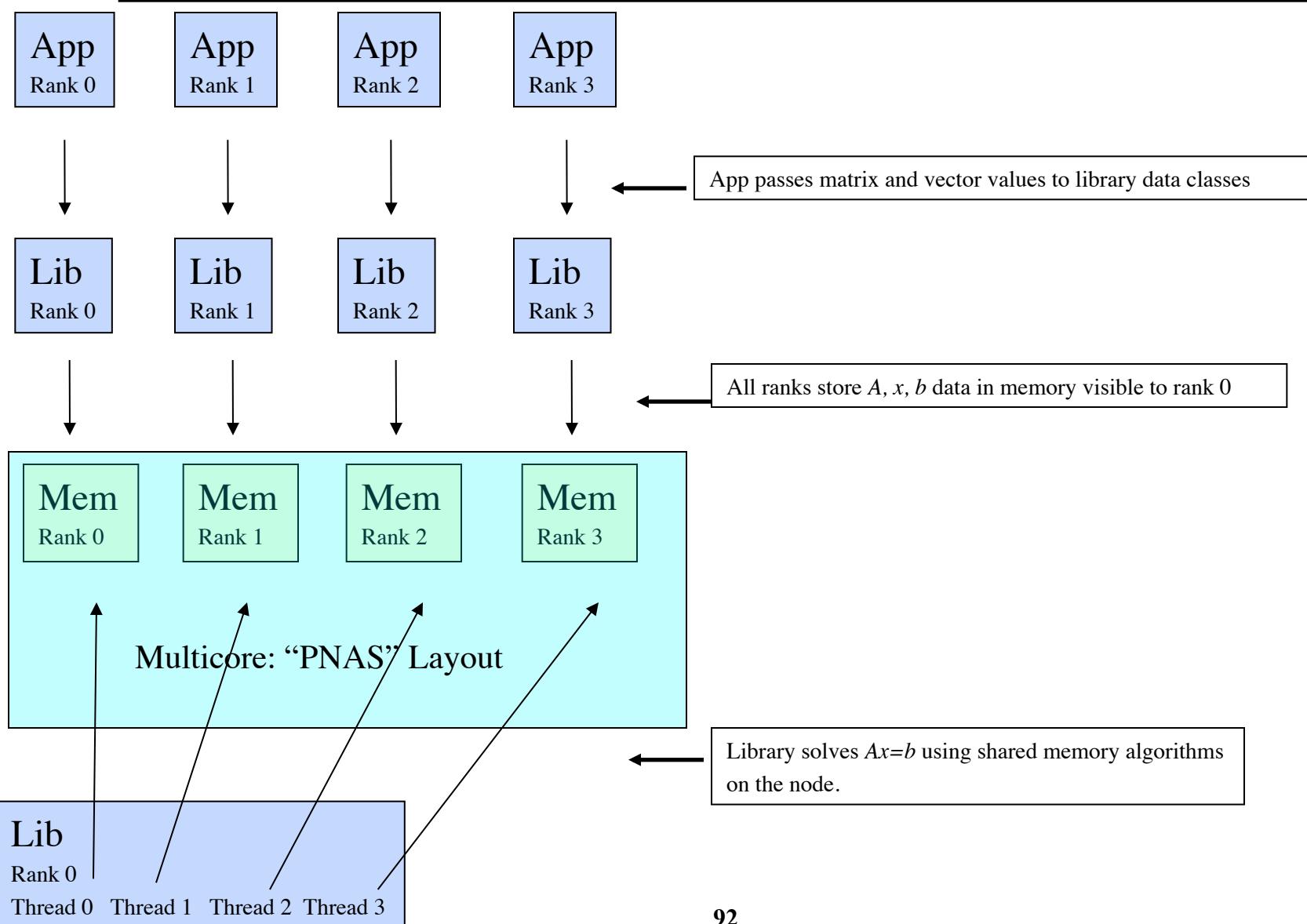
- Scales well  
(sometimes superlinear)
- MPI-only sufficient.

## Solver:

- Scales more poorly.
- Memory system-limited.
- MPI+threads can help.

\* Charon Results:  
Lin & Shadid TLCC Report

# MPI-Only + MPI/Threading: $Ax=b$



# 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.
  - First “Hello World” Program works.
  - Incorporation into standard still not certain. Need to build case.
  - Next Step: Demonstrate usage with threaded triangular solve.
- Exascale potential:
  - Incremental path to MPI+X.
  - Dial-able SMP scope.

```
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 (values).
// - Can continue in MPI mode (using shared memory algorithms) or
// - Can quiet all but one:
int rank;
MPI_Comm_rank(MPI_COMM_NODE, &rank);
if (rank==0) { // Start threaded code segment, only on rank 0 of the node
...
}

MPI_Comm_free_mem(MPI_COMM_NODE, values);
```

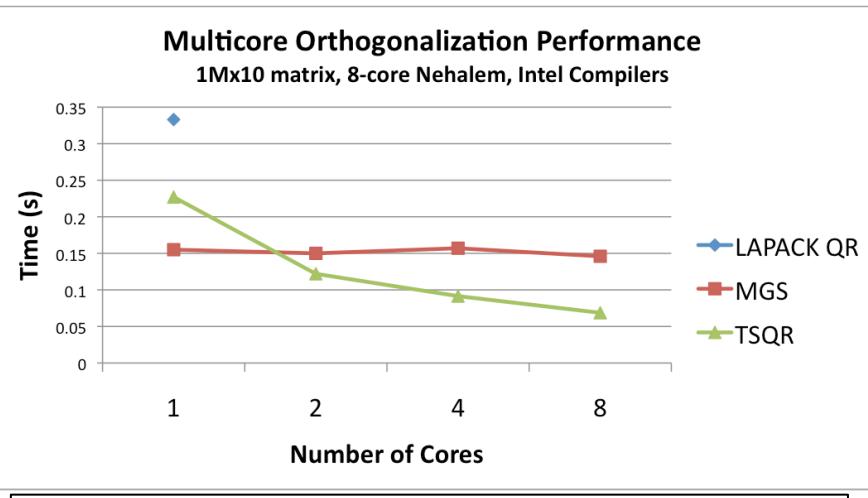
Collaborators: B. Barrett, Brightwell, Wolf - SNL; Vallee, Koenig - ORNL

---

# *Algorithms and Meta-Algorithms*

# Communication-avoiding iterative methods

- Iterative Solvers:
  - Dominant cost of many apps (up to 80+% of runtime).
- Exascale challenges for iterative solvers:
  - Collectives, synchronization.
  - Memory latency/BW.
  - Not viable on exascale systems in present forms.
- Communication-avoiding ( $s$ -step) iterative solvers:
  - Idea: Perform  $s$  steps in bulk ( $s=5$  or more):
    - $s$  times fewer synchronizations.
    - $s$  times fewer data transfers: Better latency/BW.
  - Problem: Numerical accuracy of orthogonalization.
- New orthogonalization algorithm:
  - Tall Skinny QR factorization (TSQR).
  - Communicates less *and* more accurate than previous approaches.
  - Enables reliable, efficient  $s$ -step methods.
- TSQR Implementation:
  - 2-level parallelism (Inter and intra node).
  - Memory hierarchy optimizations.
  - Flexible node-level scheduling via Intel Threading Building Blocks.
  - Generic scalar data type: supports mixed and extended precision.



LAPACK – Serial, MGS – Threaded modified Gram-Schmidt

## TSQR capability:

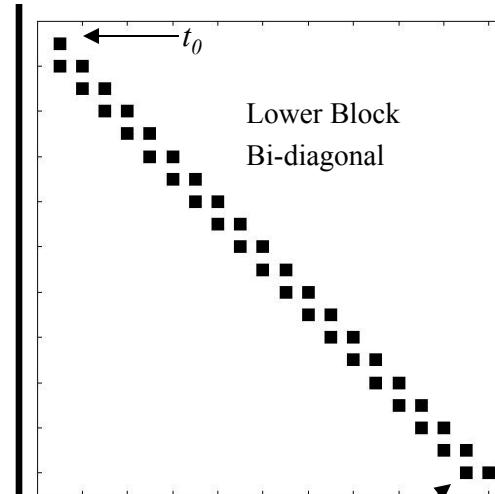
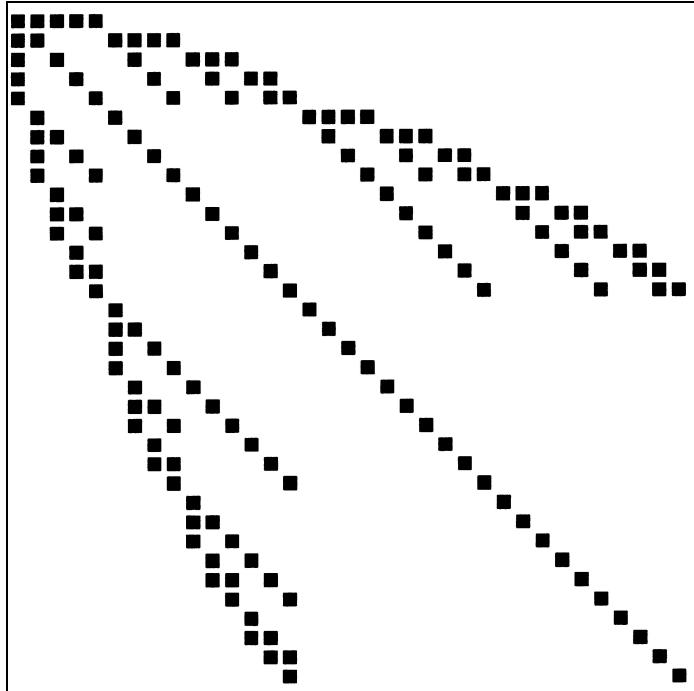
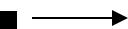
- Critical for exascale solvers.
- Part of the Trilinos scalable multicore capabilities.
- Helps all iterative solvers in Trilinos (available to external libraries, too).
- Staffing: Mark Hoemmen (lead, post-doc, UC-Berkeley), M. Heroux
- Part of Trilinos 10.6 release, Sep 2010.



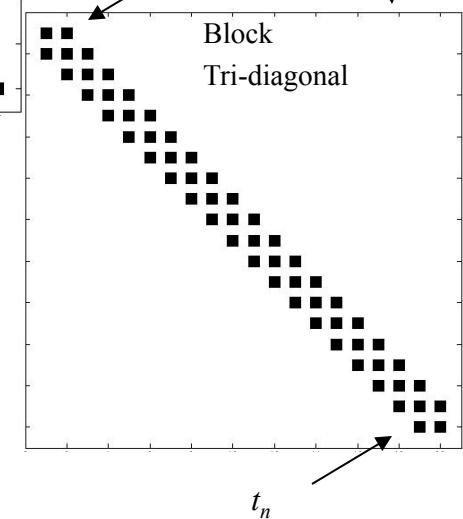
## Advanced Modeling and Simulation Capabilities: Stability, Uncertainty and Optimization

- Promise: 10-1000 times increase in parallelism (or more).

SPDEs:



Transient  
Optimization:



- Pre-requisite: High-fidelity “forward” solve:
  - Computing families of solutions to similar problems.
  - Differences in results must be meaningful.

■ - Size of a single forward problem



# Advanced Capabilities: Readiness and Importance

Modeling Area	Sufficient Fidelity?	Other concerns	Advanced capabilities priority
Seismic <i>S. Collis, C. Ober</i>	Yes.	None as big.	Top.
Shock & Multiphysics (Alegra) <i>A. Robinson, C. Ober</i>	Yes, but some concerns.	Constitutive models, material responses maturity.	Secondary now. Non-intrusive most attractive.
Multiphysics (Charon) <i>J. Shadid</i>	Reacting flow w/ simple transport, device w/ drift diffusion, ...	Higher fidelity, more accurate multiphysics.	Emerging, not top.
Solid mechanics <i>K. Pierson</i>	Yes, but...	Better contact. Better timestepping. Failure modeling.	Not high for now.



## Advanced Capabilities: Other issues

---

- Non-intrusive algorithms (e.g., Dakota):
  - Task level parallel:
    - A true peta/exa scale problem?
    - Needs a cluster of 1000 tera/peta scale nodes.
- Embedded/intrusive algorithms (e.g., Trilinos):
  - Cost of code refactoring:
    - Non-linear application becomes “subroutine”.
    - Disruptive, pervasive design changes.
- Forward problem fidelity:
  - Not uniformly available.
  - Smoothness issues.
  - Material responses.



## Advanced Capabilities: Derived Requirements

---

- Large-scale problem presents collections of related subproblems with forward problem sizes.
- Linear Solvers:  $Ax = b \rightarrow AX = B, Ax^i = b^i, A^i x^i = b^i$ 
  - Krylov methods for multiple RHS, related systems.
- Preconditioners:
$$A^i = A_0 + \Delta A^i$$
  - Preconditioners for related systems.
- Data structures/communication:
$$pattern(A^i) = pattern(A^j)$$
  - Substantial graph data reuse.

---

## *An Alternative to Data Passing*



## Additional Challenge: Math/CS Interface

---

- Data motion directive: Don't do it.
- Question:
  - How do we exchange data between app & lib in a manycore world?
  - More generally:
    - How do we reconcile a mathematical view and CS view?
    - Example: *Mathematical* vector vs. *STL* vector.
- Simple example: Odd/Even Merge Sort
- System: 48-core AMD system
  - 4 sockets.
  - Dual 6-core units per socket.
  - 8 NUMA regions.



## OEMSort (2 half-steps)

---

```
bool sort(int length, int * array) {  
    bool isSorted = true;  
    for (int j=0; j<2; ++j)  
        for (int i=0; i< (length-j)/2; ++i) {  
            int v0 = array[2*i+j];  
            int v1 = array[2*i+1+j];  
            if (v0>v1) {  
                array[2*i+j] = v1;  
                array[2*i+1+j] = v0;  
                isSorted = false; // Had to swap at least one value  
            }  
        }  
    return isSorted;  
}
```



## OEMSort main()

---

```
int main(int argc, char *argv[]) {  
    int length = atoi(argv[1]);  
    int * array = new int[length]; // Allocate array for sorting  
    // Initialize the unsorted array, putting values into opposite order  
    for (int i=0; i<length; ++i) array[i] = length - i;  
  
    while(!sort(length, array));  
    cout << "Sorted array[0:" << print_length << "] = ";  
    for (int i=0; i<print_length; ++i) cout << array[i] << " ";  
    cout << endl;  
  
    return 0 ;  
}
```

# First attempt at threading

BTW: This is the common approach today

```
bool sort(int length, int * array) {  
    bool isSorted = true;  
    for (int j=0; j<2; ++j) {  
#pragma omp parallel  
{  
    bool thread_isSorted = true;  
#pragma omp for  
        for (int i=0; i< (length-j)/2; ++i) {  
            int v0 = array[2*i+j];  
            int v1 = array[2*i+1+j];  
            if (v0>v1) {  
                array[2*i+j] = v1;  
                array[2*i+1+j] = v0;  
                thread_isSorted = false;  
            }  
        }  
    }  
}
```

```
#pragma omp critical  
    if (!thread_isSorted) isSorted = false;  
} // omp for  
} // omp parallel  
return isSorted;  
}
```

What's wrong with this?



# OEMSort main() is Still Serial !

---

```
int main(int argc, char *argv[]) {  
    int length = atoi(argv[1]);  
    int * array = new int[length]; // Allocate array for sorting  
    // Initialize the unsorted array, putting values into opposite order  
    for (int i=0; i<length; ++i) array[i] = length - i;  
  
    while(!sort(length, array));  
    cout << "Sorted array[0:" << print_length << "] = ";  
    for (int i=0; i<print_length; ++i) cout << array[i] << " ";  
    cout << endl;  
  
    return 0 ;  
}
```



# OEMSort main() Version 2 (Conditioned Data)

---

```
int main(int argc, char *argv[]) {  
    int length = atoi(argv[1]);  
    int * array = new int[length]; // Allocate array for sorting  
    // Initialize the unsorted array, putting values into opposite order  
    #pragma omp parallel for  
    for (int i=0; i<length; ++i) array[i] = length - i;  
  
    while(!sort(length, array));  
    cout << "Sorted array[0:" << print_length << "] = ";  
    for (int i=0; i<print_length; ++i) cout << array[i] << " ";  
    cout << endl;  
  
    return 0 ;  
}
```



# OEMSort main() Version 3

## Convert User data to thread-private data

---

```
int numCycles = 1; // <= Added feature allows CA-type approaches
int numGhost = numCycles*2;
int * exchangeBuffer = new int[2*numGhost*numThreads];
#pragma omp parallel
{
    int threadNum = omp_get_thread_num();
    int myStart, myStop, numLeftGhost, numRightGhost;
    computeStartStop(threadNum, numThreads, length, numGhost, myStart,
                     myStop, numLeftGhost, numRightGhost, debug);

    int myPaddedLength = myStop-myStart+numLeftGhost+numRightGhost;
    int * myPaddedArray = new int[myPaddedLength];
    int myLength = myStop-myStart;
    int * myArray = myPaddedArray+numLeftGhost;
    int * ptr = myArray;
for (int i=myStart; i<myStop; ++i) *ptr++ = array[i];
```



# ArrayInitializer.hpp

## User-defined functor

---

```
class ArrayInitializer {  
private:  
    int offset;  
  
public:  
    ArrayInitializer(int initOffset) : offset(initOffset) {  
    }  
  
    int operator () (int i) const {  
        return offset - i;  
    }  
};
```



# OEMSort main() Version 4

## Use functor to create thread-private data

---

```
ArrayInitializer arrayFunctor(length);
int numCycles = 128; // <= Do 128 steps before global sync.
int numGhost = numCycles*2;
int * exchangeBuffer = new int[2*numGhost*numThreads];
#pragma omp parallel
{
    int threadNum = omp_get_thread_num();
    int myStart, myStop, numLeftGhost, numRightGhost;
    computeStartStop(threadNum, numThreads, length, numGhost, myStart,
                     myStop, numLeftGhost, numRightGhost, debug);

    int myPaddedLength = myStop-myStart+numLeftGhost+numRightGhost;
    int * myPaddedArray = new int[myPaddedLength];
    int myLength = myStop-myStart;
    int * myArray = myPaddedArray+numLeftGhost;
    int * ptr = myArray;
    for (int i=myStart; i<myStop; ++i) *ptr++ = arrayFunctor(i);
```



# OEM Sort Results

## Length = 10M, 1K Steps

---

### 48-core AMD Node (4-socket, dual hexcore per socket)

Version	Time (sec)	Data Usage
Serial	29.44	40 MB
Simple OMP (Common today)	16.09	40 MB
Conditioned OMP	2.23	80 MB
Thread Private	0.83	80 MB
Thread Private (1 level redundancy)	0.73	80+eps MB
Thread Private with Functor	0.83	40 MB
Thread Private (1 level redundancy & Functor)	0.73	40+eps MB



# Building Next Generation Applications & Libraries: 15 Strategies to Consider

---

1. Prepare for disruptive change
2. Design to the new scalability parameters: thread count and vector lengths
3. Encapsulate all parallelizable functionality into stateless (sequential) functions
4. Organize for vectorization
5. Decide on struct of arrays or array of structs
6. Prefer computation to data storage
7. Consider lower precision storage or computation, or both
8. Consider higher precision storage or computation, or both
9. Exploit data regularity
10. Separate physics indexing from storage indexing
11. Separate definition of array contents from filling of array data structures
12. Create library interfaces, even if you only call your own libraries
13. Look for untapped resources of parallelism
14. Make template meta-programming your friend
15. Build resilience into your software