Toward Parallel Applications for the Year of Exascale

Michael A. Heroux
Scalable Algorithms Department
Sandia National Laboratories

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.

Outline

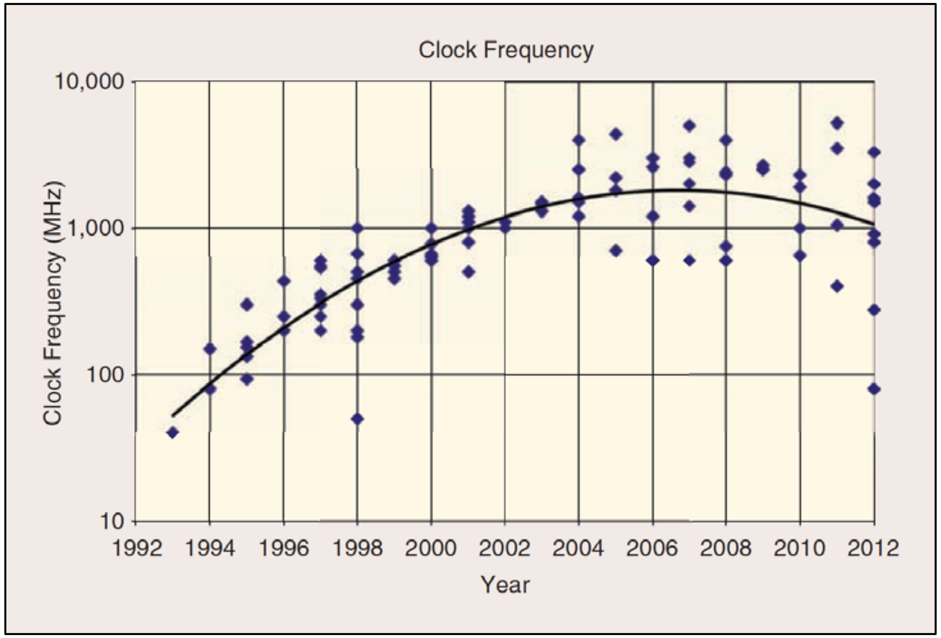
- Parallel Computing Trends and MPI+X.
- Reasoning about Parallelism.
- Programming Languages.
- Resilience.

Stein's Law: *If a trend cannot continue, it will stop.*

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

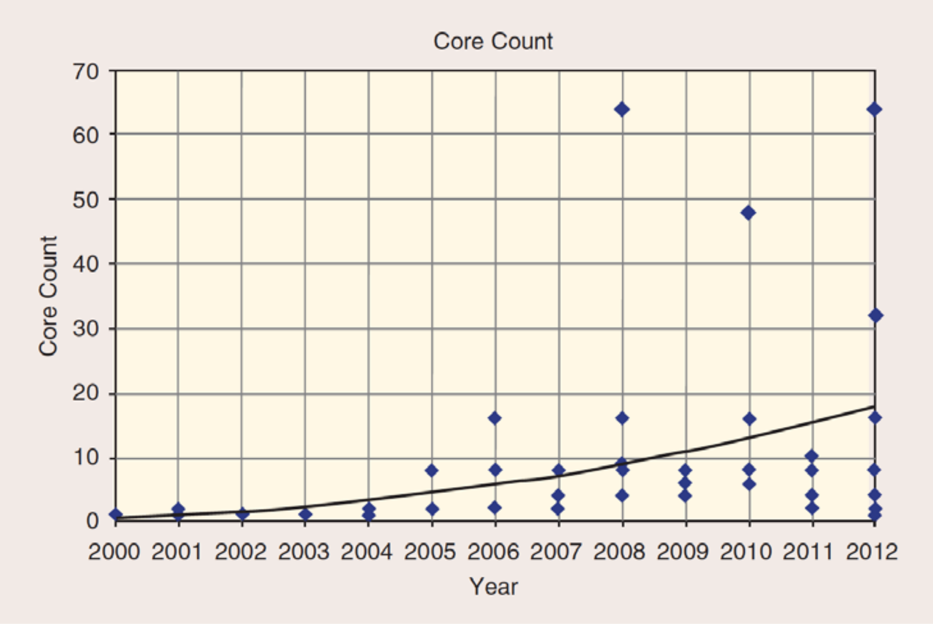
What is Different: Old Commodity Trends Failing

- Clock Speed.
  - Well-known.
  - Related: Instruction-level Parallelism (ILP).
- Number of nodes.
  - Connecting 100K nodes is complicated.
  - Electric bill is large.
- Memory per core.
  - Going down (but some hope in sight).
- Consistent performance.
  - Equal work $\not\Rightarrow$ Equal execution time.
    - Across peers or from one run to the next.



International Solid-State Circuits Conference (ISSCC 2012) Report
http://isscc.org/doc/2012/2012_Trends.pdf

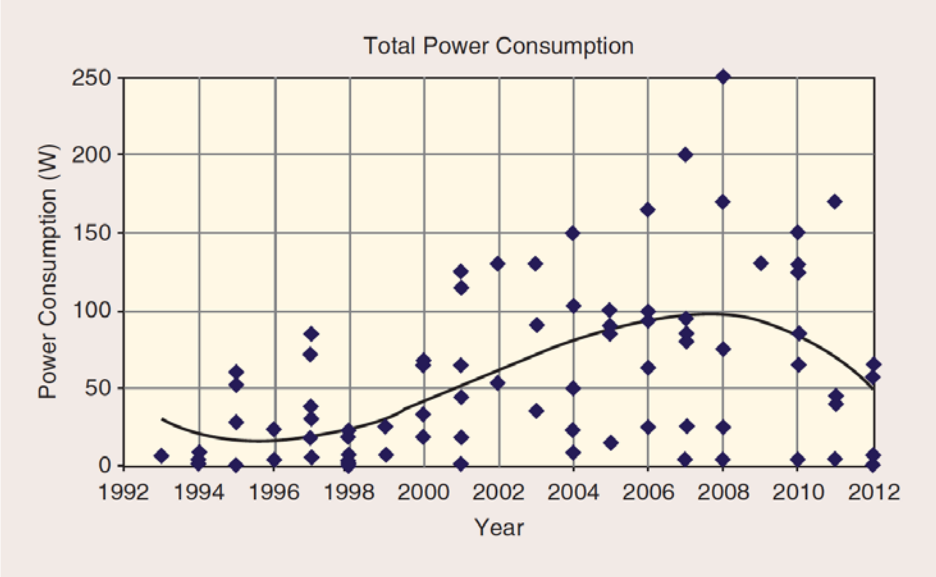
New Commodity Trends and Concerns Emerge



Big Concern: Energy Efficiency.

- Thread count.
  - – Occupancy rate.
  - – State-per-thread.
- SIMT/SIMD (Vectorization).
- Heterogeniety:
  - – Performance variability.
  - – Core specialization.
- Memory per *node* (not core).
  - – Fixed (or growing).

Take-away: Parallelism is essential.



International Solid-State Circuits Conference (ISSCC 2012) Report
http://isscc.org/doc/2012/2012_Trends.pdf

Challenge: Achieve Scalable 1B-way Concurrency

- 10^{18} Ops/sec with 10^9 clock rates: 10^9 Concurrency.
- Question: What role (if any) will MPI play?
- Answer: Major role as MPI+X.
  - MPI: Today's MPI with several key enhancements.
  - X: Industry-provided; represents numerous options.
- Why: MPI+X is leveraged, synergistic, doable.
  - Resilience: Algorithms + MPI/Runtime enhancements.
  - Programmability: There is a path.
- Urgent: Migration to manycore must begin in earnest.
  - We can't wait around for some magic exascale programming model.
  - We have to begin in earnest to learn about X options and deploy as quickly as possible.

Reasons for SPMD/MPI Success?

- Portability? Standardization? Momentum? Yes.
- Separation of Parallel & Algorithms concerns? Big Yes.
- Preserving & Extending Sequential Code Investment? Big, Big Yes.

- MPI was disruptive, but not revolutionary.
  - A meta layer encapsulating sequential code.
    - Enabled mining of vast quantities of existing code and logic.
  - Sophisticated physics added as sequential code.
    - Ratio of science experts vs. parallel experts: 10:1.
- Key goal for new parallel apps: Preserve these dynamics.

Three Parallel Computing Design Points

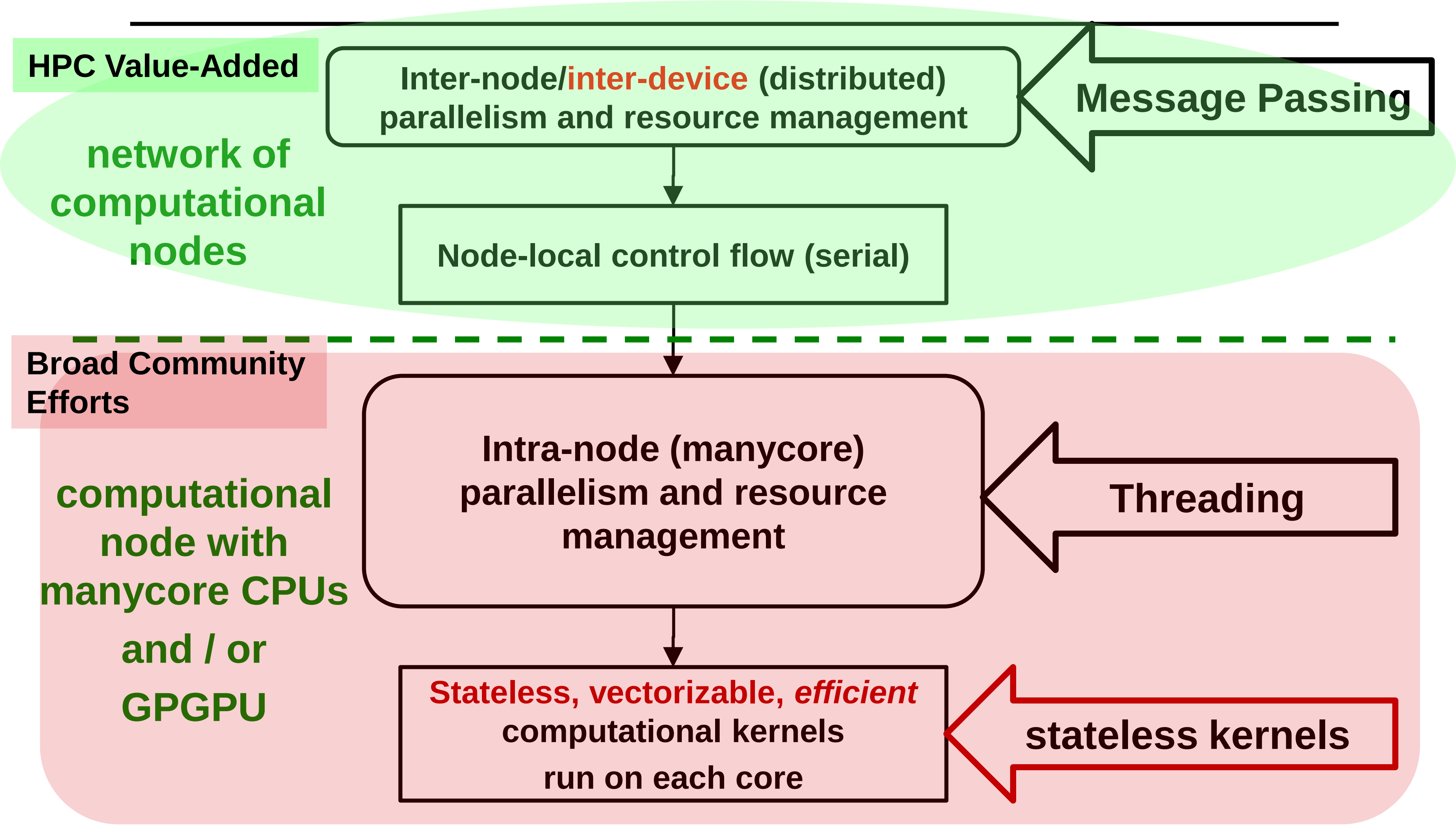
- Terascale Laptop: Uninode-Manycore
- Petascale Deskside: Multinode-Manycore
- Exascale Center: Manynode-Manycore

Goal: Make
Petascale = Terascale + more
Exascale = Petascale + more

Common Element

Applications will not adopt an exascale programming strategy that is incompatible with tera and peta scale.

MPI+X Parallel Programming Model: Multi-level/Multi-device



HPC Value-Added

network of computational nodes

Inter-node/inter-device (distributed) parallelism and resource management

Node-local control flow (serial)

Message Passing

Broad Community Efforts

computational node with manycore CPUs

and / or

GPGPU

Intra-node (manycore) parallelism and resource management

Threading

Stateless, vectorizable, *efficient* computational kernels run on each core

stateless kernels

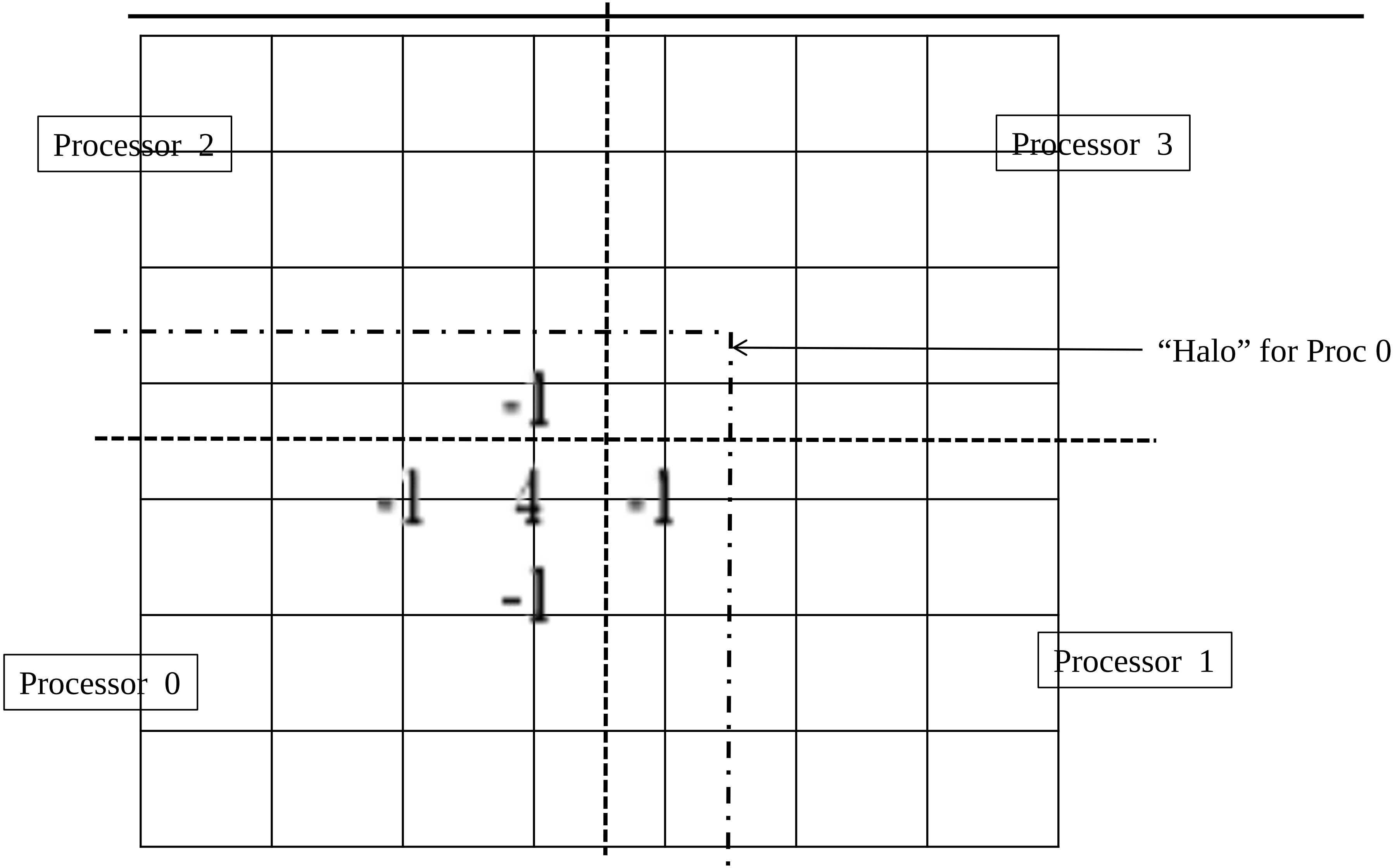
Incentives for MPI+X

- Almost all DOE scalable applications use MPI.
  - MPI provides portability layer.
  - Typically app developer accesses via conceptual layer.
  - Could swap in another SPMD approach (UPC, CAF).
  - Even dynamic SPMD is possible. Adoption expensive.
- Entire computing community is focused on X.
  - It takes a community…
  - Many promising technologies emerging.
  - Industry very interested in programmer productivity.
- MPI and X interactions well understood.
  - Straight-forward extension of existing MPI+Serial.
  - New MPI features will address specific threading needs.

Effective node-level parallelism: First priority

- Future performance is mainly from node improvements.
  - Number of nodes is not increasing dramatically.
- Application refactoring efforts on node are disruptive:
  - Almost every line of code will be displaced.
    - All current serial computations must be threaded.
  - Successful strategy similar to SPMD migration of 90s.
    - Define parallel pattern framework.
    - Make framework scalable for minimal physics.
    - Migrate large sequential fragments into new framework.
- If no node parallelism, we fail at all computing levels.

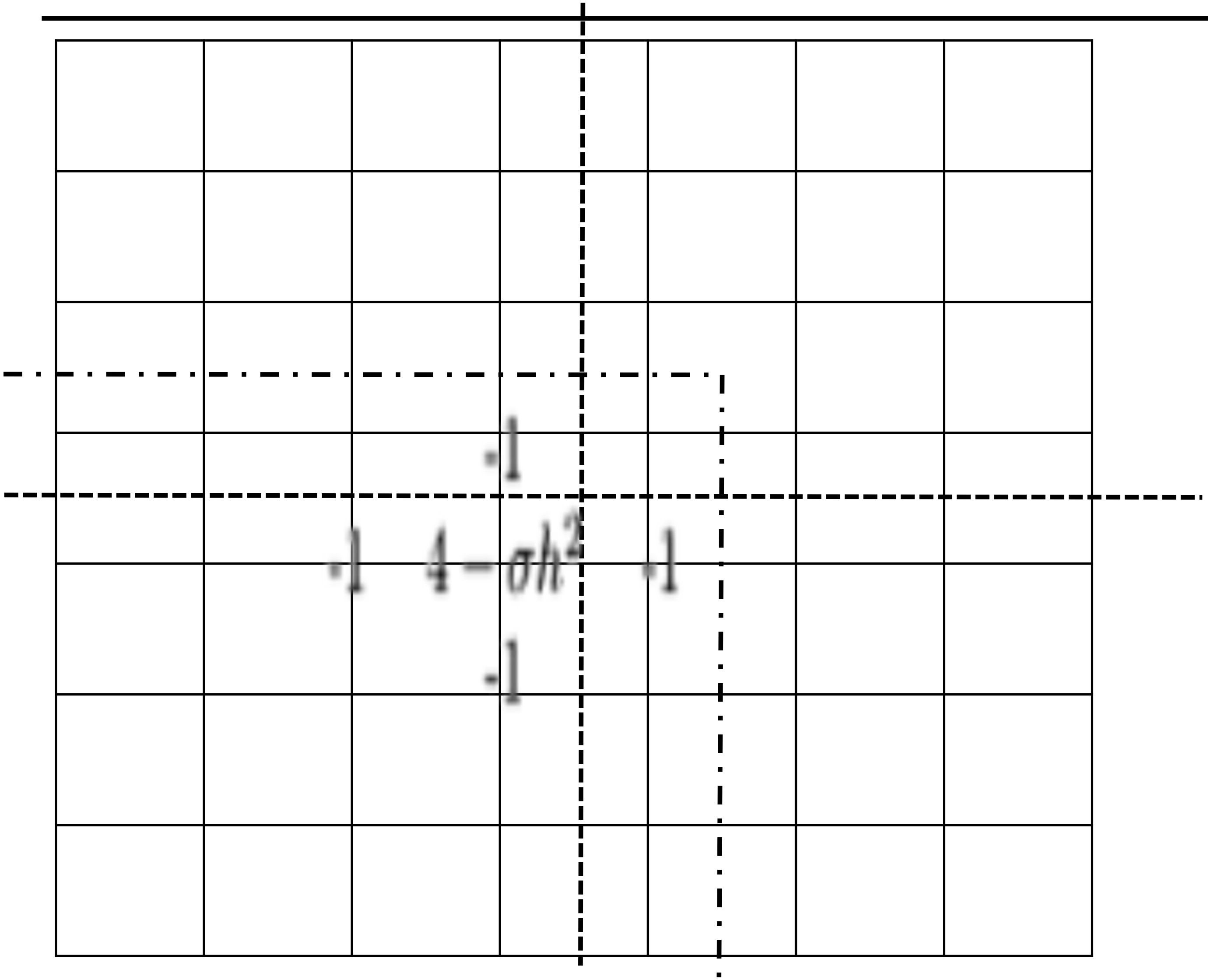
2D PDE on Regular Grid (Standard Laplace)



SPMD Patterns for Domain Decomposition

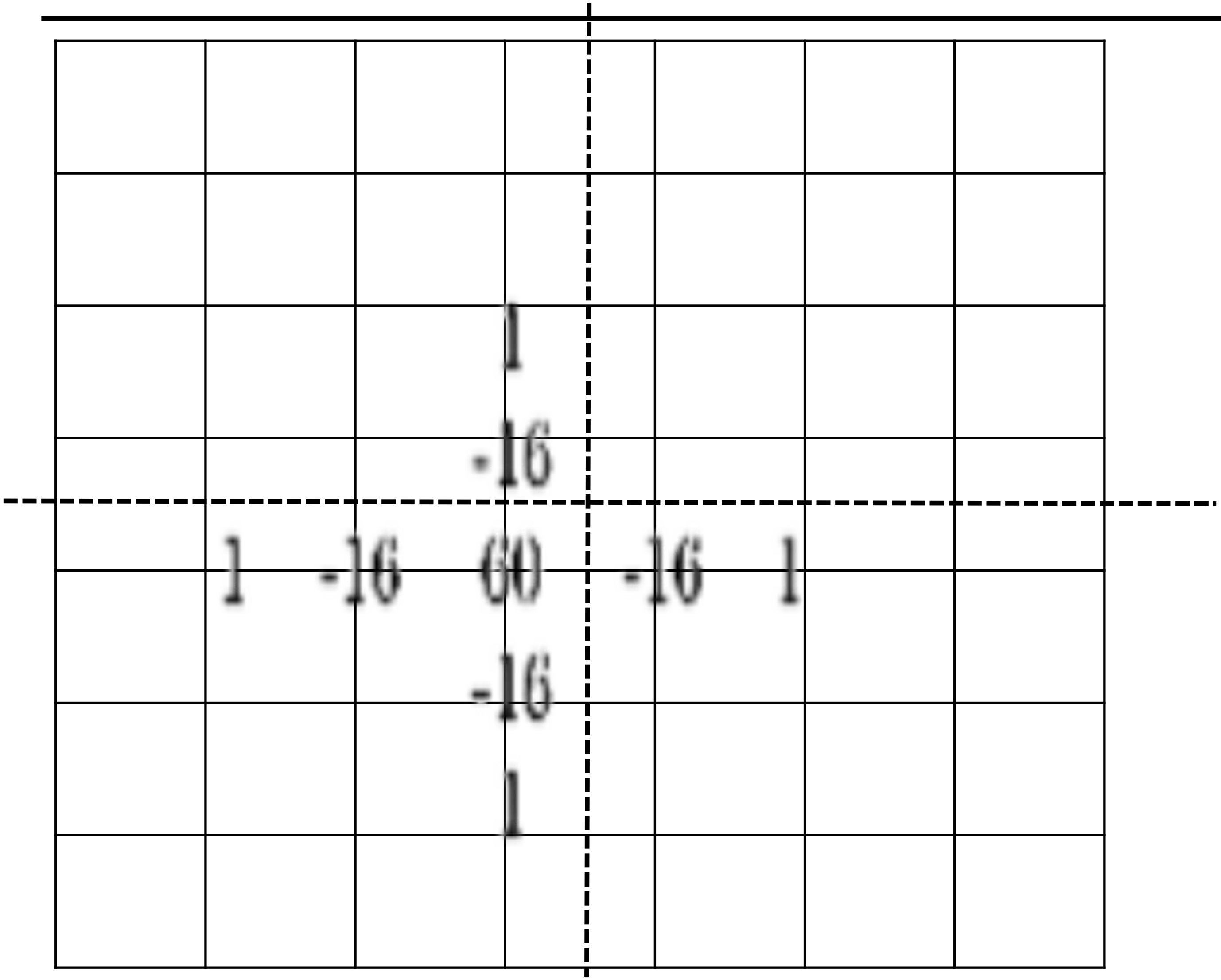
- Single Program Multiple Data (SPMD):
  - Natural fit for many differential equations.
  - All processors execute same code, different subdomains.
  - Message Passing Interface (MPI) is portability layer.
- Parallel Patterns:
  - Halo Exchange:
    - Written by parallel computing expert: Complicated code.
    - Used by domain expert: DoHaloExchange() - Conceptual.
    - Use MPI. Could be replace by PGAS, one-sided, …
  - Collectives:
    - Dot products, norms.
- All other programming:
  - Sequential!
  - Example: 5-point stencil computation is sequential.

2D PDE on Regular Grid (Helmholtz)



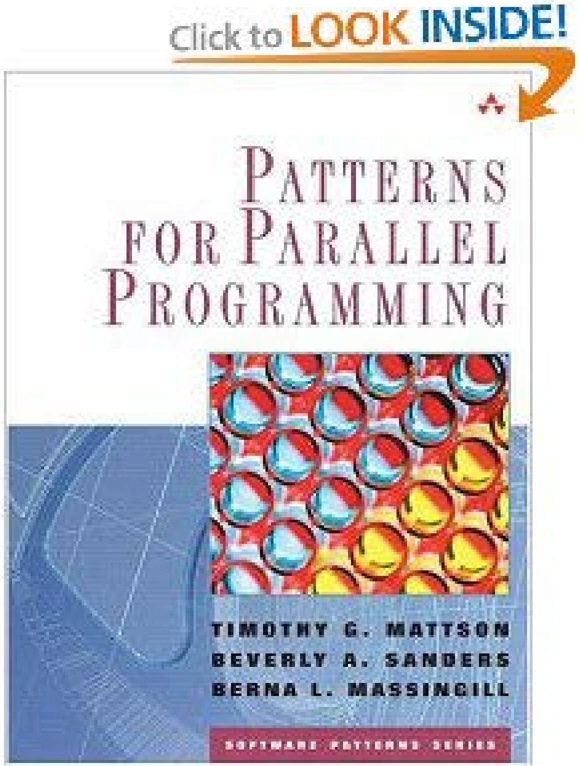
$$-\nabla u - \sigma u = f \qquad (\sigma \geq 0)$$

2D PDE on Regular Grid (4th Order Laplace)



Thinking in Patterns

- First step of parallel application design:
  - – Identify parallel patterns.
- Example: 2D Poisson (& Helmholtz!)
  - – SPMD:
    - Halo Exchange.
    - AllReduce (Dot product, norms).
  - – SPMD+X:
    - Much richer palette of patterns.
    - Choose your taxonomy.
    - Some: Parallel-For, Parallel-Reduce, Task-Graph, Pipeline.



Thinking in Parallel Patterns

- 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 >>> (…);`
- Thrust, …
- Cray Autotasking (April 1989)

```
c.....do parallel SAXPY
CMIC$ DO ALL SHARED(N, ALPHA, X, Y)
CMIC$1   PRIVATE(i)
         do 10 i = 1, n
             y(i) = y(i) + alpha*x(i)
 10      continue
```

Why Patterns

- Essential expressions of concurrency.
- Describe constraints.
- Map to many execution models.
- Example: Parallell-for.
  - Can be mapped to SIMD, SIMT, Threads, SPMD.
  - Future: Processor-in-Memory (PIM).
- Lots of ways to classify them.

Domain Scientist’s Parallel Palette

- 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.

Other construct: Thread team

- Characteristics:
  - Multiple threads.
  - Fast barrier.
  - Shared, fast access memory pool.
  - Example: Nvidia SM, Intel MIC
  - X86 more vague, emerging more clearly in future.
- Qualitatively better algorithm:
  - Threaded triangular solve scales.
  - Fewer MPI ranks means fewer iterations, better robustness.
  - Data-driven parallelism.

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.
  - MPI+X.
  - Exception: Large-scale data-intensive graph?

MPI+X Preserves Programmability

- MPI apps preserve sequential programmability via abstractions:
  - Halo exchange, app-specific collectives.
  - Domain scientists add new features: sequential code expressions.
- Most X (TBB, CUDA, OpenMP*, …) do too via patterns:
  - Parallel-for, Parallel-reduce, task graph, prefix ops, etc.
  - Basic MPI+X kernels: sequential code, mined from MPI-only code.
- Critical issues migrating to X:
  - Identifying latent node-level parallelism.
  - Identifying, replacing current, essential node-level sequentiality.
  - Isolation of computation to stateless kernels.
  - Abstraction of physics i,j,k from data structure i,j,k.
- Any beyond-MPI platform must also preserve programmability.

With C++ as your hammer,
everything looks like your thumb.

“Are C++ templates safe? No, but they are good.”

Compile-time Polymorphism

Templates and Sanity upon a shifting foundation

How can we:

- Implement mixed precision algorithms?
- Implement generic fine-grain parallelism?
- Support hybrid CPU/GPU computations?
- Support extended precision?
- Explore resilient computations?

C++ and templates most sane way.

Template Benefits:

- Compile time polymorphism.
- True generic programming.
- No runtime performance hit.
- Strong typing for mixed precision.
- Support for extended precision.
- Many more…

Template Drawbacks:

- Huge compile-time performance hit:
  - But good use of multicore :)
  - Eliminated for common data types.
- Complex notation:
  - Esp. for Fortran & C programmers.
  - Can insulate to some extent.

Resilience Problems: Already Here, Already Being Addressed, Algorithms & Co-design Are Key

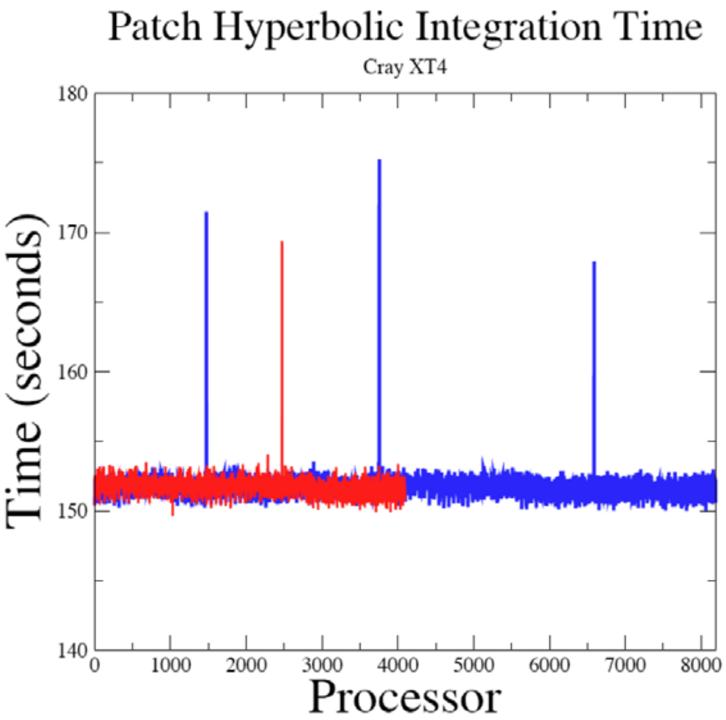
- Already impacting performance: Performance variability.
  - HW fault prevention and recovery introduces variability.
  - Latency-sensitive collectives impacted.
  - MPI non-blocking collectives + new algorithms address this.
- Localized failure:
  - Now: local failure, global recovery.
  - Needed: local recovery (via persistent local storage).
  - MPI FT features + new algorithms: Leverage algorithm reasoning.
- Soft errors:
  - Now: Undetected, or converted to hard errors.
  - Needed: Apps handle as performance optimization.
  - MPI reliable messaging + PM enhancement + new algorithms.
- *Key to addressing resilience: algorithms & co-design.*

Resilience Issues Already Here

- First impact of unreliable HW?
  - Vendor efforts to hide it.
  - Slow & correct vs. fast & wrong.

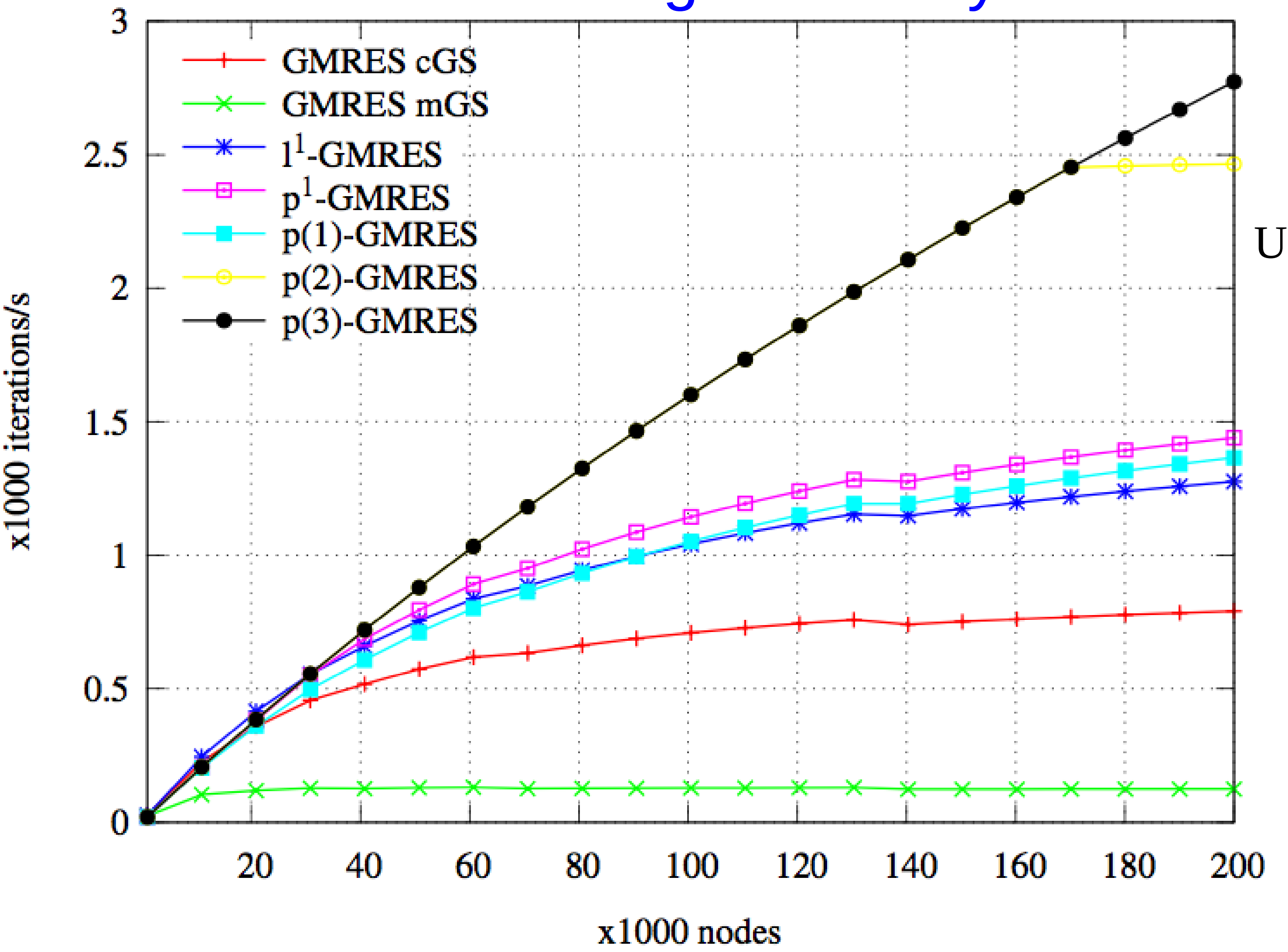
- Result:
  - Unpredictable timing.
  - Non-uniform execution across cores.

- Blocking collectives:
  - $t_c = \max_i\{t_i\}$



Brian van Straalen, DOE Exascale Research Conference, April 16-18, 2012. *Impact of persistent ECC memory faults.*

Latency-tolerant Algorithms + MPI 3: Recovering scalability



Up is good

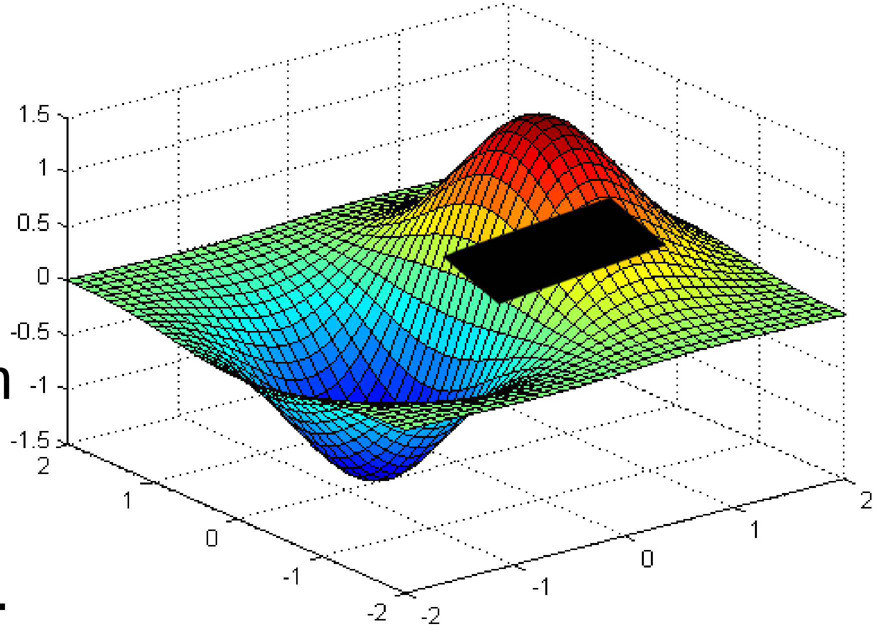
Hiding global communication latency in the GMRES algorithm on massively parallel machines,

P. Ghysels T.J. Ashby K. Meerbergen W. Vanroose, Report 04.2012.1, April 2012,

ExaScience Lab Intel Labs Europe

Enabling Local Recovery from Local Faults

- Current recovery model: Local node failure, global kill/restart.
- Different approach:
  - App stores key recovery data in persistent local (per MPI rank) storage (e.g., buddy, NVRAM), and registers recovery function.
  - Upon rank failure:
    - MPI brings in reserve HW, assigns to failed rank, calls recovery fn.
    - App restores failed process state via its persistent data (& neighbors'?).
    - All processes continue.



Local Recovery from Local Faults Advantages

- Enables fundamental algorithms work to aid fault recovery:
  - Straightforward app redesign for explicit apps.
  - Enables reasoning at approximation theory level for implicit apps:
    - What state is required?
    - What local discrete approximation is sufficiently accurate?
    - What mathematical identities can be used to restore lost state?
  - Enables practical use of many exist algorithms-based fault tolerant (ABFT) approaches in the literature.

Every calculation matters

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.

Soft Error Resilience

- 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.

M. Heroux, M. Hoemmen

Selective Reliability Enables Reasoning about Soft Errors: FT-GMRES Algorithm

```
Input: Linear system $Ax = b$ and initial guess $x_0$
  $r_0 := b - Ax_0$, $\beta := \|r_0\|_2$, $q_1 := r_0/\beta$
  for $j = 1, 2, \ldots$ until convergence do
      Inner solve: Solve for $z_j$ in $q_j = Az_j$
      $v_{j+1} := Az_j$
      for $i = 1, 2, \ldots, k$ do                          ▷ Orthogonalize $v_{j+1}$
          $H(i,j) := q_i^* v_{j+1}$, $v_{j+1} := v_{j+1} - q_i H(i,j)$
      end for
      $H(j+1,j) := \|v_{j+1}\|_2$
      Update rank-revealing decomposition of $H(1{:}j, 1{:}j)$
      if $H(j+1,j)$ is less than some tolerance then
          if $H(1{:}j, 1{:}j)$ not full rank then
              Try recovery strategies
          else
              Converged; return after end of this iteration
          end if
      else
          $q_{j+1} := v_{j+1}/H(j+1,j)$
      end if
      $y_j := \operatorname{argmin}_y \|H(1{:}j+1, 1{:}j)y - \beta e_1\|_2$    ▷ GMRES projected problem
      $x_j := x_0 + [z_1, z_2, \ldots, z_j]y_j$              ▷ Solve for approximate solution
  end for
```

"Unreliably" computed.

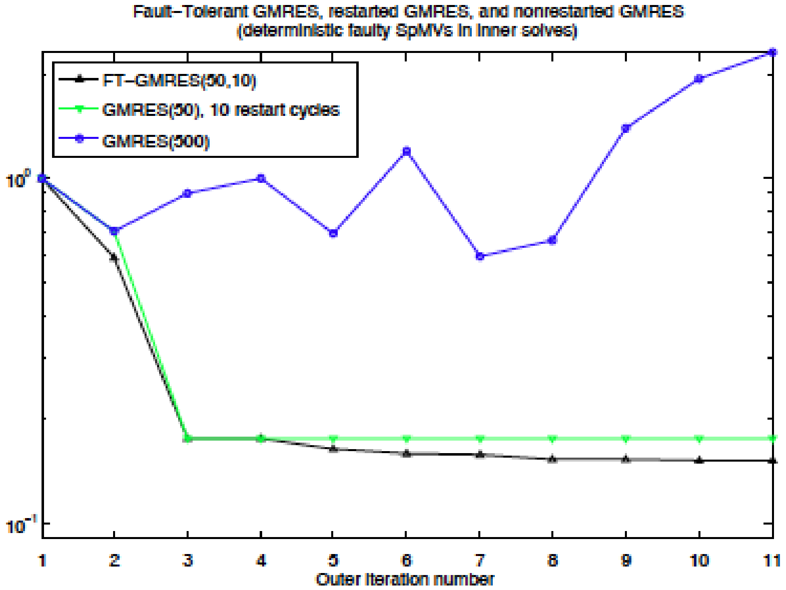
Majority of computational cost.

Captures true linear operator issues, AND

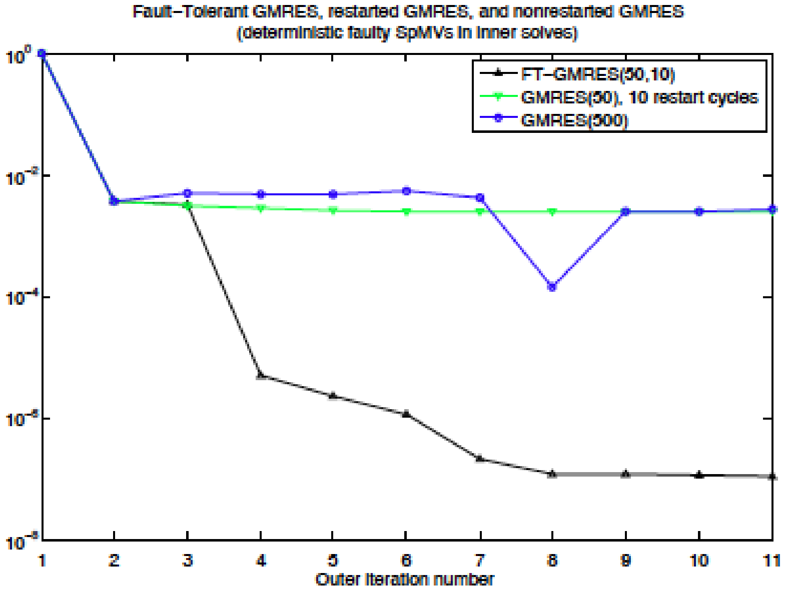
Can use some "garbage" soft error results.

Selective reliability enables “running through” faults

- FT-GMRES can run through faults and still converge.
- Standard GMRES, with or without restarting, cannot.



FT-GMRES vs. GMRES on Ill_Stokes (an ill-conditioned discretization of a Stokes PDE).



FT-GMRES vs. GMRES on mult_dcop_03 (a Xyce circuit simulation problem).

Summary

- Node-level parallelism is the new commodity curve:
  - Tasks, threads, vectors.
- Domain experts need to “think” in parallel.
  - Building a parallel pattern framework is an effective approach.
- Most future programmers won’t need to write parallel code.
  - Pattern-based framework separates concerns (parallel expert).
  - Domain expert writes sequential fragment. (Even if you are both).
- Fortran can be used for future parallel applications, but:
  - Complex parallel patterns are very challenging (impossible).
  - Parallel features lag, lack of compile-time polymorphism hurts.
- Resilience is a major front in extreme-scale computing.
  - Resilience with current algorithms base is not feasible.
  - Need algorithms-driven resilience efforts.

Summary

- MPI+X is and will be dominant platform for tera and peta scale.
- MPI+X will be a (dominant) platform for exascale:
  - Natural fit for many science & engineering apps.
  - Hierarchical composition matches tera, peta and exascale.
  - Naturally leverages industry efforts.
- Ongoing efforts needed in MPI to address emerging needs.
  - New MPI features address most important exascale concerns.
  - Co-design from discretizations to low-level HW enables resilience.
- Migrating to emerging industry X platforms: Critical, urgent.
  - Good preparation for beyond MPI:
    - Isolation of computation to stateless kernels.
    - Abstraction of data layout.
  - Requires investment outside of day-to-day apps efforts.
  - Essential now for near-term manycore success.

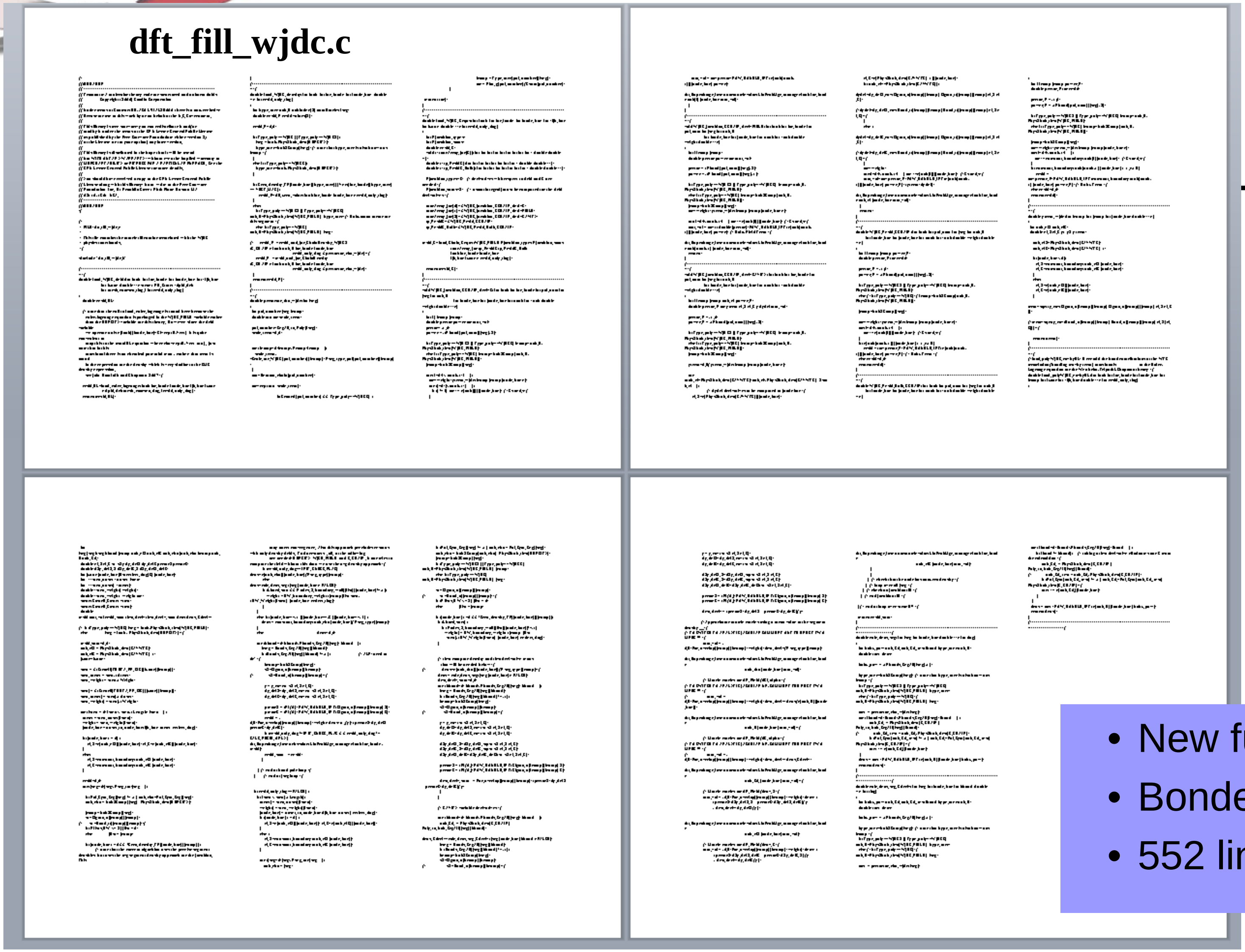
Extra Slides

Notable New MPI Features

- Non-blocking collectives #109.
- Neighborhood collectives (aka, sparse) #258.
- Updated One-sided features #270.
- Shared memory window #284.
- Noncollective Comm Creation #286.
- Nonblocking Comm Dup #168.
- Fault-tolerance.
- …

http://www.unixer.de/blog/index.php/2012/02/06/mpi-3-0-is-coming-an-overview-of-new-and-old-features
Torsten Hoefler Blog

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.**

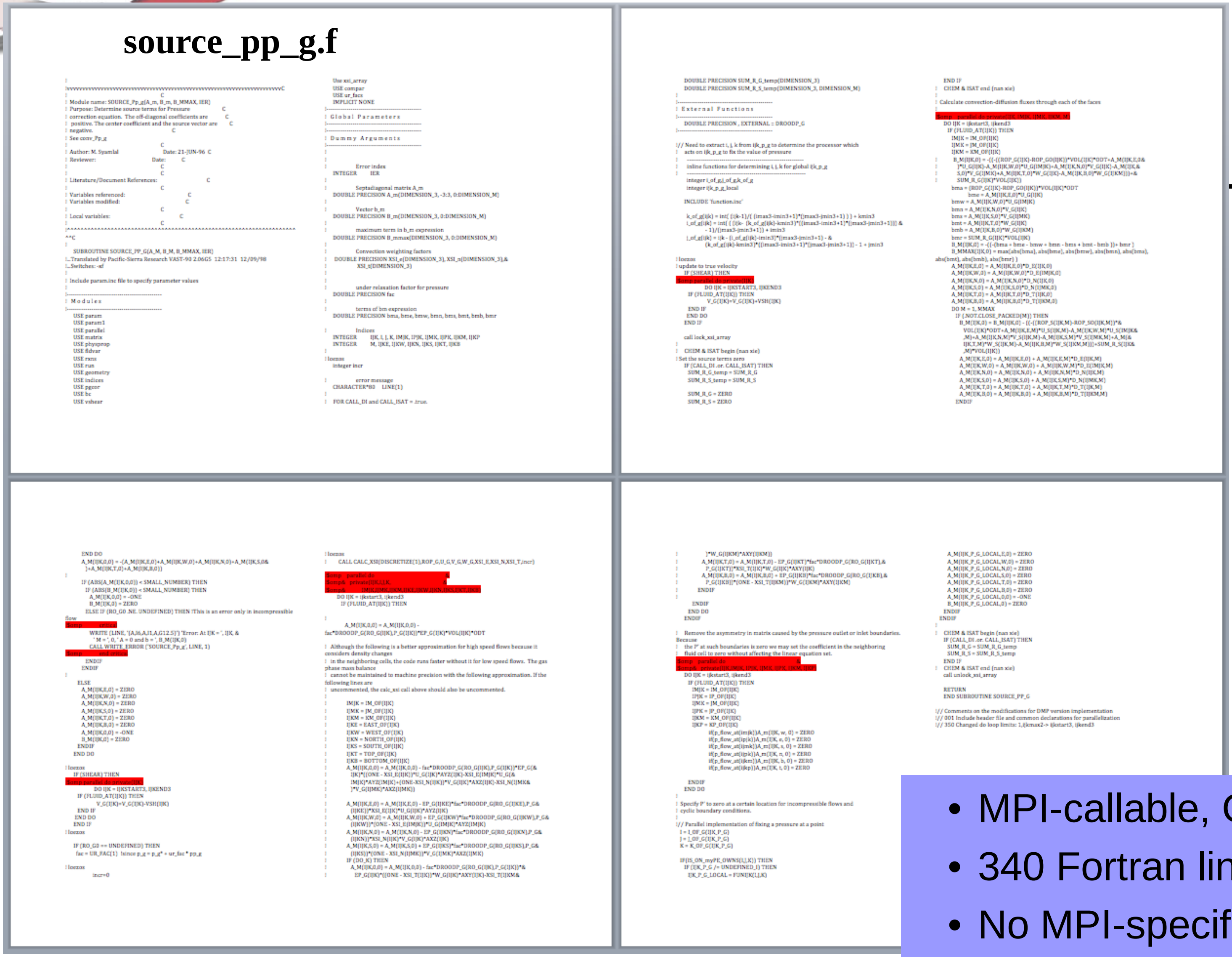
How much MPI-specific code?

dft_fill_wjdc.c MPI-specific code

MFIX

Source term for pressure correction

source_pp_g.f

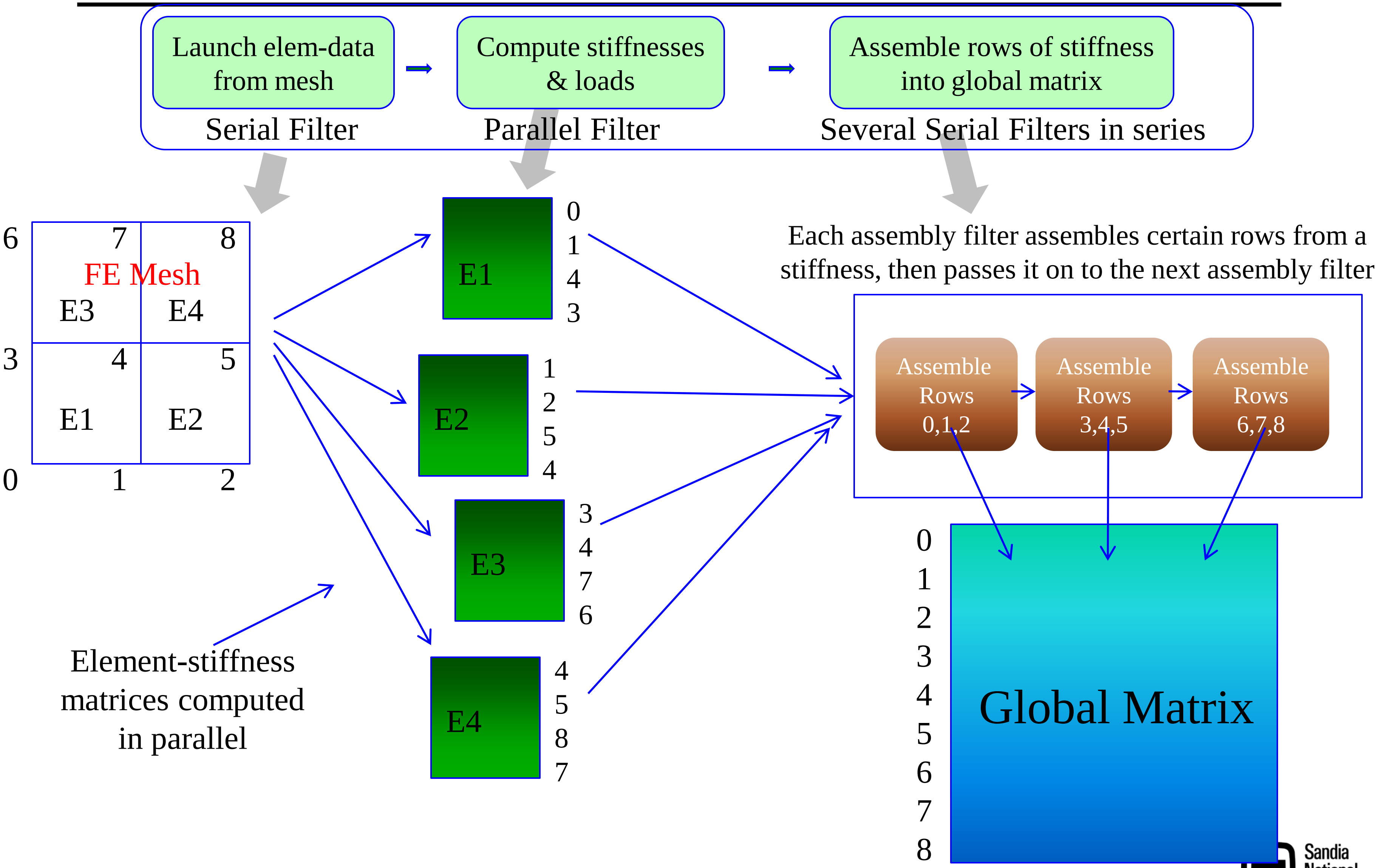


- MPI-callable, OpenMP-enabled.
- 340 Fortran lines.
- No MPI-specific code.
- Ubiquitous OpenMP markup (red regions).

MFIX: Multiphase Flows with Interphase eXchanges (https://www.mfix.org/)

Sandia National Laboratories

TBB Pipeline for FE assembly



Launch elem-data from mesh → Compute stiffnesses & loads → Assemble rows of stiffness into global matrix

Serial Filter

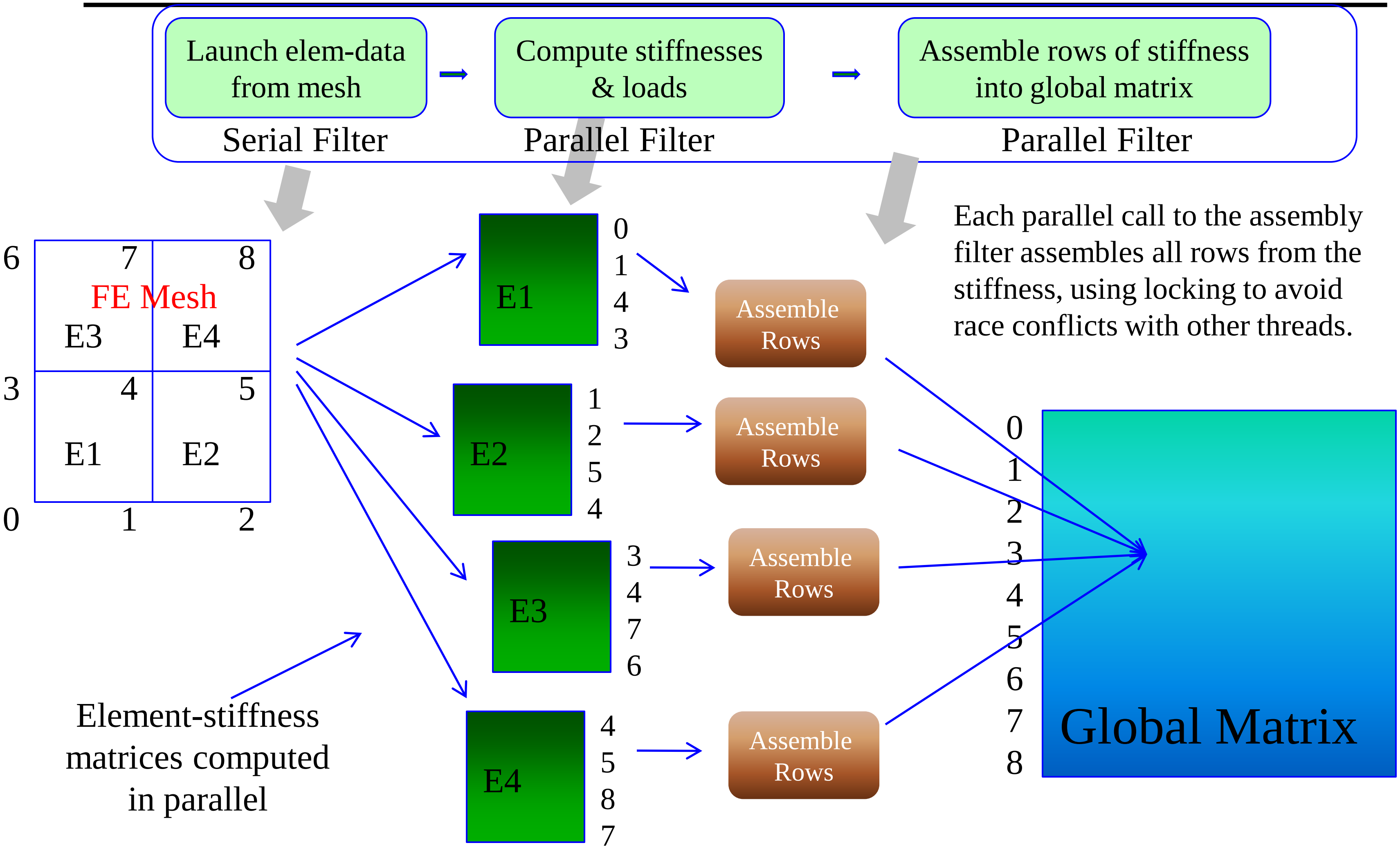
Parallel Filter

Several Serial Filters in series

Each assembly filter assembles certain rows from a stiffness, then passes it on to the next assembly filter

Element-stiffness matrices computed in parallel

Alternative
TBB Pipeline for FE assembly



Launch elem-data from mesh → Compute stiffnesses & loads → Assemble rows of stiffness into global matrix

Serial Filter — Parallel Filter — Parallel Filter

Each parallel call to the assembly filter assembles all rows from the stiffness, using locking to avoid race conflicts with other threads.

Element-stiffness matrices computed in parallel

Finite Elements/Volumes/Differences and parallel node constructs

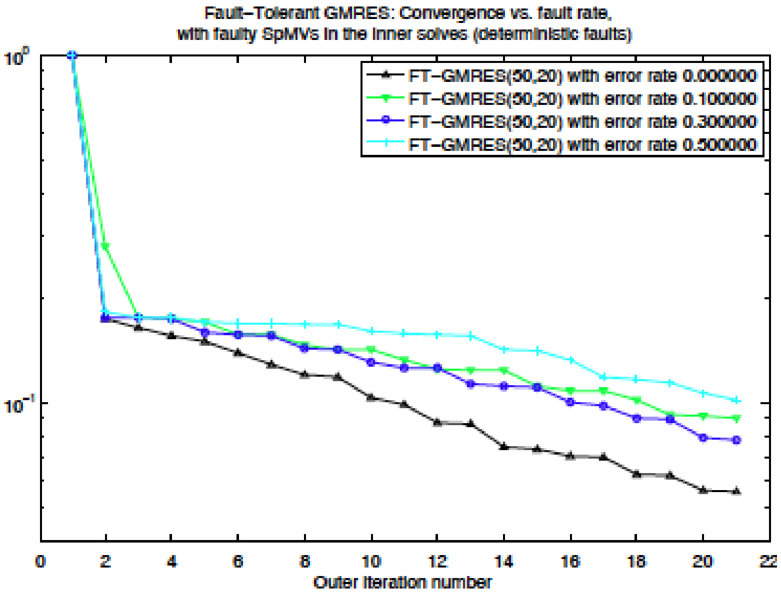
- Parallel for, reduce, pipeline:
  - Sufficient for vast majority of node level computation.
  - Supports:
    - Complex modeling expression.
    - Vanilla parallelism.
  - Must be “stencil-aware” for temporal locality.
- Thread team:
  - Complicated.
  - Requires true parallel algorithm knowledge.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

Reliability Model

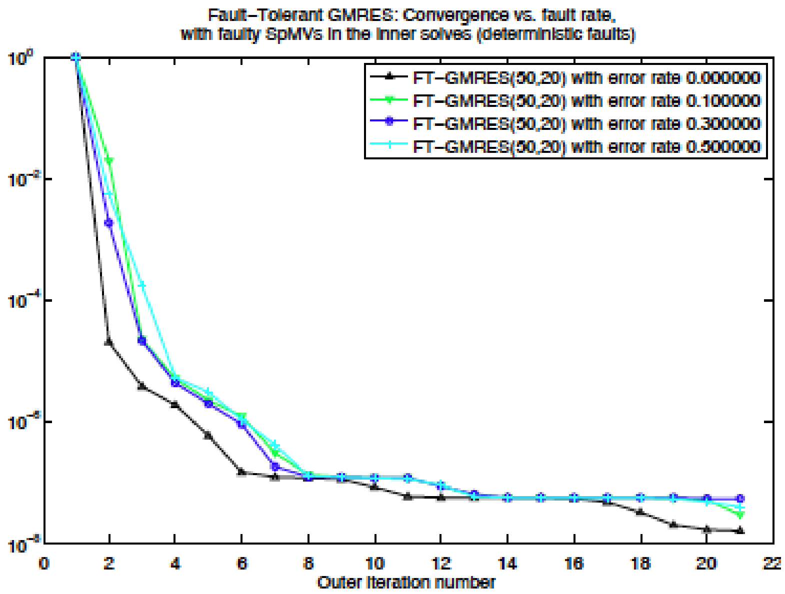
- Can’t reason about code behavior without a model
- Current model: “Fail-stop”
  - System tries to detect all soft faults
  - *Turn all detected soft faults into hard faults*
- Our basic model: “Sandbox”
  - Isolate unreliable computation in a box
  - Reliable code invokes box as a function
- Additional desired features of a model
  - Detection: report faults to application
  - Transience: refresh / recompute unreliable data periodically
  - Embed into type system: compiler can help you reason
- Our challenge goal:
    - *Turn all detected hard faults into soft faults*

Gradual Convergence Degradation

- Empirical observation: FT-GMRES convergence slows gradually as fault rate increases.



FT-GMRES on Ill_Stokes problem, with different fault rates in inner solves' SpMVs.

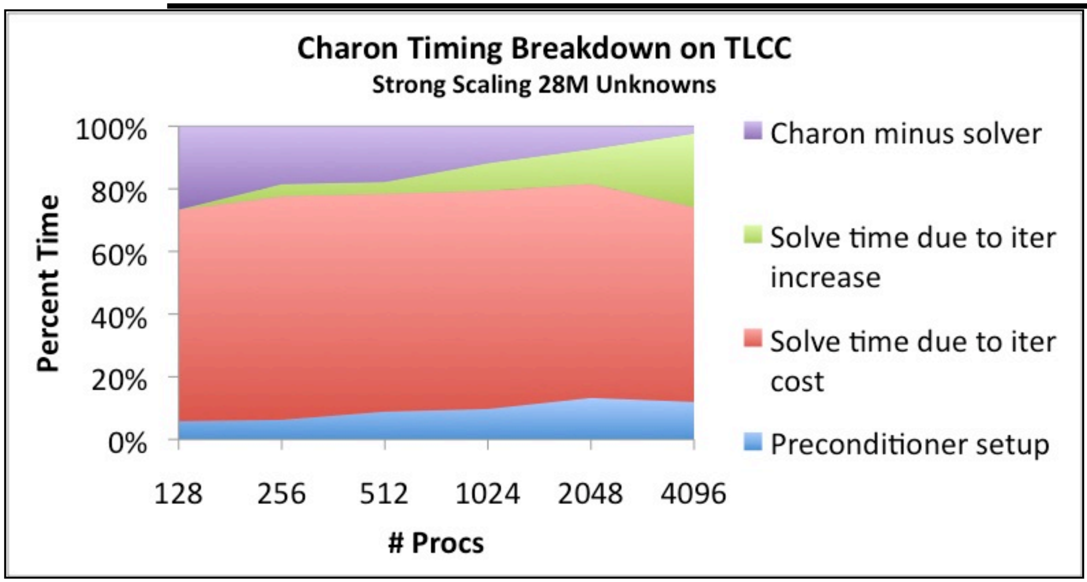


FT-GMRES on mult_dcop_03 problem, with different fault rates in inner solves' SpMVs.

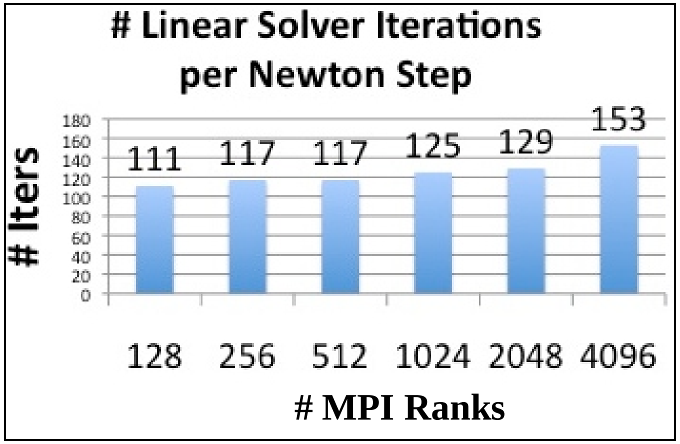
Selective Reliability Programming

- Standard approach:
  - System over-constrains reliability
  - “Fail-stop” model
  - Checkpoint / restart
  - Application is ignorant of faults
- New approach:
  - System lets app control reliability
  - Tiered reliability
  - “Run through” faults
  - App listens and responds to faults

Challenges for Coarse Grain Dynamic Parallelism

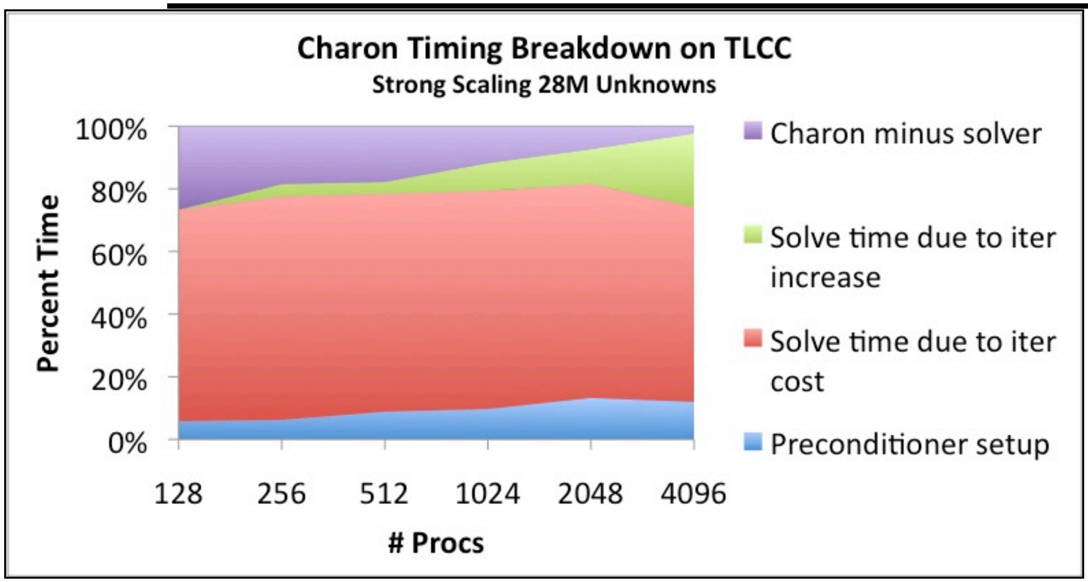


Strong scaling of Charon on TLCC (P. Lin, J. Shadid 2009)

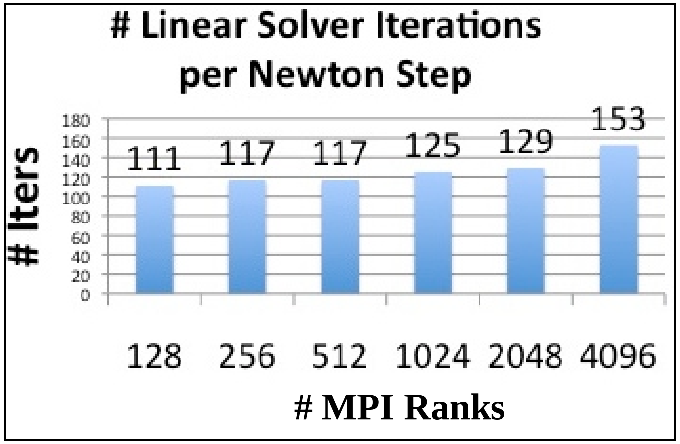


- Observe: Iteration count increases with number of subdomains.
- Dynamic parallelism implies over-decomposing.
- Example:
  - 4X over-decomposition, 1024 processors.
  - 20% increase in aggregate computational cost (125 iters becomes 153).
  - Can dynamic execution overcome this?
- Coarse grain dynamic parallelism degrades robustness!

Opportunities for Fine Grain Dynamic Parallelism



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



- 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

Factors Impacting Performance of Multithreaded Sparse Triangular Solve, **Michael M. Wolf and Michael A. Heroux and Erik G. Boman, VECPAR 2010.**