



System Software: A Necessary but Ill-prepared Hero

Michael A. Heroux
Computing Research Division
Sandia National Laboratories

Collaborators: Erik Boman, Irina Demeshko, Carter Edwards, James Elliot, Mark Hoemmen, Siva Rajamanickam, Keita Teranishi, Christian Trott, Alan Williams



Scope of Consideration

- All Software Except:
 - Application itself.
 - Domain specific libraries: Math, Physics, etc.
- Includes:
 - Threading support.
 - Programming Environments.

Non-incremental Efforts

- Use of Futures:
 - ◆ Exploit previously inaccessible, fine-grain dynamic parallelism.
 - ◆ Natural framework for expressing data-driven parallelism.
- Better than MPI:
 - ◆ Beyond functional mimic of MPI.
 - ◆ AGAS: Truly adaptive mesh refinement.
- Overarching goal: Show that non-incremental approaches
 - ◆ Work.
 - ◆ Superior to MPI+X in one or more metric:
 - Performance: Extracting latent parallelism.
 - Portability: Performance obtained from system's underlying runtime.
 - Productivity: Easier to write, understand, maintain.
- Won't talk about this effort today.

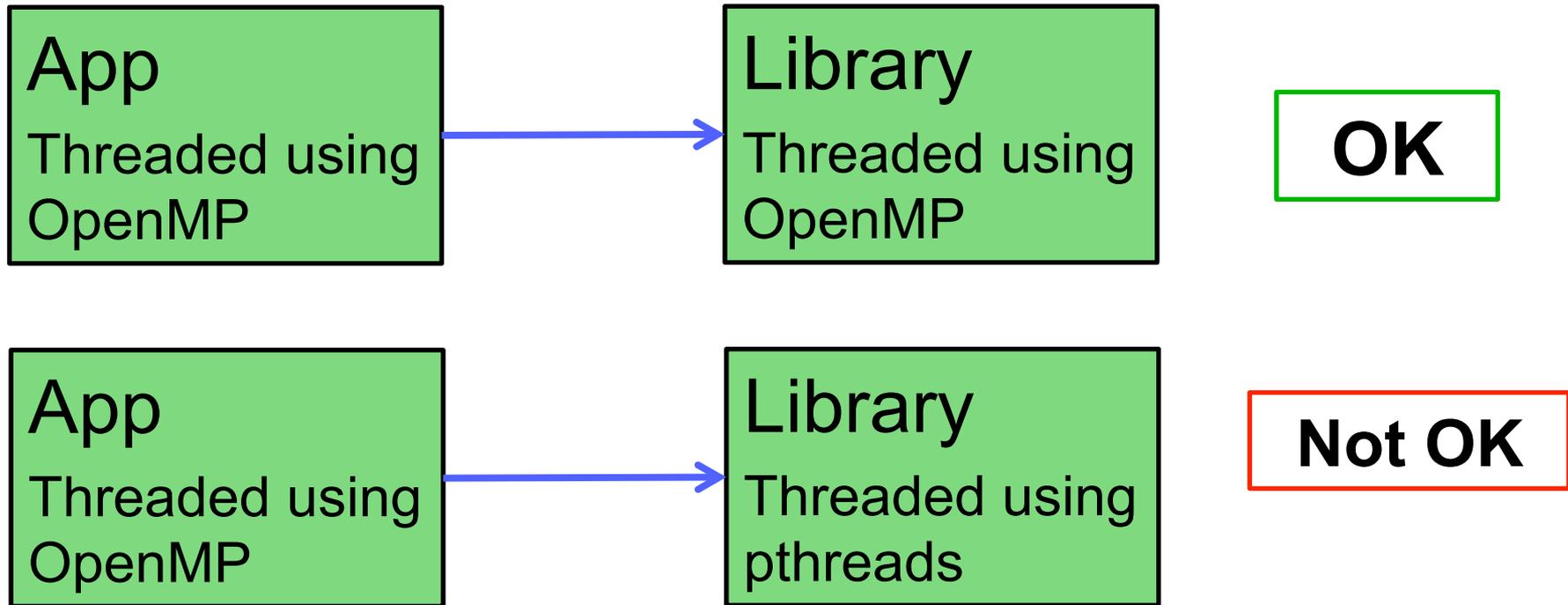


General Reality of Multicore Parallelism

- Best single shared memory parallel programming environment:
 - MPI.
- But:
 - Two level parallelism (MPI+X) is generally more effective.
- But, the best option for X (if explored at all) is:
 - MPI.
- Furthermore, for an $(N_1 \times N_2)$ MPI+X decomposition:

“For a given number of core counts, the best performance is achieved with the smallest possible N_2 for both hybrid [MPI +OpenMP] and MPI [MPI+MPI] versions. As N_2 increases, the runtime also increases.”

Threading Multi-API Usage: Needs to work



- Problem: App uses all threads in one phase, library in another phase.
- Intel Sandy Bridge: **Not OK** 1.16 to 2.5 X slower than **OK** .
- Intel Phi: **Not OK** 1.33 to 1.8 X slower than **OK** .
- Implication:
 - Libraries must pick an API.
 - Or support many. Possible, but complicated.



Data Placement & Alignment

- First Touch is not sufficient.
 - Happens at initialization. Hard to locate, control.
- Really need placement as first class concept.
 - Create object with specific layout.
 - Create objects compatible with existing object.
- Lack of support limits MPI+OpenMP.
 - OpenMP restricted to single UMA core set.



Nvidia GPUs

- Supports mixed environments: All that it handles.
- Has good performance model support.
- Has flexible data placement model.
- C++ support is good, waiting for lambdas.
- Severe environment, but results in general goodness.



OpenMP 4.0, OpenACC

- Active, addressing highest priority requirements.
- Incompatible, even conceptually.
- Best hope: Compiler recognizes both.



Ill-prepared but moving in the right direction

- Features are coming, but slowly.
- Performance models coming too.
- Happy to be a C++ developer.
 - Fortran support always lags.
 - Fortran features arrive a decade late.
- Missing piece: Higher level APIs. More in a bit.

Resilience & RT/OS Interactions



Four Resilient Programming Models

- Skeptical Programming. (SP)
- Relaxed Bulk Synchronous (rBSP)
- Local-Failure, Local-Recovery (LFLR)
- Selective (Un)reliability (SU/R)

Toward Resilient Algorithms and Applications

Michael A. Heroux

arXiv:1402.3809v2 [cs.MS]

Skeptical Programming

I might not have a reliable digital machine

- Expect rare faulty computations
- Use analysis to derive cheap “detectors” to filter large errors
- Use numerical methods that can absorb *bounded error*

Algorithm 1: GMRES algorithm

```
for  $l = 1$  to do
   $\mathbf{r} := \mathbf{b} - \mathbf{A}\mathbf{x}^{(j-1)}$ 
   $\mathbf{q}_1 := \mathbf{r} / \|\mathbf{r}\|_2$ 
  for  $j = 1$  to restart do
     $\mathbf{w}_0 := \mathbf{A}\mathbf{q}_j$ 
    for  $i = 1$  to  $j$  do
       $h_{i,j} := \mathbf{q}_i \cdot \mathbf{w}_{i-1}$ 
       $\mathbf{w}_i := \mathbf{w}_{i-1} - h_{i,j}\mathbf{q}_i$ 
    end
     $h_{j+1,j} := \|\mathbf{w}\|_2$ 
     $\mathbf{q}_{j+1} := \mathbf{w} / h_{j+1,j}$ 
    Find  $\mathbf{y} = \min \|\mathbf{H}_j\mathbf{y} - \|\mathbf{b}\| \mathbf{e}_1\|_2$ 
    Evaluate convergence criteria
    Optionally, compute  $\mathbf{x}_j = \mathbf{Q}_j\mathbf{y}$ 
  end
end
```

GMRES

Theoretical Bounds on the Arnoldi Process

$$\|\mathbf{w}_0\| = \|\mathbf{A}\mathbf{q}_j\| \leq \|\mathbf{A}\|_2 \|\mathbf{q}_j\|_2$$

$$\|\mathbf{w}_0\| \leq \|\mathbf{A}\|_2 \leq \|\mathbf{A}\|_F$$

From isometry of orthogonal projections,

$$|h_{i,j}| \leq \|\mathbf{A}\|_F$$

- h_{ij} form Hessenberg Matrix
- Bound only computed once, valid for entire solve

Evaluating the Impact of SDC in Numerical Methods

J. Elliott, M. Hoemmen, F. Mueller, SC'13



What is Needed for Skeptical Programming?

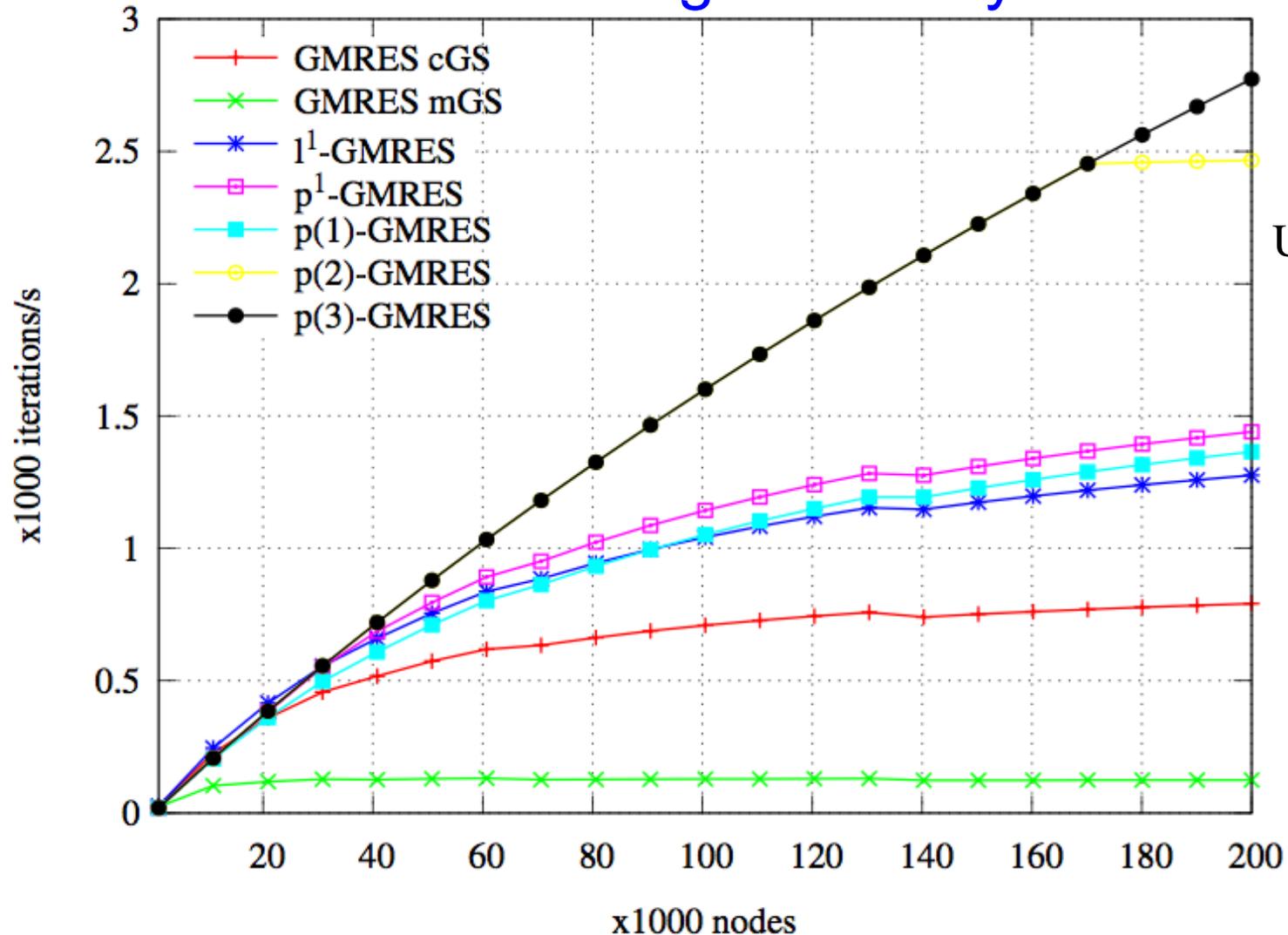
- Skepticism.
- Meta-knowledge:
 - Algorithms,
 - Mathematics,
 - Problem domain.
- Nothing else, at least to get started.

- RT/OS interactions:
 - Nothing first order.
 - Still worth mentioning.

Performance Variability is a Resilience Issue

- 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:
 - $t_c = \max_i \{t_i\}$
 - Also called “Limpware”:
 - Haryadi Gunawi, University of Chicago
 - <http://www.anl.gov/events/lights-case-limping-hardware-tolerant-systems>
- Ideal:
equal work +
equal data access =>
equal execution time.
 - Reality:
 - Lots of variation.
 - Variations increasing.

Latency-tolerant Algorithms + MPI 3: Recovering scalability



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

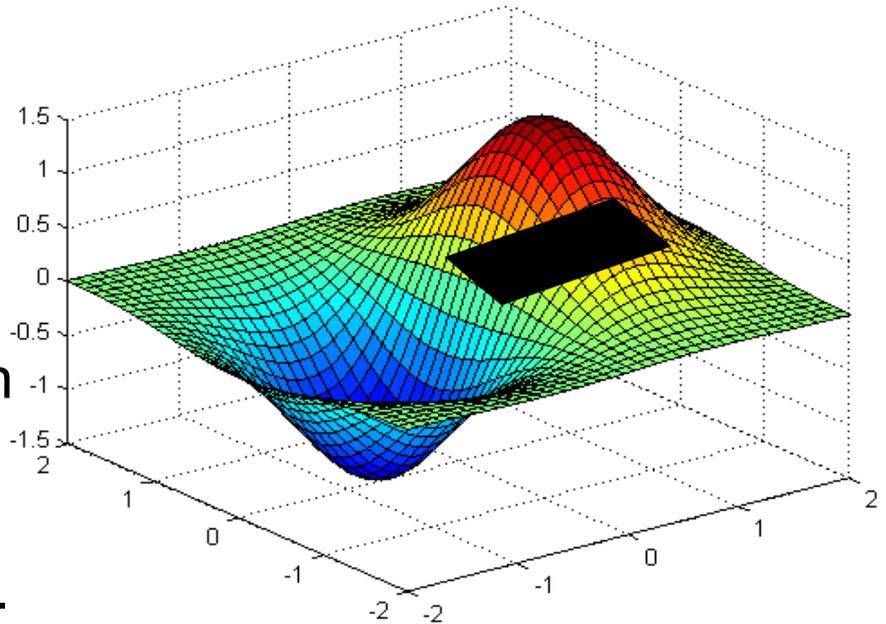


What is Needed to Support Latency Tolerance?

- MPI 3 (SPMD):
 - Asynchronous global and neighborhood collectives.
- A “relaxed” BSP programming model:
 - Start a collective operation (global or neighborhood).
 - Do “something useful”.
 - Complete the collective.
- The pieces are coming online.
- With new algorithms we can recover some scalability.
- **RTI/OS implications:**
 - Success gives you a license to detect/correct more errors.**

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.





LFLR Algorithm Opportunities & Challenges

- 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.



What is Needed for Local Failure Local Recovery (LFLR)?

- LFLR realization is non-trivial.
- Programming API (but not complicated). ULFM helps.
- Lots of runtime/OS infrastructure.
 - Persistent storage API (frequent brainstorming outcome).
- Research into messaging state and recovery? No.
- New algorithms, apps re-work.
- But:
 - Can leverage global CP/R logic in apps.
- This approach is the next natural step in beyond CP/R.



Requirements for LFLR

- If a process dies, don't kill the rest.
- Ability to query who is alive among the MPI processes.
- Persistent access to the seminal state of the lost process.
- Support to re-constitute the lost local state:
 - This is the app's responsibility, but may require some assistance from system.
- System resources to execute recovered process.
 - Can be a spare process.
 - If over-decomposed, can distribute to work across existing processes.
 - Nonfunctional requirements:
 - Scalable implementations of the above functions.
- Current status:
 - ULFM provides message passing support.
 - We provide *ad hoc* persistent storage solution.

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.

Fault-tolerant linear solvers via selective reliability,

Patrick G. Bridges, Kurt B. Ferreira,
Michael A. Heroux, Mark Hoemmen
arXiv:1206.1390v1 [math.NA]

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 $j = 1, 2, \dots$ until convergence **do**

Inner solve: Solve for z_j in $q_j = Az_j$

$v_{j+1} := Az_j$

for $i = 1, 2, \dots, k$ **do**

$H(i, j) := q_i^* v_{j+1}$, $v_{j+1} := v_{j+1} - q_i H(i, j)$

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

$y_j := \operatorname{argmin}_y \|H(1:j+1, 1:j)y - \beta e_1\|_2$ \triangleright GMRES projected problem

$x_j := x_0 + [z_1, z_2, \dots, z_j]y_j$ \triangleright Solve for approximate solution

end for

“Unreliably” computed.

Standard solver library call.

Majority of computational cost.

\triangleright Orthogonalize v_{j+1}

Captures true linear operator issues, AND
Can use some “garbage” soft error results.



What is Needed for Selective Reliability?

- A lot, lot.
- A programming model.
 - Expressing data/code reliability or unreliability.
- Algorithms.
 - Basic approaches:
 - Nest an unreliable algorithm in a reliable version of the same.
 - Dispatch unreliable task subgraph from reliable graph node.
- Lots of runtime/OS infrastructure.
 - Provision of reliable data, paths, execution.
 - Portable interfaces to HW solutions.
- Hardware support?
 - Special HW components that are
 - slower and more reliable or
 - faster and less reliable



Resilient Application Programming

- Standard approach:

- System over-constrains reliability
- “Fail-stop” model
- Checkpoint / restart
- Application is ignorant of faults

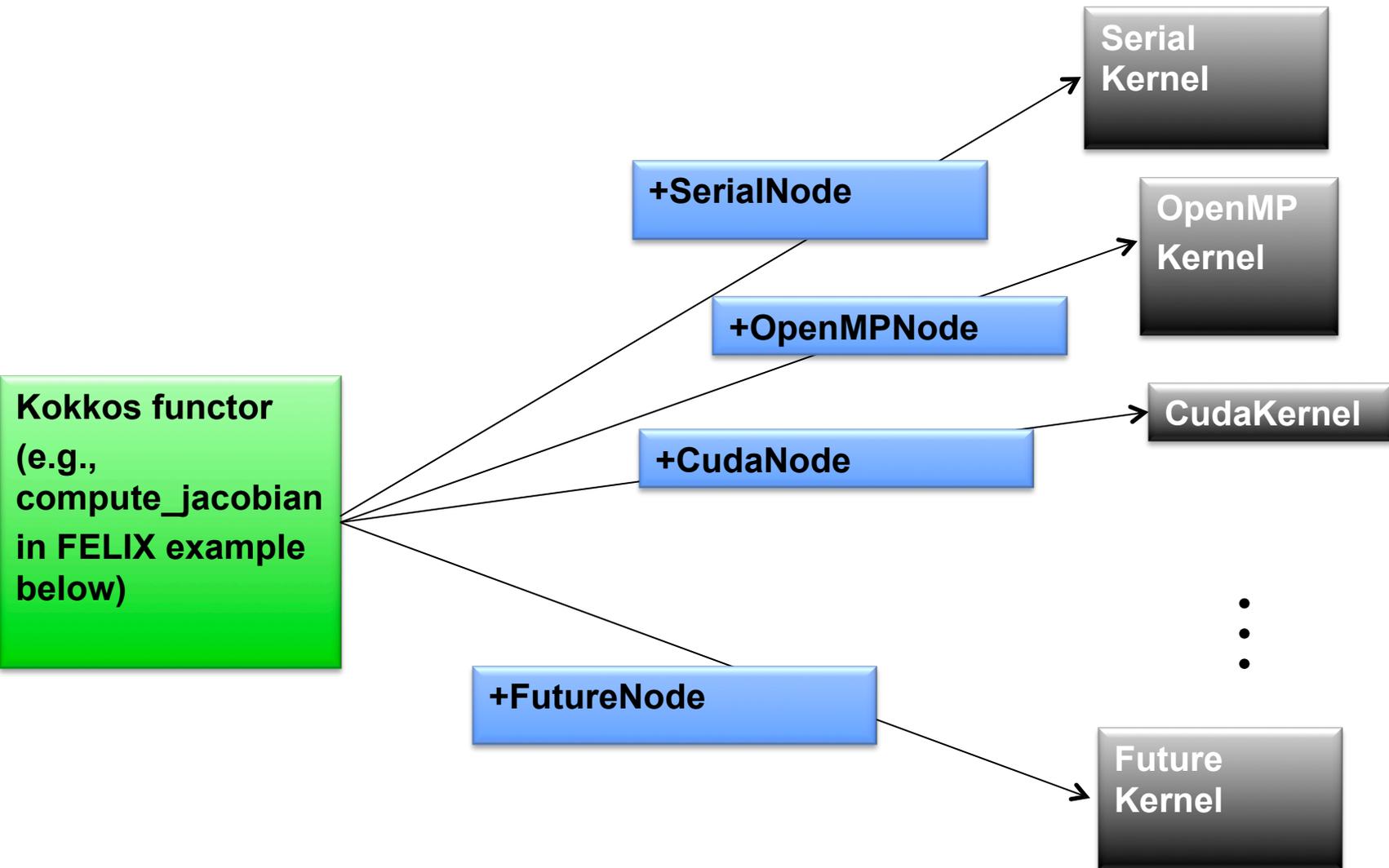
- New approach:

- System lets app control reliability
- Tiered reliability
- “Run through” faults
- App listens and responds to faults



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

Compile-time Polymorphism



Kokkos_functor example: compute_jacobian

```
for(int cell = 0; cell < worksetNumCells; cell++) {  
  for(int qp = 0; qp < numQPs; qp++) {  
    for(int row = 0; row < numDims; row++){  
      for(int col = 0; col < numDims; col++){  
        for(int node = 0; node < numNodes; node++){  
          jacobian(cell, qp, row, col) +=  
  
            coordVec(cell, node, row)  
              *basisGrads(node, qp, col);  
  
        } // node  
      } // col  
    } // row  
  } // qp  
} // cell
```

```
Kokkos::parallel_for ( worksetNumCells,  
  compute_jacobian<ScalarT, Device, numQPs, numDims,  
  numNodes> (basisGrads, jacobian, coordVec));
```

```
template < typename ScalarType, class DeviceType, int numQPs_,  
           int numDims_, int numNodes_ >  
  
class compute_jacobian {  
  Array3 basisGrads_;  
  Array4 jacobian_;  
  Array3_const coordVec_;  
public:  
  typedef DeviceType device_type;  
  compute_jacobian(Array3 &basisGrads, Array4 &jacobian,  
                  Array3 &coordVec)  
    : basisGrads_(basisGrads)  
    , jacobian_(jacobian)  
    , coordVec_(coordVec){}  
  
  KOKKOS_INLINE_FUNCTION  
  void operator () (const std::size_t i) const  
  {  
    for(int qp = 0; qp < numQPs_; qp++) {  
      for(int row = 0; row < numDims_; row++){  
        for(int col = 0; col < numDims_; col++){  
          for(int node = 0; node < numNodes_; node++){  
            jacobian_(i, qp, row, col) += coordVec_(i, node, row)  
              *basisGrads_(node, qp, col);  
  
            } // node  
          } // col  
        } // row  
      } // qp  
    }  
  }  
};
```

Node API Futures: MPI Analogy

- `ls Trilinos/packages/aztecoo/src/md_wrap_*`

```
md_wrap_intel_c.c md_wrap_ncube_c.c md_wrap_scalar_c.c  
md_wrap_mpi_c.c md_wrap_puma_c.c md_wrap_sp2_c.c
```

- Excerpt from `md_wrap_sp2_c.c` (next slide).

```
int md_write(char *buf, int bytes, int dest, int type, int *flag)
{
    int err, buffer;

    if (bytes == 0) {
        err = mpc_bsend(&buffer, 1, dest, type);
    } else {
        err = mpc_bsend(buf, bytes, dest, type);
    }
    if (err!=0) (void) fprintf(stderr, "mpc_bsend error = %d\n", mperrno);
    return 0;
}
```

Original md_write function (prior to MPI)

```

int md_write(char *buf, int bytes, int dest, int type, int *flag)
{
#if defined (MPL)
    int err, buffer;

    if (bytes == 0) {
        err = mpc_bsend(&buffer, 1, dest, type);
    } else {
        err = mpc_bsend(buf, bytes, dest, type);
    }
    if (err!=0) (void) fprintf(stderr, "mpc_bsend error = %d\n", mperrno);
#elif defined (MPI)
    int err, buffer;
    if (bytes == 0) {
        err = MPI_Send(&buffer, 1, MPI_BYTE, dest, type, MPI_COMM_WORLD);
    } else {
        err = MPI_Send(buf, bytes, MPI_BYTE, dest, type, MPI_COMM_WORLD);
    }
    if (err != 0) (void) fprintf(stderr, "MPI_Send error = %d\n", err);
#endif
    return 0;
}

```

All node APIs are transitional...

- Except the one(s) that becomes the standard.
- Recommended activities:
 - ◆ Write a light-weight API:
 - Compile-time polymorphism.
 - Match your needs and nothing more.
 - Wrap other functionality: CUDA, OpenMP, OpenCL, pthreads, ...
 - Don't fall in love with your implementation!
 - Examples: OCCA, Mint, Kokkos, RAJA, and more.
 - ◆ Participate in standards committees.
 - ◆ Prepare for eventual replacement by a standard.



The Programming Language War is Over

- There is only one Parallel Programming Language of the Future: C++
 - Supports all the programming models we care about.
 - With obvious extensions: MPI, CUDA, OpenMP, OpenACC, ...
 - Ability to extend the language is powerful:
 - Kokkos::parallel_for becomes std::parallel_for
 - Movement to std:: transfers ownership to compiler and runtime!
 - Templates are essential.
 - Dr Dobbs writes articles: We are part of a (much) larger group.
 - C++ Standards committee is aggressive at adoption.
- C can live as a subset of C++.



Fortran codes should have a sunset plan

- Fortran is no longer part of many application ecosystems.
 - “But Fortran 2008 has [fill in the blank] feature.”
 - Fortran 2003 features are barely portable.
 - ForTrilinos became a compiler debugging project!
- Millions of Fortran lines of code: Don’t abandon.
- But: Have a controlled end-of-life plan.
- Fortran features should only be funded in support of sunset efforts.
- One feature should be migrated: F--
 - (C++)--
 - Cray already has working version.



Summary

- Node-level parallel computing is:
 - Low performing, out of the box: Lots of work required for scalability.
 - Simplistic: Can't mix models, incompatible Open efforts.
 - Opaque: Abstract performance models are still not widely available.
 - Making progress.
- Resilient computing and OS/RT:
 - rBSP:
 - Effective non-blocking communication.
 - Permits more aggressive error detection/correction.
 - LFLR:
 - Persistent storage.
 - Fault tolerant message passing.
 - Recovery mechanisms.
 - Selective Reliability:
 - A lot, requires co-design.



Summary

- OS/RT role summary:
 - Must provide much more functionality at node level.
 - Performance is critical (slow used to be OK, better scaling).
 - Must “let go” of reliable digital machine illusion, a bit.
- C++ offers the elements needed for parallel computing, now and in the future.
 - Supports all the parallel models we care about, right now.
 - Language is natural extensible.
 - Large community cares about it.
- One way to unify the HPC community is to let go of Fortran.
 - Features arrive a decade or more late, never reach true portability.
 - Protect existing code investments, but no more.