Exceptional service in the national interest

Sandia National Laboratories

Trilinos

Task-centric Application Design for Emerging HPC systems

Michael Heroux, Sandia National Laboratories, USA

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

New Trends and Responses

- Increasing data parallelism:
  - Design for vectorization and increasing vector lengths.
  - SIMT 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
- Memory architecture complexity:
  - Localize allocation/initialization.
  - Favor algorithms with higher compute/communication ratio.
- Resilience:
  - Distinguish what must be reliably computed.
  - Incorporate bit-state uncertainty into broader UQ contexts?

FUTURE PARALLEL APPLICATION DESIGN: SUGGESTED PRACTICES

#1: Encapsulate All Computation

- Fortran/C functions, done. IF no globals/commons.
- Methods in classes:
  - Extract Loops.
  - Create catalog of functions.
  - Functions usable as:
    - Kernels from OpenMP, TBB, etc.
    - Starting point for lambda/functor based design.
  - Starting point for thread-safe methods.

A Simple Epetra/AztecOO Program

```
// Header files omitted…
int main(int argc, char *argv[]) {
  MPI_Init(&argc,&argv); // Initialize MPI, MpiComm
  Epetra_MpiComm Comm( MPI_COMM_WORLD );
```

```
// ***** Map puts same number of equations on each pe *****

  int NumMyElements = 1000 ;
  Epetra_Map Map(-1, NumMyElements, 0, Comm);
  int NumGlobalElements = Map.NumGlobalElements();
```

```
// ***** Create an Epetra_Matrix  tridiag(-1,2,-1) *****

  Epetra_CrsMatrix A(Copy, Map, 3);
  double negOne = -1.0; double posTwo = 2.0;

  for (int i=0; i<NumMyElements; i++) {
    int GlobalRow = A.GRID(i);
    int RowLess1 = GlobalRow - 1;
    int RowPlus1 = GlobalRow + 1;
    if (RowLess1!=-1)
       A.InsertGlobalValues(GlobalRow, 1, &negOne, &RowLess1);
    if (RowPlus1!=NumGlobalElements)
       A.InsertGlobalValues(GlobalRow, 1, &negOne, &RowPlus1);
    A.InsertGlobalValues(GlobalRow, 1, &posTwo, &GlobalRow);
  }
A.FillComplete(); // Transform from GIDs to LIDs
```

```
  // ***** Create x and b vectors *****
  Epetra_Vector x(Map);
  Epetra_Vector b(Map);
  b.Random(); // Fill RHS with random #s
```

```
// ***** Create Linear Problem *****
  Epetra_LinearProblem problem(&A, &x, &b);
```

```
  // ***** Create/define AztecOO instance, solve *****
  AztecOO solver(problem);
  solver.SetAztecOption(AZ_precond, AZ_Jacobi);
  solver.Iterate(1000, 1.0E-8);
```

```
// ***** Report results, finish ***********************
  cout << "Solver performed " << solver.NumIters()
          << " iterations." << endl
          << "Norm of true residual = "
          << solver.TrueResidual()
          << endl;

  MPI_Finalize() ;
  return 0;
}
```

Construction for Irregular Data: Common Pattern

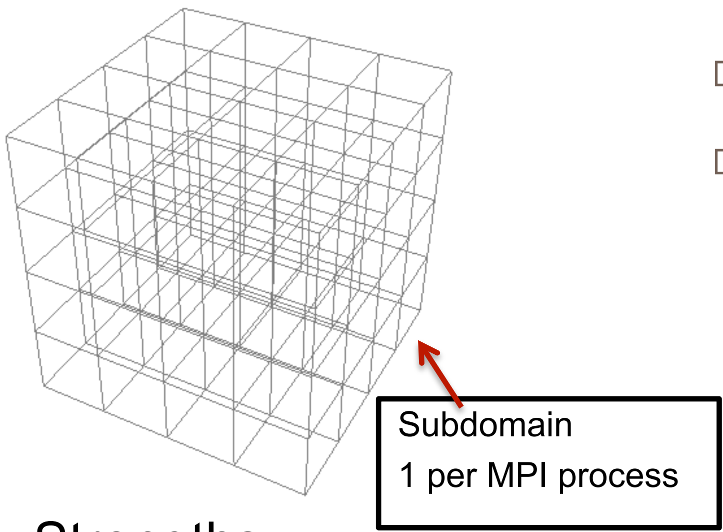
- Fill: Insert data.
- Analyze II: Graphs.
- Compute: Use the data object.

#2 Construction for Irregular Data: Bit by Bit The Path to Scalable Threading

- Count:
  - “Dry-run of allocation and fill.
  - Resist allocating storage.
- Analyze I:
  - Determine required storage, who should allocate.
- Allocate:
  - Coordinated, varies across platforms.
- Initialize:
  - Improved locality.
- Fill: Insert data.
- Analyze II: Graphs.
- Compute: Finally.

3: TASK-CENTRIC/DATAFLOW DESIGN

Classic HPC Application Architecture

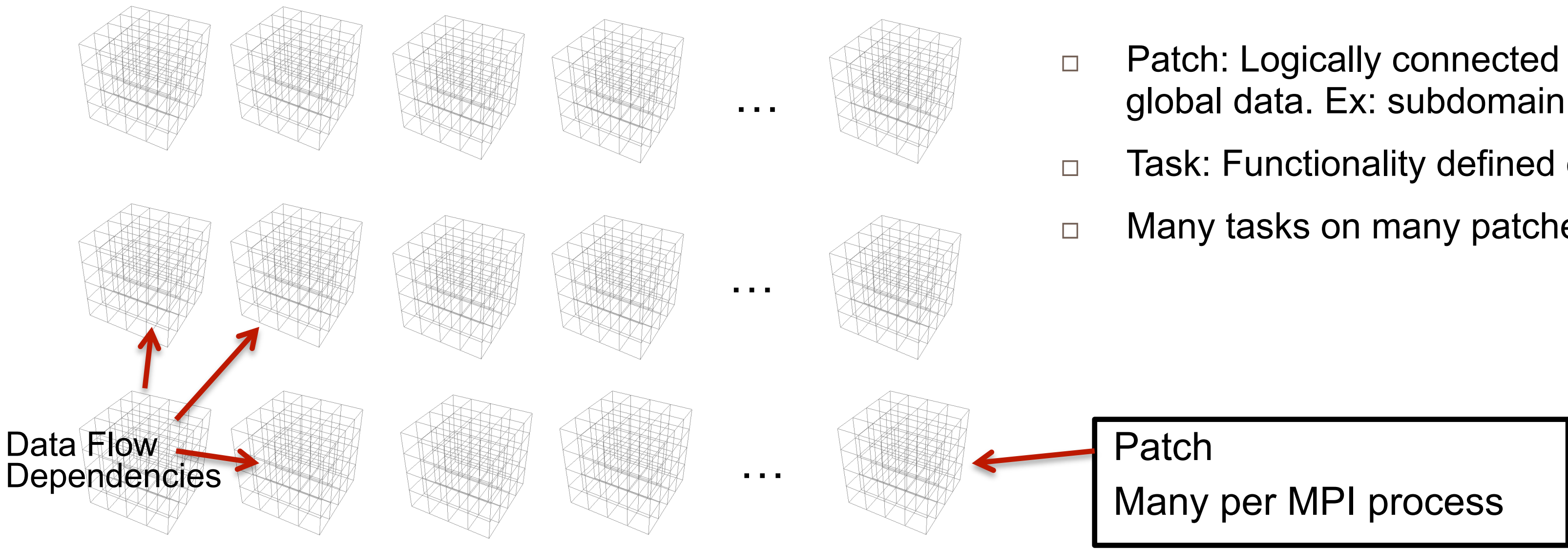


- Logically Bulk-Synchronous, SPMD
- Basic Attributes:
  - Halo exchange.
  - Local compute.
  - Global collective.
  - Halo exchange.

- Strengths:
  - Portable to many specific system architectures.
  - Separation of parallel model (SPMD) from implementation (e.g., message passing).
  - Domain scientists write sequential code within a parallel SPMD framework.
  - Supports traditional languages (Fortran, C).
  - Many more, well known.

- Weaknesses:
  - Not well suited (as-is) to emerging manycore systems.
  - Unable to exploit functional on-chip parallelism.
  - Difficult to tolerate dynamic latencies.
  - Difficult to support task/compute heterogeneity.

Task-centric/Dataflow Application Architecture



- Patch: Logically connected portion of global data. Ex: subdomain, subgraph.
- Task: Functionality defined on a patch.
- Many tasks on many patches.

- Strengths:
  - Portable to many specific system architectures.
  - Separation of parallel model from implementation.
  - Domain scientists write sequential code within a parallel framework.
  - Supports traditional languages (Fortran, C).
  - Similar to SPMD in many ways.

- More strengths:
  - Well suited to emerging manycore systems.
  - Can exploit functional on-chip parallelism.
  - Can tolerate dynamic latencies.
  - Can support task/compute heterogeneity.

Task on a Patch

- Patch: Small subdomain or subgraph.
  - Big enough to run efficiently once its starts execution.
    - CPU core: Need ~1 millisecond for today's best runtimes (e.g. Legion).
    - GPU: Give it big patches. GPU runtime does manytasking very well on its own.
- Task code (Domain scientist writes most of this code):
  - Standard Fortran, C, C++ code.
  - E.g. FEM stiffness matrix setup on a "workset" of elements.
  - Should vectorize (CPUs) or SIMT (GPUs).
  - Should have small thread-count parallel (OpenMP)
    - Take advantage of shared cache/DRAM for UMA cores.
  - Source line count of task code should be tunable.
    - Too coarse grain task:
      - GPU: Too much register state, register spills.
      - CPU: Poor temporal locality. Not enough tasks for latency hiding.
    - Too fine grain:
      - Too much overhead or
      - Patches too big to keep task execution at 1 millisec.

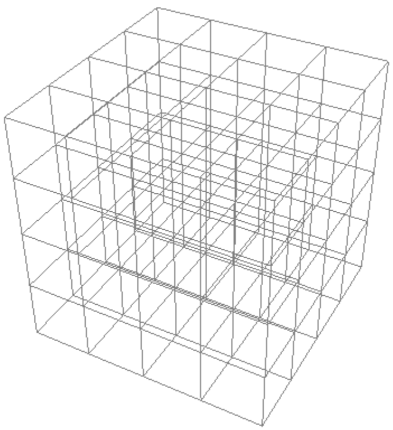
Portable Task Coding Environment

- Task code must run on many types of cores:
  - Standard multicore (e.g., Haswell).
  - Manycore (Intel PHI, KNC, KNL).
  - GPU (Nvidia).
- Desire:
  - Write single source.
  - Compile phase adapts for target core type.
  - Sounds like what?
- Kokkos (and others: OCCA, …):
  - Enable meta programming for multiple target core architectures.
- Future: Fortran/C/C++ with OpenMP 4:
  - Limited execution patterns, but very usable.
  - Like programming MPI codes today: Déjà vu for domain scientists.
- Other future: C++ with Kokkos in std namespace.
  - Broader execution pattern selection, more complicated.

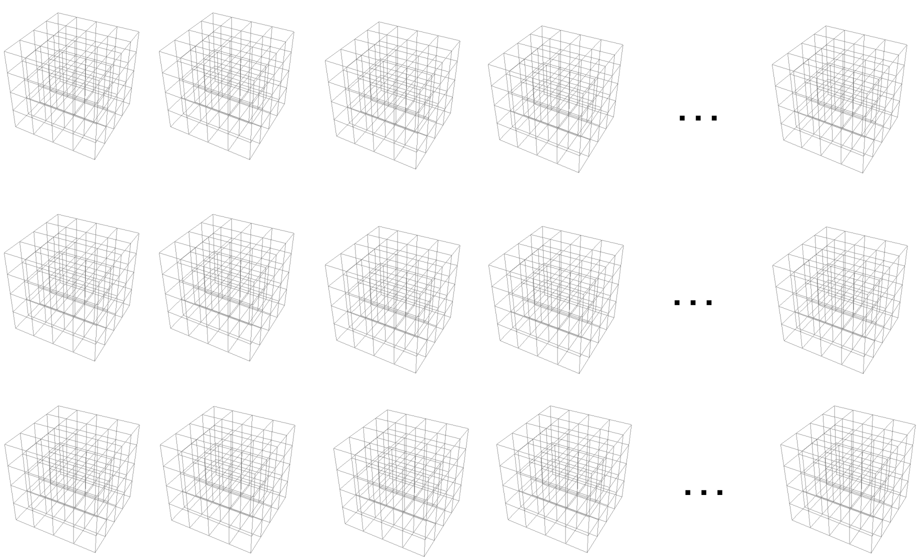
Task Management Layer

- New layer in application and runtime:
  - Enables (async) task launch: latency hiding, load balancing.
  - Provides technique for declaring inter-task dependencies:
    - Data read/write (Legion).
      - Task A writes to variable x, B depends on x. A must complete before B starts.
    - Futures:
      - Explicit encapsulation of dependency. Task B depends on A's future.
    - Alternative: Explicit DAG management.
  - Aware of temporal locality:
    - Better to run B on the same core as A to exploit cache locality.
  - Awareness of data staging requirements:
    - Task should not be scheduled until its data are ready:
      - If B depends on remote data (retrieved by A).
  - Manage heterogeneous execution: A on Haswell, B on PHI.
  - Resilience: If task A launched task B, A can relaunch B if B fails or times out.
- What are the app vs. runtime responsibilities?
- How can each assist the other?

Task-centric Benefits



- MPI:
  - Halo exchange.
  - Local compute.
  - Global collective.
  - Halo exchange.



- Task-centric: Many tasks
  - Async dispatch: Many in flight.
  - Natural latency hiding.
  - Higher message injection rates.
  - Better load balancing.
  - Compatible with "classics":
    - Fortran, vectorization, small-scale OMP.
    - Used within a task.
  - Natural resilience model:
    - Every task has a parent (can regenerate).
  - Demonstrated concept:
    - Co-Design centers, PSAAP2, others.

Task-centric/Dataflow Application Architecture Characteristics

- Task execution requirements:
  - Tunable work size: Enough to efficiently use a core once scheduled.
  - Vector/SIMT capabilities.
  - Small thread-count SMP.
  - Task data dependencies.
  - Accelerator mode: Big patch.
- Universal portability:
  - Works within node, across nodes.
  - Works across heterogeneous core types.
- Many tasks:
  - Async dispatch: Many in flight.
  - Natural latency hiding.
  - Higher message injection rates.
  - Better load balancing.
- Compatible with "classics":
  - Fortran, C, OpenMP.
  - Used within a task.
- Natural resilience model:
  - Every task has a parent (can regenerate).
- Demonstrated concept:
  - Co-Design centers, PSAAP2, others.

Open Questions for Task-Centric/Dataflow Strategies

- Functional vs. Data decomposition.
  - Over-decomposition of spatial domain:
    - Clearly useful, challenging to implement.
  - Functional decomposition:
    - Easier to implement. Challenging to execute efficiently (temporal locality).
- Dependency specification mechanism.
  - How do apps specify inter-task dependencies?
  - Futures (e.g., C++, HPX), data addresses (Legion), explicit (Uintah).
- Roles & Responsibilities: App vs Libs vs Runtime vs OS.
- Interfaces between layers.
- Huge area of R&D for many years.

Open Questions for Task-Centric/Dataflow Strategies

- Functional vs. Data decomposition.
  - Over-decomposition of spatial domain:
    - Clearly useful, challenging to implement.
  - Functional decomposition:
    - Easier to implement. Challenging to execute efficiently (temporal locality).
- Dependency specification mechanism.
  - How do apps specify inter-task dependencies?
  - Futures (e.g., C++, HPX), data addresses (Legion), explicit (Uintah).
- Roles & Responsibilities: App vs Libs vs Runtime vs OS.
- Interfaces between layers.
- Huge area of R&D for many years.

Data challenges:

- Read/write functions:
  - Must be task compatible.
  - Thread-safe, non-blocking, etc.
- Versioning:
  - Computation may be executing across multiple logically distinct phases (e.g. timesteps)
  - Example: Data must exist at each grid point and for all active timesteps.
- Global operations:
  - Coordination across task events.
  - Example: Completion of all writes at a time step.

Key messages:

- HPC App architectures are changing, adding further demands and opportunities for big data, big compute co-design efforts.
- Need to know HPC app trends to get combined big data, big compute right.

Execution Policy for Task Parallelism

- TaskManager< ExecSpace > execution policy
  - Policy object shared by potentially concurrent tasks

```
TaskManager<...> tm( exec_space , ... );
Future<> fa = spawn( tm , task_functor_a ); // single-thread task
Future<> fb = spawn( tm , task_functor_b );
```

  - Tasks may be data parallel

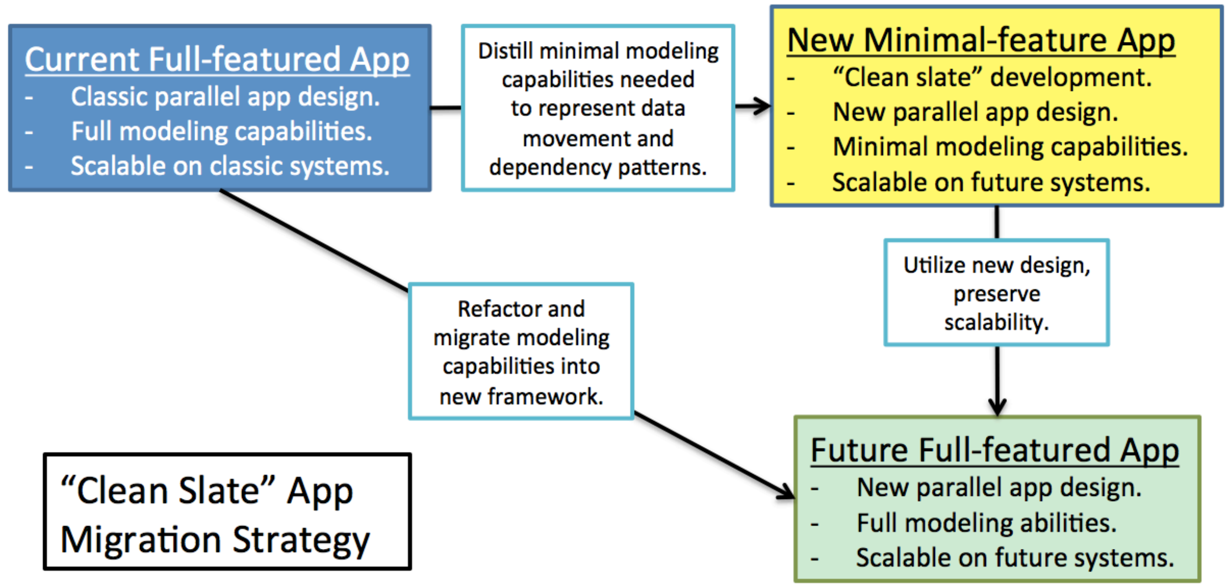
```
Future<> fc = spawn_for( tm.range(0..N) , functor_c );
Future<value_type> fd = spawn_reduce( tm.team(N,M) , functor_d );
wait( tm ); // wait for all tasks to complete
```

  - Destruction of task manager object waits for concurrent tasks to complete
- Task Managers
  - Define a scope for a collection of potentially concurrent tasks
  - Have configuration options for task management and scheduling
  - Manage resources for scheduling queue

Movement to Task-centric/Dataflow is Disruptive: Use Clean-slate strategies

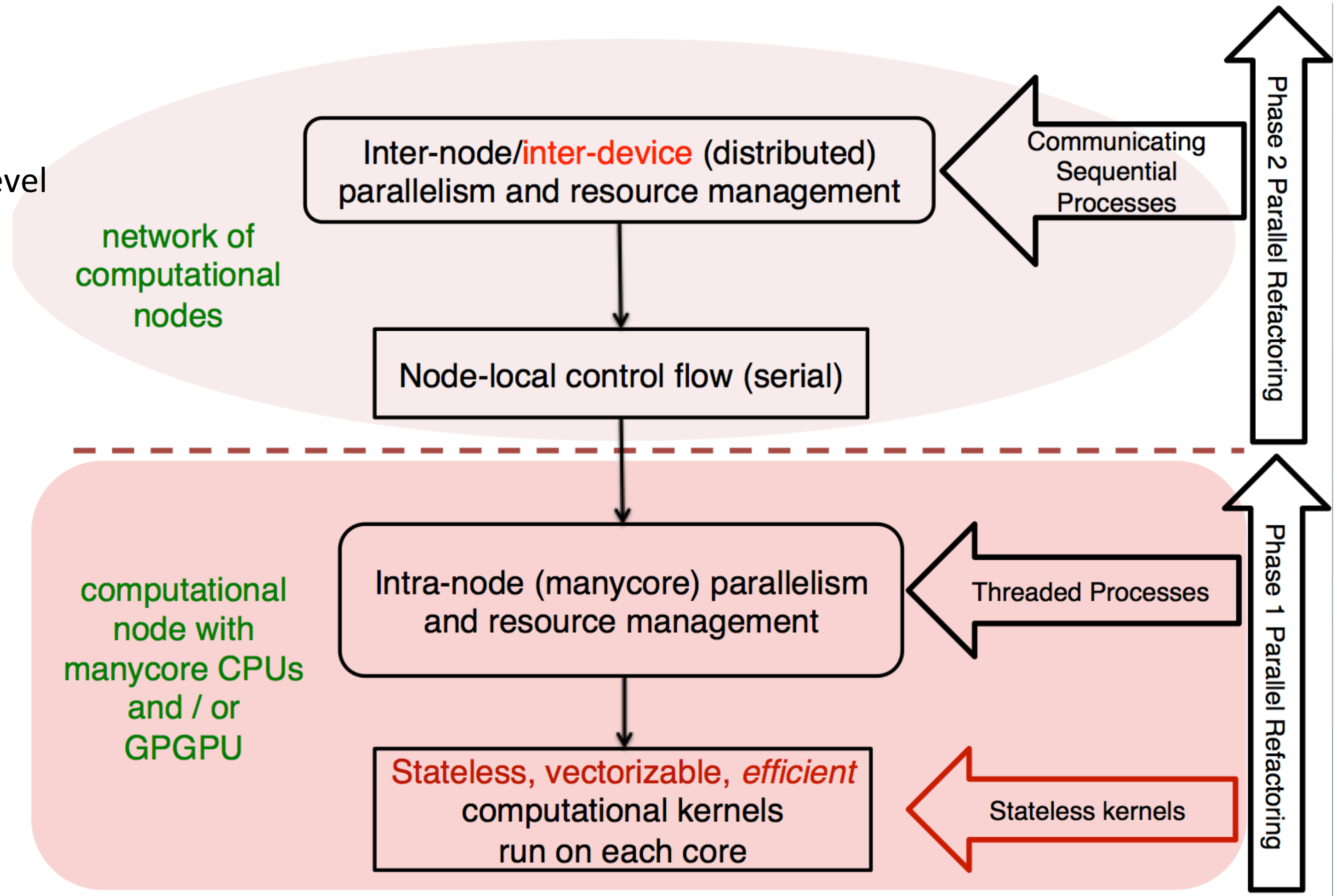
Sandia National Laboratories

- Best path to task-centric/dataflow.
- Stand up new framework:
  - Minimal, *representative* functionality.
  - Make it scale.
- Mine functionality from previous app.
  - May need to refactor a bit.
  - May want to refactor substantially.
- Historical note:
  - This was the successful approach in 1990s migration from vector multiprocessors (Cray) to distributed memory clusters.
  - In-place migration approach provided early distributed memory functionality. Failed long-term scalability needs.



Phased Migration to Task-centric/ Dataflow

- All Apps Looking for new Node-level programming environments.
- Exploring standards, emerging:
  - OpenMP, pthreads.
  - OpenMP 4, OpenACC.
- Exploring non-standard:
  - HPX (Parallex).
  - Legion.
- Brute force:
  - Uintah framework.
- Strategy:
  - Phase 1: On-node.
  - Phase 2: Inter-node.



Task-centric/dataflow & Trilinos

- Kokkos:
  - Task launch/futures.
  - Provided for Trilinos users (or independently).
  - Used by Trilinos itself.
- Thread-safe methods:
  - Class methods, e.g., matrix fill, must be thread-safe.
    - Task A and B should be able to call matrix insertion at the same time.
  - BUT: Using Kokkos directly for these operations is even better.
    - Then Tpetra must accept Kokkos arrays for it object pieces.
- Solvers must be threaded:
  - If application is using MPI+X, we must use MPI+X.
    - Same MPI ranks. Same definition of X.
  - Must perform efficiently with MPI+X.

Summary: Task-centric app design

- Scalable application design will move to a task-centric architecture:
  - Provides a sequential view for domain scientists.
    - Looks a lot like MPI programming.
    - Only added requirements: Consumer/producer dependencies.
  - Support vectorization/SIMT within a task.
  - Supports many (all, really) threading environments.
  - Permits continued use of Fortran.
  - Provides a resilience-capability architecture.
- Challenges to developing task-centric apps:
  - Much more complicated MPI node-level interactions:
  - OS/RT support for task-DAGS:
    - What are the Apps responsibility? How can OS/RT assist?
    - Concurrent execution is essential for scalability.
      - Must be reading/writing from memory, computing simultaneously.