Thread-scalable programming with Tpetra and Kokkos Introduction

Michael A. Heroux, Roger Pawlowski

Sandia National Laboratories

Collaborators:

Erik Boman, Carter Edwards, James, Elliot, Mark Hoemmen, Siva Rajamanickam, Keita Teranishi, Christian Trott, Alan Williams (SNL)

Factoring 1K to1B-Way Parallelism

- Why 1K to 1B?
  - Clock rate: O(1GHz) → O(10^9) ops/sec sequential
  - Terascale: 10^{12} ops/sec → O(10^3) simultaneous ops
    - 1K parallel intra-node.
  - Petascale: 10^{15} ops/sec → O(10^6) simultaneous ops
    - 1K-10K parallel intra-node.
    - 100-1K parallel inter-node.
  - Exascale: 10^{18} ops/sec → O(10^9) simultaneous ops
    - 1K-10K parallel intra-node.
    - 100K-1M parallel inter-node.

Three Parallel Computing Design Points

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

Goal: Make

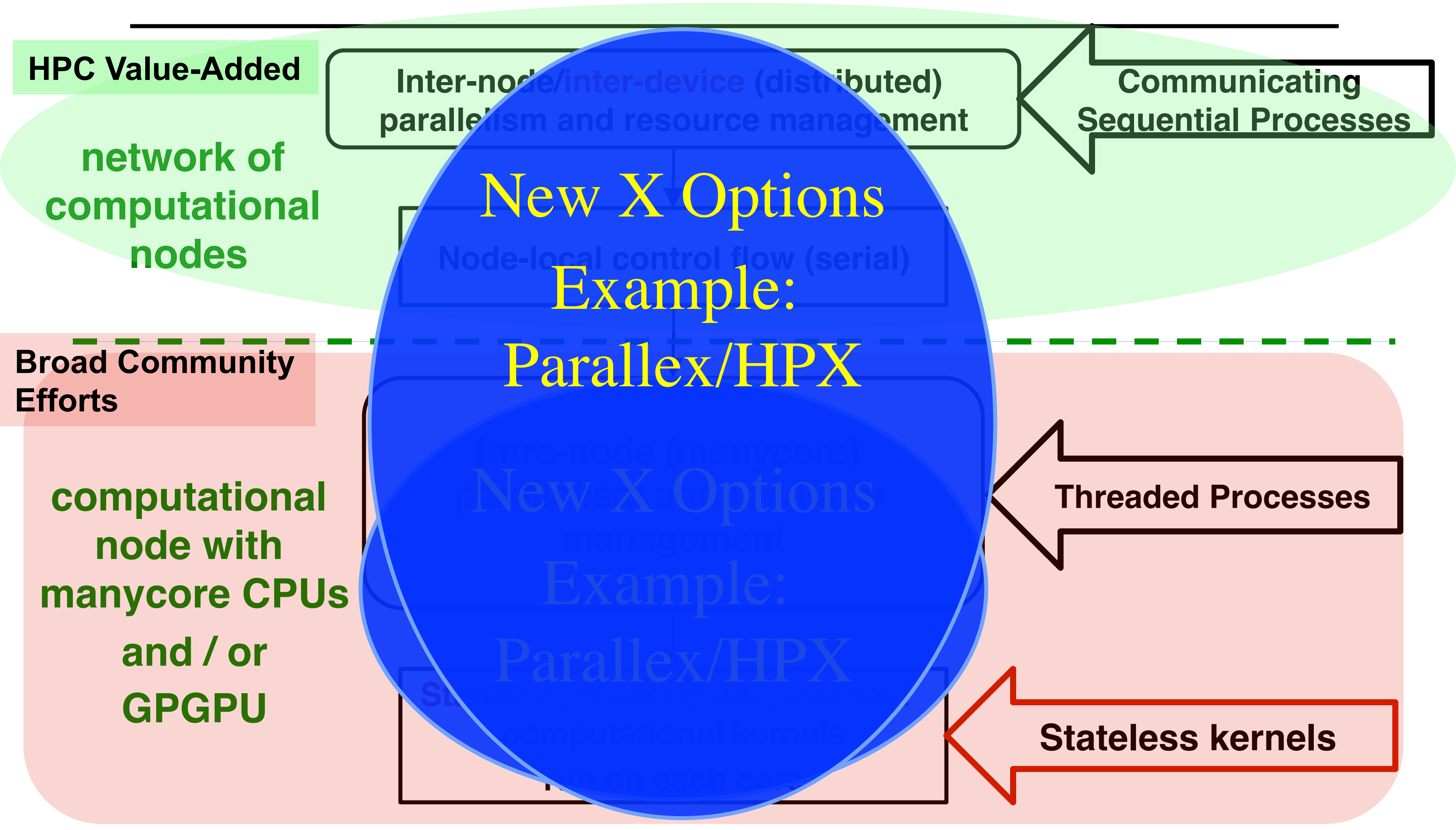
Petascale = Terascale + more

Exascale = Petascale + more

Common Element

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

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


HPC Value-Added
network of computational nodes
Inter-node/inter-device (distributed) parallelism and resource management
Communicating Sequential Processes
Node-local control flow (serial)
New X Options Example: Parallex/HPX
Broad Community Efforts
computational node with manycore CPUs and / or GPGPU
Threaded Processes
Stateless kernels

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.

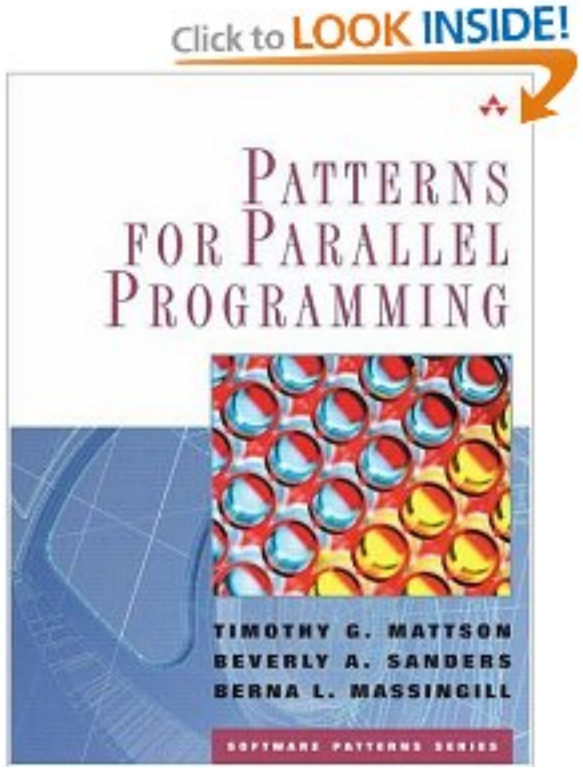
Sandia National Laboratories

Overarching (unachievable) Goal: Domain Scientists Write No Parallel Code

Reasoning About Parallelism

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 (also called *Map* pattern).
  - 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?

Sandia National Laboratories

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.

Programming Environment Deficiencies

Needs: Data management

- Break storage association:
  - Physics i,j,k should not be storage i,j,k.
- Layout as a first-class concept:
  - Construct layout, then data objects.
  - Chapel has this right.
- Better NUMA awareness/resilience:
  - Ability to "see" work/data placement.
  - Ability to migrate data: MONT
- 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.
  - Note: Dynamic work-stealing is not as easy as it seems.
- Maybe better thread local allocation will mitigate impact.

Sandia National Laboratories

Multi-dimensional Dense Arrays

- Many computations work on data stored in multi-dimensional arrays:
  - Finite differences, volumes, elements.
  - Sparse iterative solvers.
- Dimension are (k,l,m,…) where one dimension is long:
  - A(3,1000000)
  - 3 degrees of freedom (DOFs) on 1 million mesh nodes.
- A classic data structure issue is:
  - Order by DOF: A(1,1), A(2,1), A(3,1); A(1,2) … or
  - By node: A(1,1), A(1,2), …
- **Adherence to raw language arrays forces a choice.**

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

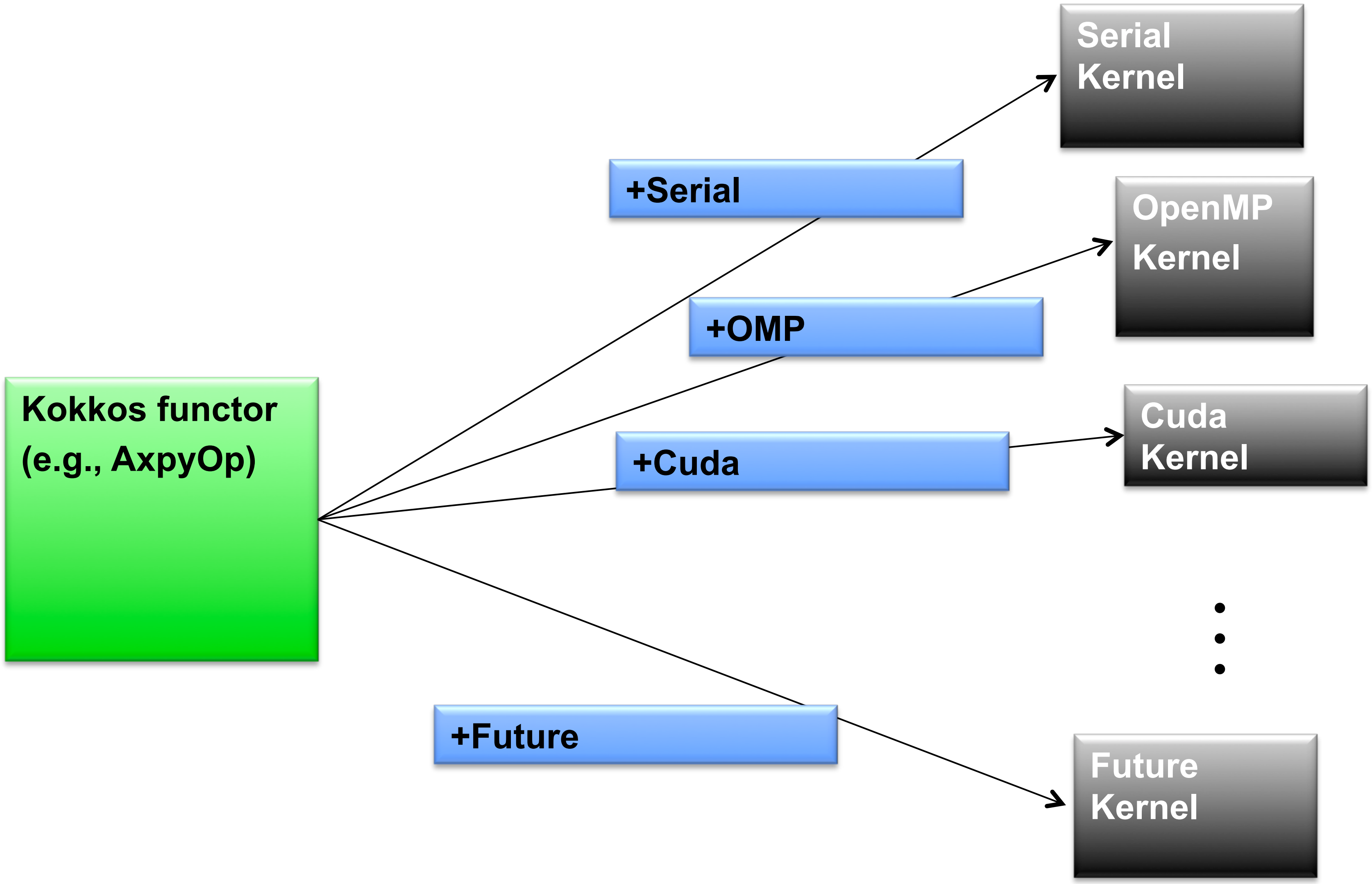
Multi-dimensional Dense Arrays

- Many computations work on data stored in multi-dimensional arrays:
  - Finite differences, volumes, elements.
  - Sparse iterative solvers.
- Dimension are (k,l,m,…) where one dimension is long:
  - A(3,1000000)
  - 3 degrees of freedom (DOFs) on 1 million mesh nodes.
- A classic data structure issue is:
  - Order by DOF: A(1,1), A(2,1), A(3,1); A(1,2) … or
  - By node: A(1,1), A(1,2), …
- Adherence to raw language arrays force a choice.

Struct-of-Arrays vs. Array-of-Structs

A False Dilemma

Compile-time Polymorphism



A Bit about Functors

Classic function “ComputeWAXPBY_ref.cpp”

```
/*!
  Routine to compute the update of a vector with the sum of two
  scaled vectors where: w = alpha*x + beta*y

 @param[in] n the number of vector elements (on this processor)
  @param[in] alpha, beta the scalars applied to x and y respectively.
  @param[in] x, y the input vectors
  @param[out] w the output vector.
  @return returns 0 upon success and non-zero otherwise
*/
int ComputeWAXPBY_ref(const local_int_t n, const double alpha, const double *
const x,  const double beta, const double * const y, double * const w) {

for (local_int_t i=0; i<n; i++) w[i] = alpha * x[i] + beta * y[i];

  return(0);
}
```

A Bit about Functors

Functor-calling function “ComputeWAXPBY.cpp”

```
/*!
  Routine to compute the update of a vector with the sum of two
  scaled vectors where: w = alpha*x + beta*y

 @param[in] n the number of vector elements (on this processor)
  @param[in] alpha, beta the scalars applied to x and y respectively.
  @param[in] x, y the input vectors
  @param[out] w the output vector.
  @return returns 0 upon success and non-zero otherwise
*/
int ComputeWAXPBY(const local_int_t n, const double alpha, const double * const x,  const
double beta, const double * const y, double * const w) {

// for (local_int_t i=0; i<n; i++) w[i] = alpha * x[i] + beta * y[i];
 tbb::parallel_for(tbb::blocked_range<size_t>(0,n), waxpby_body(n, alpha, x, beta, y, w) );

  return(0);
}
```

A Bit about Functors

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h”
  class waxpby_body{
    size_t n_;
    double alpha_;
    double beta_;
    const double * const x_;
    const double * const y_;
    double * const w_;  public:
  waxpby_body(size_t n, const double alpha, const double * const x, const double beta,
const double * const y, double * const w)
      : n_(n), alpha_(alpha), x_(x), beta_(beta), y_(y), w_(w) {  }
  void operator() (const tbb::blocked_range<size_t> &r) const {
    const double * const x = x_;
    const double * const y = y_;
    double * const w = w_;
    double alpha = alpha_;
    double beta = beta_;
    for(size_t i=r.begin(); i!=r.end(); i++) w[i] = alpha * x[i] + beta * y[i];
  }
};
```

A Bit about ~~Functors~~ Lambdas

Lambda version "ComputeWAXPBY.cpp"

```
/*!
  Routine to compute the update of a vector with the sum of two
  scaled vectors where: w = alpha*x + beta*y

 @param[in] n the number of vector elements (on this processor)
  @param[in] alpha, beta the scalars applied to x and y respectively.
  @param[in] x, y the input vectors
  @param[out] w the output vector.
  @return returns 0 upon success and non-zero otherwise
*/
int ComputeWAXPBY(const local_int_t n, const double alpha, const double * const x,  const
double beta, const double * const y, double * const w) {

// for (local_int_t i=0; i<n; i++) w[i] = alpha * x[i] + beta * y[i];
tbb::parallel_for (size_t(0), n, [=](size_t i) {w[i] = alpha * x[i] + beta * y[i];});
return(0);
}
```

Transition to Kokkos

Kokkos is the Trilinos foundation for thread-scalable programming