ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

A Brief Compendium of GPU-enabled Numerical Libraries

CSCS-USI Autumn School, 15.09.2013

Dr. William Sawyer, Dr. Karl Rupp, Dr. Michael Heroux

Tutorial Summary

- Introduction and library overview
  - Typical numerical and semi-numerical problems
  - Survey of some gpu-enabled libraries:
    - MAGMA
    - Thrust / CuSP
    - ViennaCL
    - Paralution
    - PETSc
    - Trilinos
- Case studies: Thrust, ViennaCL

Poll

- What programming languages do you work with?
- What parallel programming paradigms?
- What problems do you want to solve ?
- What algorithms do you want to use?
- What libraries, if any, do you use?

Objectives of this tutorial

- Awareness of the available libraries
- Realization that it is not necessary to "recreate the wheel"
- Bonus: some hands-on experiences

Problems you might like to solve

- Partial differential equations
- Dense systems of linear equations
- Sparse systems of linear equations
- Preconditioning of large systems
- Eigenvalue / singular value decompositions of sparse/ dense matrices
- Partitioning large graphs
- Non-linear systems and optimization
- ...

Typical dense linear algebra operations

- Cholesky factorization: $A = A^T = LL^T \qquad i < j \Rightarrow L_{i,j} = 0$
- QR factorization: $A = QR \qquad Q^T Q = I \qquad i > j \Rightarrow R_{i,j} = 0$
- LU factorization: $A = P^T LU \qquad P^T P = I$
- Forward/back-substitution: $Ax = y \Rightarrow LUx = y \Rightarrow w = L^{-1}y \Rightarrow x = R^{-1}w$
- Eigenvalue decomposition: $Ax = \lambda x \Rightarrow A = QDQ^T$
- Generalized eigen-problem: $Ax = \lambda Bx$
- Singular value decomposition: $A = U\Sigma V^T \quad U^T U = I \quad V^T V = I$

MAGMA: Matrix Algebra on GPU and Multicore Architecture

- *MAGMA:* Matrix Algebra on GPU and Multicore Architectures http://icl.cs.utk.edu/magma
- Soon: *D-PLASMA*, *D-MAGMA* for distributed memory platforms
- *See subsequent tutorial by Stan Tomov!*

Thrust: Standard Template Library for GPUs

- A library of parallel algorithms resembling the C++ STL
- Allows easy access/manipulation of vectors on both host (CPU) and device (GPU); based on data *iterators*
- Defines straightforward vector data operators, e.g., :
  - ✳ initialize vectors
  - ✳ exchange existing values
  - ✳ copy one to another
  - ✳ transform with an operator
  - ✳ perform reductions (e.g., one-dimensional to scalar)
  - ✳ sorting and other operators

Thrust: typical operations

- Declare arrays on host or device

```
thrust::host_vector<int> H(4);
thrust::device_vector<int> D = H;
thrust::device_vector<int> Z(4, 1);  // All ones
```

- Initialize arrays, use iterators

```
thrust::sequence(H.begin(), H.end()); // H = (0,1,2,3)
thrust::fill(D.begin(), D.end(), 2);  // Fill with twos
```

- Transform arrays

```
thrust::transform(D.begin(), D.end(), Z.begin(), thrust::negate<int>());
thrust::replace(H.begin(), H.end(), 2, -2);
```

- Perform a reduction

```
int sum = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());
```

- Sort array

```
thrust::sort(H, H + 4);
```

Thrust: simple manipulations

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <iostream>
int main(void)
{
  // initialize all ten integers of a device_vector to 1
  thrust::device_vector<int> D(10, 1);
  // initialize a host_vector with the first five elements of D
  thrust::host_vector<int> H(D.begin(), D.begin() + 5);
  // set the first seven elements of a vector to 9
  thrust::fill(D.begin(), D.begin() + 7, 9);
  // copy all of H back to the beginning of D
  thrust::copy(H.begin(), H.end(), D.begin());
  // print D
  for(int i = 0; i < D.size(); i++)
  {
    std::cout << "D[" << i << "] = " << D[i] << std::endl;
  }
  return 0;
}
```

Assignment: what values are printed?

Thrust: reductions operations

Sum array

```
#include <thrust/sort.h>
...
// initial value of the reduction
int init = 0;
// binary operation used to reduce values
thrust::plus<int> binary_op;
// compute sum on the device
int sum = thrust::reduce(d_vec.begin(), d_vec.end(), init, binary_op);
```

```
#include <thrust/iterator/permutation_iterator.h>
...
// gather locations
thrust::device_vector<int> map(4);
map[0] = 3; map[1] = 1; map[2] = 0; map[3] = 5;

// array to gather from
thrust::device_vector<int> source(6);
source[0] = 10; source[1] = 20; source[2] = 30;
source[3] = 40; source[4] = 50; source[5] = 60;

// fuse gather with reduction:
//   sum = source[map[0]] + source[map[1]] + ...
int sum = thrust::reduce(thrust::make_permutation_iterator(source.begin(), map.begin()),
                         thrust::make_permutation_iterator(source.begin(), map.end()));
```

Perform a reduction on an indirectly addressed array

Thrust: Exercise

- Compare host and device sorting of integer vectors
- Create device version from host version (given)
- `https://github.com/fomics/GPU_Libraries_2013/wiki/Thrust-Exercise`
- Performance comparison

Thrust: advanced operators

- Scan operations (parse and alter vector)

```
#include <thrust/scan.h>
int data[6] = {1, 0, 2, 2, 1, 3};
thrust::inclusive_scan(data, data + 6, data); // data now {1, 1, 3, 5, 6, 9}
thrust::exclusive_scan(data, data + 6, data); // data now {0, 1, 2, 5, 10, 16}
```

- Iterator transformation -- bind an operator to an iterator.

```
#include <thrust/transform_iterator.h>
thrust::device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;
thrust::device_vector<int>::iterator first = thrust::make_transform_iterator(vec.begin(), negate<int>());
thrust::device_vector<int>::iterator last  = thrust::make_transform_iterator(vec.end(), negate<int>());
// first[0] returns -10, first[1] returns -20, first[2] returns -30
thrust::reduce(first, last);   // returns -60 (i.e. -10 + -20 + -30)
```

- Zip operator: turns multiple input arguments into tuples

```
#include <thrust/iterator/zip_iterator.h>
thrust::device_vector<int>  A(3); A[0] = 10;  A[1] = 20;  A[2] = 30;
thrust::device_vector<char> B(3); rust::device_vector<char> B(3);
first = thrust::make_zip_iterator(thrust::make_tuple(A.begin(), B.begin()));
last  = thrust::make_zip_iterator(thrust::make_tuple(A.end(),   B.end()));
thrust::maximum< tuple<int,char> > binary_op;
thrust::tuple<int,char> init = first[0];
thrust::reduce(first, last, init, binary_op); // returns tuple(30, 'z')
```

Thrust: conclusions

- Attempt to extend C++ STL functionality for host/device
- Based on CUDA, thus bound to NVIDIA GPUs
- Uses template meta-programming to find correct implementation at compile time
- Is an community, open-source project, but appears to have long-term approval from NVIDIA (bundled in SDK releases)
- Development is demand-driven by community; let your opinions / needs / usages / suggestions be known!

Linear solvers

Goal: Support the solution of linear systems,

$$Ax=b,$$

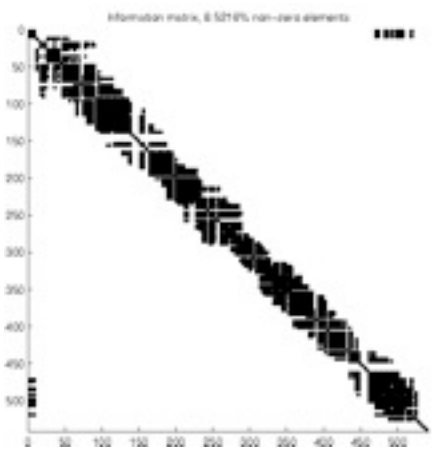
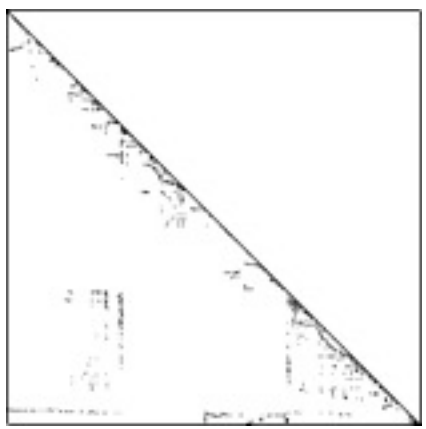
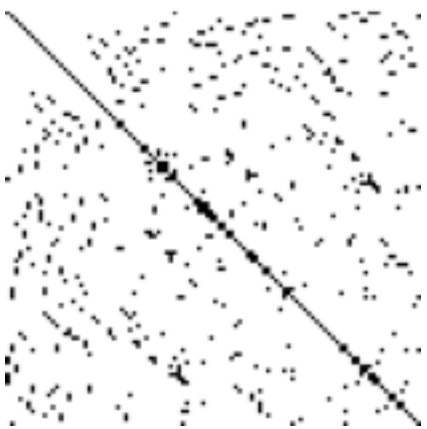
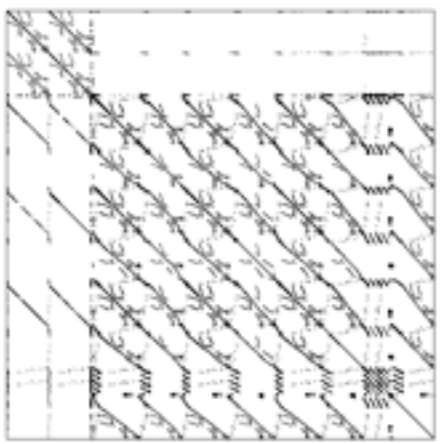
particularly for *sparse,* parallel problems arising from PDE-based models.

User provides:

- *A (matrix or operator)*
- *b (right-hand side)*
- *u (initial guess)*

Libraries for Sparse Linear Algebra

- MAGMA limited to $m \times n$ matrices with $m,n = O(10^4)$
- Sparse matrices typically contain at least 90% zeros
- Number of non-zero (nz) elements, large: $nz = O(10^7)$
- Matrix market http://math.nist.gov/MatrixMarket/

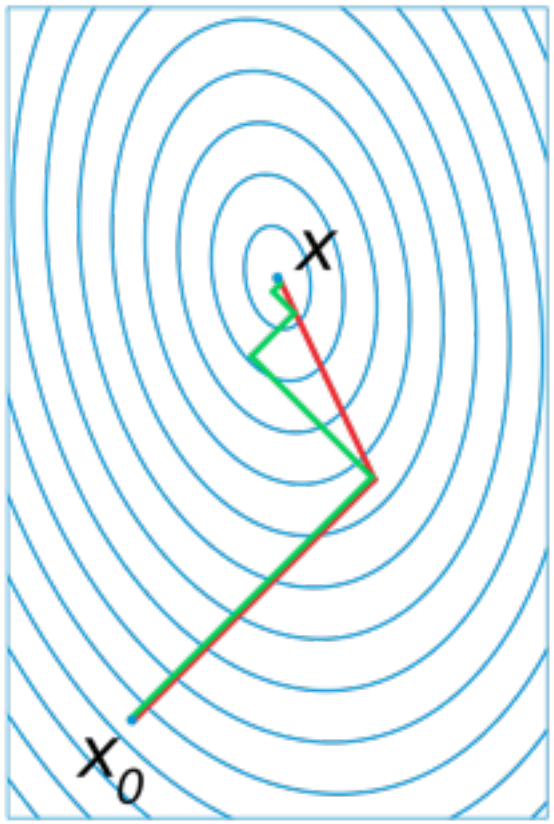





Linear System Solution: $Ax = b$

Two basic techniques:

- Direct methods, i.e. factorize matrix
  - good for multiple right hand sides
  - tend to be more robust
- Iterative methods
  - good if matrix known via operators
  - possibilities for approximate solutions

Hestenes/Stiefel, 1952: Conjugate Gradient



$$k = 0; \quad x_0 = 0; \quad r_0 = 0$$

$$while \quad r_k \neq 0 \quad \{$$

$$k = k + 1$$

$$if\,(k = 0) \Rightarrow p_1 = r_0$$

$$if\,(k > 0) \Rightarrow \beta_k = r_{k-1}^T r_{k-1} / r_{k-2}^T r_{k-2}; \quad p_k = r_{k-1} + \beta_k p_{k-1}$$

$$\alpha_k = r_{k-1}^T r_{k-1} / p_k^T A p_k$$

$$x_k = x_{k-1} + \alpha_k p_k$$

$$r_k = r_{k-1} - \alpha_k A p_k$$

$$\}$$

1980's: led to a wide class of iterative *Krylov subspace methods*

Preconditioners: KSM alone insufficient!

- CG method initially ignored due to slow convergence
  - Theoretical convergence after 2*n steps, but n is huge
  - Convergence rate related to ratio largest/smallest eigenvalue
- Easier problem: preconditioner $M \approx A$ $\quad Ax = b \Rightarrow M^{-1}Ax = M^{-1}b$
  - Find an approximation for A where $M^{-1}x$ is 'easily' calculated
  - Possibilities:
    - Approximate inverse known through physical description
    - Incomplete LU decomposition
    - Sparse approximative inverse (assume inverse also sparse)
    - Multilevel (multigrid) preconditioners
    - More…

CUSP: Sparse Lin. Alg. for GPUs

- CUda SParse: a templated library for GPUs and CPUs, providing a high-level interface that hides GPU complexities (NVIDIA, Apache license)
- Built on top of Thrust (NVIDIA)

```
#include <cusp/hyb_matrix.h>
#include <cusp/io/matrix_market.h>
#include <cusp/krylov/cg.h>
int main(void)
{
    // create an empty sparse matrix structure (HYB format)
    cusp::hyb_matrix<int, float, cusp::device_memory> A;

    // load a matrix stored in MatrixMarket format
    cusp::io::read_matrix_market_file(A, "5pt_10x10.mtx");

    // allocate storage for solution (x) and right hand side (b)
    cusp::array1d<float, cusp::device_memory> x(A.num_rows, 0);
    cusp::array1d<float, cusp::device_memory> b(A.num_rows, 1);

    // solve the linear system A * x = b with the Conjugate Gradient
method
    cusp::krylov::cg(A, x, b);
    return 0;
}
```

CUSP: some extensions

- CUSP implementation of sparse approximate inverse preconditioner at CSCS
- Requires least-squares minimization (QR factorization)
- GMRES solver (like CG for non-symmetric matrices)

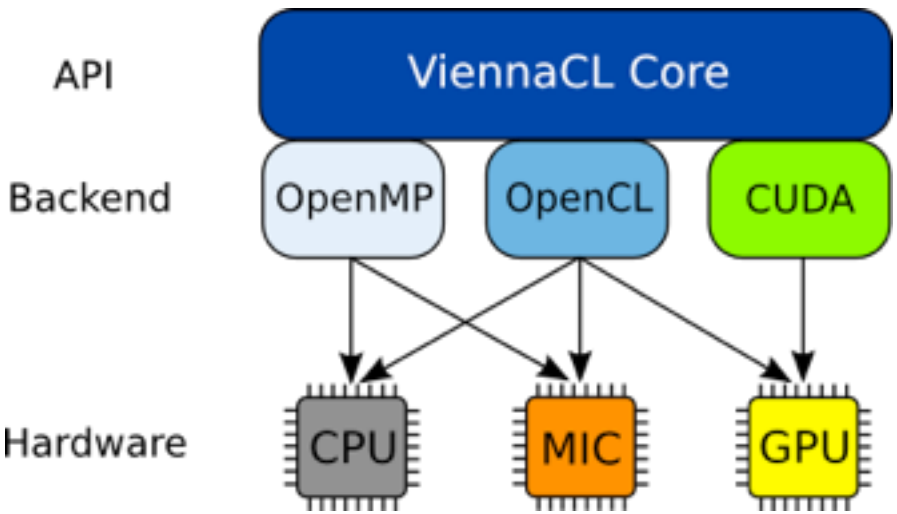
```
        std::cout << "\nSolving with SPAI preconditioner" << std::endl;
// allocate storage for solution (x) and right hand side (b)
        cusp::array1d<ValueType, MemorySpace> x(A.num_rows, 0);
        cusp::array1d<ValueType, MemorySpace> b(A.num_rows, 1);
              // set stopping criteria (iteration_limit = 1000, relative_tolerance = 1e-6)
        cusp::default_monitor<ValueType> monitor(b, 1000, 1e-6)

// setup preconditioner
        cusp::precond::spai<ValueType, MemorySpace> M(A, A);

// solve
        cusp::krylov::gmres(A, x, b, monitor, M);
```

CUSP: Some conclusions

- Template metaprogramming: conceptually easy to specify one template for different data types, e.g., single/double precision
- Only supports single node execution (multi-node implementation should be at a higher level, anyway)
- Has only CUDA backends: only for NVIDIA GPUs
- Not supported by NVIDIA! Future: uncertain
- Community effort, driven by user demand
- Location: `https://github.com/cusplibrary`

ViennaCL: Sparse Lin. Algebra on multiple platforms

- Linear algebra library for many core architectures (GPUs, Intel Xeon Phi)
- Supports BLAS 1-3
- Iterative solvers
- Sparse row matrix-vector multiplication
- Goals:
  - ➡ Simplicity, minimal dependencies
  - ➡ Compatible with Boost.uBLAS
  - ➡ Open source, header-only library



Boost: Solve linear system

```
using namespace boost::numeric::ublas;
matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);

/* Fill A, x, rhs here */

// Some operations
rhs += 2.0 * x;
double val = inner_prod(x, rhs);
matrix += val * outer_prod(x, rhs);

x = solve(A, rhs, upper_tag()); // Upper triangular solver

std::cout << " 2-norm: " << norm_2(x) << std::endl;
std::cout << "sup-norm: " << norm_inf(x) << std::endl;
```

ViennaCL: Solve linear system

```
using namespace viennacl;
using namespace viennacl::linalg;

matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);

/* Fill A, x, rhs here */

// Some operations
rhs += 2.0 * x;
double val = inner_prod(x, rhs);
matrix += val * outer_prod(x, rhs);

x = solve(A, rhs, upper_tag()); // Upper triangular solver

std::cout << " 2-norm: " << norm_2(x) << std::endl;
std::cout << "sup-norm: " << norm_inf(x) << std::endl;
```

ViennaCL: Memory Model

Memory buffers need to be managed differently for each of the compute backends (OpenMP, CUDA, OpenCL)

- Memory domain abstraction in class `viennacl::backend::mem_handle`
- Raw handles from `cuda_handle()`, `opencl_handle()` and `ram_handle()`
- backend is required to support:
  - ➡ `memory_create()`: Create a memory buffer
  - ➡ `memory_copy()`: Copy the (partial) contents of one buffer to another
  - ➡ `memory_write()`: Write from a memory location in CPU RAM to the buffer
  - ➡ `memory_read()`: Read from the buffer to a memory location in CPU RAM

ViennaCL: Interoperability

Standard C++ vectors and Boost uBLAS vectors can be passed to/from ViennaCL vectors:

```
std::vector<double> std_x(100)
ublas::vector<double> ublas_x(100);
viennacl::vector<double> vcl_x1, vcl_x2;

/* setup of std_x and ublas_x omitted */
viennacl::copy(std_x.begin(), std_x.end(), vcl_x1.begin());
viennacl::copy(ublas_x.begin(), ublas_x.end(), vcl_x2.begin());
```

ViennaCL: Solve sparse system

```
using namespace viennacl;
using namespace viennacl::linalg;
compressed_matrix<double> A(1000, 1000);  // sparse matrix format
vector<double> x(1000), rhs(1000);
/* Fill A, x, rhs here */
x = solve(A, rhs, cg_tag()); // Conjugate Gradient solver
x = solve(A, rhs, bicgstab_tag()); // BiCGStab solve
x = solve(A, rhs, gmres_tag()); // GMRES solver
```

uBLAS has no iterative solvers, but thanks to compatibility

```
using namespace boost::numeric::ublas;
using namespace viennacl::linalg;
compressed_matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);
/* Fill A, x, rhs here */
x = solve(A, rhs, cg_tag()); // Conjugate Gradient solver
x = solve(A, rhs, bicgstab_tag()); // BiCGStab solver
x = solve(A, rhs, gmres_tag()); // GMRES solver
```

ViennaCL: temporaries

Consider the expression

```
vec1 = vec2 + alpha * vec3 - beta * vec4;
```

With naive C++ this could be equivalent to

```
tmp1 <- alpha * vec3
tmp2 <- beta * vec4;
tmp3 <- tmp1 - tmp2;
tmp4 <- vec2 + tmp3;
vec1 <- tmp4;
```

Temporaries are costly on CPUs, even more so on GPUs

➡ *Expression templates* reduce usage of temporaries

ViennaCL: expression templates

Example expression

```
vec1 += alpha * vec3;
```

Typical operator overload signatures

```
vector & vector::operator+=(vector const & T);
vector operator*(double value, vector const & v);
```

Avoid the temporary returned by `operator*` as follows

```
vector &
vector::operator+=(vector_expression<double, op_mult,vector> const & expr);
vector_expression<double, op_mult, vector> operator*(double value, vector const & v);
```

Implementation of `operator+=` calls *in-place* mult-add kernel

```
vector &
vector::operator+=(vector_expression<double, op_mult,vector> const & expr)
{
     inplace_mult_add(*this, expr.lhs(), expr.rhs());
}
```

ViennaCL: Exercise

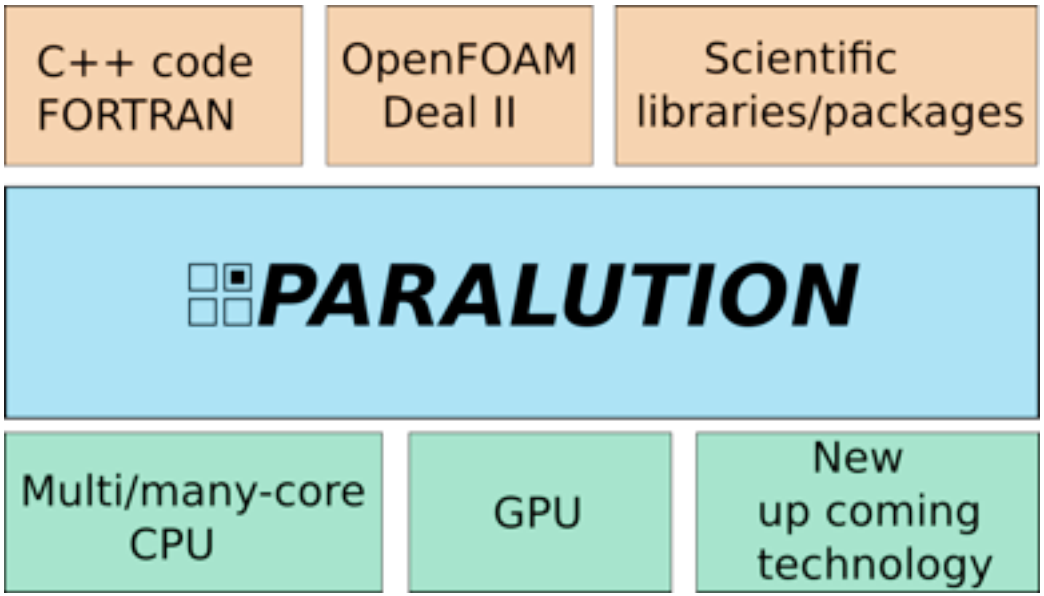
- Compile and run uBLAS version of sparse matrix-vector multiply version on CPU
- Create device version from host version (given)
- `https://github.com/fomics/GPU_Libraries_2013/wiki/Thrust-Exercise`
- Performance comparison

ViennaCL Conclusions

- A logical extension of Boost uBLAS template library
- Backends for OpenMP, OpenCL and CUDA
- Runs on CPUs, Intel Xeon Phi, NVIDIA + AMD GPUs
- Actively supported by ViennaCL team
- Krylov-subspace solvers, minimal spectrum of preconditioners
- Performance results are favorable
- Interoperates with other libraries:
  - Eigen
  - PETSc
  - others...

➡ Promising library supporting high-level linear algebra

PARALUTION: Sparse Linear Algebra on multiple platforms



- Sparse Iterative solvers
- Preconditioners
- *Will be treated in tutorial of Dimitar Lukarski*

Beyond linear algebra

So perhaps single-node sparse/dense linear algebra is covered

- What about distributed memory parallelism?
- What about problems beyond linear algebra?
- Fact: there are solid MPI-GPU development efforts ongoing:
  - ➡ D-MAGMA (see Stan's tutorial)
  - ➡ PETSc (next slides)
  - ➡ Trilinos (next slides)
- *Fact: current multi-node GPU support for non-linear problems, optimizations, eigenvalue problems and others is provisional at best!*

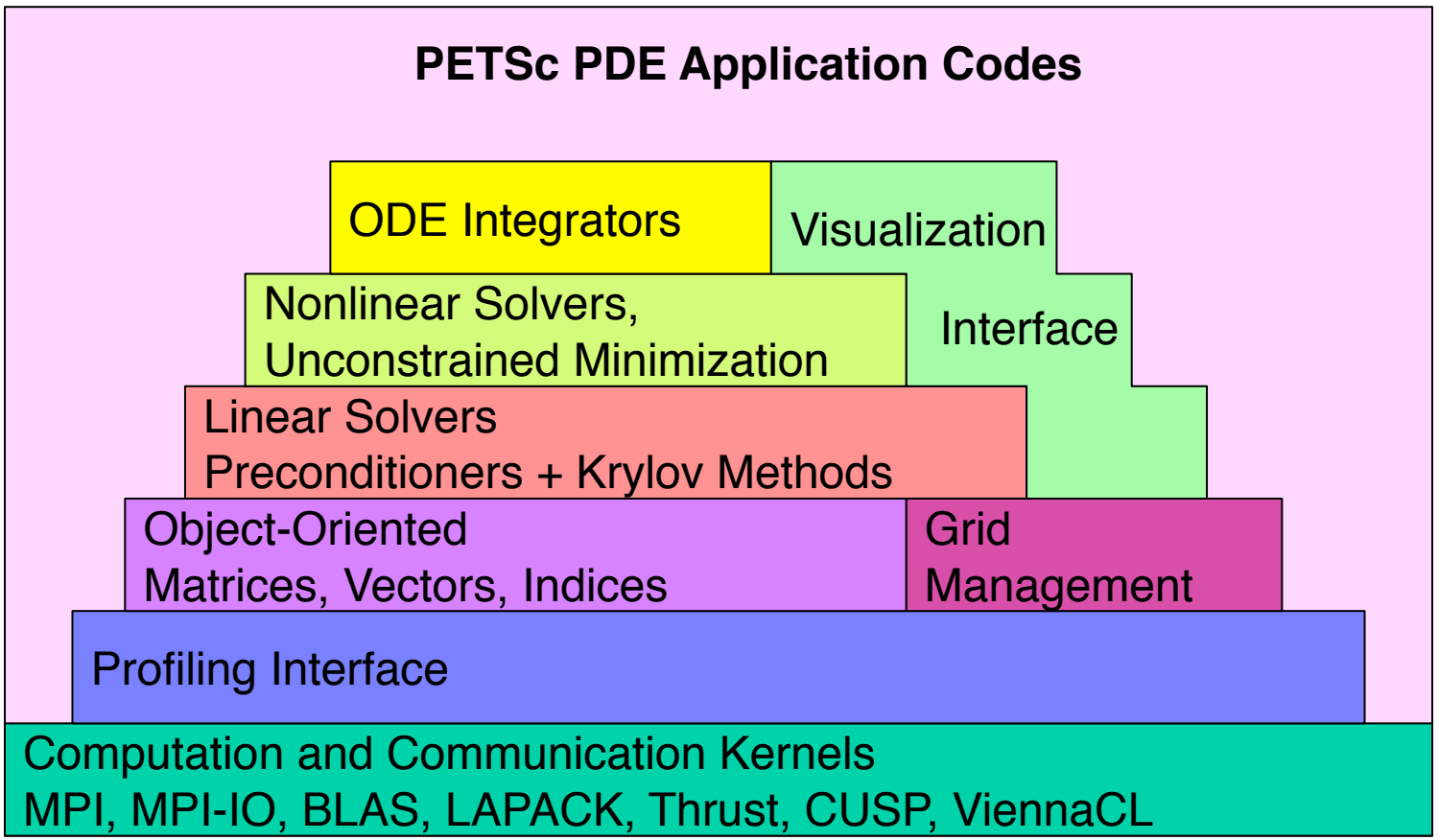
What is PETSc ?

- Supports non-linear PDE problems
- A freely available (and supported!) research code
  - Available via http://www.mcs.anl.gov/petsc
  - Free for everyone, including industrial users
  - Hyperlinked documentation and manual pages for all routines
  - Many tutorial-style examples
  - Support via email: petsc-maint@mcs.anl.gov
  - Current version: 3.3 (released Jun. 5, 2012)
- Portable to any parallel system supporting MPI
  - Tightly coupled systems, e.g., Cray XK6, XE6

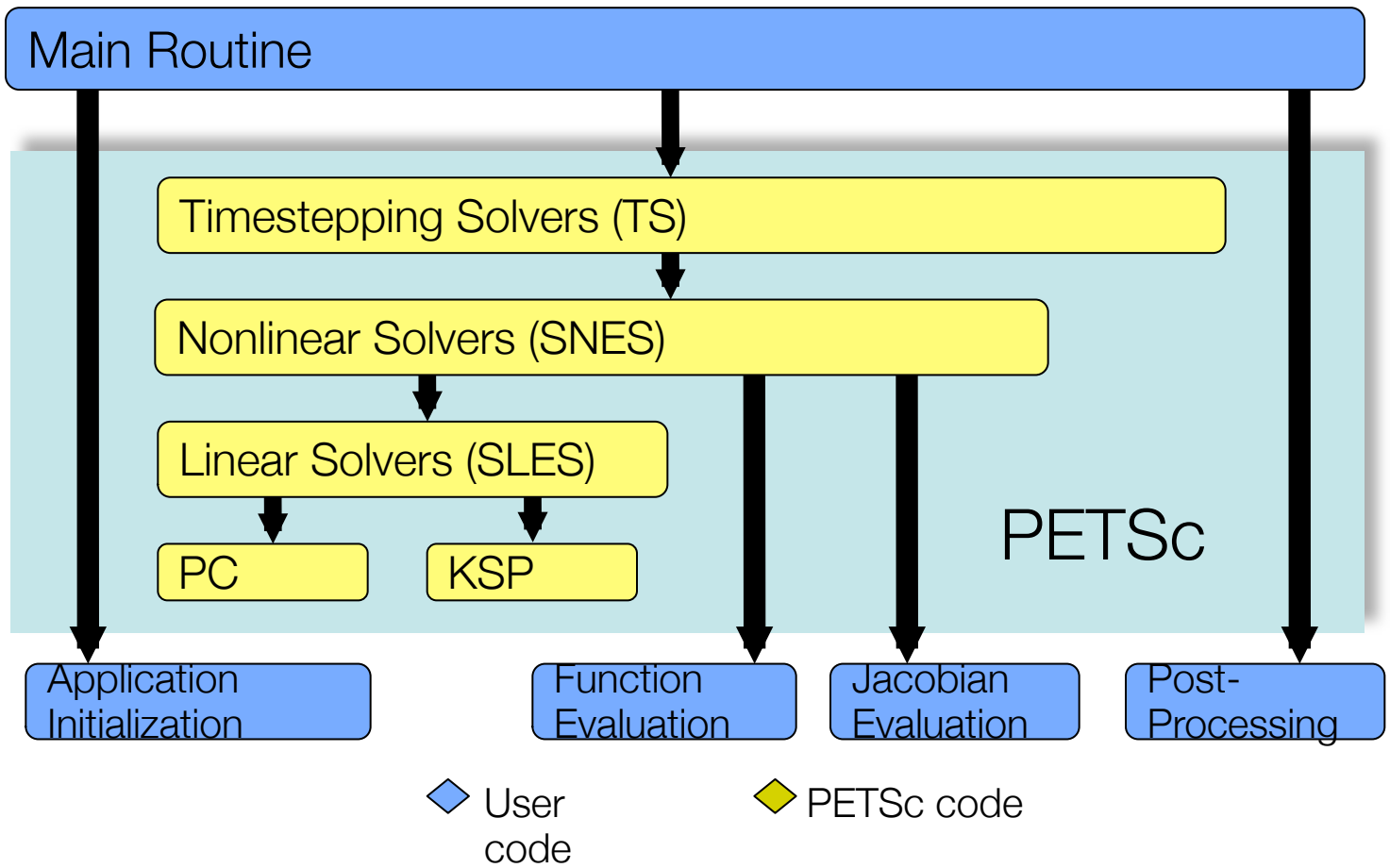
PETSc Concepts

- To specify the mathematics of the problem:
  - Programmer manipulates mathematical objects (sparse matrices, nonlinear equations), algorithmic objects (solvers) and discrete geometry (meshes)
- To solve the problem:
  - Solvers: linear, nonlinear, and time stepping (ODE)
- Parallel computing considerations:
  - Parallel data layout, e.g., structured and unstructured meshes

Structure of PETSc



Flow control for PDE solution



PETSc Programming Model

- **Goals**
  - Portable, runs everywhere, including heterogeneous multi-core
  - Performance
  - Scalable parallelism
- **Approach**
  - Distributed memory, “shared-nothing”
    - Access to data on remote machines or nodes through MPI
  - Can still exploit node parallelism on each node (e.g., SMP), with limitations (see PETSc home page)
  - Hide within parallel objects the details of the communication
  - User orchestrates communication at a higher abstract level than message passing
  - Additional classes added for GPU support

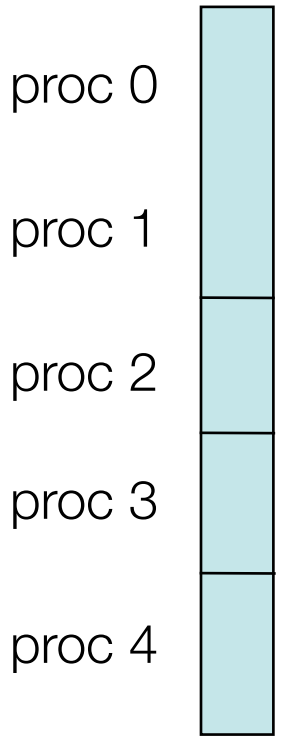
PETSc Data Objects

- Vectors (**Vec**)
  - focus: field data arising in nonlinear PDEs
- Matrices (**Mat**)
  - focus: linear operators arising in nonlinear PDEs (i.e., Jacobians)

beginner	• Object creation
beginner	• Object assembly
intermediate	• Setting options
intermediate	• Viewing
advanced	• User-defined customizations

PETSc Data Objects

- What are PETSc vectors?
  - Fundamental objects for storing field solutions, right-hand sides, etc.
  - Each process locally owns a subvector of contiguously numbered global indices
- Create vectors via
  - VecCreate(...,Vec *)
    - MPI_Comm - processors that share the vector
    - number of elements local to this processor
    - or total number of elements
  - VecSetType(Vec,VecType)
    - Where VecType is
      - VEC_SEQ, VEC_MPI, or VEC_SHARED



Parallel Vector (and Matrix) Assembly

- Processors may generate any entries in vectors and matrices
- Entries need not be generated on the processor on which they ultimately will be stored
- PETSc automatically moves data during the assembly process if necessary

PETSc Communication

- MPI communicators (MPI_Comm) specify collectivity (processors involved in a computation)
- All PETSc creation routines for solver and data objects are collective with respect to a communicator, e.g.,

– `VecCreate(MPI_Comm comm, int m, int M, Vec *x)`

- Some operations are collective, while others are not, e.g.,

– collective: `VecNorm()`

– not collective: `VecGetLocalSize()`

- If a sequence of collective routines is used, they **must** be called in the same order on each processor

PETSc Vector Assembly

- `VecSetValues(Vec,…)`
  - number of entries to insert/add
  - indices of entries
  - values to add
  - mode: `[INSERT_VALUES,ADD_VALUES]`
- `VecAssemblyBegin(Vec)`
- `VecAssemblyEnd(Vec)`

PETSc Matrices

- PETSc matrices are fundamental objects for storing linear operators (e.g., Jacobians)
- Create matrices via

– MatCreate(…,Mat *)
  - MPI_Comm - processors that share the matrix
  - number of local/global rows and columns

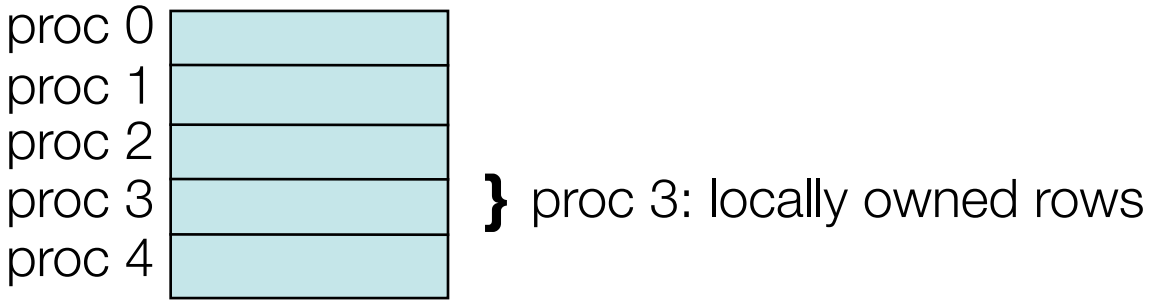
– MatSetType(Mat, MatType), where MatType is one of
  - default sparse AIJ: MPIAIJ, SEQAIJ
  - block sparse AIJ (for multi-component PDEs): MPIAIJ, SEQAIJ
  - symmetric block sparse AIJ: MPISBAIJ, SAEQSBAIJ
  - block diagonal: MPIBDIAG, SEQBDIAG
  - dense: MPIDENSE, SEQDENSE
  - matrix-free

PETSc Matrix Assembly

- MatSetValues(Mat,…)
  - number of rows to insert/add
  - indices of rows and columns
  - number of columns to insert/add
  - values to add
  - mode: [INSERT_VALUES, ADD_VALUES]
- MatAssemblyBegin(Mat, MAT_FINAL_ASSEMBLY)
- MatAssemblyEnd(Mat, MAT_FINAL_ASSEMBLY)

PETSc Parallel Matrix Distribution

Each process locally owns a submatrix of contiguously numbered global rows.



`MatGetOwnershipRange(Mat A, int *rstart, int *rend)`

- `rstart`: first locally owned row of global matrix
- `rend -1`: last locally owned row of global matrix

PETSc Linear Solvers (subset)

Krylov Methods (KSP)

- Conjugate Gradient
- GMRES
- CG-Squared
- Bi-CG-stab
- Transpose-free QMR
- etc.

Preconditioners (PC)

- Block Jacobi
- Overlapping Additive Schwarz
- ICC, ILU via BlockSolve95
- ILU(k), LU (sequential only)
- etc.

PETSc Krylov Subspace Solvers

Only KSP solvers discussed here

- Create a KSP solver:

```
ierr = KSPCreate(PETSC_COMM_WORLD,&ksp)
```

- Use matrix to define an operator

```
ierr = KSPSetOperators(ksp,A,A,XXXX)
```

- Set the solver from the command-line options

```
ierr = KSPSetFromOptions(ksp);
```

- Set the solver from the command-line options

```
ierr = KSPSolve(ksp,b,u);
```

PETSc: GPU support

PETSc GPU Model

- Each MPI process has access to a single GPU, which has its own memory
- Backends for CUSP, CUSparse and ViennaCL available
- New implementations of **Vec** and **Mat (type at run-time)**
  - ➡ Vectors with types `VECSEQCUSP`, `VECMPICUSP`, or `VECCUSP`
  - ➡ Matrices with types `MATSEQAIJCUSP`, `MATMPIAIJCUSP`, or `MATAIJCUSP`
  - ➡ Matrices with types `MATSEQAIJCUSPARSE`, `MATMPIAIJCUSPARSE`, or `MATAIJCUSPARSE`

PETSc: GPU support

Objects support both CPU and GPU copy of data, and carry flags indicated the validity of the data

PETSC_CUDA_UNALLOCATED	Memory not allocated on GPU
PETSC_CUDA_GPU	Values on GPU are current
PETSC_CUDA_CPU	Values on CPU are current
PETSC_CUDA_BOTH	Values on both devices current

Implementations for GPU-CPU data movement

- `VecCUDACopyToGPU`
- `VecCUDACopyFromGPU`
- `...`

these are generally used internally in solvers

PETSc: GPU support

How it works, at least conceptually

- User needs to specify types of vectors and matrices at run time
- Functionality from backend (CUSP, CUSparse, ViennaCL invoked, if available
- Implementation can be transparent to user
- Alternatively, user can program in CUDA and access device objects directly, directly call thrust:: and cusp:: operators, etc.

PETSc Summary

- PETSc library of PDE/ODE solvers
- Extensive selection of solvers, high quality, good support, free
- However: much more effective for new code
- Saddled by design choices, i.e., not thread-safe
- Monolithic: one package tries to solve all, though there are adaptors to other libraries
- *MPI-GPU support in development version, in principle can be invoked at run-time. Backends for CUSP, CUSparse, ViennaCL*

Trilinos: a ‘pearl necklace’ of packages

Object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems

- http://trilinos.sandia.gov
- http://code.google.com/p/trilinos/wiki/TrilinosHandsOnTutorial

Trilinos: parallel packages (with GPU support)

- Basic linear algebra: *Epetra/EpetraExt (C++), Tpetra (C++ templates)*
- Preconditioners: *AztecOO, Ifpack2, ML, Meros*
- Iterative linear solvers: *AztecOO, Belos*
- Direct linear solvers: *Amesos (SuperLU, UMFPACK, MUMPS, ScaLAPACK, …)*
- Non-linear / optimization solvers: *NOX, MOOCHO*
- Eigensolvers: *Anasazi*
- Mesh generation / adaptivity: *Mesquite, PAMGEN*
- Domain decomposition: *Claps*
- Partitioning / load balance: *Isorropia, Zoltan2*

Trilinos: *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.

Trilinos: Generic Parallel Constructs

Node provides generic parallel constructs, user fills in the rest

Parallel FOR:

```
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)`

Parallel REDUCE:

```
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)`

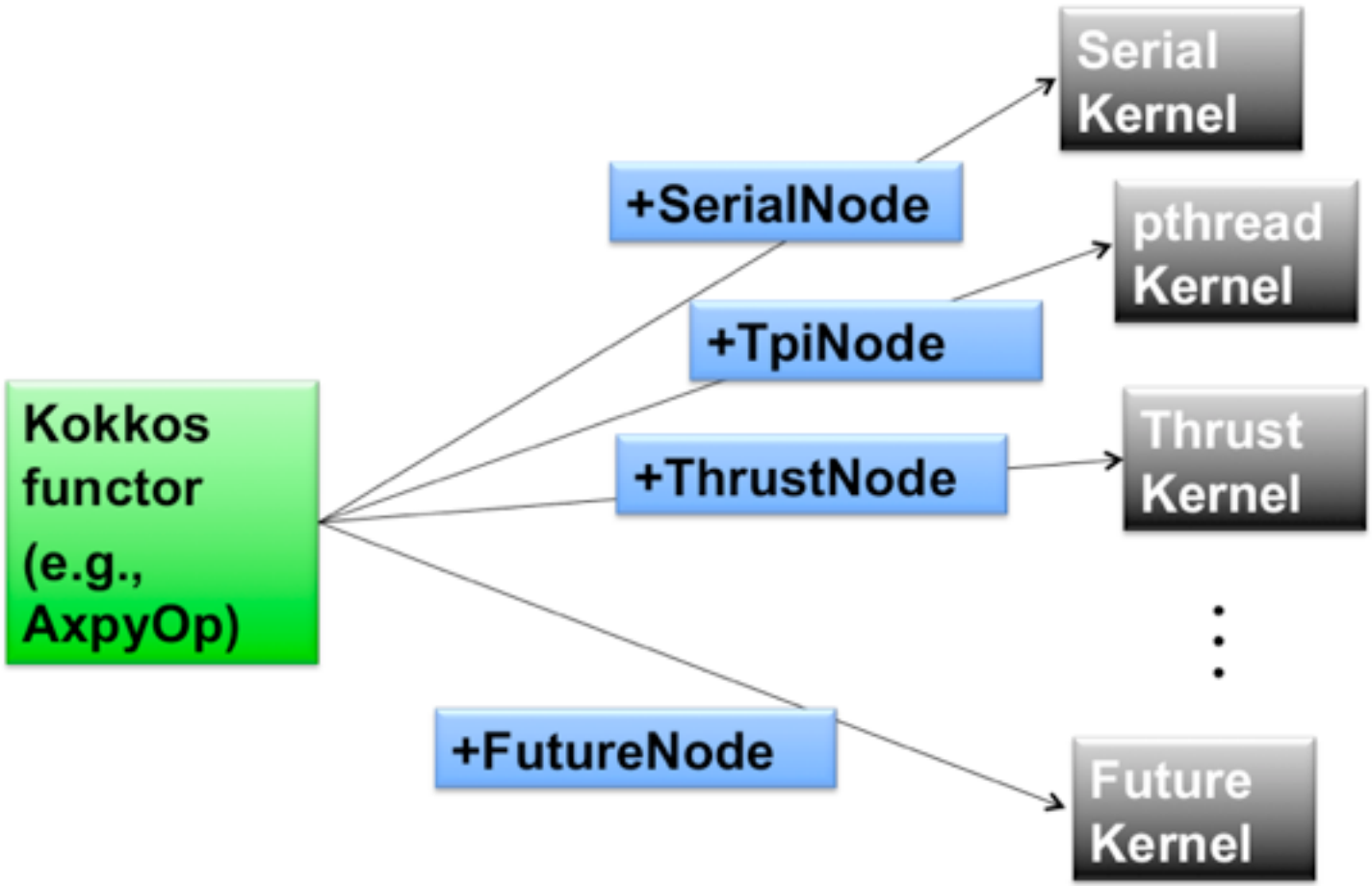
Kokkos: axpy implementation

```
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);
```

Kokkos: dot product

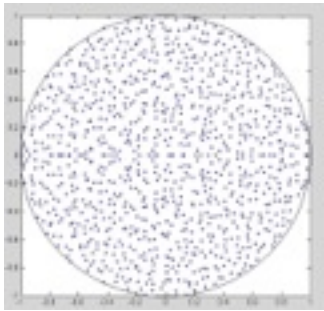
```
template <class WDP> WDP::ReductionType
Node::parallel_reduce(int beg, int end, WDP workdata);

template <class T> struct DotOp
{typedef T ReductionType;
  const T* x, y;
  T identity() { return (T)0;}T alpha, beta;
  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);
```

Kokkos: compile-time kernel specialization



Eigen- and Singular values/vectors

- Formulations:
  - Non-symmetric, real:
  - Symmetric, real: $Ax = \lambda x \Rightarrow Q^T AQ = diag(\lambda_1,..,\lambda_n)$
  - Non-symmetric, non-defective: $Ax = \lambda x \Rightarrow X^{-1} AX = diag(\lambda_1,..,\lambda_n)$
  - Generalized, symm: $Ax = \lambda Bx \Rightarrow Q^T AQ = diag(a_1,..,a_n) \quad Q^T BQ = diag(b_1,..,b_n)$
  - Singular values: $U^T AV = diag(\sigma_1,..,\sigma_n)$
- Interpretation: eigenvectors of A when multiplied by A are parallel to themselves
- Applications in numerous fields



Eigenvalues/vectors of large, sparse matrices

- Techniques based on Lanczos iteration (symm. A)

$$r_0 = q_1; \quad \beta_0 = 1; \quad q_0 = 0; \quad j = 0$$

$$while \quad \beta_j \neq 0 \quad \{$$

$$q_{j+1} = r_j / \beta_j; \quad j = j + 1; \quad \alpha_j = q_j^T A q_j$$

$$r_j = (A - \alpha_j I) q_j - \beta_{j-1} q_{j-1}; \quad \beta_j = \left\| r_j \right\|_2$$

$$\}$$

- Lanczos vectors: q_j
- Form tridiagonal matrix T: diagonal α_j subdiagonal β_j
- Diagonalization of T is stable iterative procedure

Anasazi: design objectives

- Opaque objects: hide low level complexity
- Flexibility to allow for various linear algebra primitives; ease of incorporation with other frameworks
- Provide a small set of ‘turn-key’ eigen-solvers for large (sparse) matrices
- Provide a ‘workbench’ framework in which new methods can be implemented

Anasazi: classes for $Ax = \lambda Bx$

- `Anasazi::Eigenproblem`
  - Contains components of eigen-problem
  - setOperator, SetA, SetB, setPrec, setInitVec
- `Anasazi::Eigensolution`
  - Manages the solution of the eigen-problem
- `Anasazi::Eigensolver`
  - Defines interface which must be met by any solver
  - Currently implemented solvers: BlockDavidson, BlockKrylovSchur, LOBPCG
- `Anasazi::SolverManager`
  - 'Turn-key' class to use existing eigen-solvers

NOX: non-linear equations

- Solve $F(x) = 0$ with $F(x) = \begin{pmatrix} f_1(x_1,\cdots,x_n) \\ \vdots \\ f_n(x_1,\cdots,x_n) \end{pmatrix}$ and $J_{i,j} = \frac{\partial F_i}{\partial x_j}(x)$
- User supplies:
  - Function F(x) evaluation
  - Optional: Jacobian evaluation, preconditioner
- With good guess, convergence quadratic
- Heuristics used to improve first guess
- PETSc interface available
- http://trilinos.sandia.gov/packages/nox/

Trilinos Summary

- Non-monolithic set of packages, some interoperating tightly, some loosely, some not at all
- Large development team, free software, technically advanced, latest solvers, following emerging technologies (e.g. GPUs)
- Solvers are opaque, hard to see internals, bugs can be hard to deal with
- GPU implementation through Kokkos abstraction, only *NOX, Tpetra,Belos, Anasazi, Ifpack2, Zoltan2*

GPU-enabled Libraries Summary

- The take home message is:

  Don't recreate the wheel

- But: a limited number of GPU-libraries available, e.g. CUxxxx (vendor), Thrust, ViennaCL, ...
- Parallel message-passing libraries in development, e.g., D-MAGMA, PETSc, Trilinos

Acknowledgments

- The PETSc team
- Trilinos team
- Roberto Croce: Autumn School organization
- FoMICS and CSCS: sponsoring Autumn School

… and thanks to you for attending!