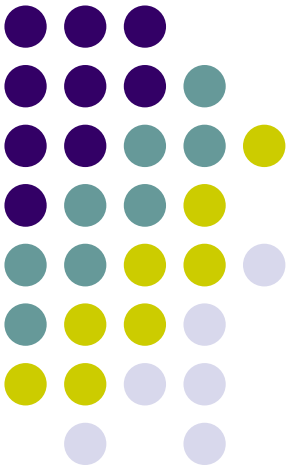
ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Parallel Computing: ways and means

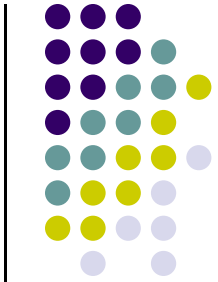
Other options: TBB, c++11, cilk, Chapel

November 16, 2015

Lecture 27



Quote of the Day

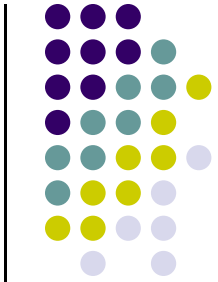


“Choose a job you love, and you will never have to work a day in your life.”

-- Confucius

551 BC – 479 BC

Before We Get Started

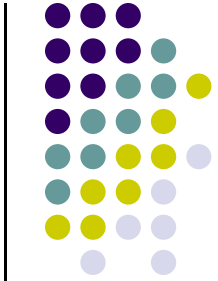


- Issues covered last time:
  - MPI point-to-point communication: non-blocking send/receive
  - MPI collective actions:
    - Synchronization (barriers, wait, etc)
    - Communication (gather, scatter, etc.)
    - Operations (reduce, scan, etc.)

- Today's topics
  - CUDA, OpenMP, MPI: putting things in perspective
  - Other parallel programming models, quick overview
    - TBB
    - C++11
    - Cilk
    - Chapel
    - Charm++ (next lecture)

- Other issues:
  - Wednesday is the last lecture of the semester
  - HW09, due on Wd, Nov. 18, at 11:59 PM
  - Final Project Proposal: if you don't hear from me by Friday, it means that your proposal was fine
  - Second (and last) exam: coming up on 11/23 (Monday) at 7:15 PM (Room: 1610EH)
    - Review on 11/23 in 1610EH during regular lecture hours

MPI – We Scratched the Surface



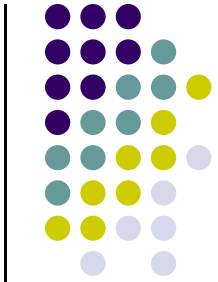
- In some MPI implementations there are more than 300 MPI functions
  - Not all of them part of the MPI standard though, some vendor specific

MPI_Abort, MPI_Accumulate, MPI_Add_error_class, MPI_Add_error_code, MPI_Add_error_string, MPI_Address, MPI_Allgather, MPI_Allgatherv, MPI_Alloc_mem, MPI_Allreduce, MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw, MPI_Attr_delete, MPI_Attr_get, MPI_Attr_put, MPI_Barrier, MPI_Bcast, MPI_Bsend, MPI_Bsend_init, MPI_Buffer_attach, MPI_Buffer_detach, MPI_Cancel, MPI_Cart_coords, MPI_Cart_create, MPI_Cart_get, MPI_Cart_map, MPI_Cart_rank, MPI_Cart_shift, MPI_Cart_sub, MPI_Cartdim_get, MPI_Comm_call_errhandler, MPI_Comm_compare, MPI_Comm_create, MPI_Comm_create_errhandler, MPI_Comm_create_keyval, MPI_Comm_delete_attr, MPI_Comm_dup, MPI_Comm_free, MPI_Comm_free_keyval, MPI_Comm_get_attr, MPI_Comm_get_errhandler, MPI_Comm_get_name, MPI_Comm_group, MPI_Comm_rank, MPI_Comm_remote_group, MPI_Comm_remote_size, MPI_Comm_set_attr, MPI_Comm_set_errhandler, MPI_Comm_set_name, MPI_Comm_size, MPI_Comm_split, MPI_Comm_test_inter, MPI_Dims_create, MPI_Errhandler_create, MPI_Errhandler_free, MPI_Errhandler_get, MPI_Errhandler_set, MPI_Error_class, MPI_Error_string, MPI_Exscan, MPI_File_call_errhandler, MPI_File_close, MPI_File_create_errhandler, MPI_File_delete, MPI_File_get_amode, MPI_File_get_atomicity, MPI_File_get_byte_offset, MPI_File_get_errhandler, MPI_File_get_group, MPI_File_get_info, MPI_File_get_position, MPI_File_get_position_shared, MPI_File_get_size, MPI_File_get_type_extent, MPI_File_get_view, MPI_File_iread, MPI_File_iread_at, MPI_File_iread_shared, MPI_File_iwrite, MPI_File_iwrite_at, MPI_File_iwrite_shared, MPI_File_open, MPI_File_preallocate, MPI_File_read, MPI_File_read_all, MPI_File_read_all_begin, MPI_File_read_all_end, MPI_File_read_at, MPI_File_read_at_all, MPI_File_read_at_all_begin, MPI_File_read_at_all_end, MPI_File_read_ordered, MPI_File_read_ordered_begin, MPI_File_read_ordered_end, MPI_File_read_shared, MPI_File_seek, MPI_File_seek_shared, MPI_File_set_atomicity, MPI_File_set_errhandler, MPI_File_set_info, MPI_File_set_size, MPI_File_set_view, MPI_File_sync, MPI_File_write, MPI_File_write_all, MPI_File_write_all_begin, MPI_File_write_all_end, MPI_File_write_at, MPI_File_write_at_all, MPI_File_write_at_all_begin, MPI_File_write_at_all_end, MPI_File_write_ordered, MPI_File_write_ordered_begin, MPI_File_write_ordered_end, MPI_File_write_shared, MPI_Finalize, MPI_Finalized, MPI_Free_mem, MPI_Gather, MPI_Gatherv, MPI_Get, MPI_Get_address, MPI_Get_count, MPI_Get_elements, MPI_Get_processor_name, MPI_Get_version, MPI_Graph_create, MPI_Graph_get, MPI_Graph_map, MPI_Graph_neighbors, MPI_Graph_neighbors_count, MPI_Graphdims_get, MPI_Grequest_complete, MPI_Grequest_start, MPI_Group_compare, MPI_Group_difference, MPI_Group_excl, MPI_Group_free, MPI_Group_incl, MPI_Group_intersection, MPI_Group_range_excl, MPI_Group_range_incl, MPI_Group_rank, MPI_Group_size, MPI_Group_translate_ranks, MPI_Group_union, MPI_Ibsend, MPI_Info_create, MPI_Info_delete, MPI_Info_dup, MPI_Info_free, MPI_Info_get, MPI_Info_get_nkeys, MPI_Info_get_nthkey, MPI_Info_get_valuelen, MPI_Info_set, MPI_Init, MPI_Init_thread, MPI_Initialized, MPI_Intercomm_create, MPI_Intercomm_merge, MPI_Iprobe, MPI_Irecv, MPI_Irsend, MPI_Is_thread_main, MPI_Isend, MPI_Issend, MPI_Keyval_create, MPI_Keyval_free, MPI_Op_create, MPI_Op_free, MPI_Pack, MPI_Pack_external, MPI_Pack_external_size, MPI_Pack_size, MPI_Pcontrol, MPI_Probe, MPI_Put, MPI_Query_thread, MPI_Recv, MPI_Recv_init, MPI_Reduce, MPI_Reduce_scatter, MPI_Register_datarep, MPI_Request_free, MPI_Request_get_status, MPI_Rsend, MPI_Rsend_init, MPI_Scan, MPI_Scatter, MPI_Scatterv, MPI_Send, MPI_Send_init, MPI_Sendrecv, MPI_Sendrecv_replace, MPI_Ssend, MPI_Ssend_init, MPI_Start, MPI_Startall, MPI_Status_set_cancelled, MPI_Status_set_elements, MPI_Test, MPI_Test_cancelled, MPI_Testall, MPI_Testany, MPI_Testsome, MPI_Topo_test, MPI_Type_commit, MPI_Type_contiguous, MPI_Type_create_darray, MPI_Type_create_f90_complex, MPI_Type_create_f90_integer, MPI_Type_create_f90_real, MPI_Type_create_hindexed, MPI_Type_create_hvector, MPI_Type_create_indexed_block, MPI_Type_create_keyval, MPI_Type_create_resized, MPI_Type_create_struct, MPI_Type_create_subarray, MPI_Type_delete_attr, MPI_Type_dup, MPI_Type_extent, MPI_Type_free, MPI_Type_free_keyval, MPI_Type_get_attr, MPI_Type_get_contents, MPI_Type_get_envelope, MPI_Type_get_extent, MPI_Type_get_name, MPI_Type_get_true_extent, MPI_Type_hindexed, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_lb, MPI_Type_match_size, MPI_Type_set_attr, MPI_Type_set_name, MPI_Type_size, MPI_Type_struct, MPI_Type_ub, MPI_Type_vector, MPI_Unpack, MPI_Unpack_external, MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome, MPI_Win_call_errhandler, MPI_Win_complete, MPI_Win_create, MPI_Win_create_errhandler, MPI_Win_create_keyval, MPI_Win_delete_attr, MPI_Win_fence, MPI_Win_free, MPI_Win_free_keyval, MPI_Win_get_attr, MPI_Win_get_errhandler, MPI_Win_get_group, MPI_Win_get_name, MPI_Win_lock, MPI_Win_post, MPI_Win_set_attr, MPI_Win_set_errhandler, MPI_Win_set_name, MPI_Win_start, MPI_Win_test, MPI_Win_unlock, MPI_Win_wait, MPI_Wtick, MPI_Wtime

- Recall the 20/80 rule: six calls is probably what you need to implement a decent MPI code…
  - MPI_Init, MPI_Comm_Size, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Finalize

The PETSc Library

[The message: Use libraries if available]

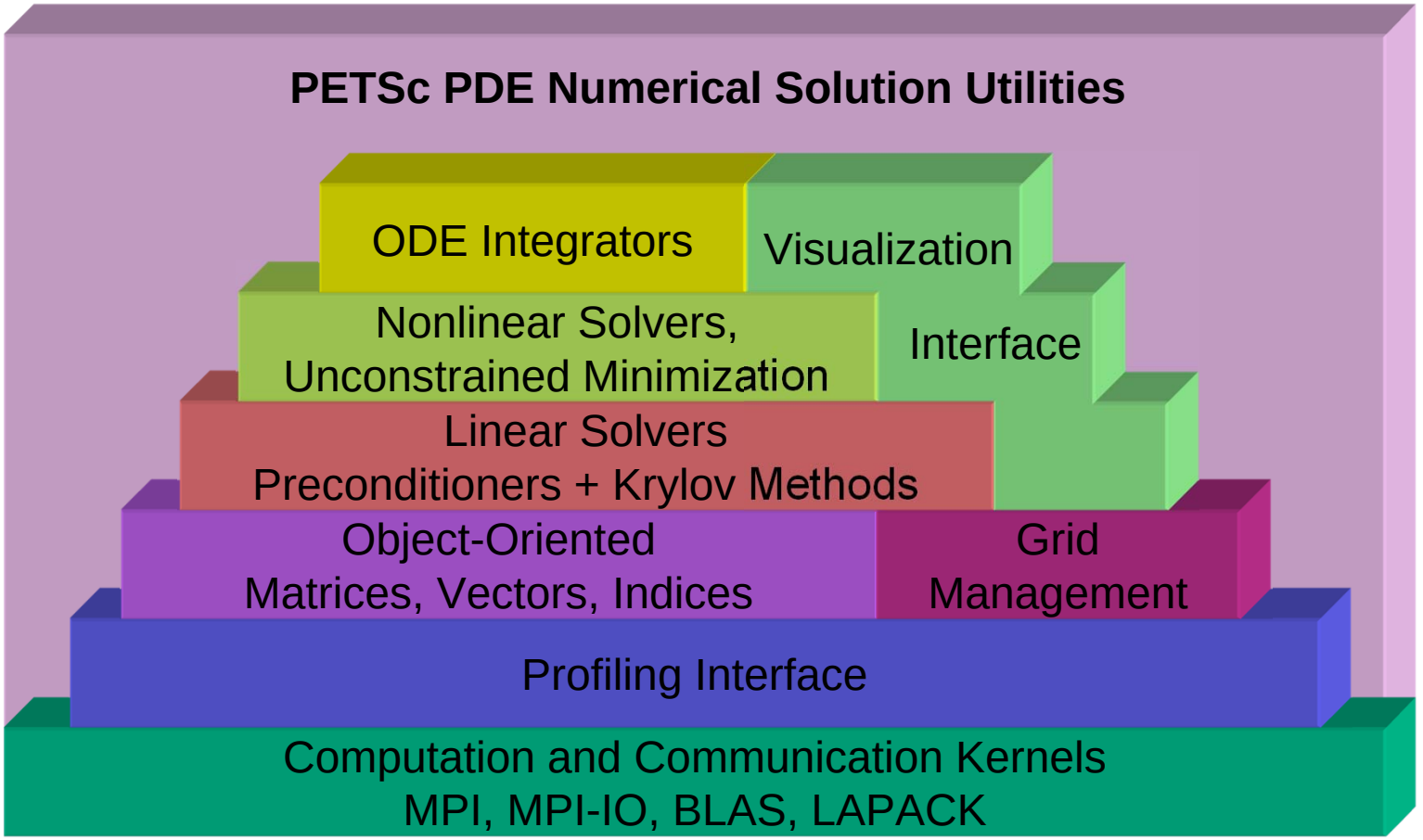


- PETSc: Portable, Extensible Toolkit for Scientific Computation
  - One of the most successful libraries built on top of MPI
  - Intended for use in large-scale application projects,
  - Developed at Argonne National Lab (Barry Smith, technical lead)
  - Open source, available for download at http://www.mcs.anl.gov/petsc/petsc-as/

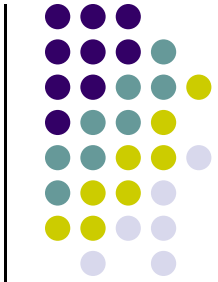
- Provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution (numerical integration)

- Also provides utility routines, such as
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)

Structure of PETSc



PETSc Numerical Components



Nonlinear Solvers		
Newton-based Methods		Other
Line Search	Trust Region	

Time Steppers			
Euler	Backward Euler	Pseudo Time Stepping	Other

Krylov Subspace Methods							
GMRES	CG	CGS	Bi-CG-STAB	TFQMR	Richardson	Chebychev	Other

Preconditioners						
Additive Schwartz	Block Jacobi	Jacobi	ILU	ICC	LU (Sequential only)	Others

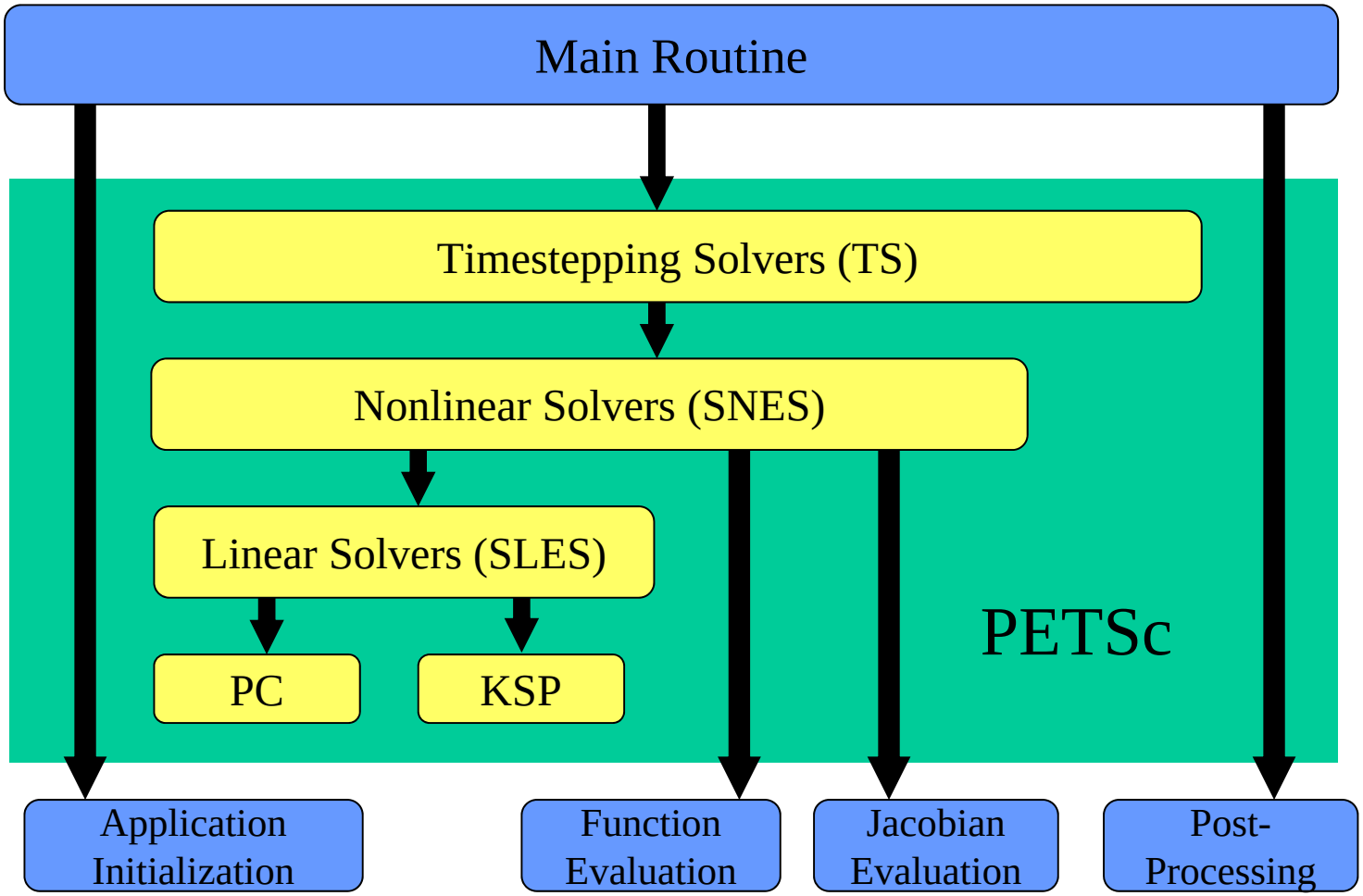
Matrices					
Compressed Sparse Row (AIJ)	Blocked Compressed Sparse Row (BAIJ)	Block Diagonal (BDIAG)	Dense	Matrix-free	Other

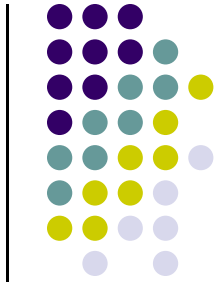
Distributed Arrays

Index Sets			
Indices	Block Indices	Stride	Other

Vectors

Flow Control for PDE Solution

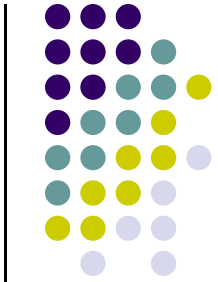




CUDA, OpenMP, MPI:

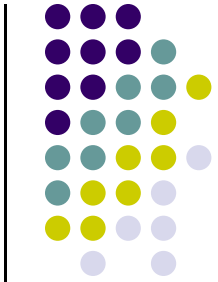
Putting Things in Perspective

Pros, CUDA



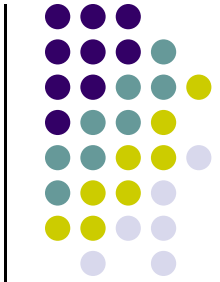
- Many remarkable success stories when the application targeted is data parallel and has high arithmetic intensity
  - Can provide sometimes one order of magnitude speed-ups
- Very affordable – democratization of parallel computing
  - At a price of $10K you get half the flop rate of what an IBM BlueGene/L delivered six or seven years ago
- Ubiquitous
  - Present on more than 100 million computers today support CUDA
- Good productivity tools

Cons, CUDA



- To extract last ounce of performance that makes GPU computing great you need to understand the computational model and the underlying hardware
- Somewhat limited amount of device memory
  - 12 GB on K40 or 24 GB on K80
- Until the CPU and GPU are fully integrated, the PCI connection is impacting performance
  - To be addressed by NVLink next year: higher bandwidth, lower latency
- For true HPC, using CUDA in conjunction with MPI not a stroll in the park

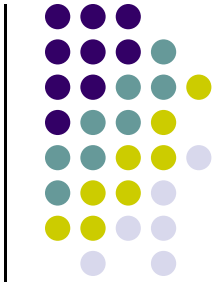
What Would Be Nice…



- CPU and GPU operate in the same physical memory space – essentially no penalty for main memory access from the GPU
- Get 3D memory out fast
- NOTE: what I had here two years ago got implemented

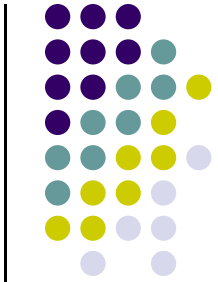
What Would Be Nice…

[the 2013 slide, two years ago]



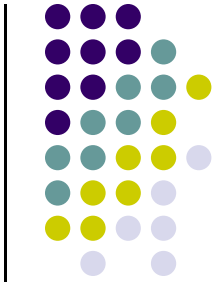
- The global memory bandwidth should increase at least as fast as the rate at which the number of scalar processors increases
- Integrate CPU & GPU so that concept of global device memory disappears
- Have the OpenACC standard succeed for seamless parallel accelerator and/or many-core programming

Pros of OpenMP

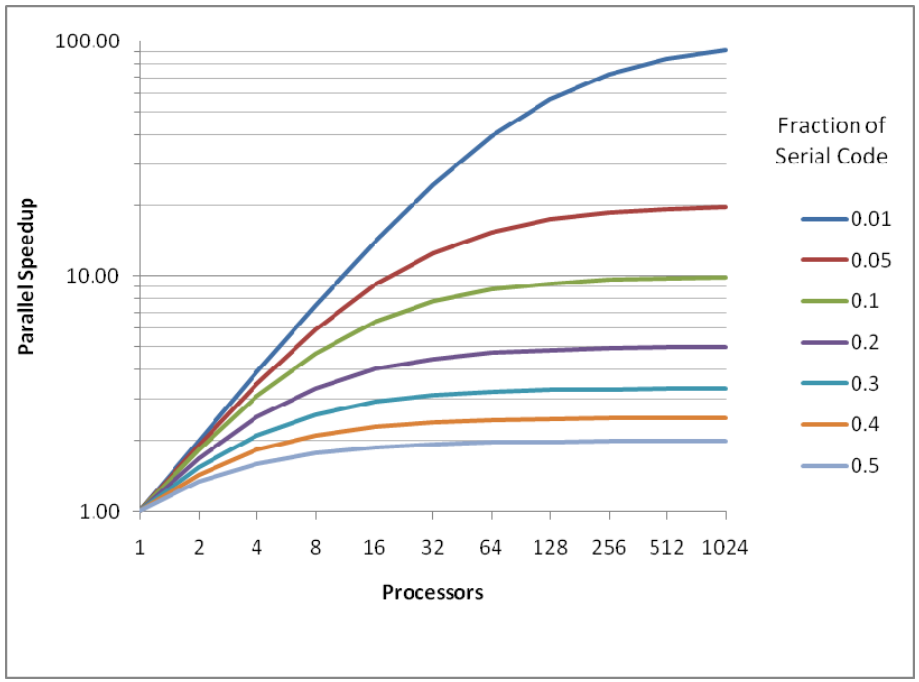


- Because it takes advantage of shared memory, the programmer does not need to worry (that much) about data placement
- Programming model is “serial-like”, thus conceptually simpler than message passing
- Compiler directives are generally simple and straightforward to use
- Legacy serial code does not need to be rewritten

Cons of OpenMP



- The model doesn't scale up all that well
- In general, only moderate speedups can be achieved
  - Because OpenMP codes tend to have serial-only portions, Amdahl's Law prohibits substantial speedups



- Amdahl's Law:

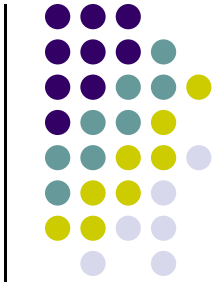
  s = Fraction of serial execution time that cannot be parallelized

  N = Number of processors

$$\text{Execution speedup: } = \frac{1}{s + \frac{1-s}{N}}$$

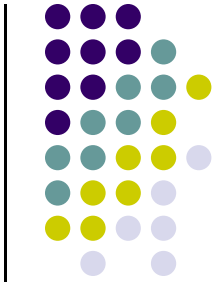
- Ideal targets for OpenMP: big loops that dominate execution time

Why Plot on Previous Slide Doesn't Apply to MPI?



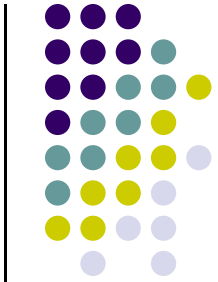
- Source of problem is different
  - MPI – multiple programs running on different nodes
    - The show stopper is the communication latency over interconnect
    - Each program most often sequential yet it might choose to use OpenMP
      - For the latter, the plot becomes relevant
  - OpenMP – running on one node
    - The show stoppers:
      - How much code you get to run under OpenMP (fraction of parallelism)
      - Cache coherence issues

Pros of MPI



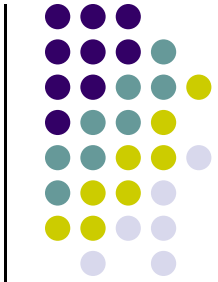
- Good vendor support for the standard
  - It was great that the community converged upon a standard (something that can't be said about GPU computing)
- Proven parallel computing solution, demonstrated to scale up to hundreds of thousands of cores
- Can be deployed both for distributed as well as shared memory architectures
- Today it is synonym with High Performance Computing
  - Provided a clear and relatively straightforward framework for reaching Petaflops grade computing

Cons of MPI



- Very targeted towards solving a certain type of computing problem
  - Very “Scientific Computing” oriented, particularly serving the domain decomposition folks (solution of PDEs, finite element)
    - Doesn’t handle heterogeneity all that well
- The interconnect is Achilles' heel. Top bandwidths today are comparable to what you get over PCI-Express
  - Latency typically is significantly worse though
- Like CUDA, works well only for applications where you don’t have to communicate all that much (high arithmetic intensity)

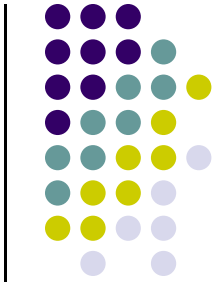
HPC and MPI related blog post



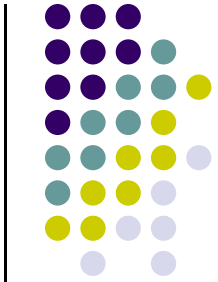
- Please make sure you read this post:

http://www.dursi.ca/hpc-is-dying-and-mpi-is-killing-it/

General Remarks on Parallel Computing



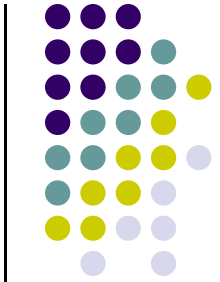
- Parallel Computing is and will be relevant at least for this decade
- Nonetheless, it continues to be challenging
  - Switching your thinking about getting a job done from sequential to parallel mode takes some time but it's a skill that must be acquired
    - Parallel Programming more difficult than programming for Sequential Computing
  - Productivity tools (debuggers, profilers, build solutions) more challenging to master
  - Need to understand the problem that you solve, the pros/cons of the parallel programming models available, and of the hardware on which your code will run



Other Parallel Programming Alternatives

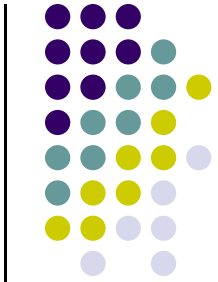
- Brought to you w/ input from Hammad and Tim Haines

Goal of This Segment



- “good to know” material, might provide some feature that simplifies your life
- No time for in-depth coverage
- Most often very similar in functionality provided to something that we have already covered
  - Chapel is somewhat different
    - Quite different than MPI, more akin to Charm++
    - We’ll discuss Charm++ on Wd

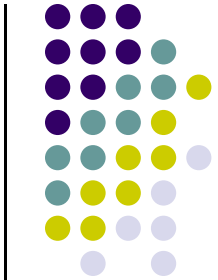
Intel® Threading Building Blocks (TBB)



- C++ library that abstracts loop-based, shared-memory parallelism like OpenMP into dynamic, composable parallelism
- Supports data-parallel and task-parallel programming paradigms
- You can explicitly control the threads, yet you don't have to
  - TBB Mantra: “Think tasks, not threads”
- Runtime system keeps track of threads and uses **work stealing** to keep each thread busy
  - Similar to OpenMP “tasks”
- Some features are drop-in replacements for standard C++ algorithms

```
#include <tbb/parallel_sort.h>
#include <vector>
#include <algorithm>


int main() {
    std::vector<int> x(1000000);
    // fill vector (maybe from file, etc.)


    // Standard C++ sort is serial
    std::sort(x.begin(),x.end());


    // TBB sort is parallel
    tbb::parallel_sort(x.begin(),x.end());
}
```

[Tim Haines]→



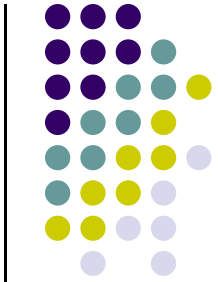
```cpp
#include <tbb/parallel_for.h>
#include <tbb/task_scheduler_init.h>
#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator>


int main() {
    //Initialize the TBB runtime system with two threads
    // NOTE: This is optional!
    tbb::task_scheduler_init init(2);


    constexpr size_t N = 100000;
    std::vector<float> x(N);
    using size_type = decltype(x)::size_type;
    using value_type = decltype(x)::value_type;


    // process grain_size many elements per task.
    constexpr size_type grain_size = 10000;


    /*
     * We have 10 tasks (N/grain_size), but only two threads!
     * The runtime system shuffles work between the threads
     * using work stealing.
     */
    tbb::parallel_for(size_type(0), N, grain_size,
            [&x,grain_size](size_type i) {
                auto start = x.begin() + i;
                std::transform(start,start+grain_size,start, [] (const value_type &v){ return 2+v; });
            }
    );


    // print the first few (should all be just 2)
    std::copy(x.begin(), x.begin() + 10,
            std::ostream_iterator<value_type>(std::cout, " "));
    std::cout << std::endl;
}
```

```cpp
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>
#include <iostream>


int main() {
    std::vector<int> x(1000000);
    std::iota(x.begin(),x.end(),1);
    std::shuffle(x.begin(),x.end(),std::mt19937());

    using namespace std::chrono;

    auto start = high_resolution_clock::now();
    std::sort(x.begin(),x.end());
    auto end = high_resolution_clock::now();

    std::cout << "std::sort took "
              << duration_cast<milliseconds>(end-start).count()
              << " ms\n";

    // pollute the cache and force a reload
    std::shuffle(x.begin(),x.end(),std::mt19937());

    start = high_resolution_clock::now();
    tbb::parallel_sort(x.begin(),x.end());
    end = high_resolution_clock::now();

    std::cout << "tbb::parallel_sort took "
              << duration_cast<milliseconds>(end-start).count()
              << " ms\n";
}
```

The TBB sort ~3x faster on Tim's desktop

[Tim Haines]→

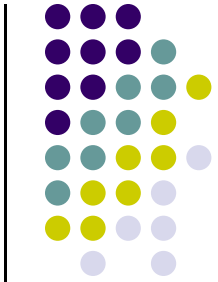
Intel® Threading Building Blocks (TBB)



- TBB Pros:
  - Simple algorithm-centric library written in portable C++11
  - Dynamic, data- and task-based parallelism
  - Provides several concurrent (lock-free) data structures
  - Can easily be incrementally introduced into existing sequential code
  - Offers an aligned, cache-aware memory allocator
  - Plays nice with MPI, processor affinity, and ccNUMA

[Tim Haines]→

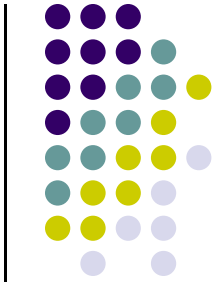
Intel® Threading Building Blocks (TBB)



- TBB Cons:
  - Target machine must have entire library installed (including platform-specific memory allocator)
  - Doesn't always play nice with other shared-memory libraries like OpenMP, `pthreads` or C++ `std::thread`

[Tim Haines]→

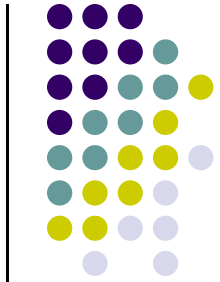
Intel® Threading Building Blocks (TBB)



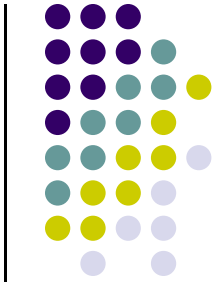
- TBB opens multiple opportunities for parallelism
  - `parallel_do`
  - `pipeline`
  - `filter`
  - `recursive splitters`
  - `etc.`

- Reference:
  - Intel Threading Building Blocks. Reinders, James. O'Reilly Media. 2007

[Tim Haines]→



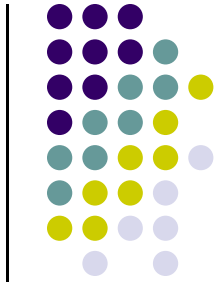
C++11 Thread Support Library



- Native, task-based parallelism
- Sophistication level:
  - Between that of `pthreads` and a “high-level” threading library like `TBB`
- Bottom line: a collection of *type-safe threading features* upon which larger libraries can be built in a platform-independent fashion
- It enables more flexible task-based parallelism than OpenMP's '`task`' or '`sections`'
  - In particular, it actually handles exceptions

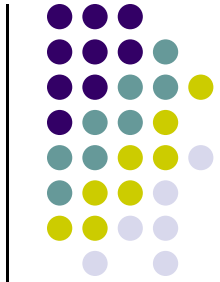
Reference: *C++ Concurrency in Action*. Williams, Anthony. Manning Publications Co. 2012.

[Tim Haines]→

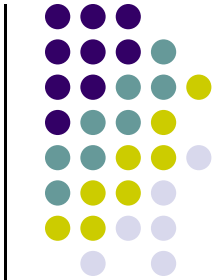


```cpp
#include <vector>
#include <algorithm>
#include <iostream>
#include <numeric>
#include <stdexcept>
#include <future>
#include <iterator>
#include <type_traits>


class empty_container: public std::exception {
    constexpr static const char* msg = "Empty container";
public:
    virtual const char* what() const noexcept override {
        return msg;
    }
};


int main() {
    auto mean = [](const auto begin, const auto end) { // C++14 polymorphic lambda
        if(auto size = std::distance(begin,end)) {
            return std::accumulate(begin,end,typename std::decay<decltype(*begin)>::type()) / size;
        }
        throw empty_container();
    };


    auto inclusive_scan = [](const auto begin, const auto end) {
        if(auto size = std::distance(begin,end)) {
            std::vector<typename std::decay<decltype(*begin)>::type> sum(size);
            std::partial_sum(begin,end,sum.begin());
            return sum;
        }
        throw empty_container();
    };


    std::vector<int> x(10000);
    std::iota(x.begin(), x.end(), 1);


    // These calls do not block
    auto avg = std::async(std::launch::async, mean, x.begin(), x.end());
    auto inc_scan = std::async(std::launch::async, inclusive_scan, x.begin(), x.end());


    // Calling get() blocks the future until the respective computation is done
    std::cout << "average = " << avg.get() << std::endl;


    auto is = inc_scan.get();
    std::copy(is.begin(), is.begin() + 10, std::ostream_iterator<decltype(x)::value_type>(std::cout, " "));
    std::cout << std::endl;
}
```

[Tim Haines]→



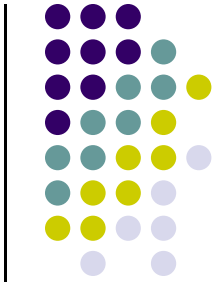
Intel Cilk™ Plus



- Extends C/C++ with a handful of keywords
- Every Cilk program has serial semantics
- Cilk provides performance guarantees based on performance abstractions
- Cilk is processor-oblivious
- Cilk's runtime system automatically manages low-level aspects of parallel execution, including protocols, load balancing, and scheduling
- Cilk supports speculative parallelism

[Hammad]→

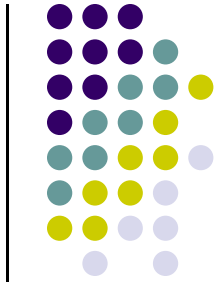
Intel Cilk™ Plus



- Cilk Plus defines three keywords
  - **cilk_spawn**
    - Allows runtime to spawn thread if needed
    - Will work steal from other cores if core doesn't have enough work
  - **cilk_sync**
    - Wait for spawned child functions to finish
  - **cilk_for**
    - Optimized for loop construct

[Hammad]→

Fibonacci Example



```
int fib (int n) {
if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }}
```

C

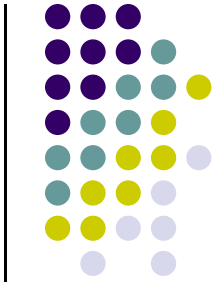
Cilk code

```
int fib (int n) {
 if (n<2) return (n);
 else {
   int x,y;
   x = cilk_spawn fib(n-1);
   y = cilk_spawn fib(n-2);
   cilk_sync;
   return (x+y);
 }}
```

- `Cilk` is a faithful extension of C
- A `Cilk` program's "serial elision" is a valid C program
- Cilk provides no new data types

[Hammad]→

Basic Cilk Keywords



Cilk code

```
int fib (int n) {
 if (n<2) return (n);
 else {
   int x,y;
   x = cilk_spawn fib(n-1);
   y = cilk_spawn fib(n-2);
   cilk_sync;
   return (x+y);
 }}
```

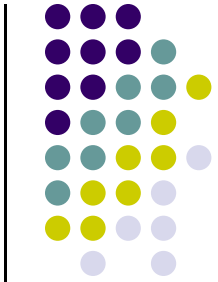
The named **child** Cilk procedure can execute in parallel with the **parent** caller.

Control cannot pass this point until all spawned children have returned.

This is basically identical to OpenMP's "task" Fibonnaci example

[Hammad]→

Parallel For Loop



C code

```
for (int i = 0; i < 8; ++i){ do_work(i); }
```

Cilk code (not optimal)

```
for (int i = 0; i < 8; ++i) {
   cilk_spawn do_work(i);
}
cilk_sync;
```

Similar to spawning OpenMP tasks

Optimal cilk code

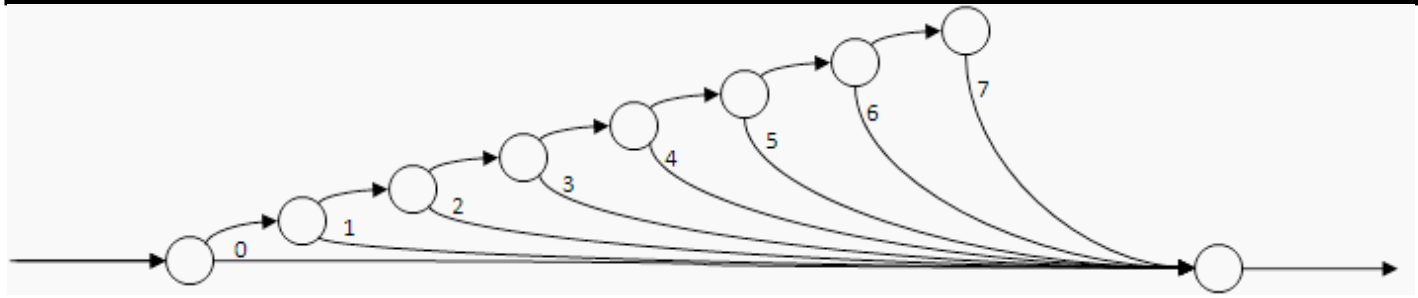
```
 cilk_for (int i = 0; i < 8; ++i
 { do_work(i); }
```

Similar to OpenMP "`parallel for`"

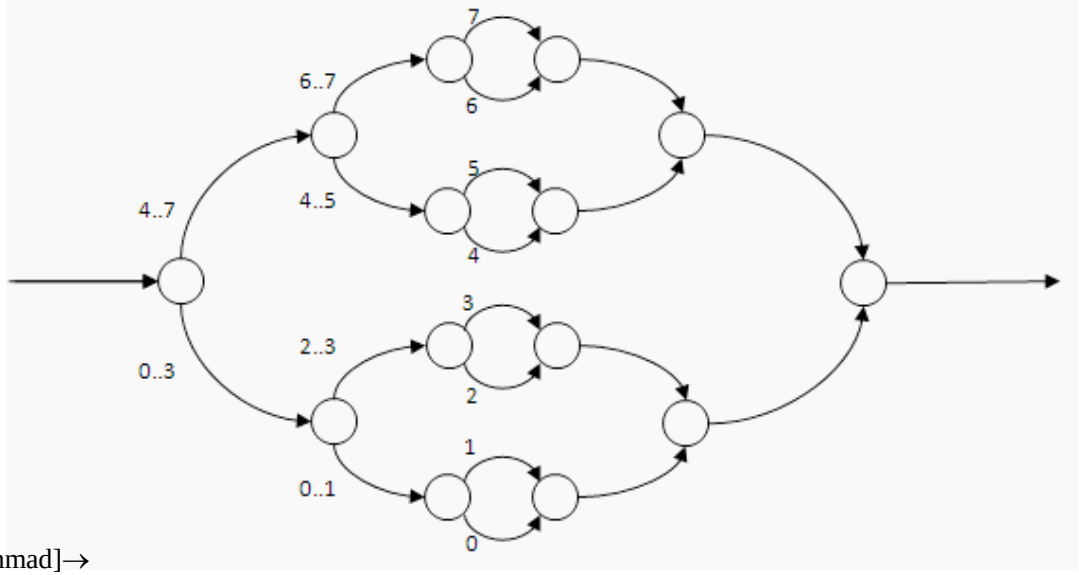
[Hammad]→

Difference Between spawn and for

```
for (int i = 0; i < 8; ++i)
{cilk_spawn do_work(i); } cilk_sync;
```

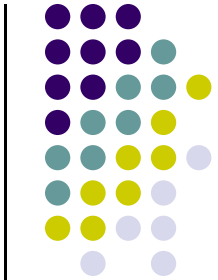



```
cilk_for (int i=0;i<8;++i){ do_work(i);}
```



[Hammad]→

Intel Cilk Plus Reducer Library

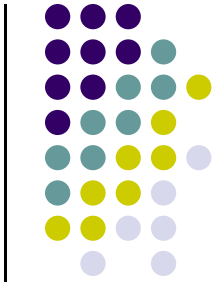


- Reducers are lock free structures for parallel applications
- Example: Fibonacci with an add reducer

```
cilk::reducer< cilk::op_add<int> > fib_sum(0);

void fib_with_reducer_internal(int n) {
    if (n < 2) {
        *fib_sum += n;
    }
    else {
        cilk_spawn fib_with_reducer_internal(n-1);
        fib_with_reducer_internal(n-2);
        cilk_sync;
    }
}
```


[Hammad]→

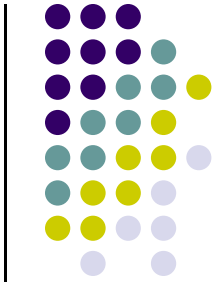
Intel Cilk Plus Reducer Library



reducer_list_append	Creates a list by adding elements to the back.
reducer_list_prepend	Creates a list by adding elements to the front.
reducer_max	Calc max value of a set of values.
reducer_max_index	Calc max value and index of that value of a set of values.
reducer_min	Calc min value of a set of values.
reducer_min_index	Calc min value and index of that value of a set of values.
reducer_opadd	Calc sum of a set of values.
reducer_opand	Calc binary AND of a set of values.
reducer_opor	Calc binary OR of a set of values.
eeducer_opxor	Calc binary XOR of a set of values.
reducer_string	Accumulates a string using append operations.
reducer_wstring	Accumulates a "wide" string using append operations.
reducer_ostream	An output stream that can be written in parallel.

[Hammad]→

The Intel Cilk™ Plus Development Tools

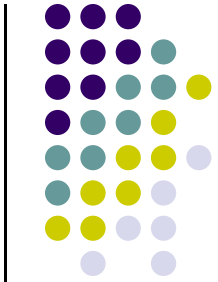


- **Cilk Screen**
  - Reports race conditions in Cilk Plus constructs that could be encountered during execution

- **Cilk View**
  - Helps w/ gauging of the performance of a Cilk Plus parallel program
  - Predicts how performance will scale on multiple processor systems
  - Automatically benchmarks a Cilk Plus program running on one or more processors

[Hammad]→

Cilk Plus Array Notation



- Extension to C++ for parallel operations on arrays

```
A[start_index:length:stride] = …
```

- Static array operations can be simplified

```
for (i = 0; i < 10; i++)
D[i] = A[i] + B[i];
```

Standard C/C++

```
D[:] = A[:] + B[:];
```

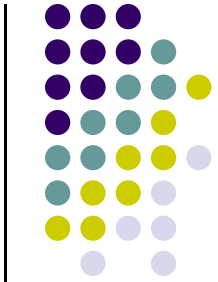
Cilk Plus

- Dynamic arrays need a start and length

```
A[start_index:length] = …
```


[Hammad]→

Cilk Plus Array Notation



- Scatter and gather operations are simplified

```
C[:] = A[B[:]]
```

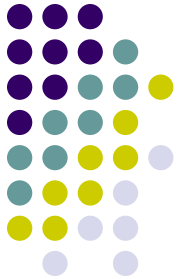
  - Gather elements to C from A given indices in B

```
A[B[:]] = C[:]
```

  - Scatter elements to A given indices in B from C

[Hammad]→

Cilk Plus
Built-in Functions for Array Sections



- Build in reduce functions for arrays

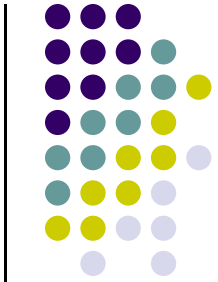
`__sec_reduce_add (A[:])`
  - Returns sum of elements

`__sec_reduce_max (A[:])`
  - Returns max element

`__sec_reduce_max_index (A[:])`
  - Returns index of max element

[Hammad]→

SIMD enabled functions



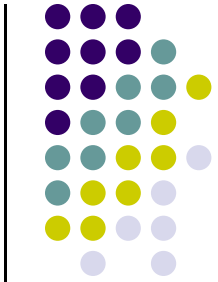
- Adding **`__declspec(vector)`** to a function will generate vectorized code

```
__declspec(vector) float myfunction(float s, float k, float r);

void Fun(float S[SIZE],float r, float out[SIZE]){
    out[:] = myfunction(S[:], r);
}
```

`SIZE` is **`define`**d elsewhere.
The vectorization magic happens at compile time.

[Hammad]→

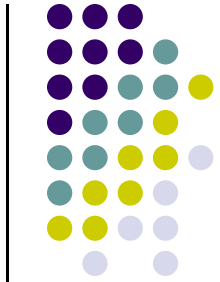
Force Loop Vectorization



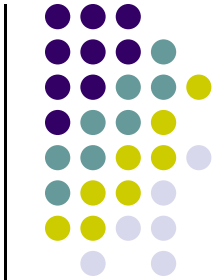
- Use **`#pragma simd`** to enforce loop vectorization
- Compile will give warning if code cannot be vectorized

```
void add_floats(float *a, float *b, float *c, float *d, float *e, int n)
{
    int i;
    #pragma simd
    for(i=0;i<n;i++){
      a[i]=a[i]+b[i]+c[i]+d[i]+e[i];
    }
}
```


[Hammad]→



Chapel



- Chapel*:* Cascade High-Productivity Language
  - Promoted by Cray (since 2009)

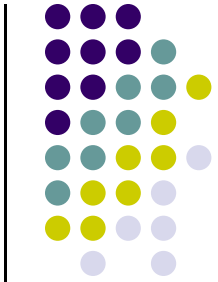
- Overall goal: “solve the parallel programming problem”
  - Simplify the creation of parallel programs
  - Support their evolution to extreme-performance, production-grade codes
  - Emphasize generality

- Desirable attributes, guiding the language design process:

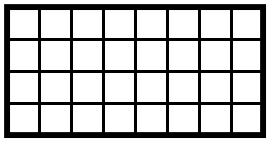
  1) Multithreaded parallel programming
  2) Locality-aware programming
  3) Object-oriented programming
  4) Generic programming and type inference

[Hammad]→

A Simple Domain Declaration

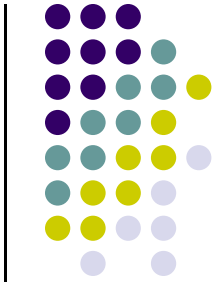


```
var m: integer = 4;
var n: integer = 8;

var D: domain(2) = [1..m, 1..n];
```

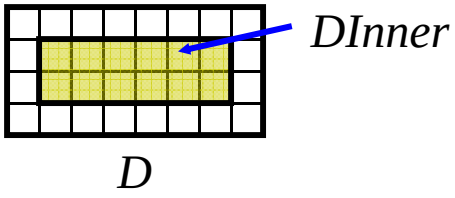


D

A Simple Domain Declaration



```
var m: integer = 4;
var n: integer = 8;

var D: domain(2) = [1..m, 1..n];
var DInner: domain(D) = [2..m-1, 2..n-1];
```



Domain Uses

- Declaring arrays:

```
var A, B: [D] float;
```

- Sub-array references:

```
A(DInner) = B(DInner);
```

- Sequential iteration:

```
for (i,j) in DInner { …A(i,j)… }
```

or: `for ij in DInner { …A(ij)… }`

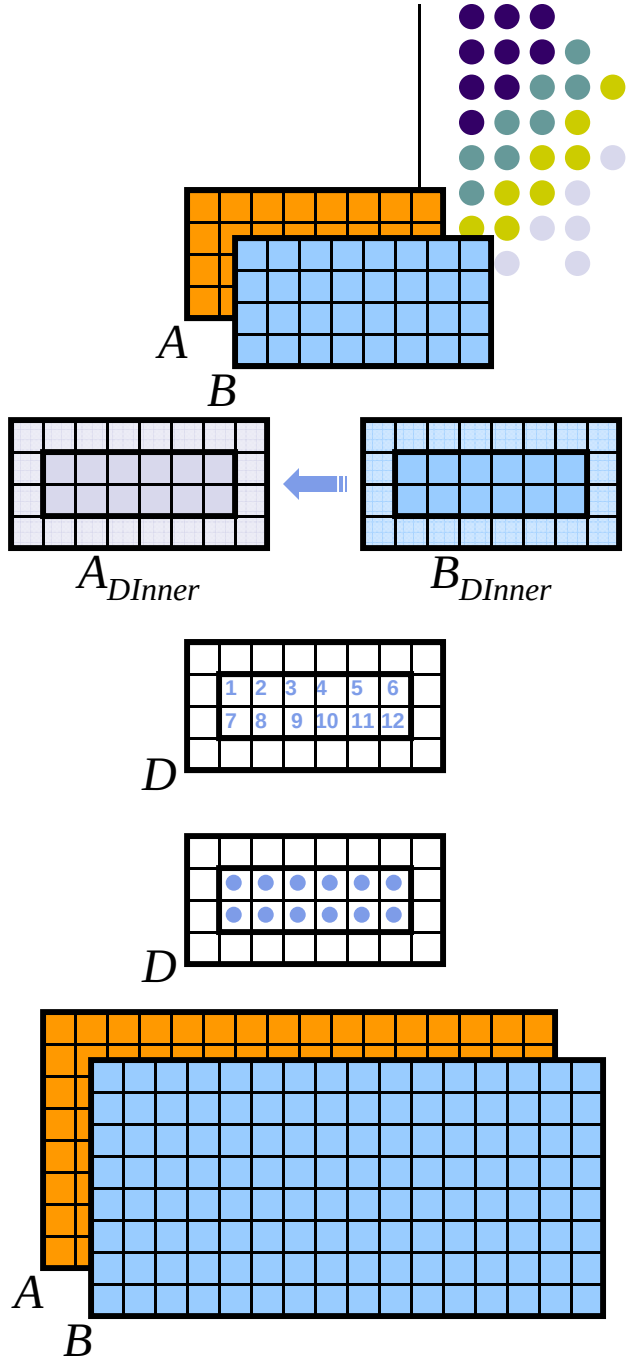
- Parallel iteration:

```
forall ij in DInner { …A(ij)… }
```

or: `[ij in DInner] …A(ij)…`

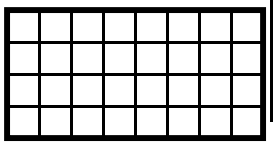
- Array reallocation:

```
D = [1..2*m, 1..2*n];
```



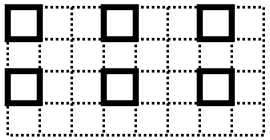
Other Arithmetic Domains

```
var D2: domain(2) = (1,1)..(m,n);
```



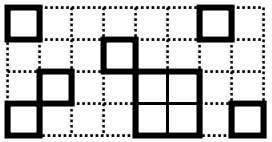
D2

```
var StridedD: domain(D) = D by (2,3);
```



StridedD

```
var indexList: seq(index(D)) = …;
var SparseD: sparse domain(D) = indexList;
```

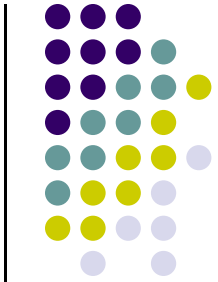


SparseD

- These are language facilities to map computations to domains

[Hammad]→

Task Parallelism



- **co-begins:** indicate statements that may run in parallel:

```
computePivot(lo, hi, data);
cobegin {
  Quicksort(lo, pivot, data);
  Quicksort(pivot, hi, data);
}
```

```
cobegin {
  ComputeTaskA(…);
  ComputeTaskB(…);
}
```

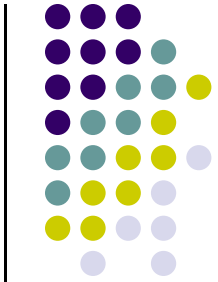
- **atomic sections:** support atomic transactions

```
atomic {
  newnode.next = insertpt;
  newnode.prev = insertpt.prev;
  insertpt.prev.next = newnode;
  insertpt.prev = newnode;
}
```

- **sync and single-assignment variables:** synchronize tasks

[Hammad]→

Locality-aware Programming



- **locale:** machine unit of storage and processing
- Programmer specifies number of locales on executable command-line

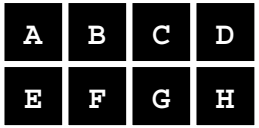
```
prompt> myChapelProg -nl=8
```

- Chapel programs provided with built-in locale array:

```
const Locales: [1..numLocales] locale;
```

- Users may use this to create their own locale arrays:

```
var CompGrid: [1..GridRows, 1..GridCols] locale = …;
```

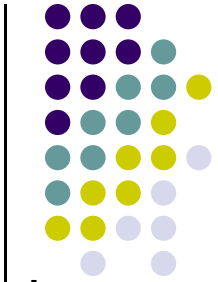


CompGrid

```
var TaskALocs: [1..numTaskALocs] locale = …;
var TaskBLocs: [1..numTaskBLocs] locale = …;
```

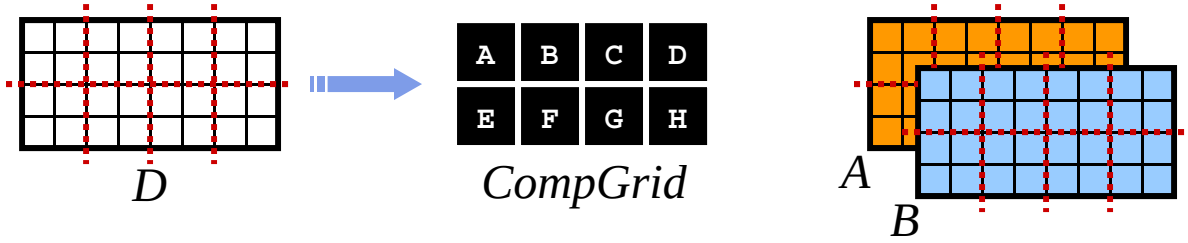
TaskALocs *TaskBLocs*

Data Distribution



- domains may be distributed across locales

```
var D: domain(2) distributed(Block(2) to CompGrid) = …;
```

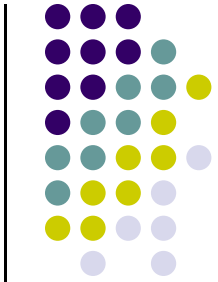


- Distributions specify…
  - …mapping of indices to locales
  - …per-locale storage layout of domain indices and array elements

- Distributions implemented as a class hierarchy
  - Chapel provides a number of standard distributions
  - Users may also write their own

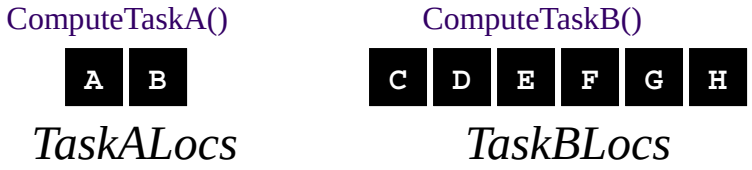
[Hammad]→

Computation Distribution



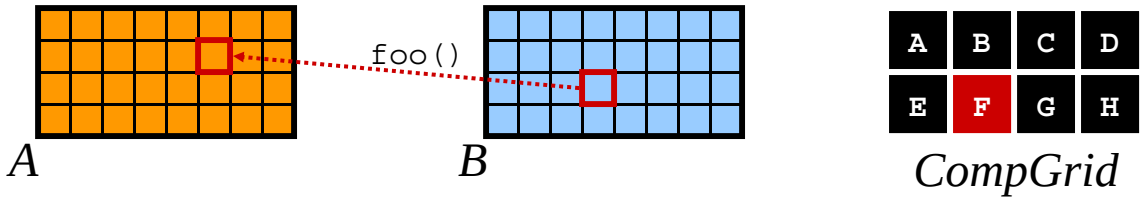
- "on" keyword binds computation to locale(s):

```
cobegin {
  on TaskALocs do ComputeTaskA(…);
  on TaskBLocs do ComputeTaskB(…);
}
```



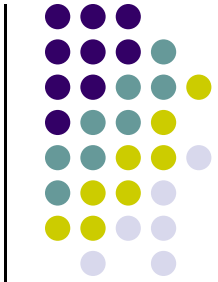
- "on" can also be used in a data-driven manner:

```
forall (i,j) in D {
  on B(j/2,i*2) do A(i,j) = foo(B(j/2,i*2));
}
```



[Hammad]→

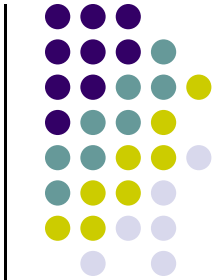
Other Chapel Features



- Tuple types, type unions, and typeselect statements
- Sequences, user-defined iterators
- Support for reductions and scans (parallel prefix)
  - including user-defined operations
- Default arguments, name-based argument passing
- Function and operator overloading
- Modules (for namespace management)
- Interoperability with other languages
- Garbage Collection

[Hammad]→

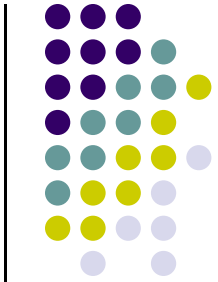
Chapel Challenges



- User Acceptance
  - True of any new language
  - Skeptical audience
- Commodity Architecture Implementation
  - Chapel designed with idealized architecture in mind
  - Clusters are not ideal in many respects
  - Results in implementation and performance challenges
- Cascade Implementation
  - Efficient user-defined domain distributions
  - Type determination w/ OOP w/ overloading w/ …
  - Parallel Garbage Collection
- And many others as well…

[Hammad]→

Summary



- Chapel designed to
  - Enhance programmer productivity
  - Address a wide range of workflows

- Relies on high-level, extensible abstractions for
  - Global-view multithreaded parallel programming
  - Locality-aware programming
  - Object-oriented programming
  - Generic programming and type inference

- Status:
  - *draft* language specification available at: http://chapel.cs.washington.edu
  - Open source implementation proceeding apace

[Hammad]→