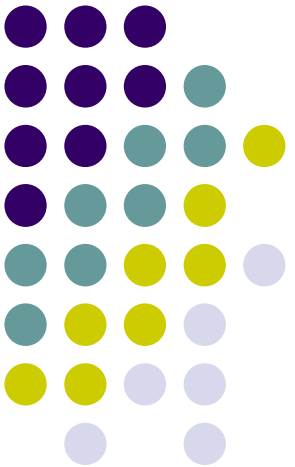
ME964
High Performance Computing for Engineering Applications

Start Parallel Computing with OpenMP

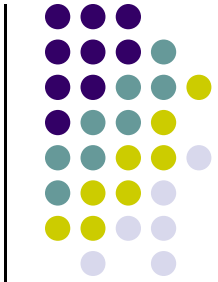
Wrap Up, MPI segment of course

April 24, 2012



"Whenever anyone says, 'theoretically,'
they really mean, 'not really.'"
Dave Parnas

Before We Get Started…

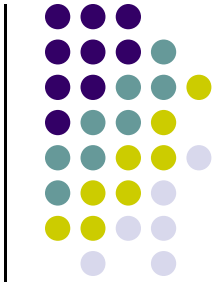


- Last lecture
  - Wrap up, MPI collective communications
  - MPI Derived Types (Handling complex data)

- Today
  - Wrap up, MPI Derived Types (Handling complex data)
  - Start OpenMP

- Other issues
  - Please see me within one week for questions regarding your exam score
  - Assignment 12 due Sunday, April 29 at 11:59 pm
    - One problem, compute integral using OpenMP
  - Lowest score assignment dropped
  - Doodle pool available soon - select time slot for your Final Project presentation

Motivation: MPI_Type_vector



- Assume you have a 2D array of integers, and want send the last column

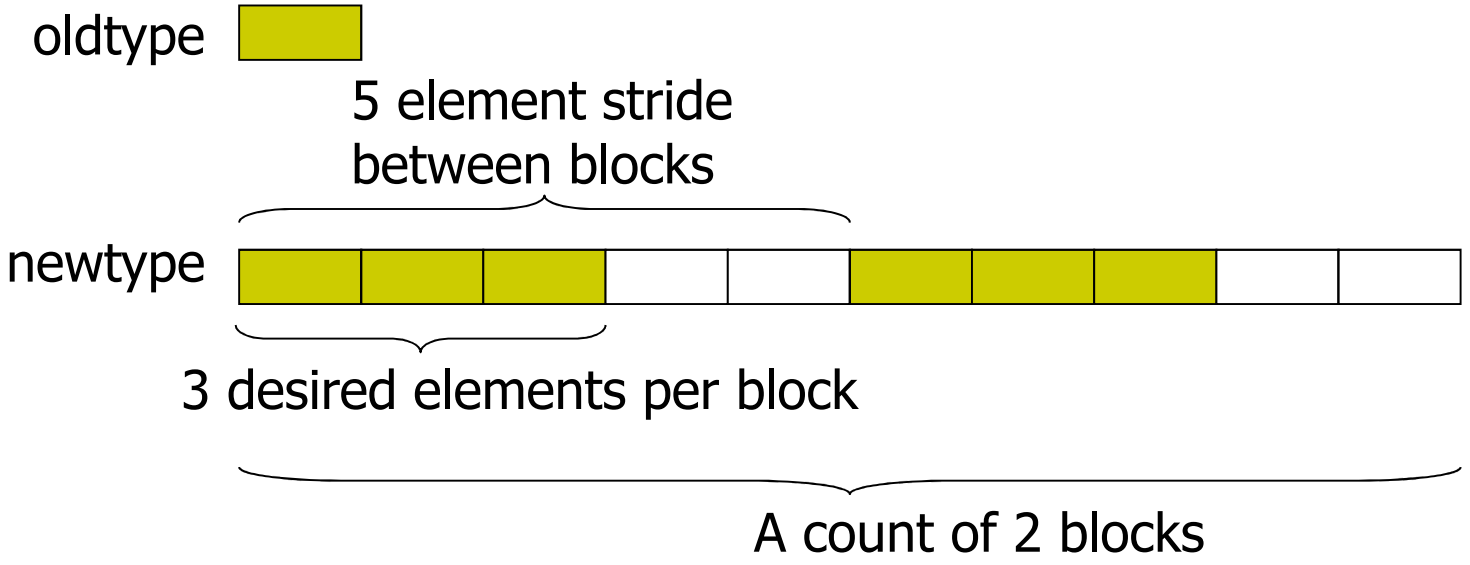
```
int x[4][8];
```

Content of `x[4][8]`:

10	11	12	13	14	15	16	17
100	101	102	103	104	105	106	107
1000	1001	1002	1003	1004	1005	1006	1007
10000	10001	10002	10003	10004	10005	10006	10007

- There should be a way to say that I want to transfer integers, 4 of them, and they are stored in x 8 integers apart (the stride)

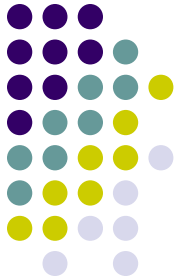
MPI_Type_vector: Example



- count = 2
- blocklength = 3
- stride = 5

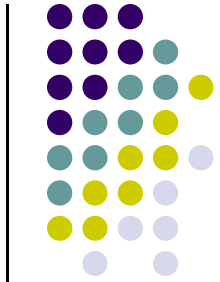
[A. Siegel]→

```c
#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rank,i,j;
    MPI_Status status;
    double x[4][8];
    MPI_Datatype coltype;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
    MPI_Type_commit(&coltype);

    if(rank==3){
        for(i=0;i<4;++i)
            for(j=0;j<8;++j) x[i][j]=pow(10.0,i+1)+j;
        MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
    }
    else if(rank==1) {
        MPI_Recv(&x[0][2],1,coltype,3,52,MPI_COMM_WORLD,&status);
        for(i=0;i<4;++i)printf("P:%d my x[%d][2]=%1f\n",rank,i,x[i][2]);
    }

    MPI_Type_free(&coltype);
    MPI_Finalize();
    return 0;
}
```




```c
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

Example: MPI_Type_vector
[Output]

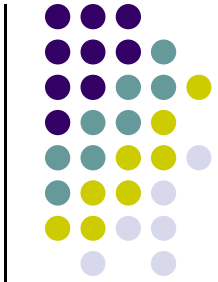


Content of x:

10	11	12	13	14	15	16	17
100	101	102	103	104	105	106	107
1000	1001	1002	1003	1004	1005	1006	1007
10000	10001	10002	10003	10004	10005	10006	10007

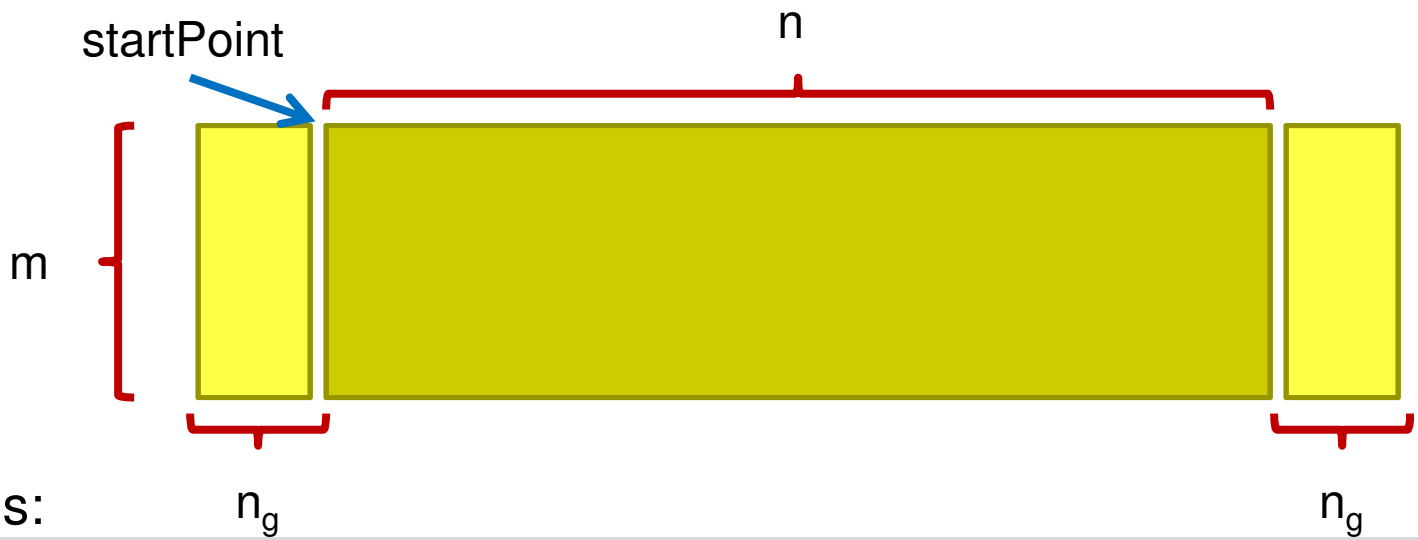
```
[negrut@euler19 CodeBits]$ mpiexec -np 12 me964.exe
P:1 my x[0][2]=17.000000
P:1 my x[1][2]=107.000000
P:1 my x[2][2]=1007.000000
P:1 my x[3][2]=10007.000000
[negrut@euler19 CodeBits]$
```

Example: MPI_Type_vector



- Given: Local 2D array of interior size **mxn** with $\mathbf{n_g}$ ghostcells at each edge
- You wish to send the interior (non ghostcell) portion of the array
- How would you describe the data type to do this in a single MPI call?

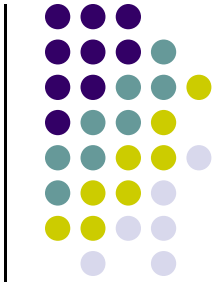
```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```



- Ans:

```
MPI_Type_vector (m, n, n+2*ng, MPI_DOUBLE, &interior);
MPI_Type_commit (&interior);
MPI_Send (startPoint, 1, interior, dest, tag, MPI_COMM_WORLD);
```

Type Map Example



- Start with `oldtype` for which
  Type Map = {(double, 0), (char, 8)}

- What is Type Map of `newtype` if defined as below?
  `MPI_Type_vector`(2,3,4,`oldtype`,&`newtype`)

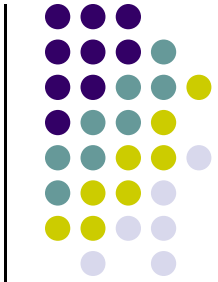
```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

Ans:

{{(double, 0), (char, 8)}, {(double,16),(char,24) }, {(double,32),(char,40) },
{(double,**64**),(char,72), {(double,80),(char,88) }, {(double,96),(char,104)}}

[A. Siegel]→

Exercise: MPI_Type_vector



- Express

```
MPI_Type_contiguous(count, old, &new);
```

…as a call to `MPI_Type_vector`

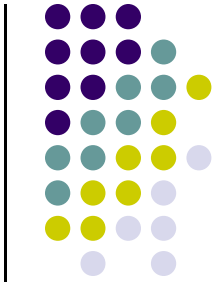
```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- Ans:

```
MPI_Type_vector (count, 1, 1, old, &new);
MPI_Type_vector (1, count, num, old, &new);
```

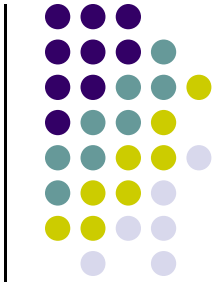

[A. Siegel]→

Outline, Coverage of MPI



- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI derived Datatypes
- MPI Closing Remarks

MPI – We're Scratching the Surface

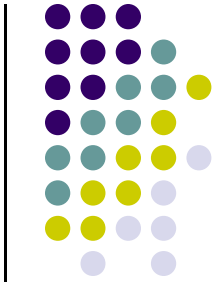


- In some MPI implementations there are more than 300 MPI functions
  - Not all of them part of the MPI standard though, some vendor specific

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]

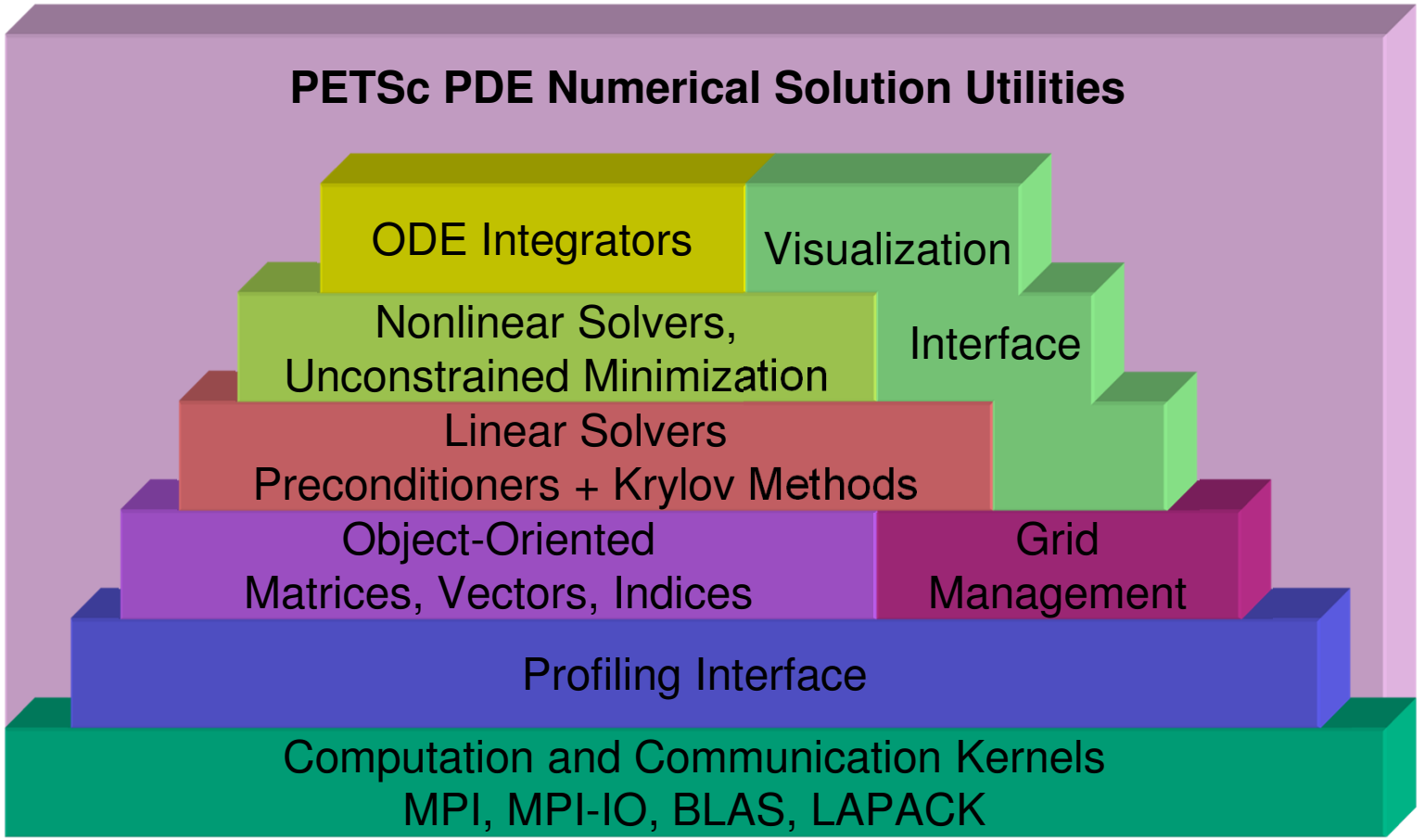


- PETSc: Portable, Extensible Toolkit for Scientific Computation
  - One of the most successful libraries built on top of MPI
  - Intended for use in large-scale application projects,
  - Developed at Argonne National Lab (Barry Smith)
  - Open source, available for download at http://www.mcs.anl.gov/petsc/petsc-as/

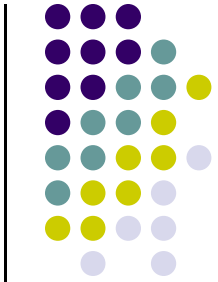
- PETSc provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution

- PETSc also provides routines for
  - 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

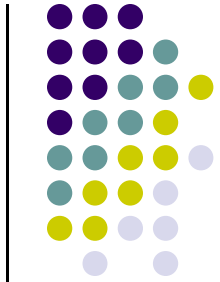
Preconditioners						
Additive Schwartz	Block Jacobi	Jacobi	ILU	ICC	LU (Sequential only)	Others

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

Distributed Arrays

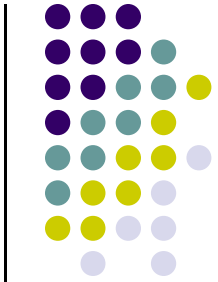
Index Sets			
Indices	Block Indices	Stride	Other

Vectors



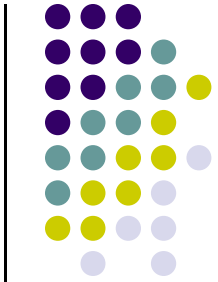
Parallel Programming Using OpenMP

Acknowledgements



- Majority of slides used for discussing OpenMP issues are from Intel's library of presentations for promoting OpenMP
  - Slides used herein with permission
- Credit given where due: IOMPP
  - IOMPP stands for "Intel OpenMP Presentation"

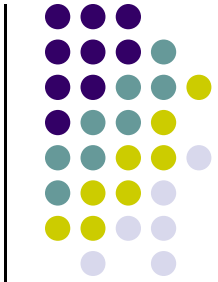
Data vs. Task Parallelism



- Data parallelism
  - You have a large amount of data elements and each data element (or possibly a subset of elements) needs to be processed to produce a result
  - When this processing can be done in parallel, we have data parallelism
  - Example:
    - Adding two long arrays of doubles to produce yet another array of doubles

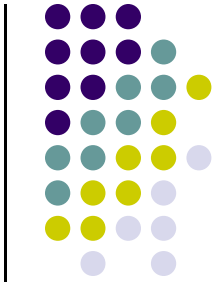
- Task parallelism
  - You have a collection of tasks that need to be completed
  - If these tasks can be performed in parallel you are faced with a task parallel job
  - Examples:
    - Reading the newspaper, drinking coffee, and scratching your back
    - The breathing your lungs, beating of your heart, liver function, controlling the swallowing, etc.

Objectives



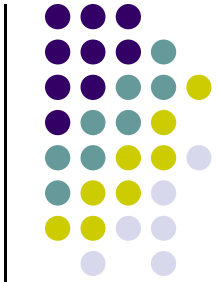
- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism
- Provide an overview of OpenMP in three lectures

Work Plan



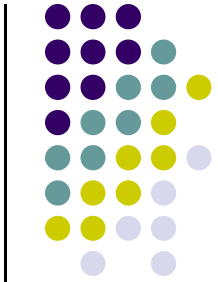
- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization
- Advanced topics

OpenMP: Target Hardware



- CUDA: targeted parallelism on the GPU
- MPI: targeted parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently a SMP architecture such as a workstation with two hexcore CPUs that draw on a good amount of shared memory
- OpenMP: targets parallelism on SMP architectures
  - Handy when
    - You have a machine that has 64 cores
    - You have a large amount of shared memory, say 128GB

OpenMP: What's Reasonable to Expect

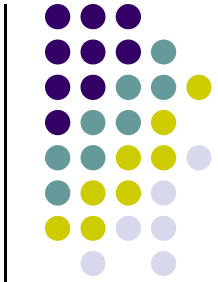


- If you have 64 cores available to you, it is *highly* unlikely to get a speedup of more than 64 (superlinear)

- Recall the trick that helped the GPU hide latency
  - Overcommitting the SPs and hiding memory access latency with warp execution

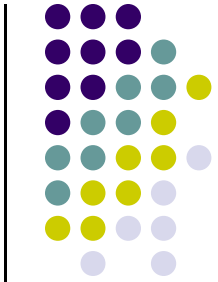
- This mechanism of hiding latency by overcommitment does not *explicitly* exist for parallel computing under OpenMP beyond what's offered by HTT

OpenMP: What Is It?



- Portable, shared-memory threading API
  - Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows
- Standardizes task & loop-level parallelism
- Supports coarse-grained parallelism
- Combines serial and parallel code in single source
- Standardizes ~ 20 years of compiler-directed threading experience
- Current spec is OpenMP 3.0
  - http://www.openmp.org
  - 318 Pages

`pthreads`: An OpenMP Precursor

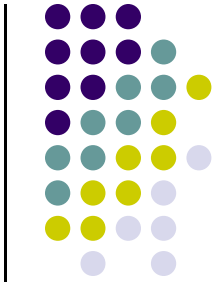


- Before there was **`OpenMP`**, a common approach to support parallel programming was by use of **`pthreads`**
  - "**`pthread`**": POSIX thread
  - POSIX: Portable Operating System Interface [for Unix]

- **`pthreads`**
  - Available originally under Unix and Linux
  - Windows ports are also available some as open source projects

- Parallel programming with **`pthreads`**: relatively cumbersome, prone to mistakes, hard to maintain/scale/expand
  - Not envisioned as a mechanism for writing scientific computing software

pthreads: Example

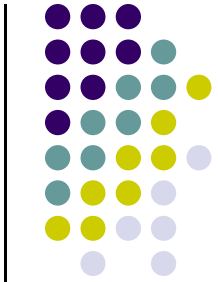


```
int main(int argc, char *argv[]) {
          parm           *arg;
          pthread_t      *threads;
          pthread_attr_t  pthread_custom_attr;

          int n = atoi(argv[1]);

          threads = (pthread_t *) malloc(n * sizeof(*threads));
          pthread_attr_init(&pthread_custom_attr);

          barrier_init(&barrier1); /* setup barrier */
          finals = (double *) malloc(n * sizeof(double)); /* allocate space for final result */

          arg=(parm *)malloc(sizeof(parm)*n);
          for( int i = 0; i < n; i++)      { /* Spawn thread */
                    arg[i].id = i;
                    arg[i].noproc = n;
                    pthread_create(&threads[i], &pthread_custom_attr, cpi, (void *)(arg+i));
          }

          for( int i = 0; i < n; i++) /* Synchronize the completion of each thread. */
                    pthread_join(threads[i], NULL);

          free(arg);
          return 0;
}
```

```c
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <sys/types.h>
#include <pthread.h>
#include <sys/time.h>

#define SOLARIS 1
#define ORIGIN  2
#define OS      SOLARIS

typedef struct {
   int          id;
   int          noproc;
   int          dim;
} parm;

typedef struct {
   int             cur_count;
   pthread_mutex_t barrier_mutex;
   pthread_cond_t  barrier_cond;
} barrier_t;


void barrier_init(barrier_t * mybarrier) { /* barrier */
   /* must run before spawning the thread */
   pthread_mutexattr_t attr;

# if (OS==ORIGIN)
   pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
   pthread_mutexattr_setprioceiling(&attr, 0);
   pthread_mutex_init(&(mybarrier->barrier_mutex), &attr);
# elif (OS==SOLARIS)
   pthread_mutex_init(&(mybarrier->barrier_mutex), NULL);
# else
# error "undefined OS"
# endif
   pthread_cond_init(&(mybarrier->barrier_cond), NULL);
   mybarrier->cur_count = 0;
}

void barrier(int numproc, barrier_t * mybarrier) {
   pthread_mutex_lock(&(mybarrier->barrier_mutex));
   mybarrier->cur_count++;
   if (mybarrier->cur_count!=numproc) {
      pthread_cond_wait(&(mybarrier->barrier_cond), &(mybarrier->barrier_mutex));
   }
   else {
      mybarrier->cur_count=0;
      pthread_cond_broadcast(&(mybarrier->barrier_cond));
   }
   pthread_mutex_unlock(&(mybarrier->barrier_mutex));
}
```

```c
void* cpi(void *arg) {
   parm           *p = (parm *) arg;
   int             myid = p->id;
   int             numprocs = p->noproc;
   double          PI25DT = 3.14159265358979323846264338327950288419716939937510;
   double          mypi, pi, h, sum, x, a;
   double          startwtime, endwtime;

   if (myid == 0)   {
      startwtime = clock();
   }
   barrier(numprocs, &barrier1);
   if (rootn==0)
      finals[myid]=0;
   else {
      h = 1.0 / (double) rootn;
      sum = 0.0;
      for(int i = myid + 1; i <=rootn; i += numprocs) {
         x = h * ((double) i - 0.5);
         sum += f(x);
      }
      mypi = h * sum;
   }
   finals[myid] = mypi;

   barrier(numprocs, &barrier1);

   if (myid == 0){
      pi = 0.0;
      for(int i=0; i < numprocs; i++) pi += finals[i];
      endwtime = clock();
      printf("pi is approx %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
      printf("wall clock time = %f\n",
          (endwtime - startwtime) / CLOCKS_PER_SEC);
   }
   return NULL;
}
```

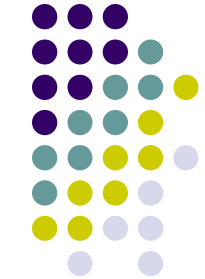
`pthreads`: leaving them behind…



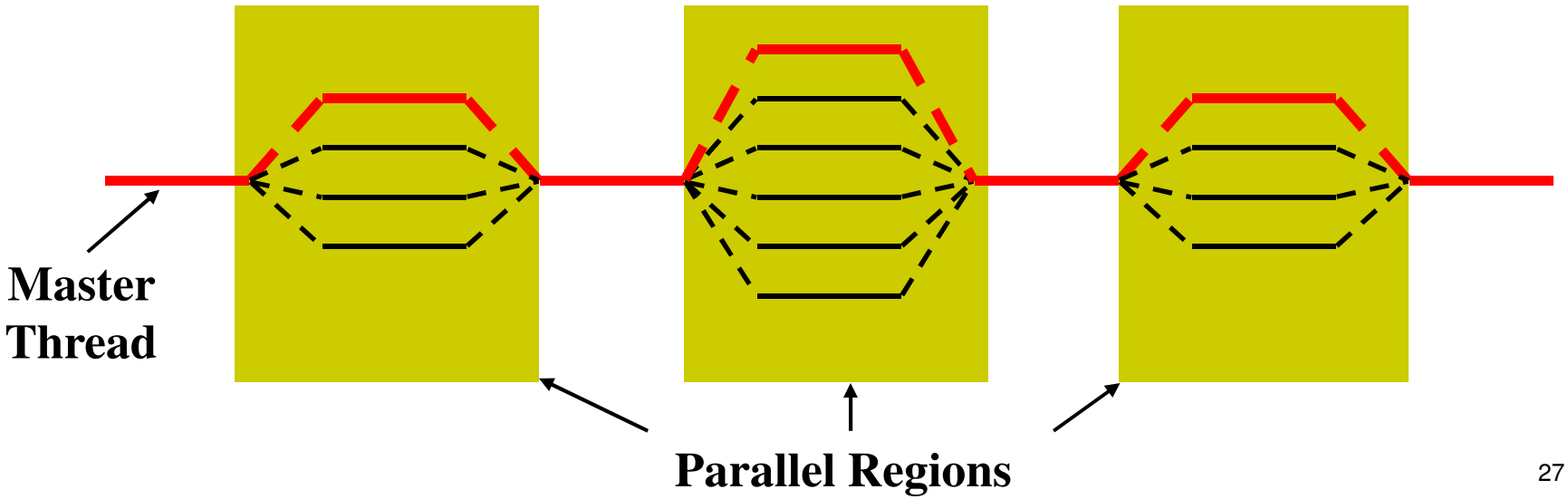
- Looking at the previous example (which is not the best written piece of code, lifted from the web…)
  - Code displays platform dependency (not portable)
  - Code is cryptic, low level, hard to read (not simple)
  - Requires busy work: fork and joining threads, etc.
    - Burdens the developer
    - Probably in the way of the compiler as well: rather low chances that the compiler will be able to optimize the implementation

- Higher level approach to SMP parallel computing for *scientific applications* was in order

OpenMP Programming Model

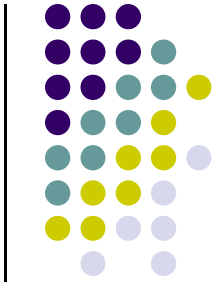


- Master thread spawns a team of threads as needed
  - Managed transparently on your behalf
  - It still relies on thread fork/join methodology to implement parallelism
    - The developer is spared the details
- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program



[IOMPP]→

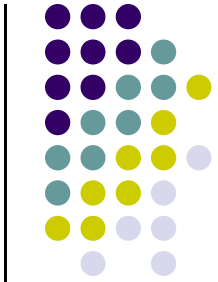
OpenMP: Library Support



- Runtime environment routines:
  - Modify/check the number of threads
    `omp_[set|get]_num_threads()`
    `omp_get_thread_num()`
    `omp_get_max_threads()`
  - Are we in a parallel region?
    `omp_in_parallel()`
  - How many processors in the system?
    `omp_get_num_procs()`
  - Explicit locks
    `omp_[set|unset]_lock()`
  - And several more...

https://computing.llnl.gov/tutorials/openMP/

A Few Syntax Details to Get Started



- Most of the constructs in OpenMP are compiler directives or pragmas
  - For C and C++, the pragmas take the form:

```
#pragma omp construct [clause [clause]…]
```

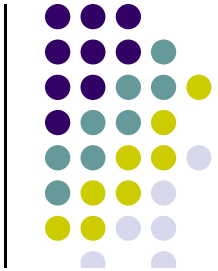
  - For Fortran, the directives take one of the forms:

```
C$OMP construct [clause [clause]…]
!$OMP construct [clause [clause]…]
*$OMP construct [clause [clause]…]
```

- Header file or Fortran 90 module

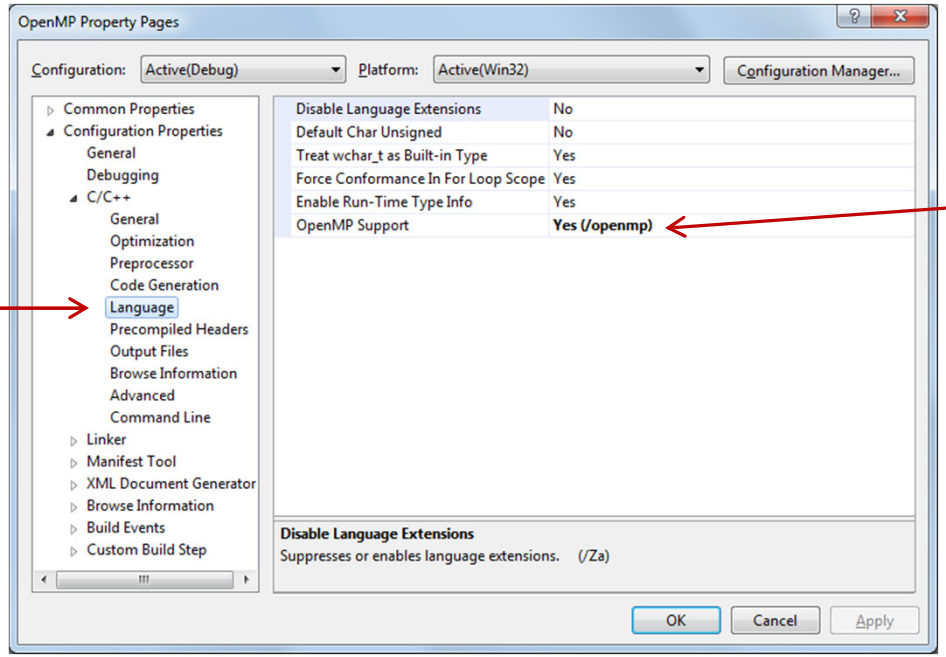
```
#include “omp.h”
use omp_lib
```

Why Compiler Directive and/or Pragmas?



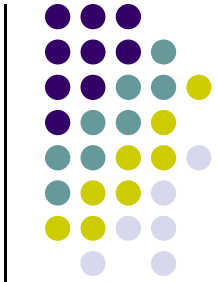
- One of OpenMP's design principles was to have the same code, with no modifications and have it run either on an one core machine, or a multiple core machine
- Therefore, you have to "hide" all the compiler directives behind Comments and/or Pragmas
- These hidden directives would be picked up by the compiler only if you instruct it to compile in OpenMP mode
  - Example: Visual Studio – you have to have the /openmp flag on in order to compile OpenMP code
  - Also need to indicate that you want to use the OpenMP API by having the right header included: #include <omp.h>

Step 1:
Go here



Step 2:
Select /openmp

OpenMP, Compiling Using the Command Line



- Method depends on compiler

- GCC:

```
$ gcc -o integrate_omp integrate_omp.c –fopenmp
```

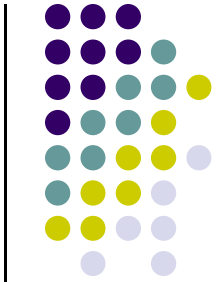
- ICC:

```
$ icc -o integrate_omp integrate_omp.c –openmp
```

- MSVC (not in the express edition):

```
$ cl /openmp integrate_omp.c
```

Enabling OpenMP with CMake



```
# Minimum version of CMake required.
cmake_minimum_required(VERSION 2.8)

# Set the name of your project
project(ME964-omp)

# Include macros from the SBEL utils library
include(SBELUtils.cmake)

# Example OpenMP program
enable_openmp_support()
add_executable(integrate_omp integrate_omp.cpp)
```

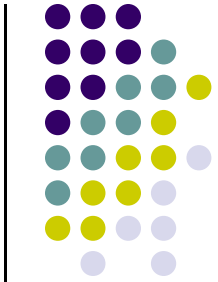
With the template

```
find_package("OpenMP" REQUIRED)

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
```

Without the template

Replaces include(SBELUtils.cmake) and enable_openmp_support() above

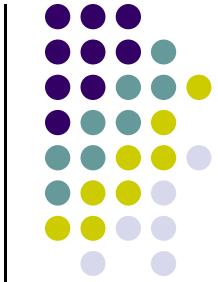
OpenMP Odds and Ends…



- Controlling the number of threads at runtime
  - The default number of threads that a program uses when it runs is the number of online processors on the machine
  - For the C Shell: **`setenv OMP_NUM_THREADS number`**
  - For the Bash Shell: **`export OMP_NUM_THREADS=number`**
- Timing:

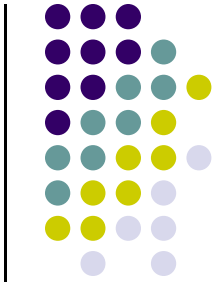
```
#include <omp.h>
stime = omp_get_wtime();
longfunction();
etime = omp_get_wtime();
total=etime-stime;
```

Work Plan



- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization
- Advanced topics

Parallel Region & Structured Blocks (C/C++)



- Most OpenMP constructs apply to structured blocks
  - Structured block: a block with one point of entry at the top and one point of exit at the bottom
  - The only "branches" allowed are `exit()` function calls in C/C++

A structured block

```
#pragma omp parallel
{
  int id = omp_get_thread_num();

more: res[id] = do_big_job (id);

  if (not_conv (res[id]) goto more;
}
printf ("All done\n");
```

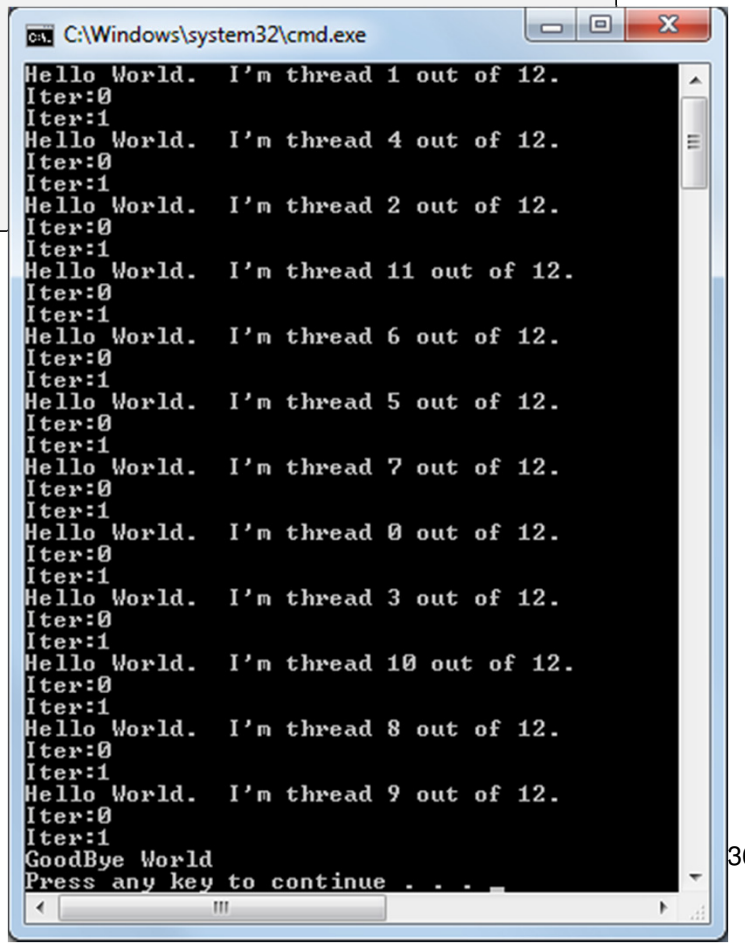
Not a structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if (conv (res[id]) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

There is an implicit barrier at the right "}" curly brace and that's the point at which the other worker threads complete execution and either go to sleep or spin or otherwise idle.

Example: Hello World on my Machine

```
#include <stdio.h>
#include <omp.h>

int main() {
#pragma omp parallel
    {
        int myId = omp_get_thread_num();
        int nThreads = omp_get_num_threads();

        printf("Hello World.  I'm thread %d out of %d.\n", myId, nThreads);
        for( int i=0; i<2 ;i++ )
            printf("Iter:%d\n",i);
    }
    printf("GoodBye World\n");
}
```

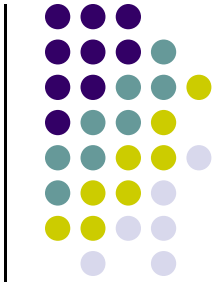


- Here's my machine (12 core machine)

Two Intel Xeon X5650 Westmere 2.66GHz 12MB L3 Cache LGA 1366 95Watts Six-Core Processors

[IOMPP]→

OpenMP: Important Remark

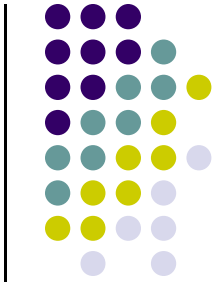


- One of the key tenets of OpenMP is that of data independence across parallel jobs

- Specifically, when distributing work among parallel threads it is assumed that there is no data dependency

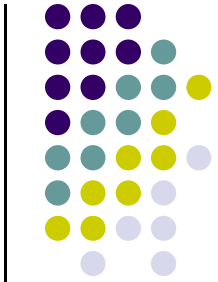
- Since you place the `omp parallel` directive around some code, it is <u>your responsibility</u> to make sure that data dependency is ruled out
  - Compilers are not smart enough and sometimes it is outright impossible to rule out data dependency between what might look as independent parallel jobs

Work Plan



- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization
- Advanced topics

Work Sharing



- **Work sharing** is the general term used in OpenMP to describe distribution of work across threads

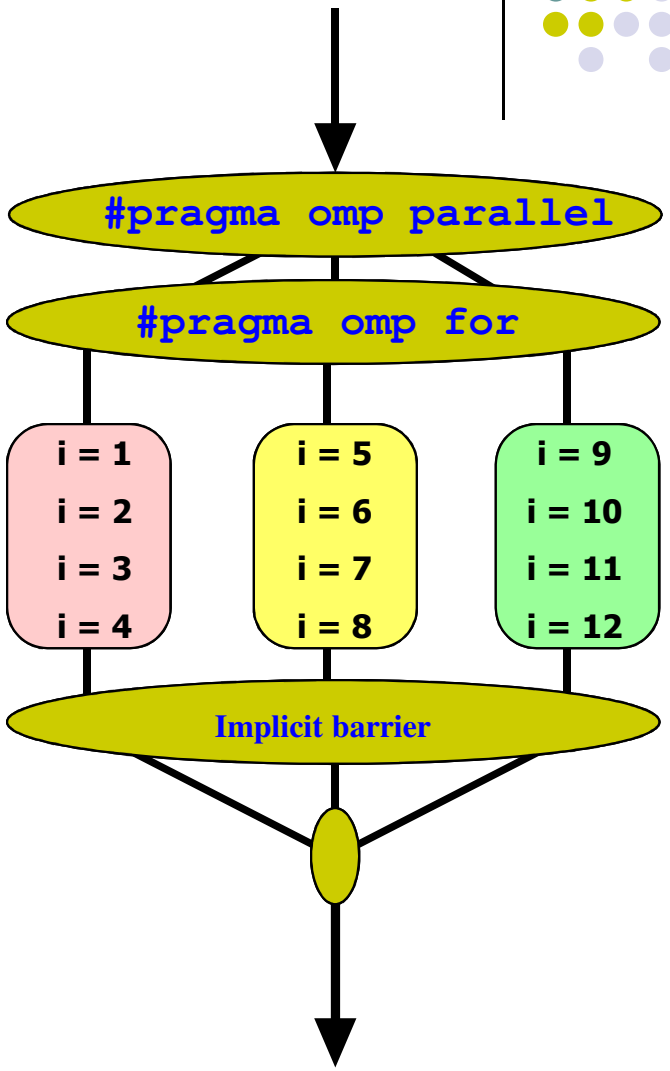
- Three categories of work sharing in OpenMP:
  - "omp for" construct
  - "omp sections" construct
  - "omp task" construct

Each of them automatically divides work among threads

[IOMPP]→

"omp for" construct

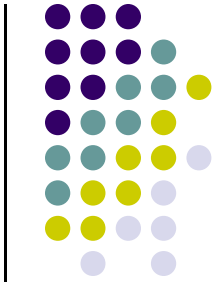
```
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1, i < N+1, i++)
        c[i] = a[i] + b[i];
```



- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

[IOMPP]→

Combining Constructs



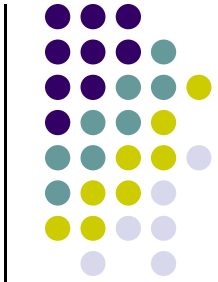
- These two code segments are equivalent

```
#pragma omp parallel
{
    #pragma omp for
    for ( int i=0;i< MAX; i++) {

        res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (int i=0;i< MAX; i++) {
         res[i] = huge();
    }
```


[IOMPP]→

The Private Clause

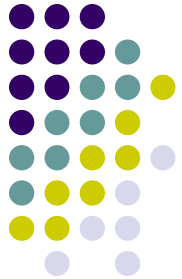


- Reproduces the variable for each task
  - Variables are un-initialized; C++ object is default constructed
  - Any variable external to the parallel region is undefined
  - By declaring a variable as being private it means that each thread will have a private copy of that variable
    - The value that thread 1 stores in x is different than the value that thread 2 stores in the variable x

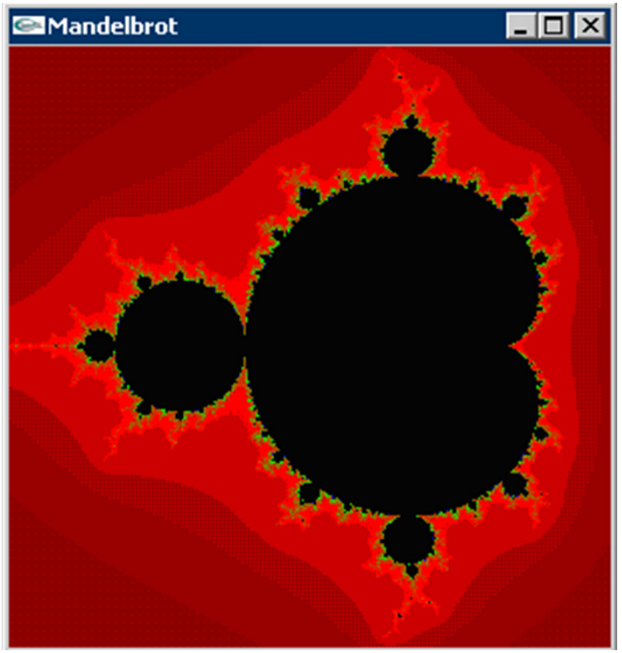
```
void* work(float* c, int N) {
    float x, y; int i;
#pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
}
```

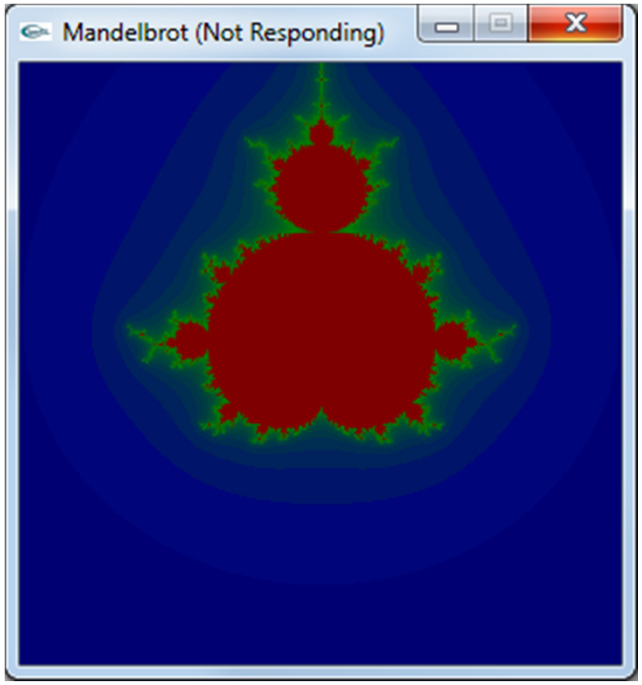

[IOMPP]→

Example: Parallel Mandelbrot



- Objective: create a parallel version of Mandelbrot using OpenMP work sharing clauses to parallelize the computation of Mandelbrot.

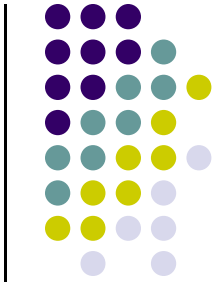




[IOMPP]→

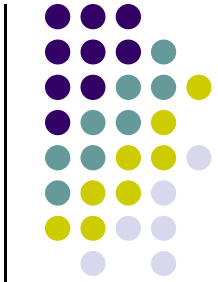
Example: Parallel Mandelbrot

[The Important Function; Includes material from IOMPP]



```
int Mandelbrot (float z_r[][JMAX],float z_i[][JMAX],float z_color[][JMAX], char gAxis ){
    float xinc = (float)XDELTA/(IMAX-1);
    float yinc = (float)YDELTA/(JMAX-1);

#pragma omp parallel for private(i,j) schedule(static,8)
    for (int i=0; i<IMAX; i++) {
        for (int j=0; j<JMAX; j++) {
            z_r[i][j] = (float) -1.0*XDELTA/2.0 + xinc * i;
            z_i[i][j] = (float) 1.0*YDELTA/2.0 - yinc * j;
            switch (gAxis) {
                case 'V':
                    z_color[i][j] = CalcMandelbrot(z_r[i][j], z_i[i][j] ) /1.0001;
                    break;
                case 'H':
                    z_color[i][j] = CalcMandelbrot(z_i[i][j], z_r[i][j] ) /1.0001;
                default:
                    break;
            }
        }
    }
    return 1;
}
```

The `schedule` Clause



- The `schedule` clause affects how loop iterations are mapped onto threads

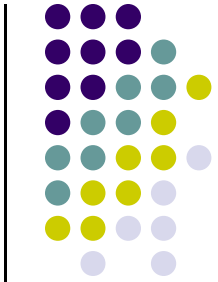
`schedule(static [,chunk])`
  - Blocks of iterations of size “chunk” assigned to each thread
  - Round robin distribution
  - Low overhead, may cause load imbalance

`schedule(dynamic[,chunk])`
  - Threads grab “chunk” iterations
  - When done with iterations, thread requests next set
  - Higher threading overhead, can reduce load imbalance

`schedule(guided[,chunk])`
  - Dynamic schedule starting with large block
  - Size of the blocks shrink; no smaller than “chunk”

Credit: IOMPP

schedule Clause Example



```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

- Iterations are divided into chunks of 8
- If start = 3, then first chunk is

$$\mathbf{i}=\{3,5,7,9,11,13,15,17\}$$

Credit: IOMPP