Hybrid Programming with OpenMP and MPI

Karl W. Schulz

Texas Advanced Computing Center
The University of Texas at Austin

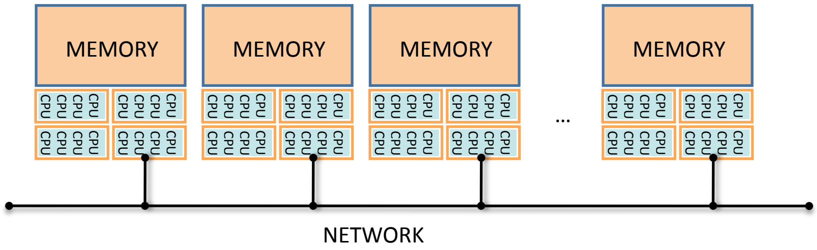
UT/Portugal Summer Institute Training
Coimbra, Portugal
July 17, 2008

Hybrid Outline

- Distributed and Shared Memory Systems
- Why Hybrid Computing
- Numa Controls (batch scripts)
- Motivation for Hybrid Computing
- Modes of Hybrid Computing
  - MPI initialization
  - Funneled, Serialized and Multi-Threaded

Distributed & Shared Memory

- Combines distributed memory parallelization with on-node shared memory parallelization
- Largest systems now employ both architectures



Ranger System

- Shared Memory component is a “cache coherent” SMP blade. Non uniform memory access (NUMA) and state (cache coherence) are the hallmarks of a global memory (within a hierarchy).
- Distributed memory component is a network of SMP blades. State(fulness) is maintained by the program.

Why Hybrid

- Eliminates domain decomposition at node (this can be a big deal, eg. factor of 16 for Ranger)
- Automatic coherency at node
- Lower memory latency and data movement within node
- Can synchronize on memory instead of barrier

Why Hybrid (cont 1)

- Only profitable if on-node aggregation of MPI parallel components is faster as a single SMP algorithm (or a single SMP algorithm on each socket).

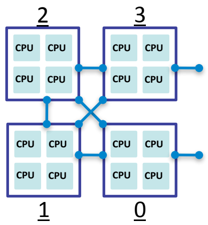
NUMA Operations

- Where do threads/processes and memory allocations go?
- Default: Decided by policy when process exec'd or thread forked, and when memory allocated. Processes and threads can be rescheduled to different sockets and cores.
- Scheduling Affinity and Memory Policy can be changed within code with (sched_get/setaffinity, get/set_memory_policy)

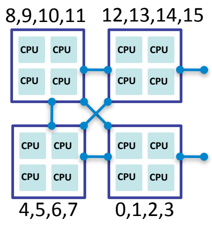
NUMA Operations (cont. 1)

- Affinity and Policy can be changed externally through numactl at the socket and core level.

```
Command:      numactl   <options>   ./a.out
```

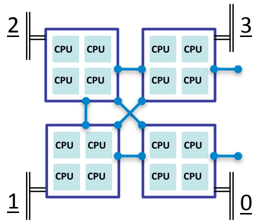


Socket References



Core References

NUMA Operations (cont. 2)

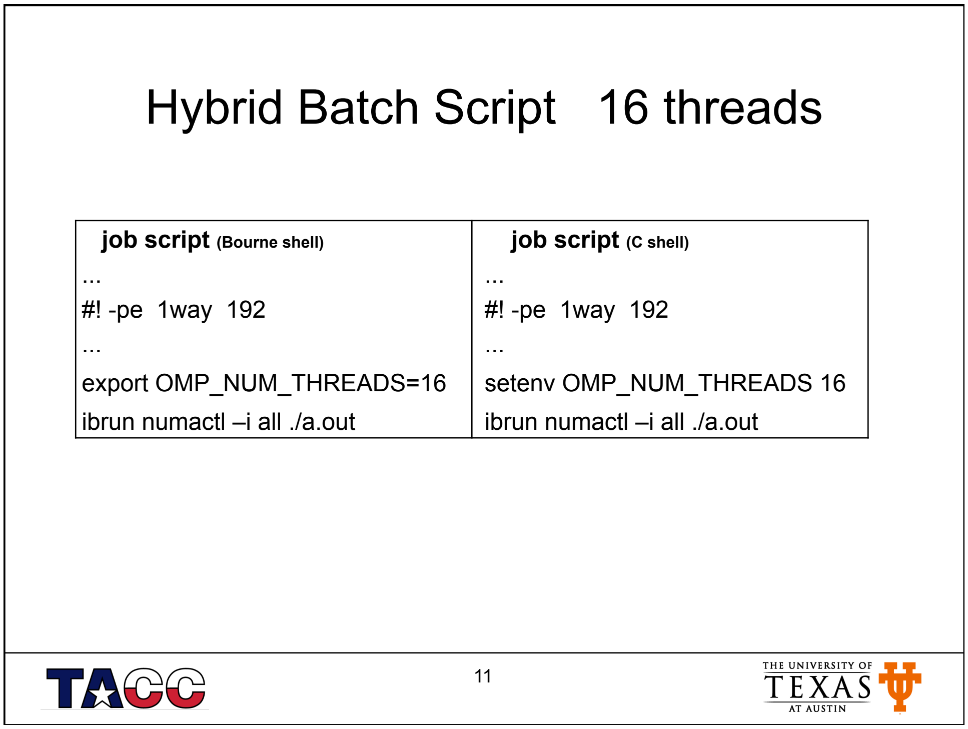


Memory: Socket References

- MPI – local is best
- SMP – Interleave best for large, completely shared arrays
- SMP – local best for private arrays
- Once allocated, a memory structure’s is fixed

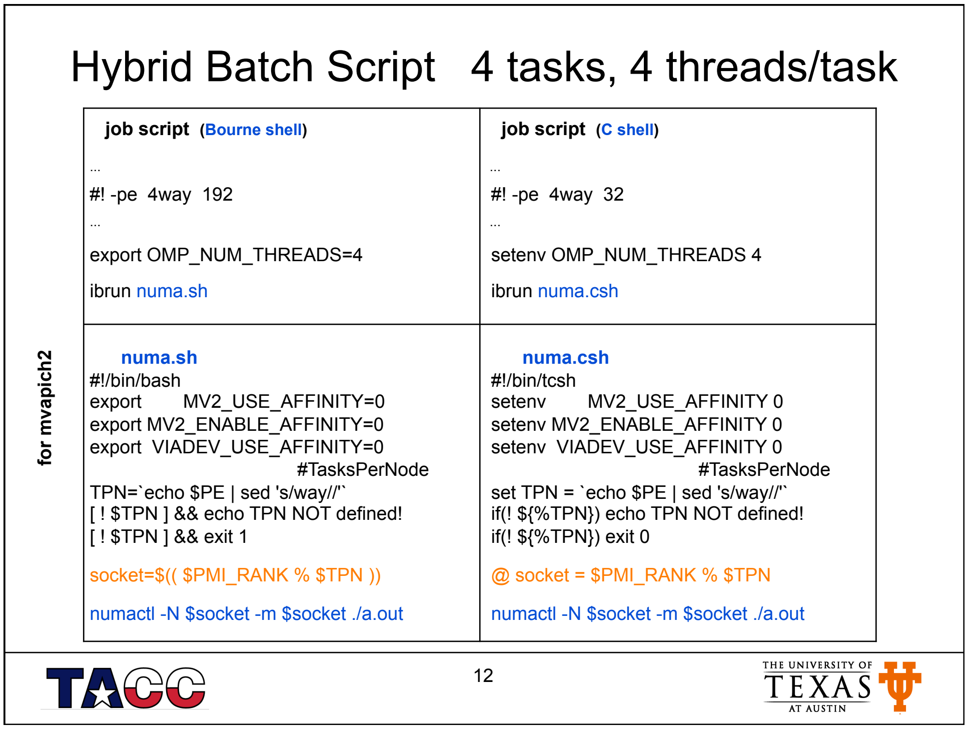
NUMA Operations (cont. 3)

	cmd	option	arguments	description
Socket Affinity	numactl	-N	{0,1,2,3}	Only execute process on cores of this (these) socket(s).
Memory Policy	numactl	-l	{no argument}	Allocate on current socket.
Memory Policy	numactl	-i	{0,1,2,3}	Allocate round robin (interleave) on these sockets.
Memory Policy	numactl	--preferred=	{0,1,2,3} select only one	Allocate on this socket; fallback to any other if full .
Memory Policy	numactl	-m	{0,1,2,3}	Only allocate on this (these) socket(s).
Core Affinity	numactl	-C	{0,1,2,3, 4,5,6,7, 8,9,10,11, 12,13,14,15}	Only execute process on this (these) Core(s).



Hybrid Batch Script   16 threads

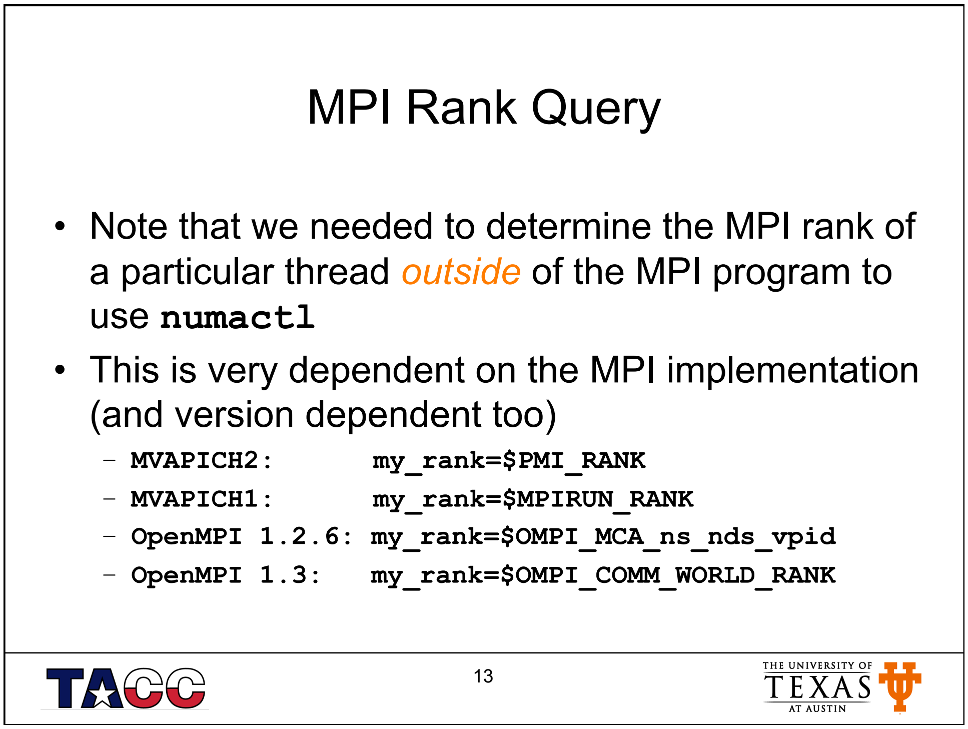
job script **(Bourne shell)**	**job script** **(C shell)**
… #! -pe 1way 192 … export OMP_NUM_THREADS=16 ibrun numactl –i all ./a.out	… #! -pe 1way 192 … setenv OMP_NUM_THREADS 16 ibrun numactl –i all ./a.out



Hybrid Batch Script   4 tasks, 4 threads/task

for mvapich2

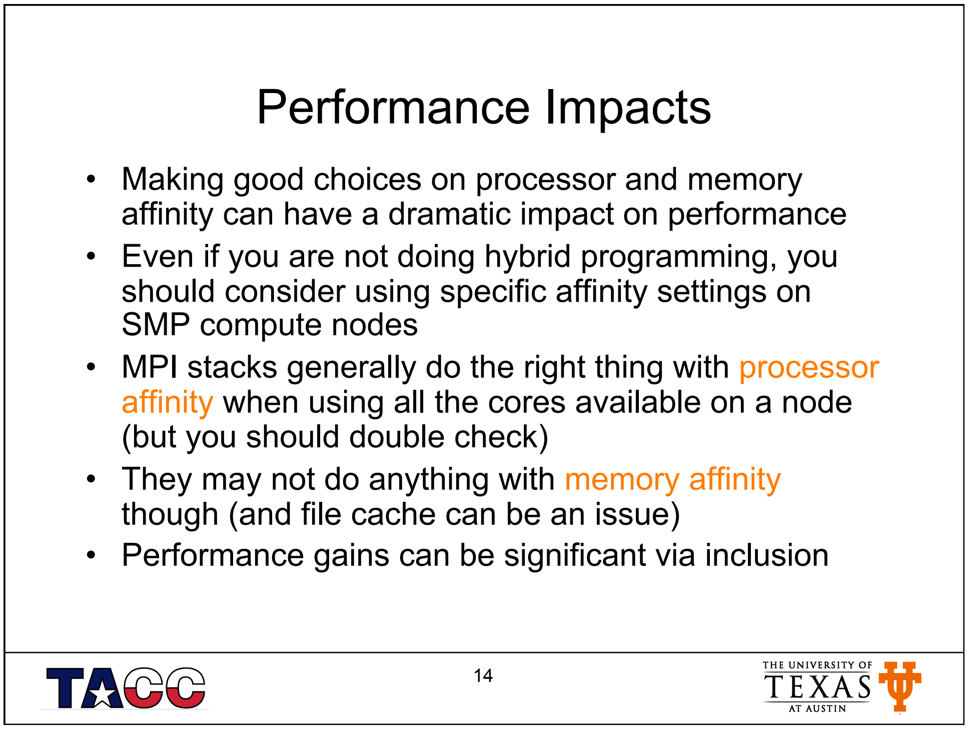
<table>
<tr><td><b>job script</b> (<b>Bourne shell</b>)
…
#! -pe 4way 192
…
export OMP_NUM_THREADS=4
ibrun numa.sh</td><td><b>job script</b> (<b>C shell</b>)
…
#! -pe 4way 32
…
setenv OMP_NUM_THREADS 4
ibrun numa.csh</td></tr>
<tr><td><b>numa.sh</b>
#!/bin/bash
export   MV2_USE_AFFINITY=0
export MV2_ENABLE_AFFINITY=0
export VIADEV_USE_AFFINITY=0
#TasksPerNode
TPN=`echo $PE | sed 's/way//'`
[! $TPN] && echo TPN NOT defined!
[! $TPN] && exit 1
socket=$(($PMI_RANK % $TPN))
numactl -N $socket -m $socket ./a.out</td><td><b>numa.csh</b>
#!/bin/tcsh
setenv   MV2_USE_AFFINITY 0
setenv MV2_ENABLE_AFFINITY 0
setenv VIADEV_USE_AFFINITY 0
#TasksPerNode
set TPN = `echo $PE | sed 's/way//'`
if(! ${%TPN}) echo TPN NOT defined!
if(! ${%TPN}) exit 0
@ socket = $PMI_RANK % $TPN
numactl -N $socket -m $socket ./a.out</td></tr>
</table>



MPI Rank Query

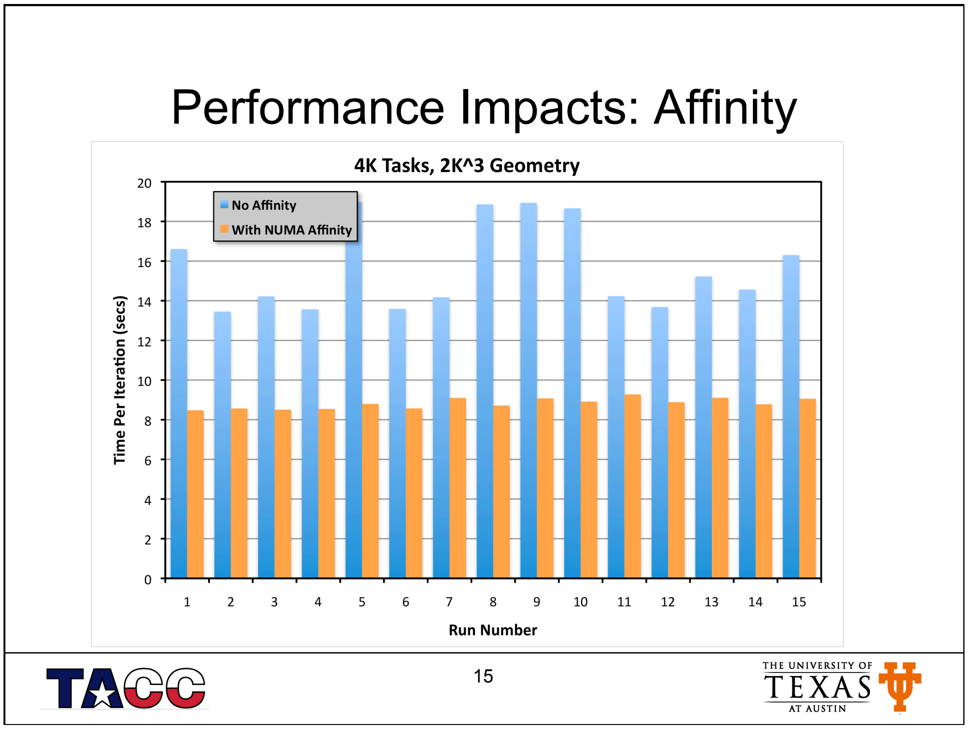
- Note that we needed to determine the MPI rank of a particular thread *outside* of the MPI program to use **numactl**
- This is very dependent on the MPI implementation (and version dependent too)

```
– MVAPICH2:       my_rank=$PMI_RANK
– MVAPICH1:       my_rank=$MPIRUN_RANK
– OpenMPI 1.2.6: my_rank=$OMPI_MCA_ns_nds_vpid
– OpenMPI 1.3:   my_rank=$OMPI_COMM_WORLD_RANK
```

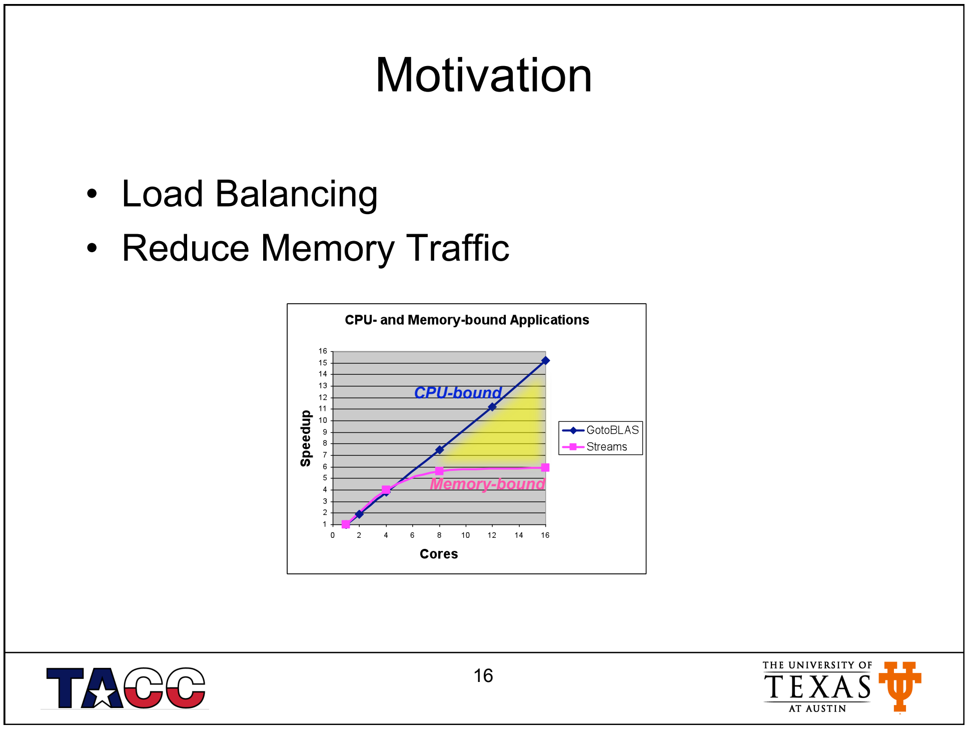


Performance Impacts

- Making good choices on processor and memory affinity can have a dramatic impact on performance
- Even if you are not doing hybrid programming, you should consider using specific affinity settings on SMP compute nodes
- MPI stacks generally do the right thing with processor affinity when using all the cores available on a node (but you should double check)
- They may not do anything with memory affinity though (and file cache can be an issue)
- Performance gains can be significant via inclusion

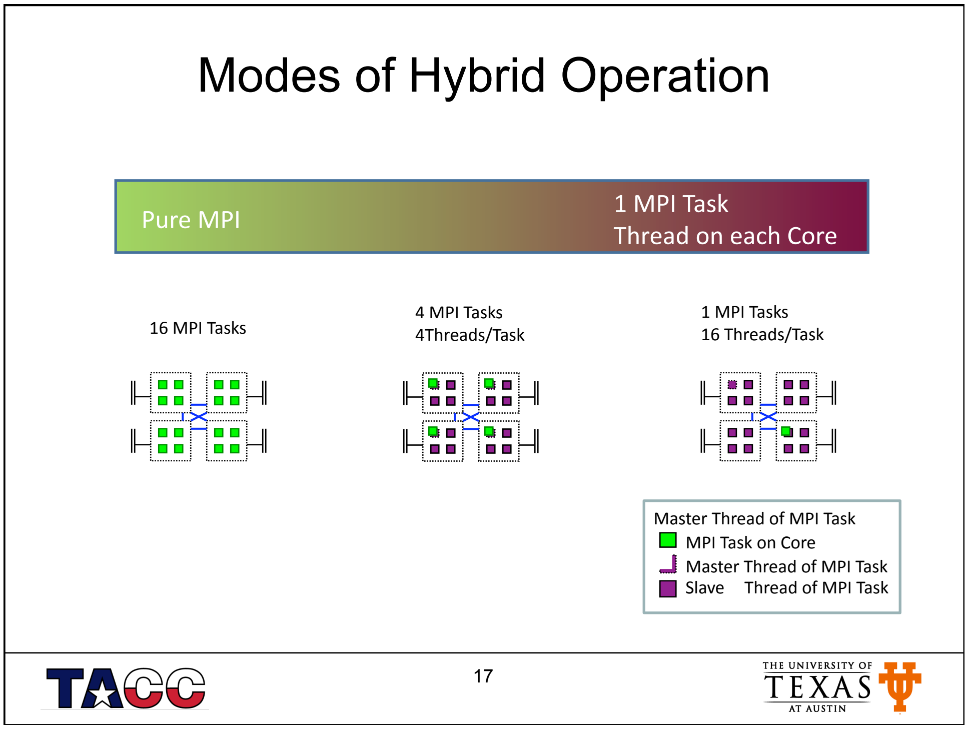


Performance Impacts: Affinity



Motivation

- Load Balancing
- Reduce Memory Traffic



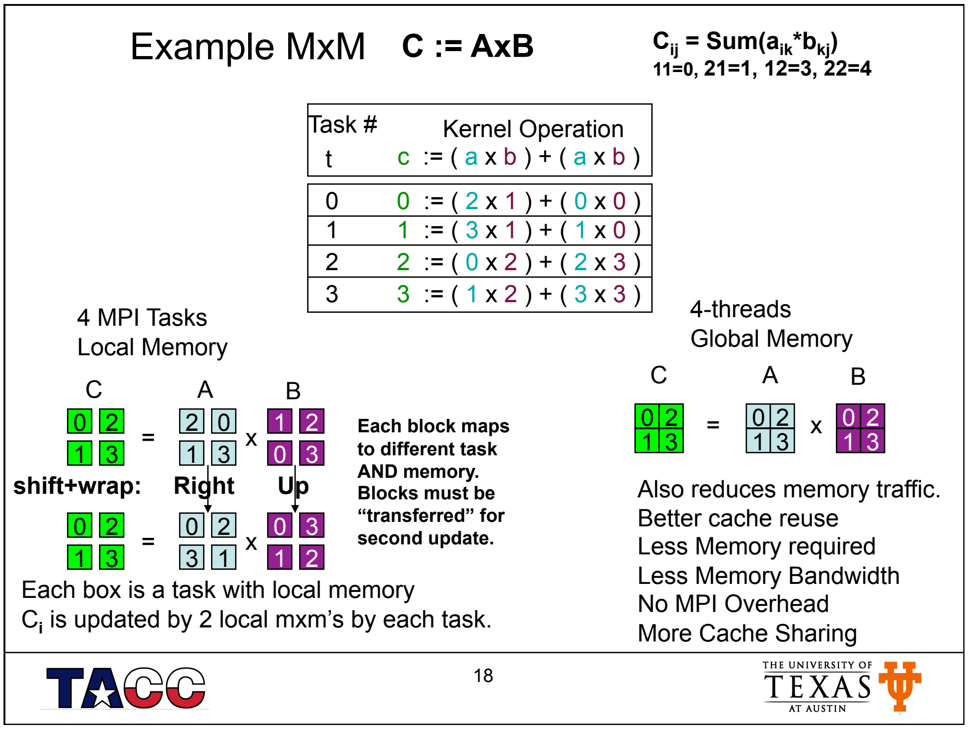
Modes of Hybrid Operation

Master Thread of MPI Task

🟩 MPI Task on Core

Master Thread of MPI Task

🟪 Slave Thread of MPI Task



Example MxM C := AxB

C_{ij} = Sum(a_{ik}*b_{kj})
11=0, 21=1, 12=3, 22=4

Task # t	Kernel Operation c := (a x b) + (a x b)
0	0 := (2 x 1) + (0 x 0)
1	1 := (3 x 1) + (1 x 0)
2	2 := (0 x 2) + (2 x 3)
3	3 := (1 x 2) + (3 x 3)

4 MPI Tasks
Local Memory

$$\begin{array}{ccccc} C & & A & & B \\ \begin{array}{|c|c|}\hline 0&2\\\hline 1&3\\\hline\end{array} & = & \begin{array}{|c|c|}\hline 2&0\\\hline 1&3\\\hline\end{array} & \times & \begin{array}{|c|c|}\hline 1&2\\\hline 0&3\\\hline\end{array}\end{array}$$

shift+wrap: Right Up

$$\begin{array}{|c|c|}\hline 0&2\\\hline 1&3\\\hline\end{array} = \begin{array}{|c|c|}\hline 0&2\\\hline 3&1\\\hline\end{array} \times \begin{array}{|c|c|}\hline 0&3\\\hline 1&2\\\hline\end{array}$$

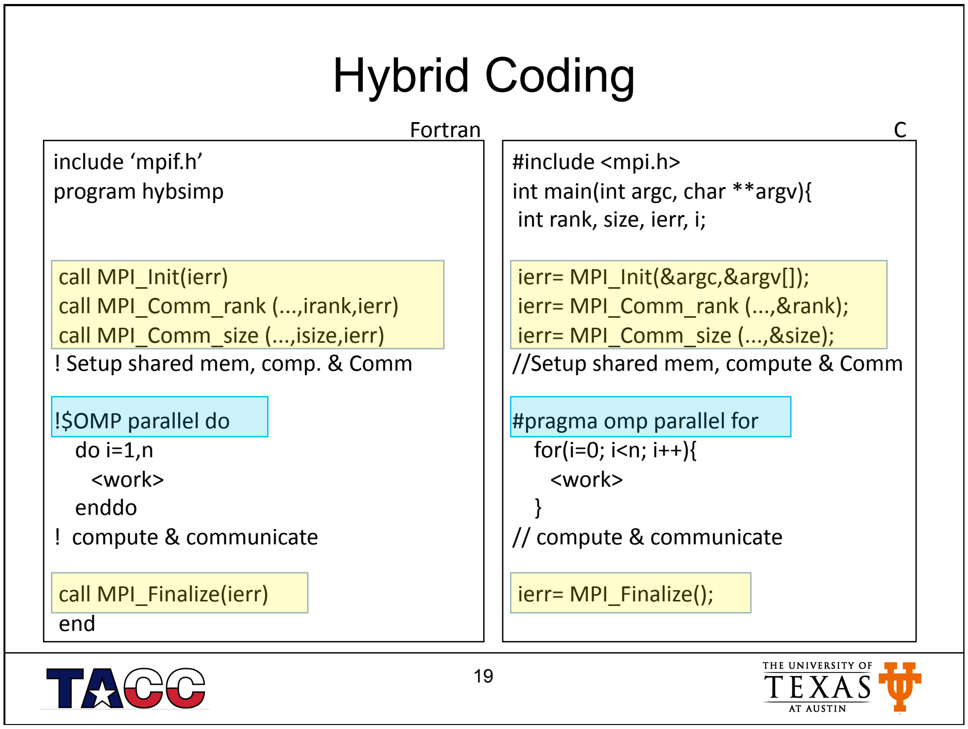
Each block maps to different task AND memory. Blocks must be "transferred" for second update.

Each box is a task with local memory
C_i is updated by 2 local mxm's by each task.

4-threads
Global Memory

$$\begin{array}{ccccc} C & & A & & B \\ \begin{array}{|c|c|}\hline 0&2\\\hline 1&3\\\hline\end{array} & = & \begin{array}{|c|c|}\hline 0&2\\\hline 1&3\\\hline\end{array} & \times & \begin{array}{|c|c|}\hline 0&2\\\hline 1&3\\\hline\end{array}\end{array}$$

Also reduces memory traffic.
Better cache reuse
Less Memory required
Less Memory Bandwidth
No MPI Overhead
More Cache Sharing



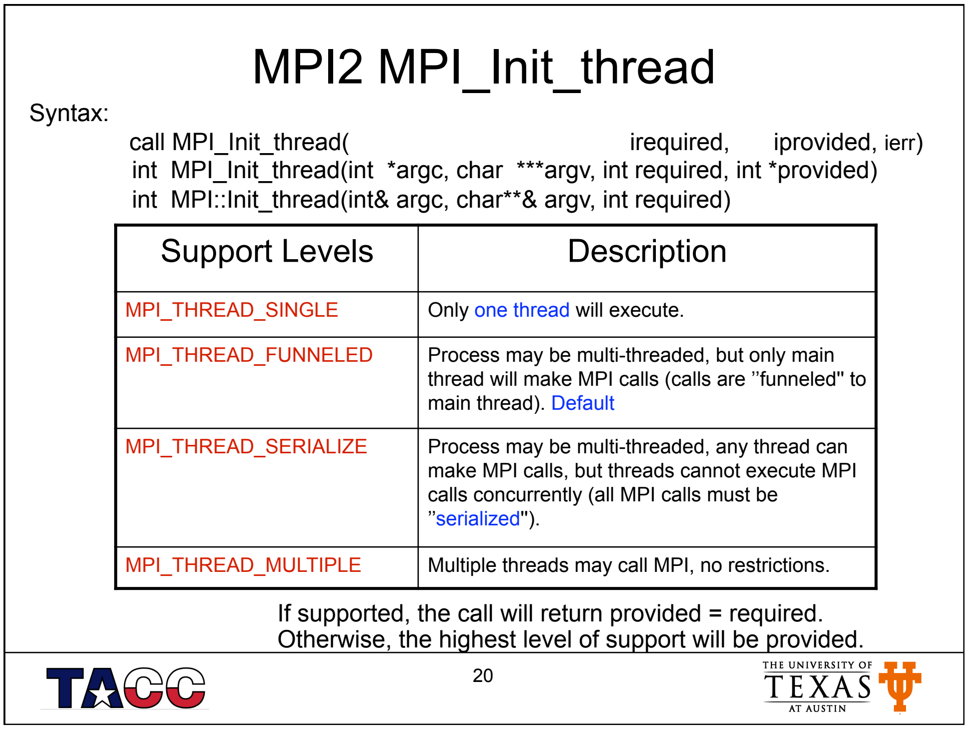
Hybrid Coding

Fortran

```
include 'mpif.h'
program hybsimp


call MPI_Init(ierr)
call MPI_Comm_rank (...,irank,ierr)
call MPI_Comm_size (...,isize,ierr)
! Setup shared mem, comp. & Comm

!$OMP parallel do
   do i=1,n
      <work>
   enddo
!  compute & communicate

call MPI_Finalize(ierr)
end
```

C

```
#include <mpi.h>
int main(int argc, char **argv){
 int rank, size, ierr, i;

 ierr= MPI_Init(&argc,&argv[]);
 ierr= MPI_Comm_rank (...,&rank);
 ierr= MPI_Comm_size (...,&size);
//Setup shared mem, compute & Comm

#pragma omp parallel for
   for(i=0; i<n; i++){
      <work>
   }
// compute & communicate

 ierr= MPI_Finalize();
```



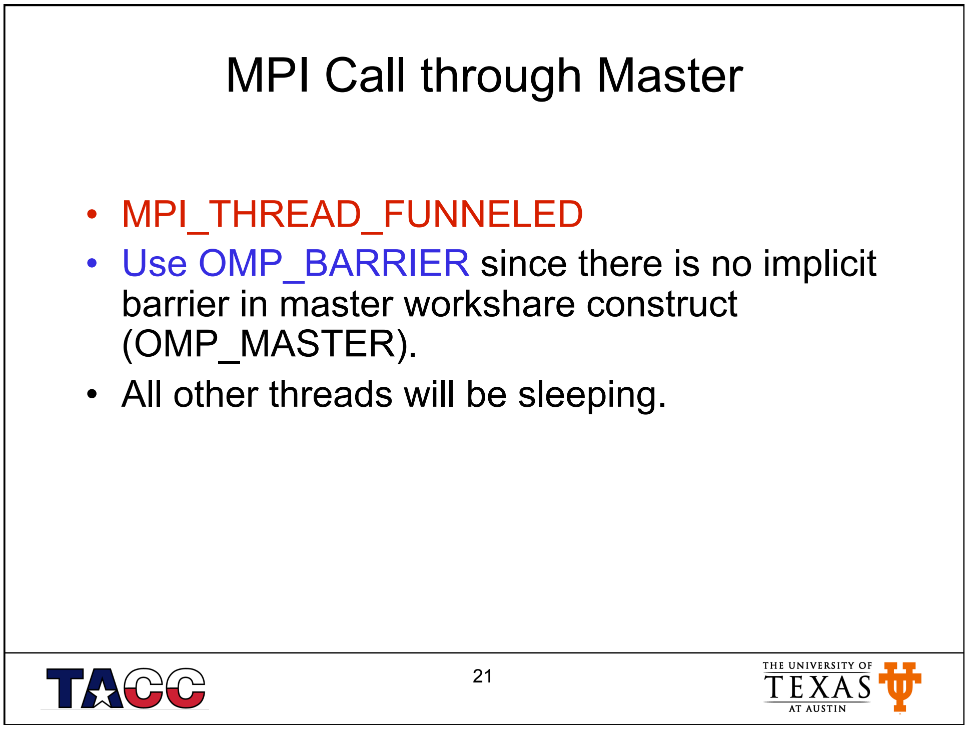
MPI2 MPI_Init_thread

Syntax:

```
call MPI_Init_thread(                          irequired,      iprovided, ierr)
 int  MPI_Init_thread(int  *argc, char  ***argv, int required, int *provided)
 int  MPI::Init_thread(int& argc, char**& argv, int required)
```

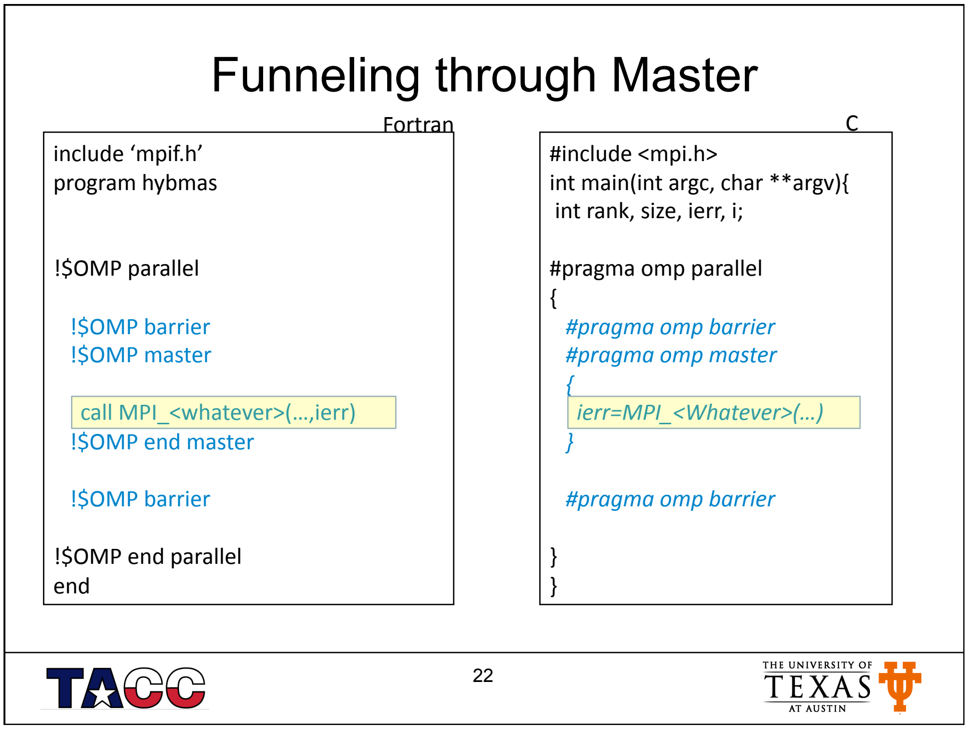
Support Levels	Description
MPI_THREAD_SINGLE	Only one thread will execute.
MPI_THREAD_FUNNELED	Process may be multi-threaded, but only main thread will make MPI calls (calls are "funneled" to main thread). Default
MPI_THREAD_SERIALIZE	Process may be multi-threaded, any thread can make MPI calls, but threads cannot execute MPI calls concurrently (all MPI calls must be "serialized").
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, no restrictions.

If supported, the call will return provided = required.
Otherwise, the highest level of support will be provided.



MPI Call through Master

- MPI_THREAD_FUNNELED
- Use OMP_BARRIER since there is no implicit barrier in master workshare construct (OMP_MASTER).
- All other threads will be sleeping.



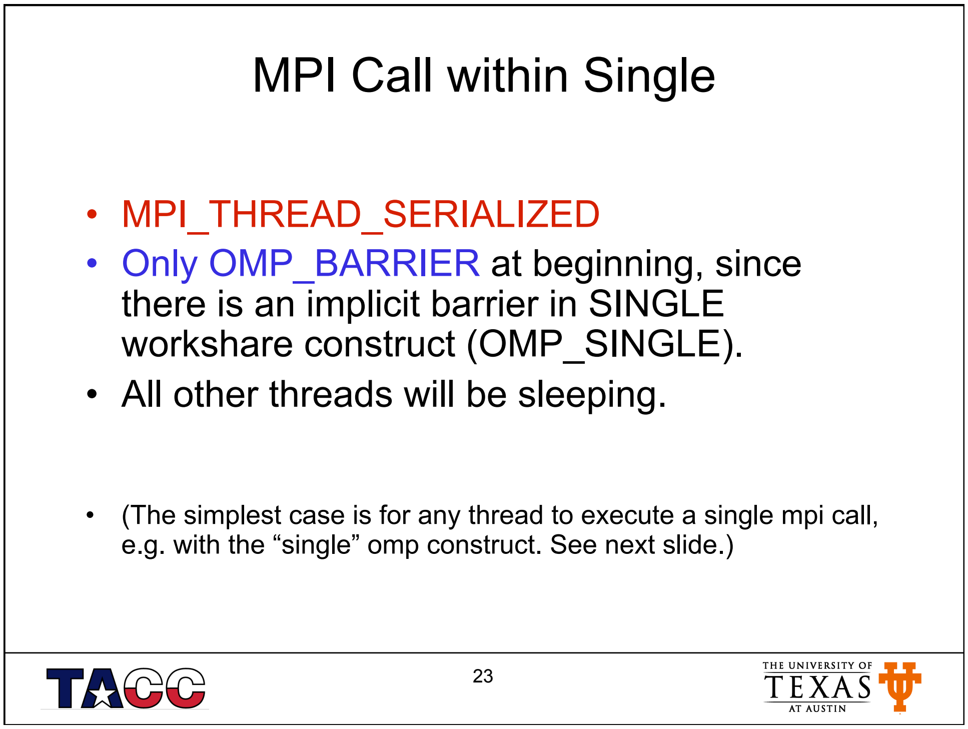
Funneling through Master

Fortran

```
include ‘mpif.h’
program hybmas


!$OMP parallel

  !$OMP barrier
  !$OMP master

  call MPI_<whatever>(…,ierr)
  !$OMP end master

  !$OMP barrier

!$OMP end parallel
end
```

C

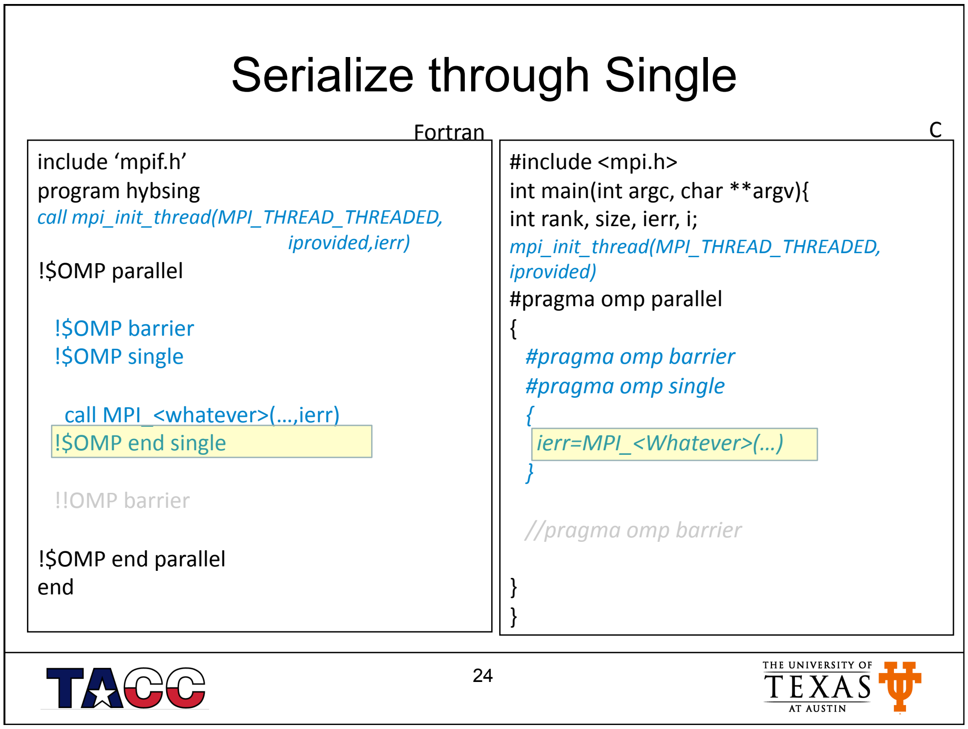
```
#include <mpi.h>
int main(int argc, char **argv){
 int rank, size, ierr, i;

#pragma omp parallel
{
  #pragma omp barrier
  #pragma omp master
  {
   ierr=MPI_<Whatever>(…)
  }

  #pragma omp barrier

}
}
```



MPI Call within Single

- MPI_THREAD_SERIALIZED
- Only OMP_BARRIER at beginning, since there is an implicit barrier in SINGLE workshare construct (OMP_SINGLE).
- All other threads will be sleeping.

- (The simplest case is for any thread to execute a single mpi call, e.g. with the "single" omp construct. See next slide.)



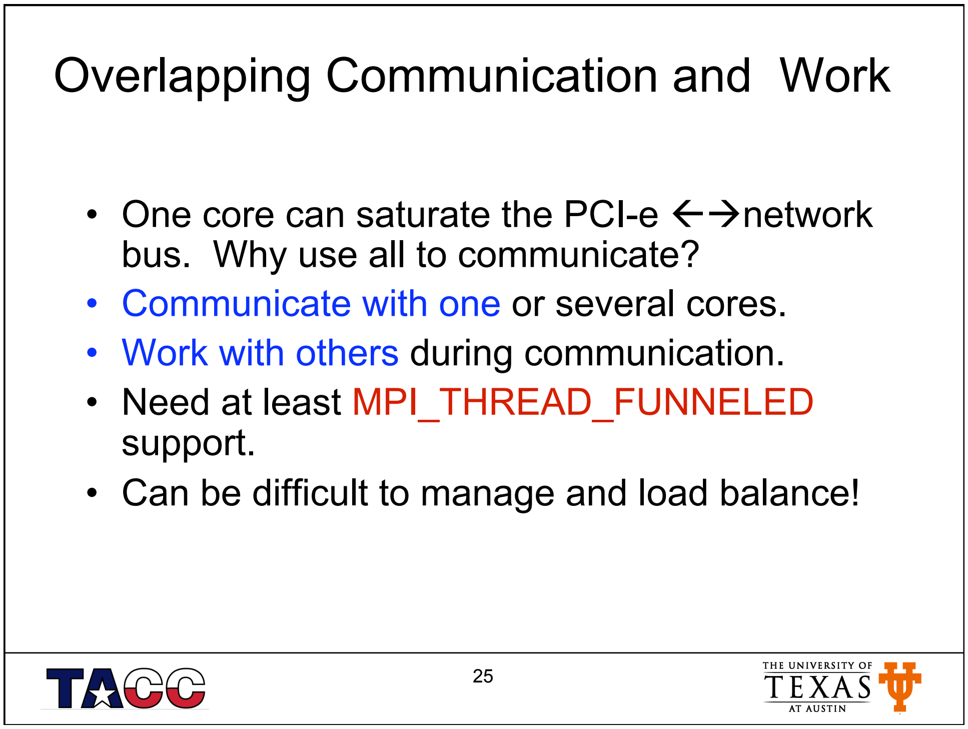
Serialize through Single

Fortran

```
include 'mpif.h'
program hybsing
call mpi_init_thread(MPI_THREAD_THREADED,
                          iprovided,ierr)
!$OMP parallel

  !$OMP barrier
  !$OMP single

   call MPI_<whatever>(…,ierr)
  !$OMP end single

  !!OMP barrier

!$OMP end parallel
end
```

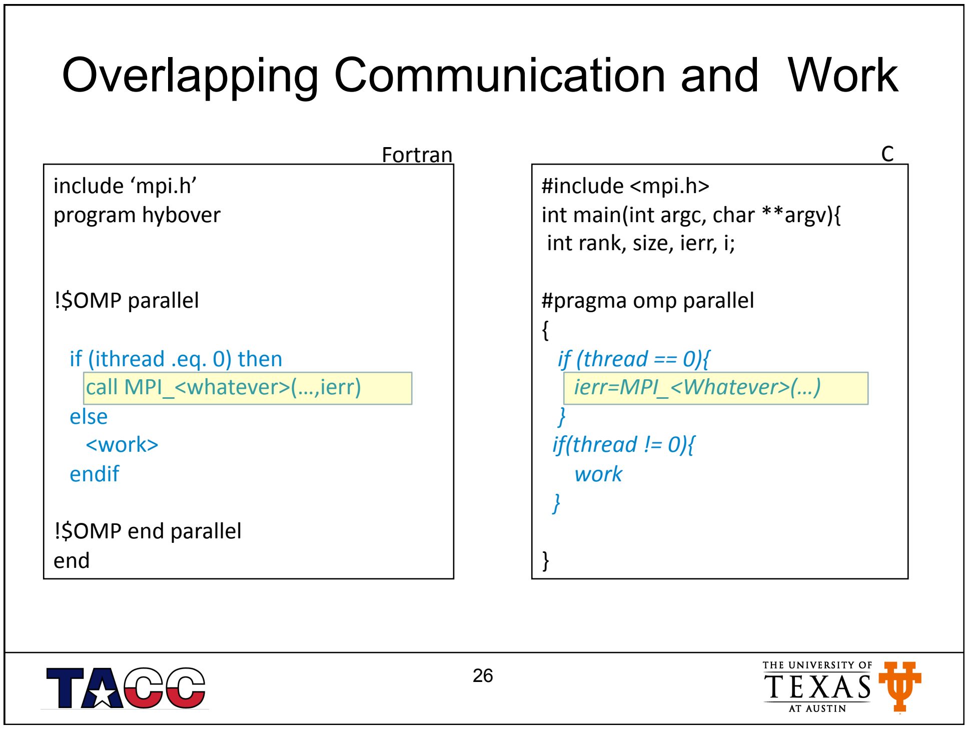
C

```
#include <mpi.h>
int main(int argc, char **argv){
int rank, size, ierr, i;
mpi_init_thread(MPI_THREAD_THREADED,
iprovided)
#pragma omp parallel
{
  #pragma omp barrier
  #pragma omp single
  {
   ierr=MPI_<Whatever>(…)
  }

  //pragma omp barrier

}
}
```



Overlapping Communication and Work

- One core can saturate the PCI-e ←→network bus. Why use all to communicate?
- Communicate with one or several cores.
- Work with others during communication.
- Need at least MPI_THREAD_FUNNELED support.
- Can be difficult to manage and load balance!



Overlapping Communication and Work

Fortran

```
include ‘mpi.h’
program hybover


!$OMP parallel

  if (ithread .eq. 0) then
    call MPI_<whatever>(…,ierr)
  else
    <work>
  endif

!$OMP end parallel
end
```

C

```
#include <mpi.h>
int main(int argc, char **argv){
 int rank, size, ierr, i;

#pragma omp parallel
{
  if (thread == 0){
    ierr=MPI_<Whatever>(…)
  }
  if(thread != 0){
     work
  }

}
```

Thread-rank Communication

- Can use thread id and rank in communication
- Next example illustrates technique in multi-thread “ping” (send/receive) example.

Thread-rank Communication

```
:
call mpi_init_thread( MPI_THREAD_MULTIPLE, iprovided,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,  irank, ierr)
call mpi_comm_size( MPI_COMM_WORLD,nranks, ierr)
:
!$OMP parallel private(i, ithread, nthreads)
:
  nthreads=OMP_GET_NUM_THREADS()
  ithread   =OMP_GET_THREAD_NUM()
  call pwork(ithread, irank, nthreads, nranks…)
  if(irank == 0) then
    call mpi_send(ithread,1,MPI_INTEGER, 1, ithread,MPI_COMM_WORLD, ierr)
  else
    call mpi_recv(         j,1,MPI_INTEGER, 0, ithread,MPI_COMM_WORLD, istatus,ierr)
    print*, "Yep, this is ",irank," thread ", ithread," I received from ", j
  endif

!$OMP END PARALLEL
end
```

Communicate between ranks.

Threads use tags to differentiate.

Conclusion

- Hybrid codes can reduce communication and memory requirements, support better cache reuse, and reduce memory traffic.
- Hybrid computing introduces another parallel layer.
- With 8-core and 16-core sockets on the way, more effort will be directed toward hybrid computing.
- Expect to see more multi-threaded libraries.

References

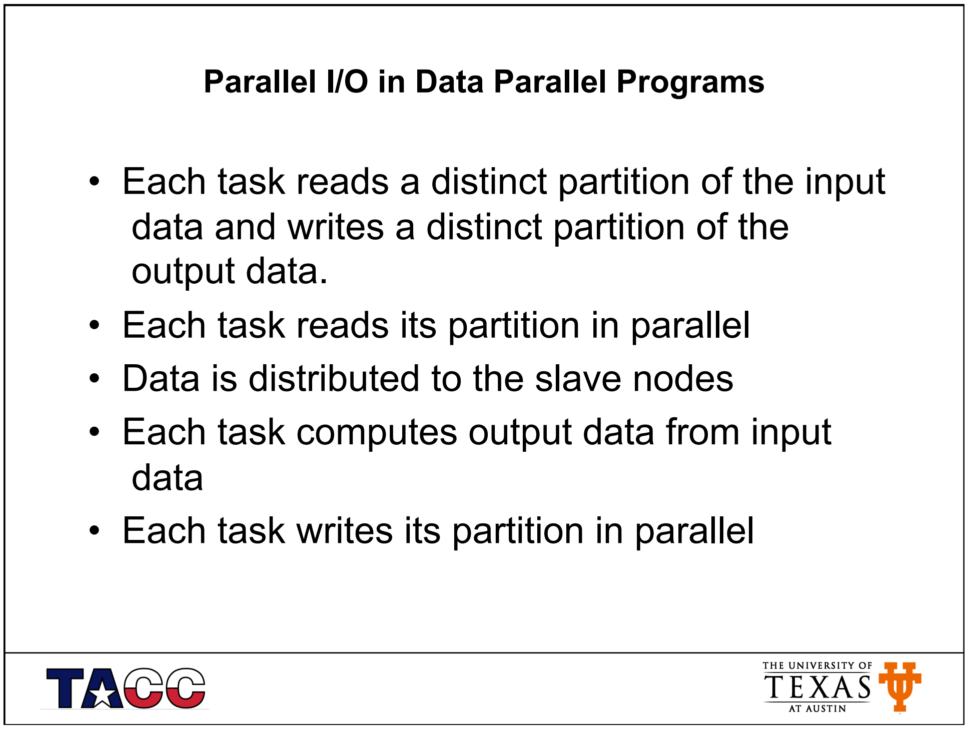
- http://www.nersc.gov/nusers/services/training/classes/NUG/Jun04/NUG2004_yhe_hybrid.ppt
  Hybrid OpenMP and MPI Programming and Tuning (NUG2004),Yun (Helen) He and Chris Ding, Lawrence Berkeley National Laboratory, June 24, 2004.
- http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/node162.htm#Node162
- www.tacc.utexas.edu/services/userguides/ranger {See numa section.}

I/O -(Parallel and Otherwise) on Large Scale Systems

Dan Stanzione
Arizona State University

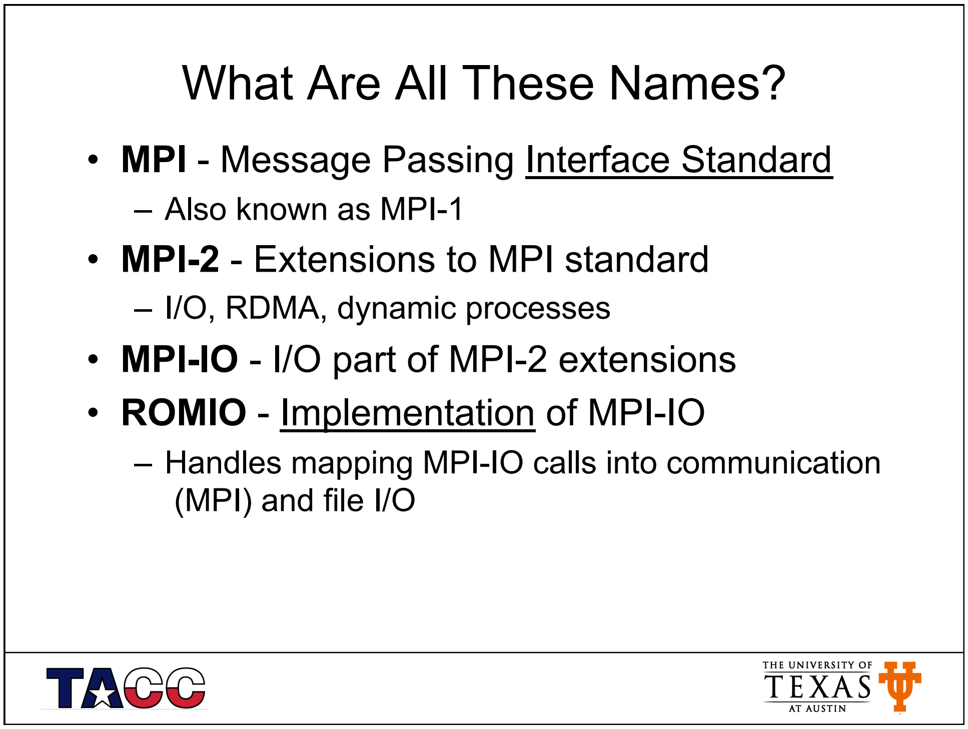
Outline

- What is Parallel I/O? Do I need it?
- Cluster Filesystem Options
- MPI I/O and ROMIO
- Example striping schemes



Parallel I/O in Data Parallel Programs

- Each task reads a distinct partition of the input data and writes a distinct partition of the output data.
- Each task reads its partition in parallel
- Data is distributed to the slave nodes
- Each task computes output data from input data
- Each task writes its partition in parallel



What Are All These Names?

- **MPI** - Message Passing <u>Interface Standard</u>
  - Also known as MPI-1
- **MPI-2** - Extensions to MPI standard
  - I/O, RDMA, dynamic processes
- **MPI-IO** - I/O part of MPI-2 extensions
- **ROMIO** - <u>Implementation</u> of MPI-IO
  - Handles mapping MPI-IO calls into communication (MPI) and file I/O

Filesystems

- Since each node in a cluster has it's own disk, making the same files available on each node can be problematic
- Three filesystem options:
  - Local
  - Remote (eg. NFS)
  - Parallel (eg. PVFS)

Filesystems (cont.)

- Local - Use storage on each node's disk
  - Relatively high performance
  - Each node has different filesystem
  - Shared datafiles must be copied to each node
  - No synchronization
  - Most useful for temporary/scratch files accessed only by copy of program running on single node
  - RANGER DOESN’T HAVE LOCAL DISKS
    - This trend may continue with other large scale systems for reliability reasons
    - Very, very small RAMdisk (300MB)

Filesystems(cont.)

- Remote - Share a single disk among all nodes
  - Every node sees same filesystem
  - Synchronization mechanisms manage changes
  - "Traditional" UNIX approach
  - Relatively low performance
  - Doesn't scale well; server becomes bottleneck in large systems
  - Simplest solution for small clusters, reading/writing small files

Filesystems(cont.)

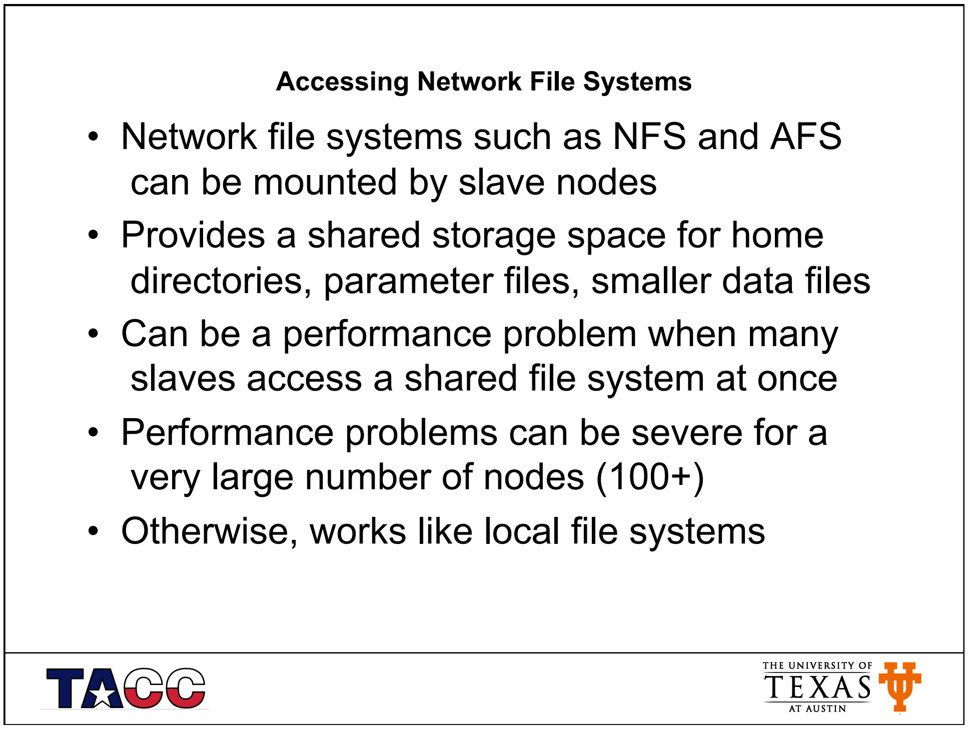
- Parallel - Stripe files across multiple disks on multiple nodes
  - Relatively high performance
  - Each node sees same filesystem
  - Works best for I/O intensive applications
  - Not a good solution for small files
  - Certain slave nodes are designated I/O nodes, local disks used to store pieces of filesystem

Using File Systems

- Local File Systems
  - EXT3, /tmp
- Network File Systems
  - NFS, AFS
- Parallel File Systems
  - PVFS, LUSTRE, IBRIX,Panasas
- I/O Libraries
  - HDF, NetCDF, Panda

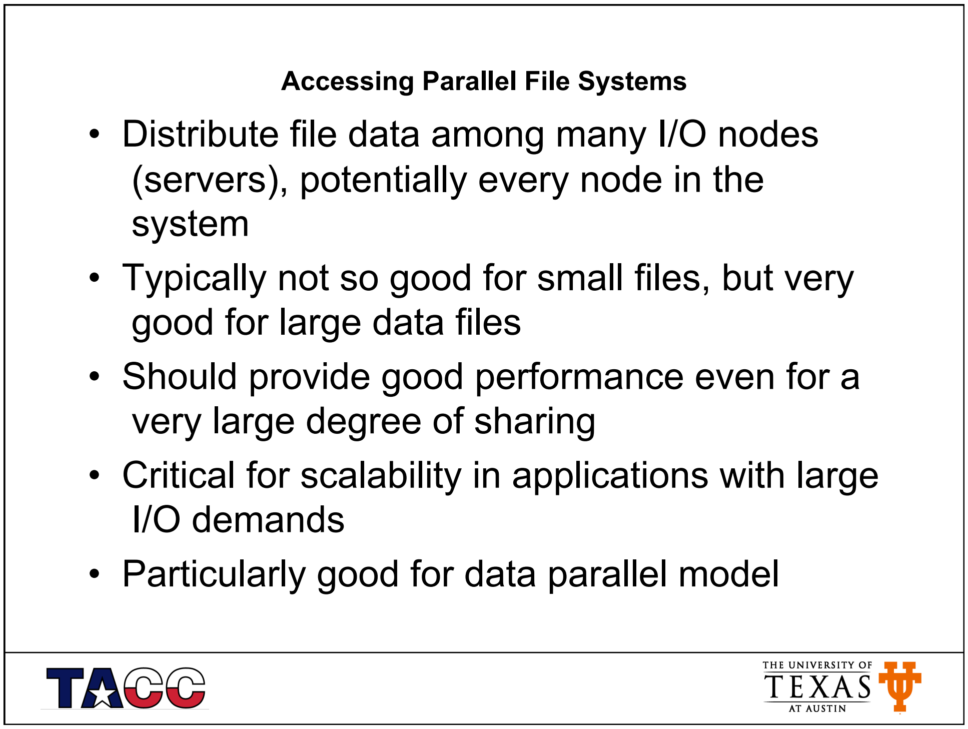
Accessing Local File Systems

- I/O system calls on compute nodes are executed on the compute node
- File systems on the slave can be made available to tasks running there and accessed as on any Linux system
- Recommended programming model does not assume that a task will run on a specific node
  - Best used for temporary storage
  - Access permissions may be a problem



Accessing Network File Systems

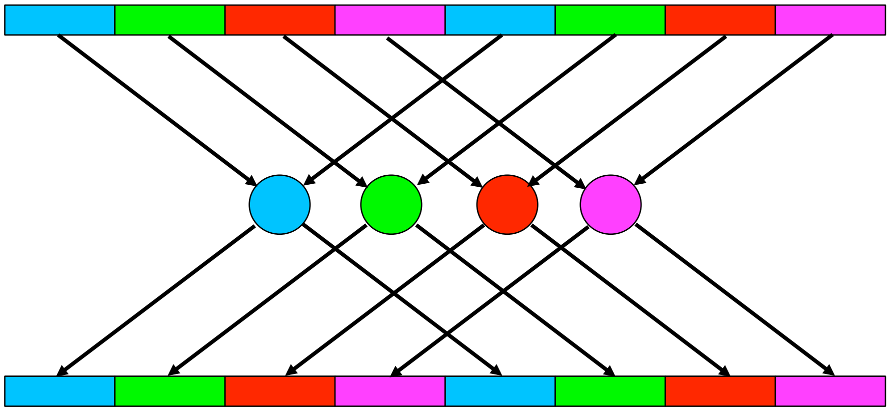
- Network file systems such as NFS and AFS can be mounted by slave nodes
- Provides a shared storage space for home directories, parameter files, smaller data files
- Can be a performance problem when many slaves access a shared file system at once
- Performance problems can be severe for a very large number of nodes (100+)
- Otherwise, works like local file systems



Accessing Parallel File Systems

- Distribute file data among many I/O nodes (servers), potentially every node in the system
- Typically not so good for small files, but very good for large data files
- Should provide good performance even for a very large degree of sharing
- Critical for scalability in applications with large I/O demands
- Particularly good for data parallel model

Example Application for Parallel I/O



Issues in Parallel I/O

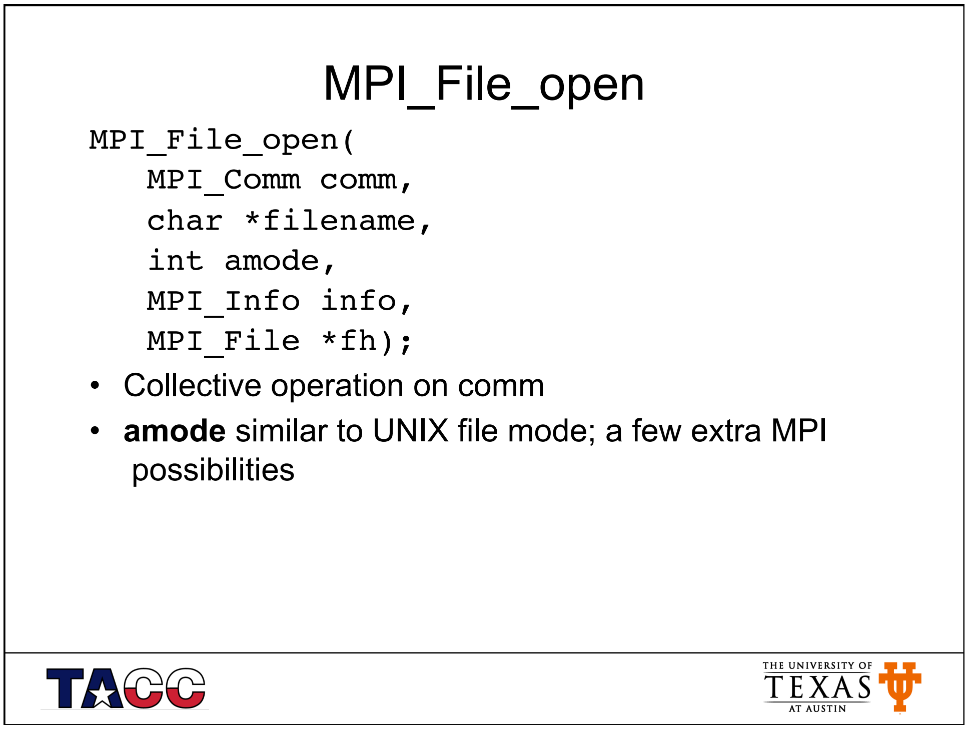
- Physical distribution of data to I/O nodes interacts with logical distribution of the I/O requests to affect performance
  - Logical record sizes should be considered in physical distribution
  - I/O buffer sizes depend on physical distribution and number of tasks
- Performance is best with rather large requests
  - Buffering should be used to get requests of 1MB or more, depending on the size of the system

I/O Libraries

- May make I/O simpler for certain applications
  - Multidimensional data sets
  - Special data formats
  - Consistent access to shared data
  - "Out-of-core" computation
- May hide some details of parallel file systems
  - Partitioning
- May provide access to special features
  - Caching, buffering, asynchronous I/O, performance

MPI-IO

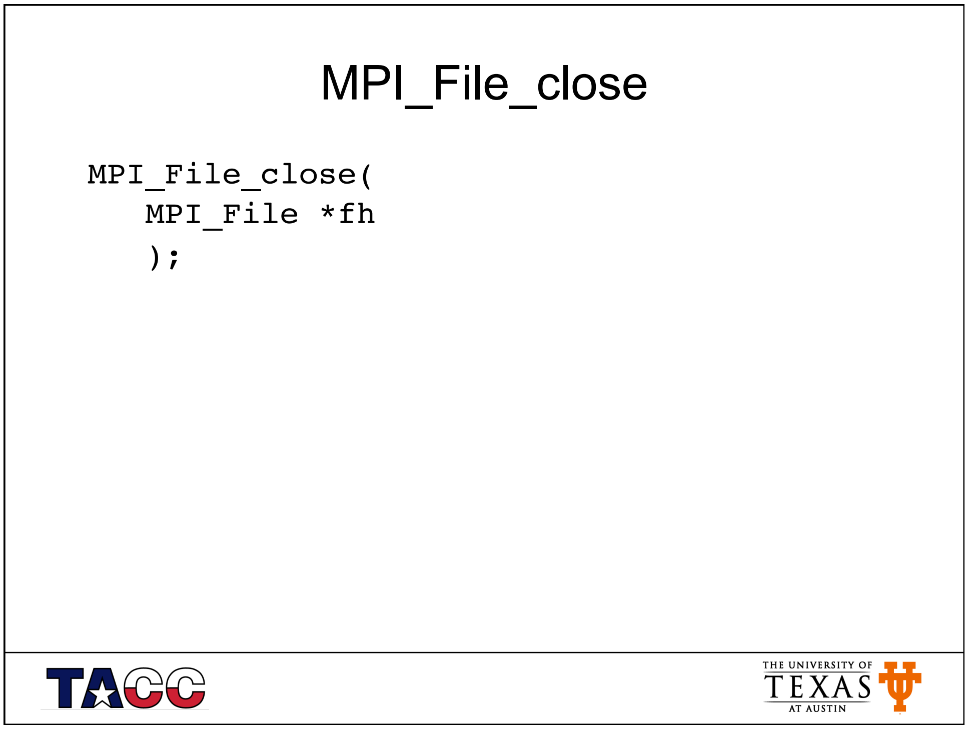
- Common file operations
  - `MPI_File_open();`
  - `MPI_File_close();`
  - `MPI_File_read();`
  - `MPI_File_write();`
  - `MPI_File_read_at();`
  - `MPI_File_write_at();`
  - `MPI_File_read_shared();`
  - `MPI_File_write_shared();`
- Open, close are collective. The rest have collective counterparts; add **_all**



MPI_File_open

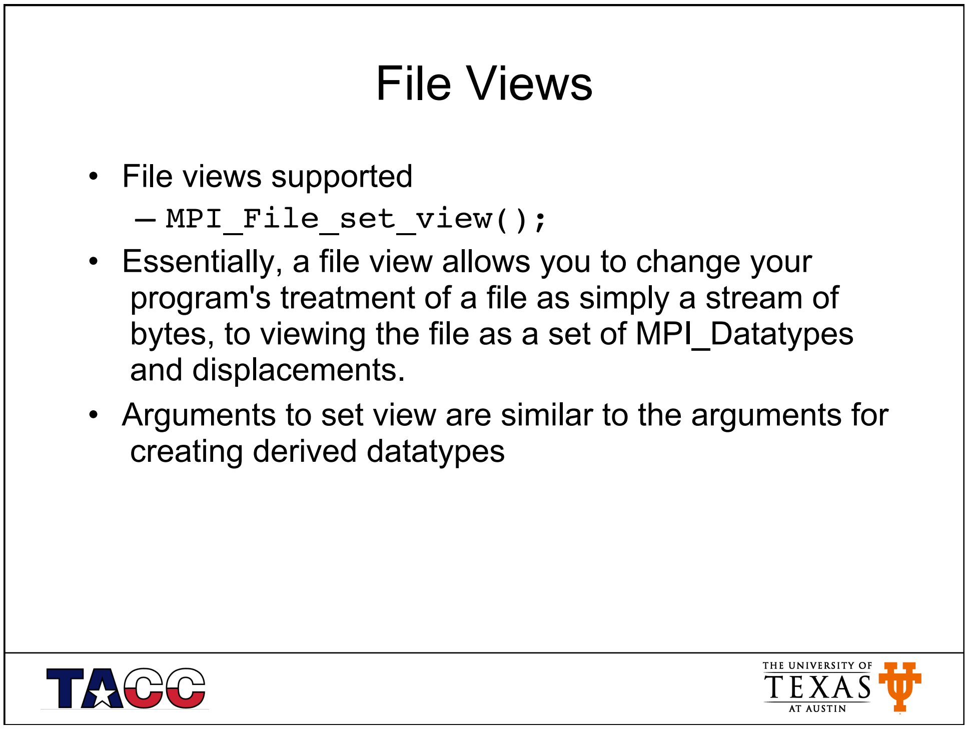
```
MPI_File_open(
   MPI_Comm comm,
   char *filename,
   int amode,
   MPI_Info info,
   MPI_File *fh);
```

- Collective operation on comm
- **amode** similar to UNIX file mode; a few extra MPI possibilities



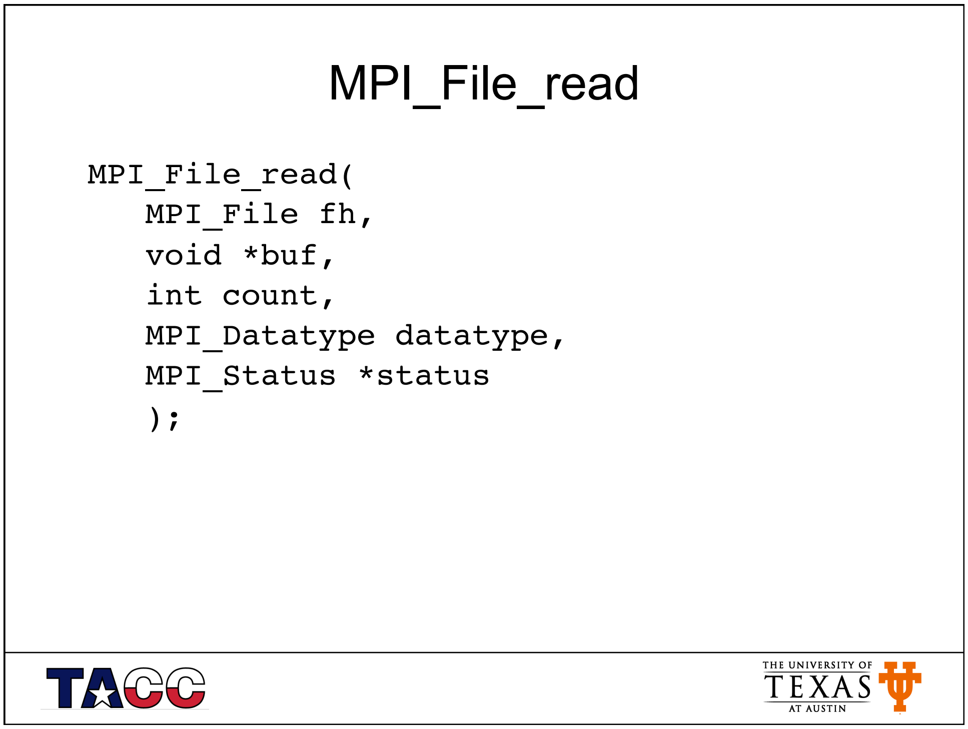
MPI_File_close

```
MPI_File_close(
   MPI_File *fh
   );
```



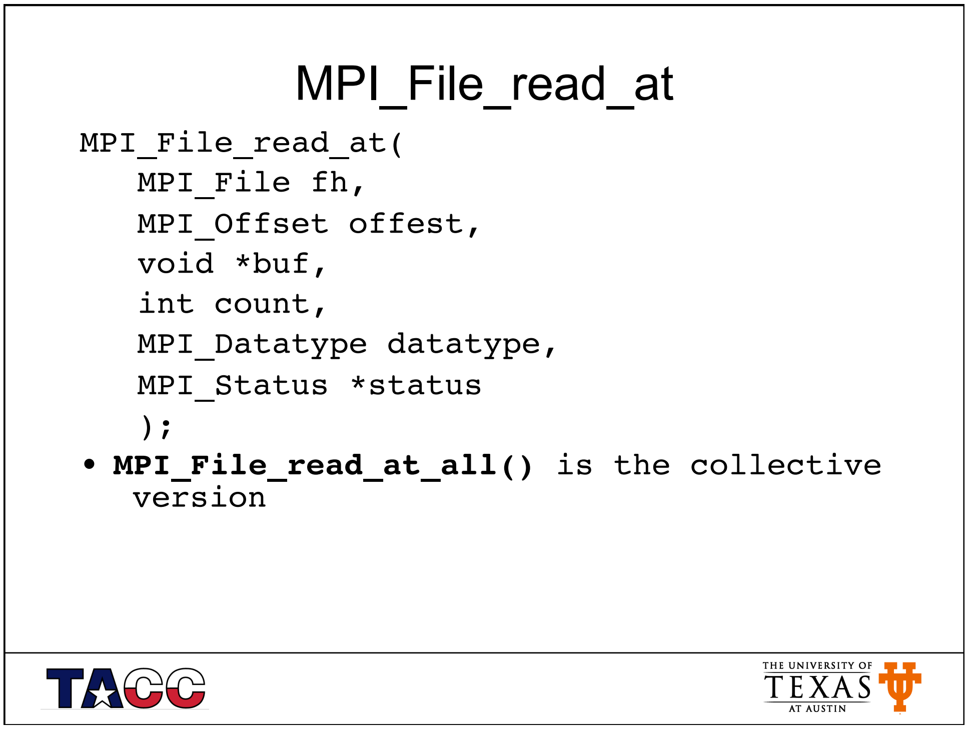
File Views

- File views supported
  - `MPI_File_set_view();`
- Essentially, a file view allows you to change your program's treatment of a file as simply a stream of bytes, to viewing the file as a set of MPI_Datatypes and displacements.
- Arguments to set view are similar to the arguments for creating derived datatypes



MPI_File_read

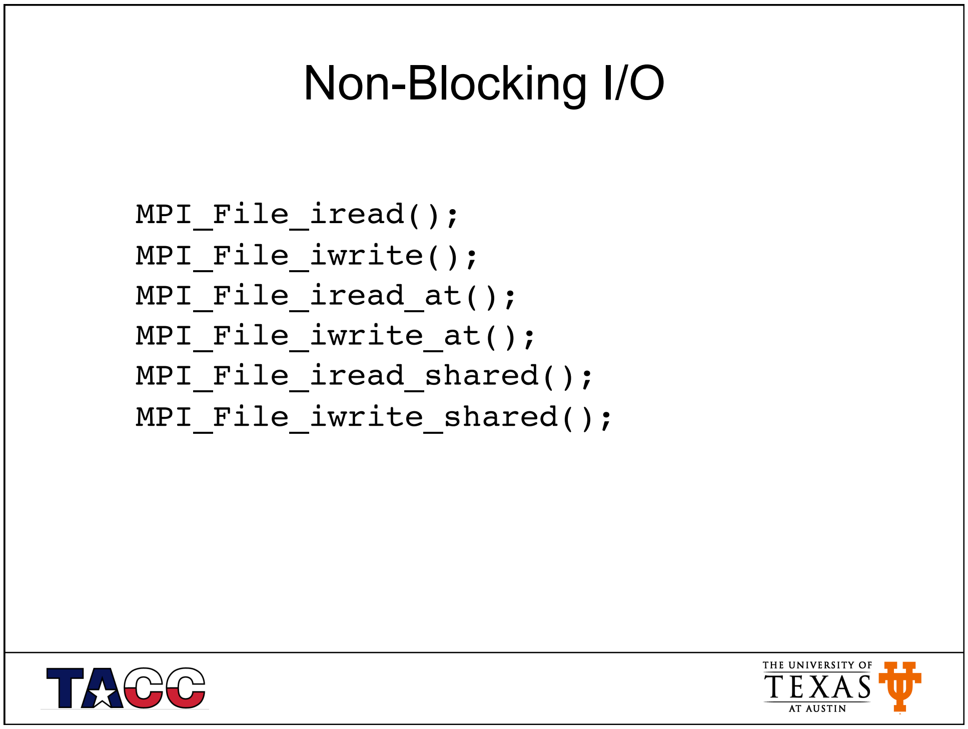
```
MPI_File_read(
   MPI_File fh,
   void *buf,
   int count,
   MPI_Datatype datatype,
   MPI_Status *status
   );
```



MPI_File_read_at

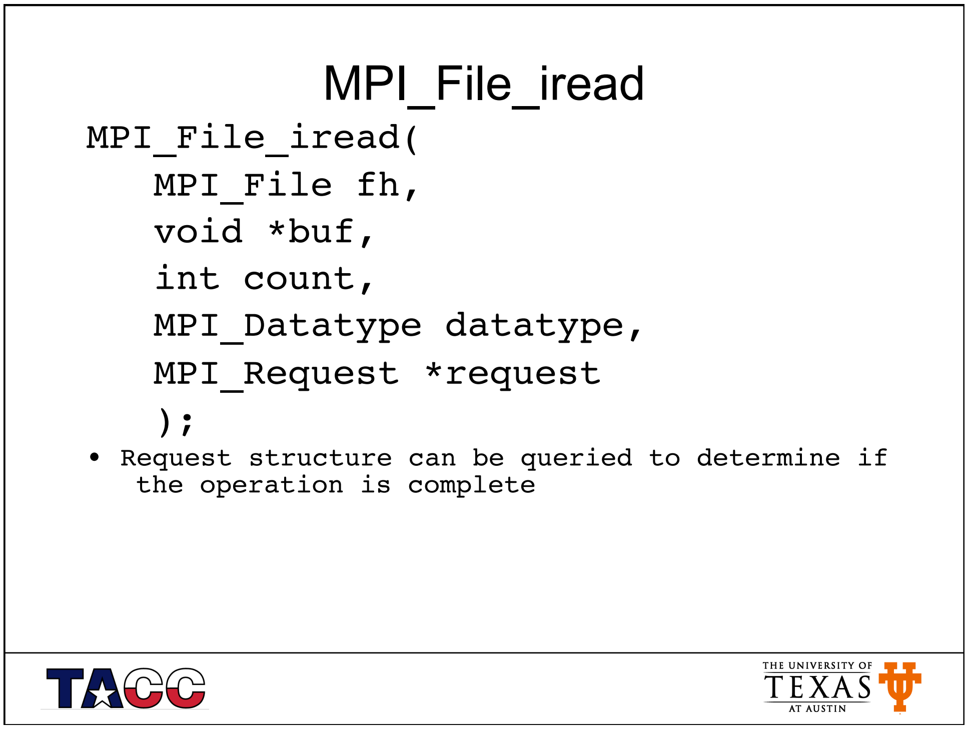
```
MPI_File_read_at(
   MPI_File fh,
   MPI_Offset offest,
   void *buf,
   int count,
   MPI_Datatype datatype,
   MPI_Status *status
   );
```

- **MPI_File_read_at_all()** is the collective version



Non-Blocking I/O

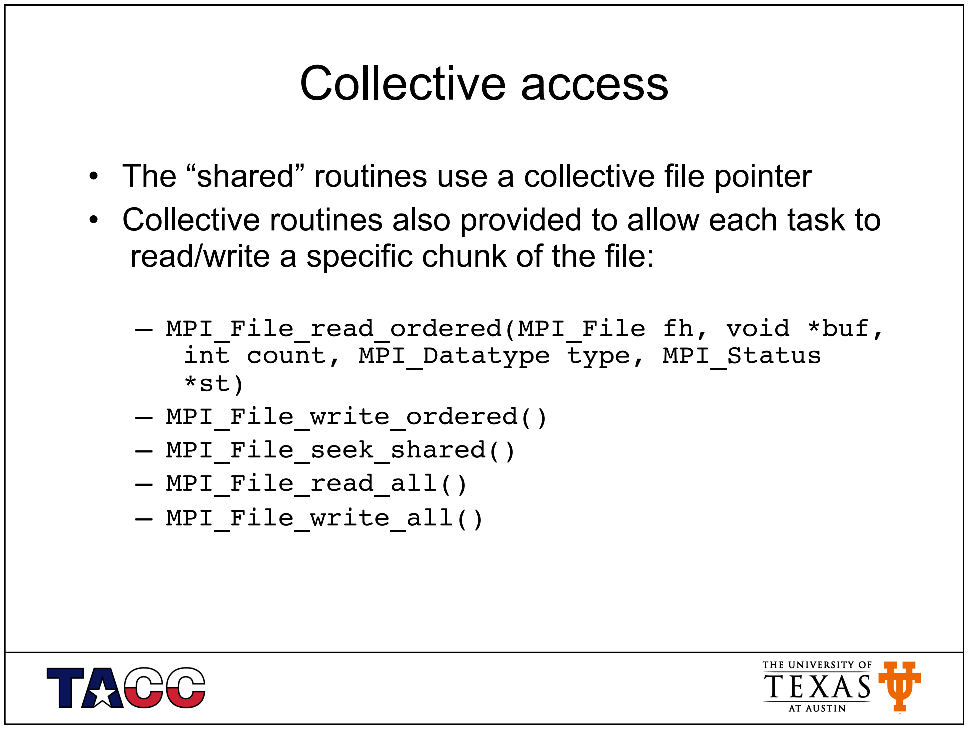
```
MPI_File_iread();
MPI_File_iwrite();
MPI_File_iread_at();
MPI_File_iwrite_at();
MPI_File_iread_shared();
MPI_File_iwrite_shared();
```



MPI_File_iread

```
MPI_File_iread(
    MPI_File fh,
    void *buf,
    int count,
    MPI_Datatype datatype,
    MPI_Request *request
    );
```

- Request structure can be queried to determine if the operation is complete



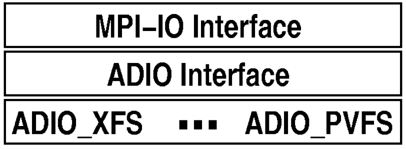
Collective access

- The “shared” routines use a collective file pointer
- Collective routines also provided to allow each task to read/write a specific chunk of the file:
  - `MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype type, MPI_Status *st)`
  - `MPI_File_write_ordered()`
  - `MPI_File_seek_shared()`
  - `MPI_File_read_all()`
  - `MPI_File_write_all()`

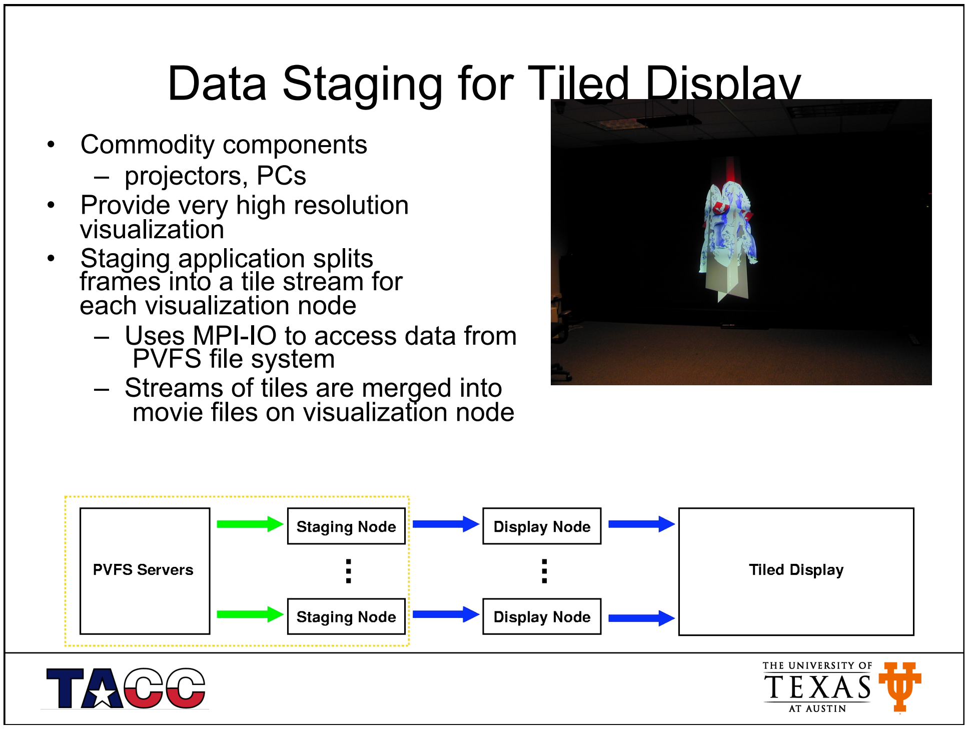
File Functions

```
–MPI_File_delete();
–MPI_File_set_size();
–MPI_File_preallocate();
–MPI_File_get_size();
–MPI_File_get_group();
–MPI_File_get_amode();
–MPI_File_set_info();
–MPI_File_get_info();
```

ROMIO MPI-IO Implementation



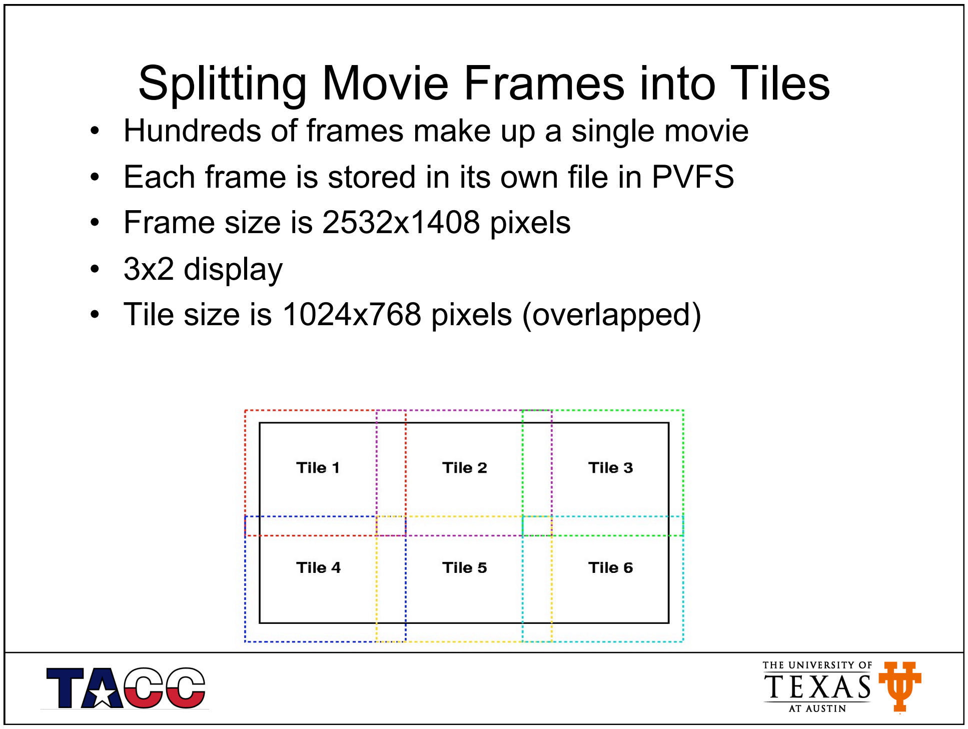
- Implementation of MPI-2 I/O specification
  - Operates on wide variety of platforms
  - Abstract Device Interface for I/O (ADIO) aids in porting to new file systems
  - Fortran and C bindings
- Successes
  - Adopted by industry (e.g. Compaq, HP, SGI)
  - Used at ASCI sites (e.g. LANL Blue Mountain)



Data Staging for Tiled Display

- Commodity components
  - projectors, PCs
- Provide very high resolution visualization
- Staging application splits frames into a tile stream for each visualization node
  - Uses MPI-IO to access data from PVFS file system
  - Streams of tiles are merged into movie files on visualization node

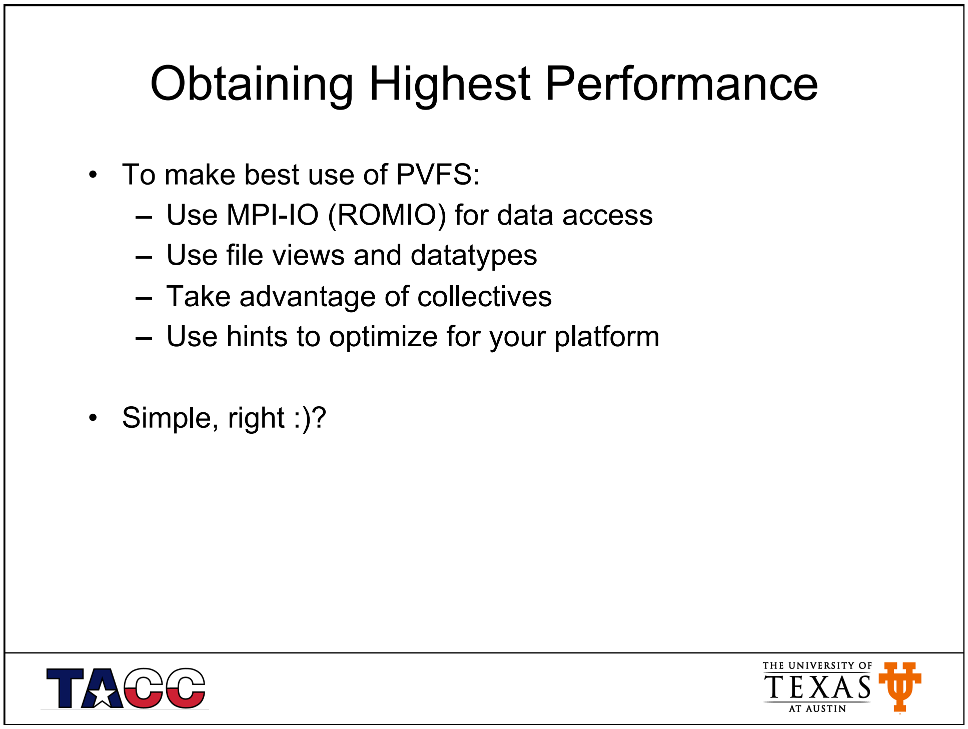
PVFS Servers → Staging Node ⋮ Staging Node → Display Node ⋮ Display Node → Tiled Display



Splitting Movie Frames into Tiles

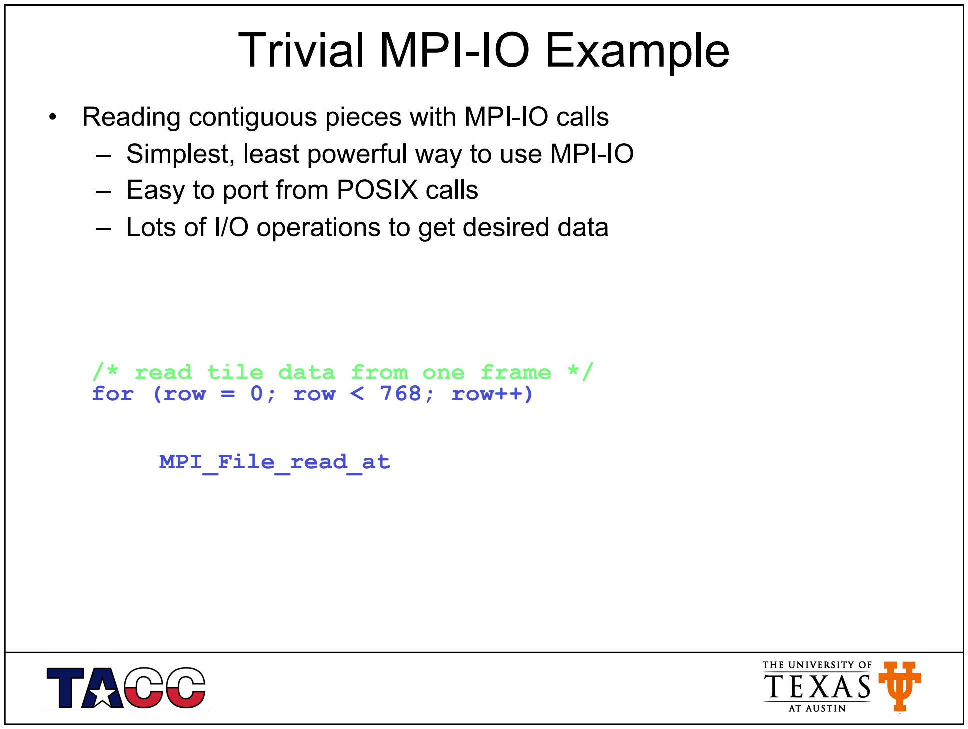
- Hundreds of frames make up a single movie
- Each frame is stored in its own file in PVFS
- Frame size is 2532x1408 pixels
- 3x2 display
- Tile size is 1024x768 pixels (overlapped)

Tile 1	Tile 2	Tile 3
Tile 4	Tile 5	Tile 6



Obtaining Highest Performance

- To make best use of PVFS:
  - Use MPI-IO (ROMIO) for data access
  - Use file views and datatypes
  - Take advantage of collectives
  - Use hints to optimize for your platform
- Simple, right :)?



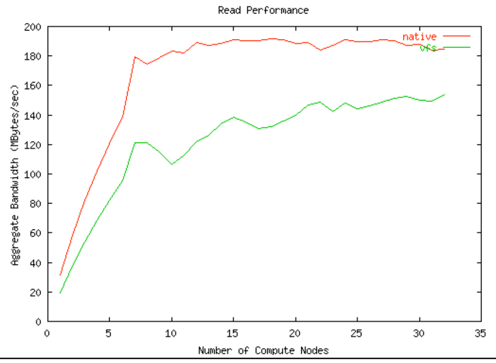
Trivial MPI-IO Example

- Reading contiguous pieces with MPI-IO calls
  - Simplest, least powerful way to use MPI-IO
  - Easy to port from POSIX calls
  - Lots of I/O operations to get desired data

```
/* read tile data from one frame */
for (row = 0; row < 768; row++)

     MPI_File_read_at
```

Avoiding the VFS Layer

- UNIX calls go through VFS layer
- MPI-IO calls use Filesystem library directly
- Significant performance gain



Why Use File Views?

- Concisely describe noncontiguous regions in a file
  - Create datatype describing region
  - Assign “view” to open file handle
- Separate description of region from I/O operation
  - Datatype can be reused on subsequent calls
- Access these regions with a single operation
  - Single MPI read call requests all data
  - Provides opportunity for optimization of access in MPI-IO implementation…

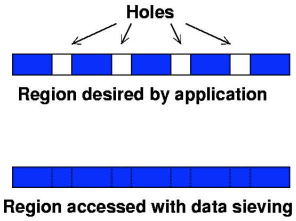
Setting a File View

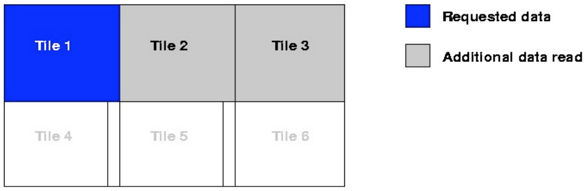
- Use MPI_Type_create_subarray() to define a datatype describing the data in the file
- Example for tile access (24-bit data):

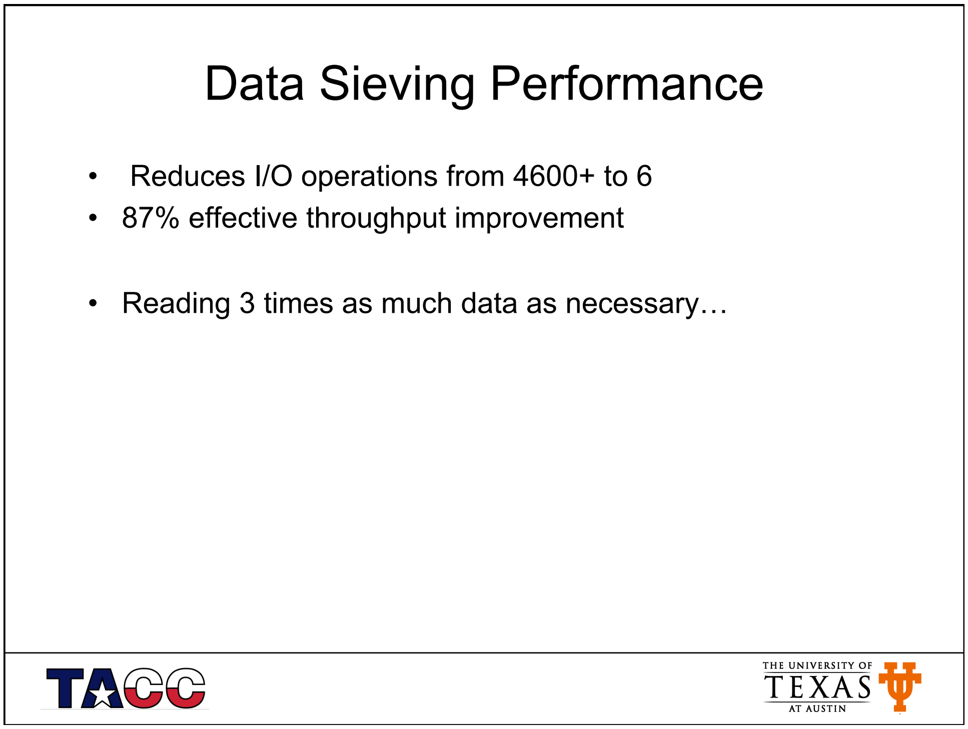
```
                      /* frame width */
                      /* frame height */
                      /* tile width */
                      /* tile height */
/* create datatype describing tile */
                                         tiletype
                 tiletype
MPI_File_set_view
     tiletype
MPI_File_read
```

Noncontiguous Access in ROMIO

- ROMIO performs “data sieving” to cut down number of I/O operations
- Uses large reads which grab multiple noncontiguous pieces
- Example, reading tile 1:


Holes
Region desired by application
Region accessed with data sieving

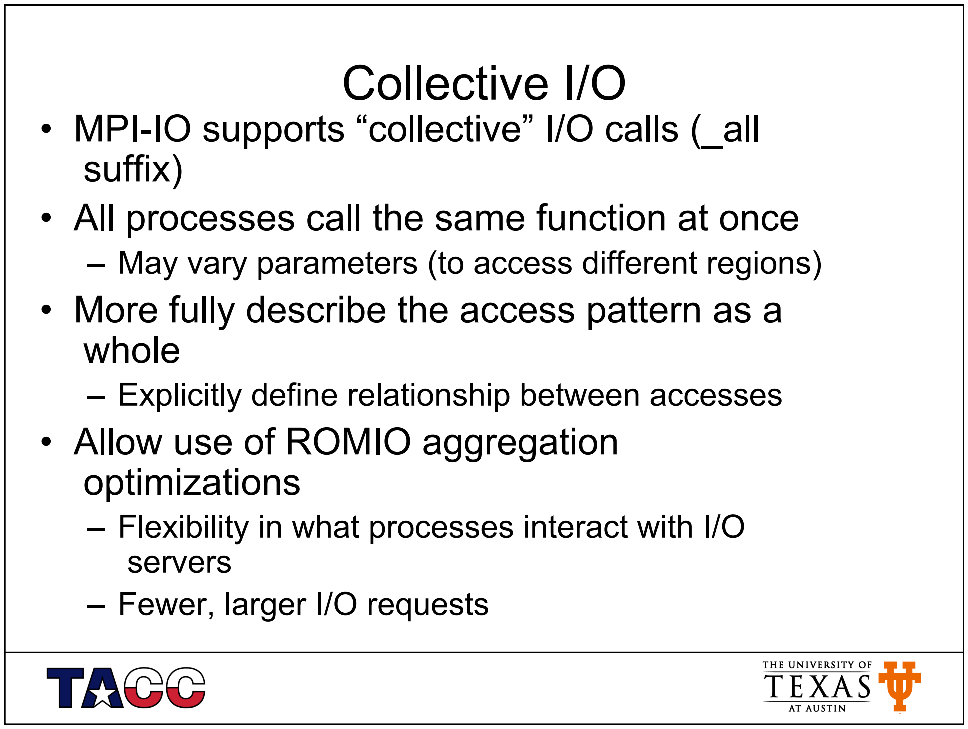




Data Sieving Performance

- Reduces I/O operations from 4600+ to 6
- 87% effective throughput improvement

- Reading 3 times as much data as necessary…



Collective I/O

- MPI-IO supports “collective” I/O calls (_all suffix)
- All processes call the same function at once
  - May vary parameters (to access different regions)
- More fully describe the access pattern as a whole
  - Explicitly define relationship between accesses
- Allow use of ROMIO aggregation optimizations
  - Flexibility in what processes interact with I/O servers
  - Fewer, larger I/O requests

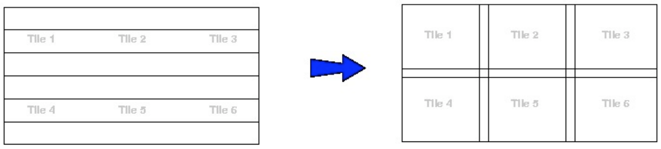
Collective I/O Example

- Single line change:

```
/* create datatype describing tile */
MPI_Type_create_subarray(2, frame_size, tile_size,
     tile_offset, MPI_ORDER_C, rgbtype, &tiletype);
MPI_Type_commit(&tiletype);

MPI_File_set_view(handle, header_size, rgbtype,
     tiletype, “native”, MPI_INFO_NULL);

#if 0
MPI_File_read(handle, buffer, buffer_size,
     rgbtype, &status);
#endif

/* collective read */
MPI_File_read_all(handle, buffer, buffer_size,
     rgbtype, &status);
```

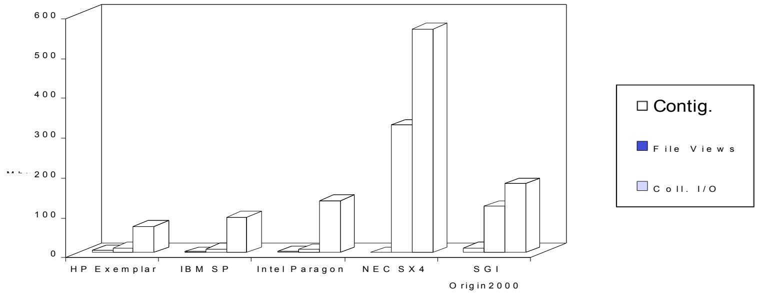
Two-Phase Access

- ROMIO implements two-phase collective I/O
  - Data is read by clients in contiguous pieces (phase 1)
  - Data is redistributed to the correct client (phase 2)
- ROMIO applies two-phase when collective accesses overlap between processes
- More efficent I/O access than data sieving alone



Two-Phase Performance

Often a big win:

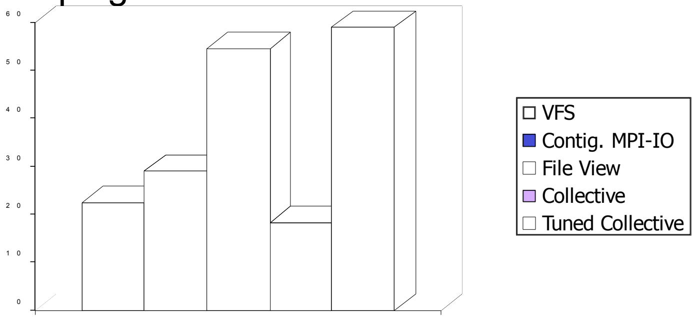


Hints

- Controlling PVFS
  - striping_factor - size of “strips” on I/O servers
  - striping_unit - number of I/O servers to stripe across
  - start_iodevice - which I/O server to start with
- Controlling aggregation
  - cb_config_list - list of aggregators
  - cb_nodes - number of aggregators (upper bound)
- Tuning ROMIO optimizations
  - romio_cb_read, romio_cb_write - aggregation on/off
  - romio_ds_read, romio_ds_write - data sieving on/off

The Proof is in the Performance

- Final performance is almost 3 times VFS access!
- Hints allowed us to turn off two-phase, modify striping of data



Summary: Why Use MPI-IO?

- Better concurrent access model than POSIX one
  - Explicit list of processes accessing concurrently
  - More lax (but still very usable) consistency model
- More descriptive power in interface
  - Derived datatypes for concise, noncontiguous file and /or memory regions
  - Collective I/O functions
- Optimizations built into MPI-IO implementation
  - Noncontiguous access
  - Collective I/O (aggregation)
- Performance portability