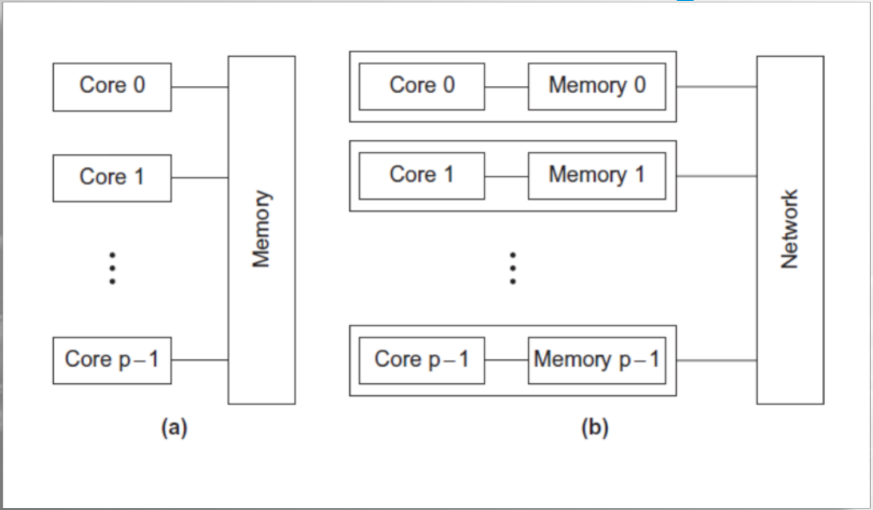
MPI

Sven Bingert

Gesellschaft für wissenschaftliche Datenverarbeitung Göttingen

2018-05-22

Shared vs. Distributed Memory

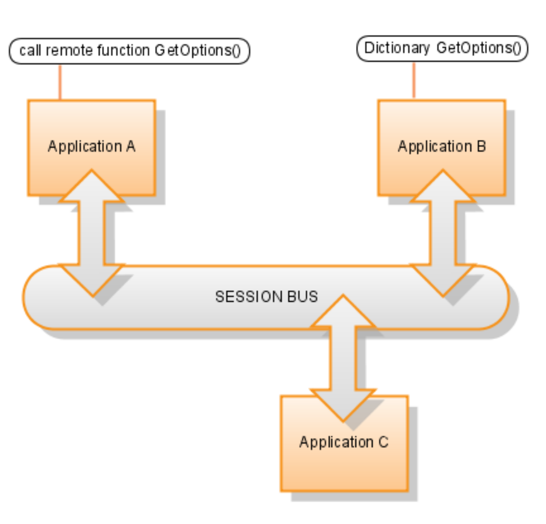


(a) Shared memory (b) Distributed Memory

Inter-process communication in DM-systems

- Only way of synchronization: Pass Messages between processes.
- Only way of sharing Data: Pass Messages between processes.
- Can happen synchronous and asynchronous
- Two major possibilities
  - Remote Procedure Calls – e.g. bus systems on the desktop
  - Message Passing – HPC
  - (RPC, Web Services, SOAP, JSON, Sockets, internet in general)
- Also works on Shared Memory systems

Bus Systems on the Desktop



https://sandersoncoelho.wordpress.com/2009/04/08/gettingsending-a-dictionary-through-dbus-with-qt/

Message Passing APIs

- Manage process spawning and process management
- Manage efficient(!) communication between processes – depending on the underlying network/ system
- Provide Function calls for this
- Two important APIs
  - Parallel Virtual Machine (PVM) – *deprecated since 2009*
  - Message Passing Interface (MPI) – *this course*

Message Passing Interface – MPI

- Standardized programming API for C and Fortran
- Version 1.0 1994
- Current Version 3.1 from June 2015
- Many implementations, open and closed source
- All share the same API but a different on-wire protocol
- Most important implementations: MPICH and OpenMPI (aka LAM/MPI)
- Bindings for all major languages exist (C++, C#, Java, Python, Perl, R, Haskell, Ruby...not: Javascript :-))
- Sadly again: *No Magic* – you have to do all parallelization yourself

Using MPI

- Compile using `mpicc`-Command!
- Run using `mpirun`-Command (sometimes called `mpiexec`)
- For GWDG resources:

  `https://info.gwdg.de/dokuwiki/doku.php?id=en:services:application_services:high_performance_computing:running_jobs`
- together with OpenMP/ pthreads:
  - MPICH is NOT thread safe
  - OpenMPI is thread safe
  - consult MPI_Init_thread(3) !

MPI slang

- Processes are often called *Processors*
- Communicator – a group of processes, default: `MPI_COMM_WORLD`
- rank – unique ID inside a communicator, integer starting with 0
- abort – MPI programs usually abort on their own if something goes wrong

Very basic program

```
#include <mpi.h>

int main(int argc, char *argv[]) {
    int  numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc,&argv); // pass argv to all processes
    if (rc != MPI_SUCCESS) {
        printf ("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
        }

    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(hostname, &len);
    printf ("Number of tasks= %d My rank= %d Running on %s\n",
              numtasks,rank,hostname);

    MPI_Finalize();
}
```

Important Functions

- `MPI_Init` (*&argc,&argv*)
- `MPI_Comm_size` (*communicator,&number_of_tasks*) – save number of tasks
- `MPI_Comm_rank` (*communicator, &rank*) – save own process id (aka rank)
- `MPI_Get_processor_name` (*&name,&length*) – save hostname in string *name* of length *length*
- `MPI_Wtime` () – return wall clock
- `MPI_Finalize` ()

Compiling/ Running/ Batch system

- `mpicc -o hello hello.c`
- `mpirun -n 8 ./hello`
- `mpirun -n 8 -hosts c025,c026 ./hello`
- Or use PBS: (example for 16 processes on 8 nodes)

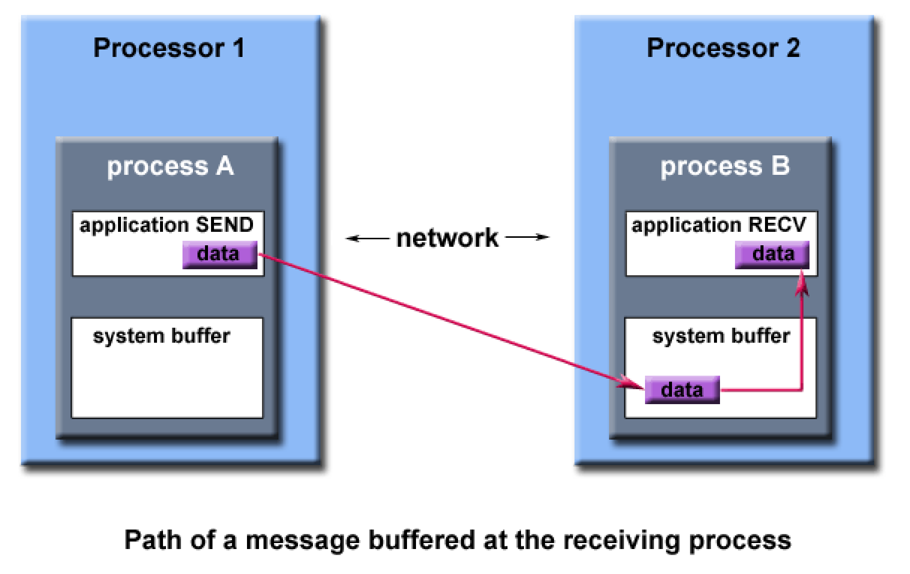
```
#!/bin/sh
#PBS -N mpi_hello
#PBS -l nodes=8

/usr/bin/mpirun -n 16 ~/mpi/a.out
```

Point-to-Point Communication

- Consists of a `Send` () and `Recv` () function
- Several types
  - Synchronous/ asynchronous send
  - Blocking/ non-blocking send/ receive
  - Combined send/recv
- Blocking send can be used with non-blocking receive and vice versa.
- Often messages are buffered on receiver side (depending on implementation).
- Messages from the same process will never overtake each other.

Buffering



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

Blocking vs. Non-Blocking

- Blocking:
  - Send will *return* after it is safe to modify the send-buffer.
  - Send can be synchronous – only returns after receiver confirms receive.
  - Send can be asynchronous if an internal system buffer is used to hold the data.
  - Receive only *returns* after the data has arrived.
- Non-blocking:
  - Always returns immediately.
  - Request the MPI library to perform the operation when it is able. The user can not predict when that will happen.
  - It is unsafe to modify the send-buffer until a flag says so. (i.e. `MPI_Wait` ())
  - Primarily used to overlap computation with communication.
  - Very tricky – meant for *pro-users*

Communication Modes

Communication Mode	Blocking Routines	Non-Blocking Routines
Synchronous	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND
Standard	MPI_SEND	MPI_ISEND
	MPI_RECV	MPI_IRECV
	MPI_SENDRECV	
	MPI_SENDRECV_REPLACE	

Send and Receive

- ```
  MPI_Send(
      void* data,               int count,
      MPI_Datatype datatype,    int destination,
      int tag,                  MPI_Comm communicator)
  MPI_Recv(
      void* data,               int count,
      MPI_Datatype datatype,    int source,
      int tag,                  MPI_Comm communicator,
      MPI_Status* status)
  ```
- *data* – pointer to the buffer for send/ receive
- *count* – number of entries of size *datatype* in buffer
- *datatype* – MPI_INT, MPI_LONG, MPI_DOUBLE, MPI_CHAR...
- *destination/ source* – rank of sender/ receiver; recv also MPI_ANY_SOURCE
- *tag* – integer to discriminate different messages
- *communicator* – MPI_COMM_WORLD
- *status* – MPI_STATUS_IGNORE
  ```

Forms of MPI Send

- *MPI_Send* will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive)
- *MPI_Bsend* may buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- *MPI_Ssend* will not return until matching receive posted
- *MPI_Rsend* may be used ONLY if matching receive already posted. User responsible for writing a correct program.
- *MPI_Isend* Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see MPI_Request_free). Note also that while the I refers to immediate, there is no performance requirement on MPI_Isend. An immediate send must return to the user without requiring a matching receive at the destination. An implementation is free to send the data to the destination before returning, as long as the send call does not block

Dynamic Receive

- Good to know the size of the received data before receiving!
- `MPI_Probe` () blocks until the next message arrives and gathers some information.
- Message stays in system-buffer.
-
```c
MPI_Status status;

// Probe for an incoming message from process zero
// MPI_Probe(source,tag,comm,status)
MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

// size of incoming message in number of MPI_INTs
MPI_Get_count(&status, MPI_INT, &size);

int* buf = (int*)malloc(sizeof(int) * size);
MPI_Recv(buf, size, MPI_INT, 0, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Interesting functions

- `MPI_Ssend` () – Synchronous blocking send
- `MPI_Bsend` ()
  `MPI_Buffer_attach` ()
  `MPI_Buffer_dettach` ()
  – Buffered blocking send – useful if system buffer is to small
- `MPI_Isend` ()
  `MPI_Irecv` () – Non-Blocking send/ receive
- `MPI_Test` (*&request,&flag,&status*)
  `MPI_Test_all` (*count,&array_of_requests,&flag,&array_of_status*)
  – Test if *request(s)* are finished (*int flag*=1)
- `MPI_Wait` (*&request,&status*) and `MPI_Wait_all` ()
  – Block until *request(s)* finish

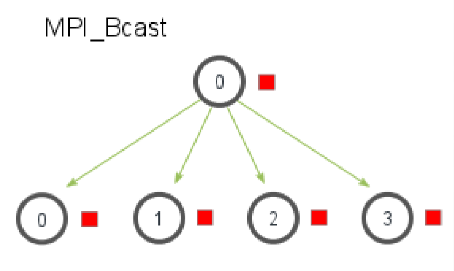
Collective Communication

- *always* involve **all** processes in a communicator
- are always blocking (MPI-3 provides non-blocking routines).
- **Sender and receiver call the same function**.
- Important types `MPI_`
  - `Barrier` (*communicator*)
  - `Broadcast` () – send same data to everybody (1:n)
  - `Scatter` () – send different data to everybody (1:n)
  - `Gather` () – receive different data from everybody (n:1)
  - `Allgather` () – everybody sends to everybody (n:n)

Broadcast

```
MPI_Bcast(          void* data, int count,
        MPI_Datatype datatype, int root,
        MPI_Comm communicator)
```

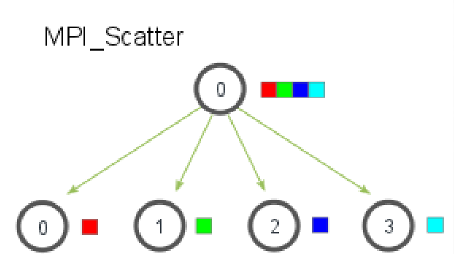
- Same function for send AND receive
- process with rank *root sends* data
- everybody else *receives* data



Scatter

```
MPI_Scatter(         void* data, int count,
    MPI_Datatype send_datatype,
               void* recv_data, int recv_count,
    MPI_Datatype recv_datatype, int root,
    MPI_Comm communicator)
```

- process with rank *root sends count* from *data* to each process
- process 0 gets the first *count* chunk, process 1 the 2nd...
- *data* has size (#processes×count)
- Usually *count==recv_count*

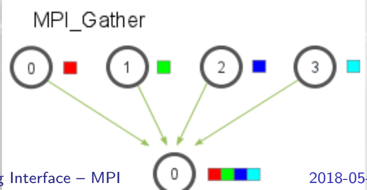


Gather

```
MPI_Gather(  void* send_data, int count,
  MPI_Datatype send_datatype,
             void* recv_data, int recv_count,
  MPI_Datatype recv_datatype, int root,
        MPI_Comm communicator)

t
```

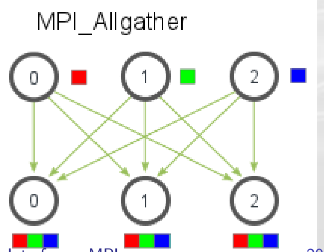
- process with rank *root receives count data* from each process
- the first *count* chunk in *recv_data* comes from process 0, the 2nd from proc 1...
- *recv_data* has size (#processes×count)
- Usually *count==recv_count*



Allgather

```
MPI_Allgather(  void* send_data, int count,
     MPI_Datatype send_datatype,
                void* recv_data, int recv_count,
     MPI_Datatype recv_datatype, MPI_Comm communicator)
```

- every process *receives count data* from each other process
- the first *count* chunk in *recv_data* comes from process 0, the 2nd from proc 1...
- *recv_data* has size (#processes×count)
- Usually *count==recv_count*



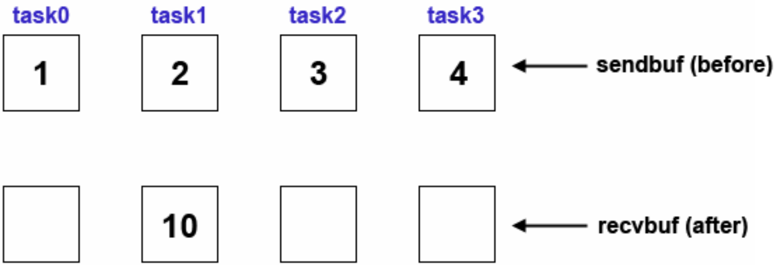
Reduction

MPI_Reduce

Perform reduction across all tasks in communicator and store result in 1 task

```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT,
           MPI_SUM, dest, MPI_COMM_WORLD);
```

task1 will contain result



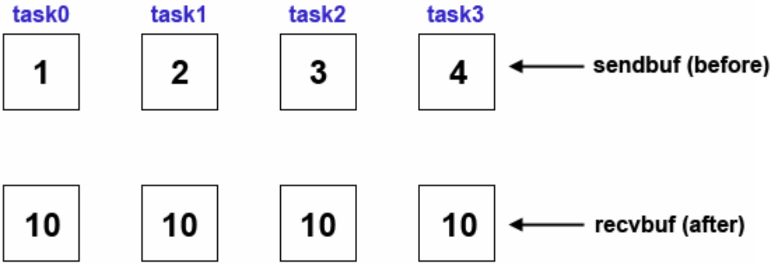
https://computing.llnl.gov/tutorials/mpi/

Allreduce

MPI_Allreduce

Perform reduction and store result across all tasks in communicator

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT,
              MPI_SUM, MPI_COMM_WORLD);
```



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

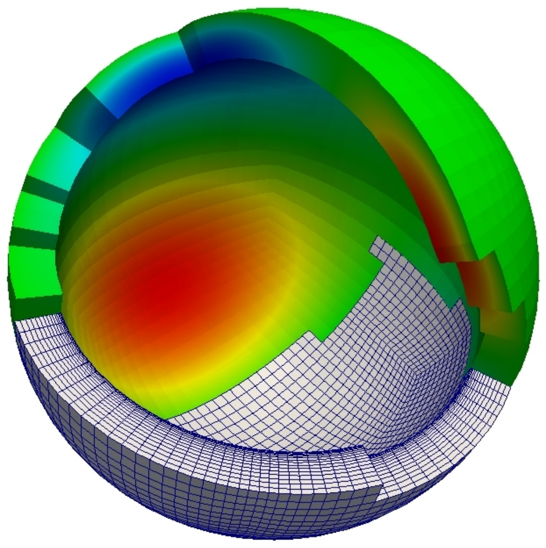
Other Things one might want to do

- Derive own data types – `MPI_Type_struct ()`
- Create own communicators/ groups – `MPI_Comm_create ()`
- Create virtual topologies to map processes e.g. on a lattice

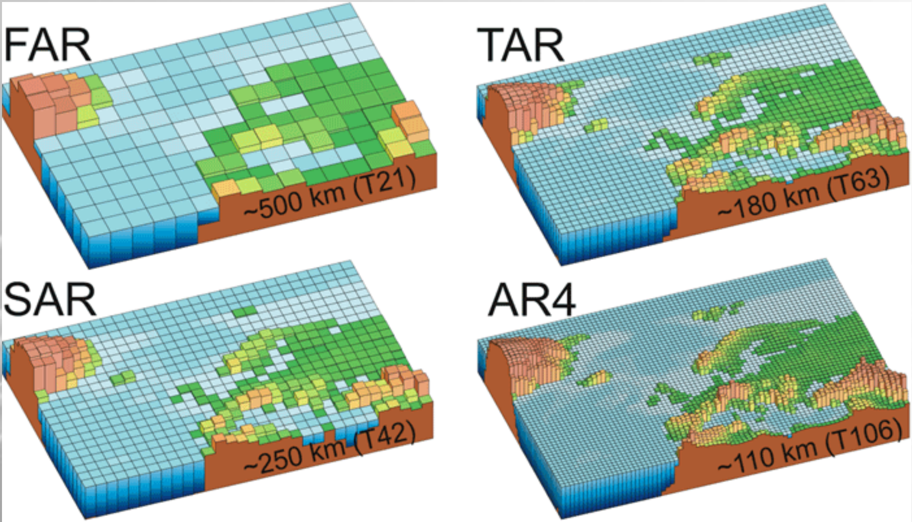
Debugging

- Run program interactive with only one process and debug `gdb ./mpiprog`
- Use `mpirun` to launch e.g. an xterm on every node and run `gdb` there: `mpirun -n 4 xterm -e gdb ./mpiprog`
- Some implementations of `mpirun` support *native* debugging: `mpirun -gdb -n 16 ./mpiprog`
- Use a full blown commercial debugger like TotalView or DDT
- Also compare `http://www.open-mpi.de/faq/?category=debugging#serial-debuggers`
- and `https://stackoverflow.com/questions/329259/how-do-i-debug-an-mpi-program/24480711#24480711`

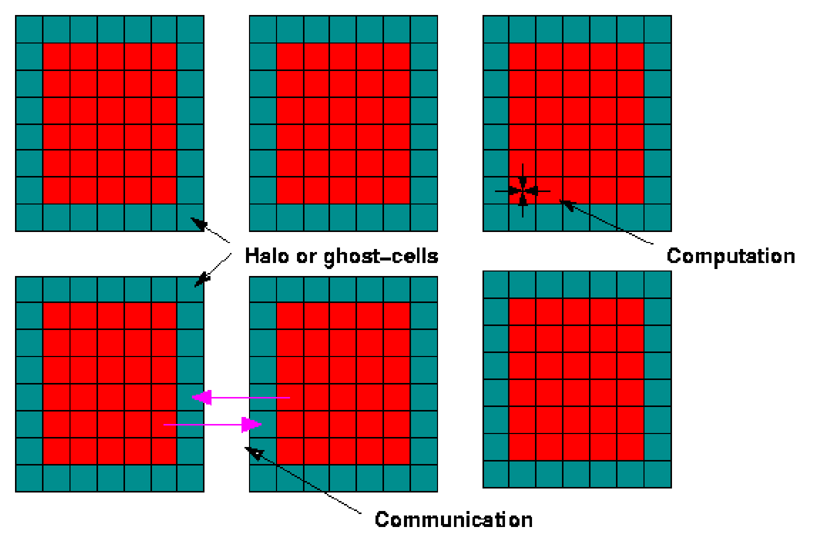
Weather



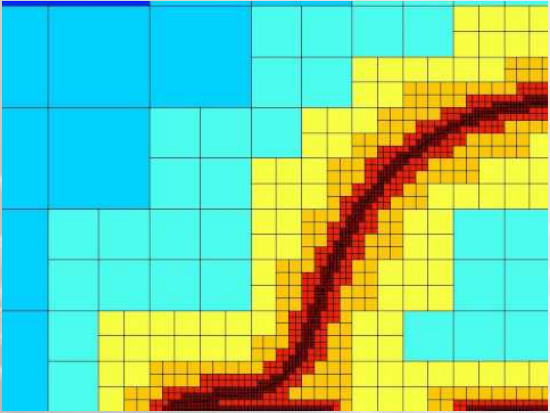
Weather Refined



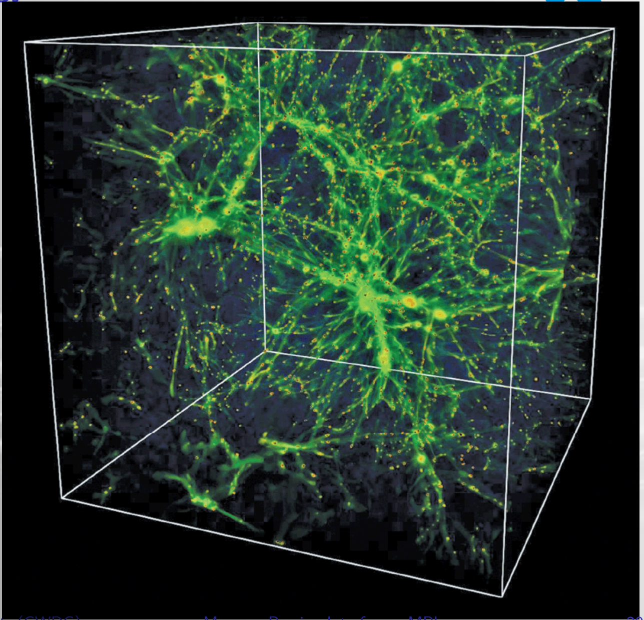
Grids and Ghost Cells



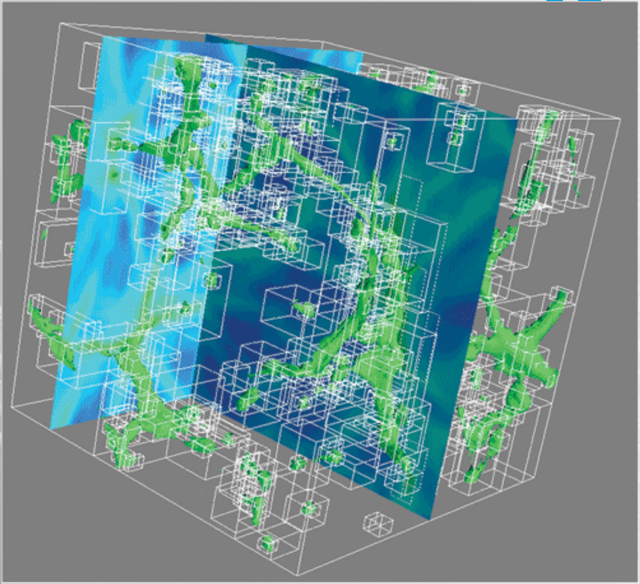
Adaptive Mesh Refinement



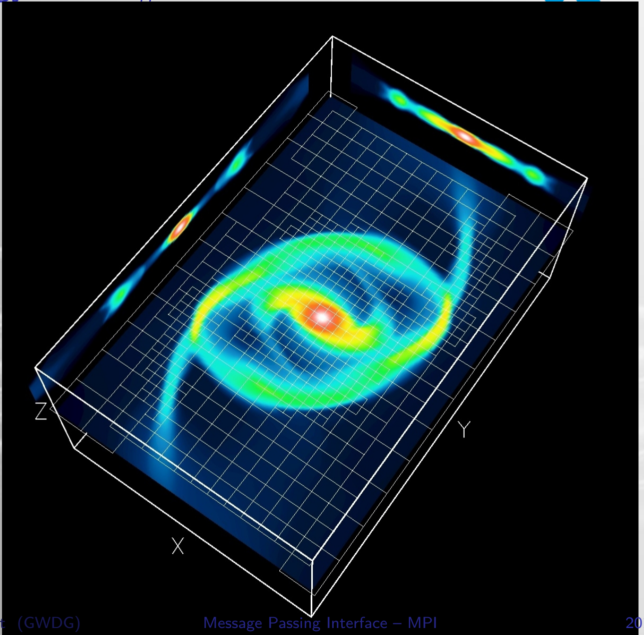
Cosmology



Cosmology AMR



Cosmology AMR #2



More Strategies

http://nf.nci.org.au/training/MPIAppOpt/slides/allslides.html

Conclusion

- Message Passing for Distributed-memory Systems
- MPI as default API for data distribution and synchronization
  - makes hard things possible
- Many possibilities to parallelize problems
  - Many Particles
  - Grids (with or without Refinement)
  - Particle In Cell
  - ...

Literature

- https://computing.llnl.gov/tutorials/mpi/
- http://www.open-mpi.de/
- http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml
- Strategies for debugging and parallelization http://nf.nci.org.au/training/MPIAppOpt/slides/allslides.html

Questions