An Introduction to MPI with an Application to DGSEM Code

James Custer

Department of Mathematics
Florida State University

04.09.2012

Flynn's Taxonomy

- SISD — **S**ingle **I**nstruction, **S**ingle **D**ata
  - Single processor system
- SIMD — **S**ingle **I**nstruction, **M**ultiple **D**ata
  - GPU
- MISD — **M**ultiple **I**nstruction, **S**ingle **D**ata
  - Not common, used for fault tolerant systems
- MIMD — **M**ultiple **I**nstruction, **M**ultiple **D**ata
  - Most common parallel computing model

The Usual Strategies

- SPMD — **S**ingle **P**rogram, **M**ultiple **D**ata
  - Most common parallel executing model
- MPMD — **M**ultiple **P**rogram, **M**ultiple **D**ata
  - Master/Worker model
- Serial programming
- One process that spawns multiple threads (OpenMP)
- Multiple parallel processes that are single-threaded (SPMD or MPMD)
- Hybrid, multiple parallel processes that use multiple threads

Parallel Programming

While developing a parallel program, one should keep in mind:

- load balancing
- communication
- synchronization

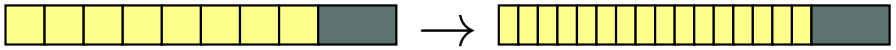
Effective parallel programming requires knowledge of

- Algorithms
- Architecture
- Languages

Scalability

Parallel portion ■ Serial portion ■

Amdahl's Law (strong scaling)



$$S(N) = \frac{t_s}{t_p} = \frac{1}{(1-P) + \frac{P}{N}}$$

where S is speedup, P is the proportion of your program that can be parallelized, and N is the number of processors.

$$P_{\text{estimated}} = \frac{\frac{1}{S} - 1}{\frac{1}{N} - 1}$$

Gustafson's Law (weak scaling)

$$S(N) = N - (1-P)(N-1)$$

Note: these neglect other limiting factors, such as: memory, network, and disk latencies

Messaging Passing Interface

Message Passing Interface (MPI) MPI is a *specification*

- *not* a language
- *not* a compiler specification
- *not* a specific implementation

MPI has implementations in:

- C
- C++
- Fortran
- Python, Perl, R, Ruby, Java, OCaml

How to Think in MPI

How to think while developing a program using MPI: Every process is executing the same program at the same time.

MPI Version of Hello, World!


```
#include <iostream>
#include <boost/mpi.hpp>

int main(int argc, char* argv[])
{
  boost::mpi::environment env(argc, argv);
  boost::mpi::communicator world;

  std::cout << "Hello from " << world.rank() << "
      of " << world.size() << "!" << std::endl;
}
```

hello-mpi.cpp

MPI Version of Hello, World!


```
PROGRAM main
  IMPLICIT NONE
!
  include 'mpif.h'
!
  INTEGER :: i
  INTEGER :: rank, size
  INTEGER :: ierr
  INTEGER :: status(MPI_STATUS_SIZE)
!
  CALL MPI_INIT( ierr )
!
  CALL MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
  CALL MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
  PRINT *, 'Hello from ', rank, ' of ', size, '!'
!
  CALL MPI_FINALIZE
!
ENDPROGRAM main
```

hello-mpi.f90

Point-to-Point Communication

Point-to-point communication involves only two different MPI tasks: one task sends a message, while another task receives. Each message contains a:

- source process
- target process
- tag
- payload containing arbitrary data.

Blocking vs. Non-Blocking

Blocking:

- Returns only after data has arrived and is ready for use by the program.

Non-Blocking:

- Returns immediately
- You should not modify your buffer during this time!
- Use wait routines to determine when it is safe to do so.
- Used to overlap computation with communication.

Blocking send/recv


```
send(dest, tag, value)

MPI_SEND(value, count, datatype, dest, tag, comm,
    ierr)
```

```
recv(source, tag, value)

MPI_RECV(value, count, datatype, source, tag,
    comm, status, ierr)
```

Blocking send/recv example


```
string message;
if (world.rank() == 0)
{
  send(1, 0, string("Hello 1, from 0"));
  recv(1, 1, message);
}
else
{
  send(0, 1, string("Hello 0, from 1"));
  recv(0, 0, message);
}
cout << message << endl;
```

Blocking send/recv example


```
string message;
if (world.rank() == 0)
{
  send(1, 0, string("Hello 1, from 0"));
  recv(1, 1, message);
}
else
{
  send(0, 1, string("Hello 0, from 1"));
  recv(0, 0, message);
}
cout << message << endl;
```

blocking-broken.cpp

Blocking send/recv example


```
string message;
if (world.rank() == 0)
{
  send(1, 0, string("Hello 1, from 0"));
  recv(1, 1, message);
}
else
{
  send(0, 1, string("Hello 0, from 1"));
  recv(0, 0, message);
}
cout << message << endl;
```

blocking-broken.cpp

Output

See blocking-deadlock.cpp for another example.

Blocking send/recv example fixed


```
string message;
if (world.rank() == 0)
{
  send(1, 0, string("Hello 1, from 0"));
  recv(1, 1, message);
}
else
{
  recv(0, 0, message);
  send(0, 1, string("Hello 0, from 1"));
}
cout << message << endl;
```

blocking-fixed.cpp

Blocking send/recv example fixed


```
string message;
if (world.rank() == 0)
{
  send(1, 0, string("Hello 1, from 0"));
  recv(1, 1, message);
}
else
{
  recv(0, 0, message);
  send(0, 1, string("Hello 0, from 1"));
}
cout << message << endl;
```

blocking-fixed.cpp

```
Hello 1, from 0
Hello 0, from 1
```

Output

Non-blocking send/recv


```
isend(dest, tag, value)

MPI_ISEND(value, count, datatype, dest, tag, comm,
    ierr)
```

```
irecv(source, tag, value)

MPI_IRECV(value, count, datatype, source, tag,
    comm, status, ierr)
```

Non-blocking send/recv example


```
vector<request> requests;
string message;
if (world.rank() == 0)
{
  requests.push_back( world.isend(1, 0,
      string("Hello 1, from 0")) );
  requests.push_back( world.irecv(1, 1, message) );
}
else {
  requests.push_back( world.isend(0, 1,
      string("Hello 0, from 1")) );
  requests.push_back( world.irecv(0, 0, message) );
}
wait_all(requests.begin(), requests.end());
for (unsigned i = 0; i < world.size(); ++i)
{
  world.barrier();
  if (world.rank() == i)
    cout << message << endl;
}
```

non-blocking.cpp

Non-blocking send/recv gotchas

Do **not** read from, nor modify, the buffers until you've waited!
If you do so, you've created a race condition.

Collective Operations

The most common collective operations:

- Broadcast
- Gather
- Scatter
- Reduce

Broadcast

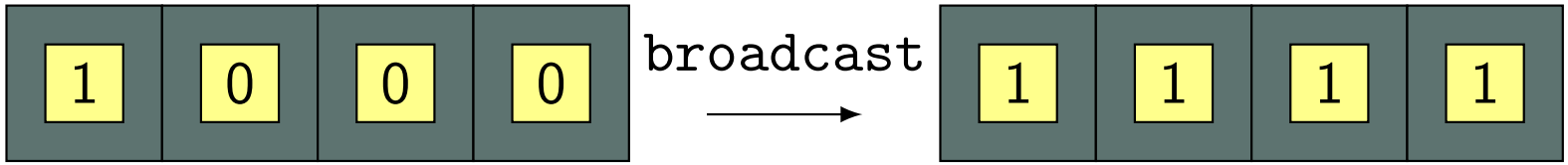

```
broadcast(comm, value, root)
```

```
int root_process = 0;
int value = 0;
if (world.rank() == root_process)
  value = 1;

broadcast(world, value, root_process);
```

broadcast.cpp

```
mpirun -np 4 ./broadcast
```



Gather

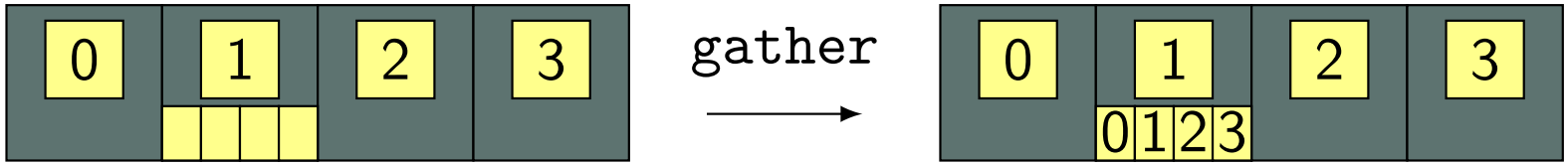

```
gather(comm, in_value, out_values, root)
```

```
int root_process = 1;
int value = world.rank();
vector<int> all_values;

gather(world, value, all_values, root_process);
```

gather.cpp

```
mpirun -np 4 ./gather
```



Gather

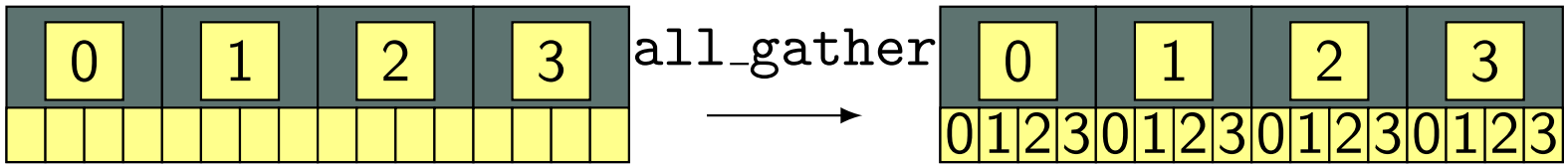

```
all_gather(comm, in_value, out_values)
```

```
int value = world.rank();
vector<int> all_values;

all_gather(world, value, all_values);
```

all_gather.cpp

```
mpirun -np 4 ./all_gather
```



Scatter

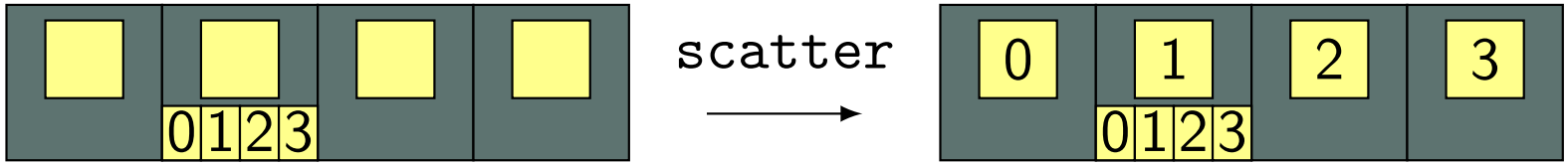

```
scatter(comm, in_values, out_value, root)
```

```
int root_process = 1;
int value;
vector<int> values;
if (world.rank() == root_process)
{
  values.resize(world.size());
  for (int i = 0; i < world.size(); ++i)
    values[i] = i;
}

scatter(world, values, value, root_process);
```

scatter.cpp

```
mpirun -np 4 ./scatter
```



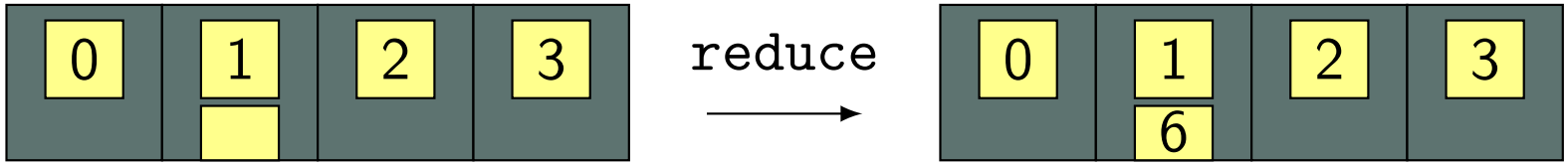
Reduce


```
reduce(comm, in_value, out_value, op, root)
```

```
int root_process = 1;
int value = world.rank();
if (world.rank() == root_process)
{
  int sum;
  reduce(world, value, sum, plus<int>(),
      root_process);
  cout << "sum: " << sum << endl;
}
else
  reduce(world, value, plus<int>(), root_process);
```

reduce.cpp

```
mpirun -np 4 ./reduce
```



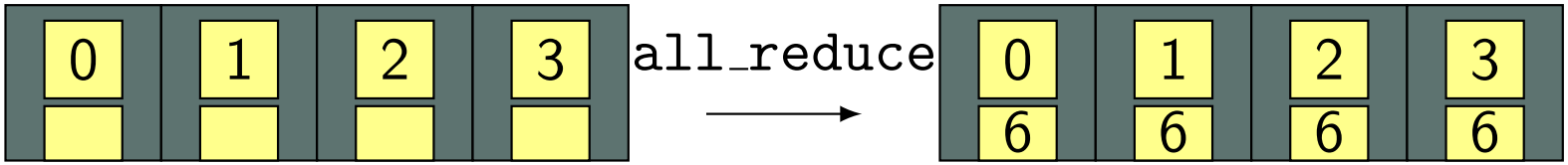
Reduce


```
all_reduce(comm, in_value, out_value, op)
```

```
int value = world.rank();
int sum;
all_reduce(world, value, sum, plus<int>());
cout << "rank: " << world.rank() << " sum: " <<
    sum << endl;
```

all_reduce.cpp

```
mpirun -np 4 ./all_reduce
```

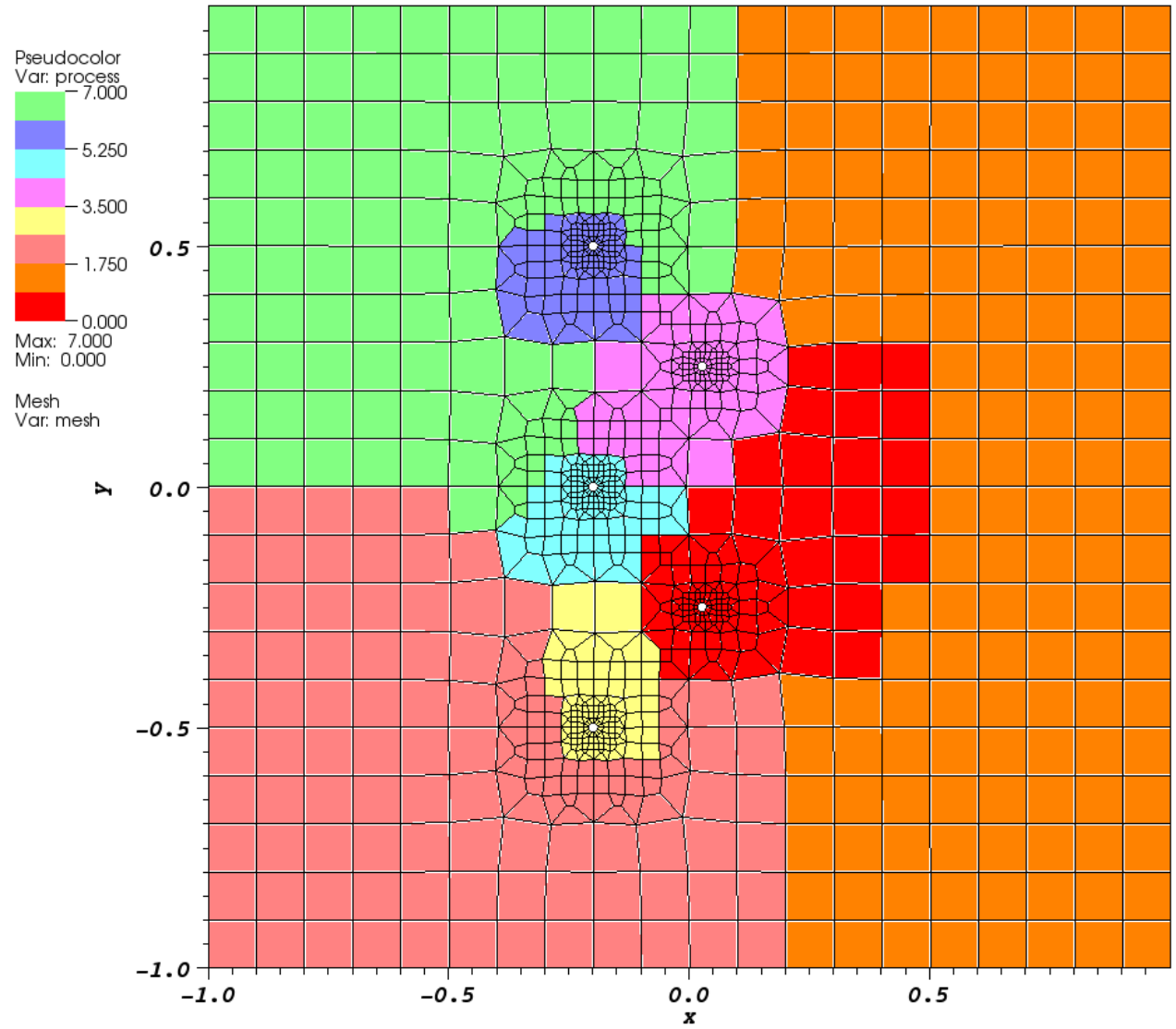


Load Balancing

Two common programs for partitioning finite element meshes:

- METIS
  - Simpler interfaces
  - Cannot freely distribute with your code unless permission is obtained
- SCOTCH
  - More complicated interfaces
  - There are METIS-style interfaces (I haven't tried them)
  - Better licensing

Load Balancing



Load Balancing

Processor	# Elements
0	144
1	141
2	144
3	148
4	140
5	148
6	145
7	144
total	1154

- Number of edge cuts: 121
- Number of edges: 2384
- Ratio: 5%

Serial Version

- Prolong to faces
- Compute edge fluxes
- Compute local time derivative

Mesh Construction

- Partition the mesh (METIS/SCOTCH)
- Separate storage into:
  - elements along partition
  - elements in interior
  - edges along partition
  - edges in interior
- Create maps that translate global element ids to local element ids

Parallel Version

- Prolong to faces for each element along partition boundary
- Send JQb to matching edges along partition boundary
  - `isend/request`
- Prolong to faces for each element on interior of partition
- Compute Edge fluxes for edges on interior of partition
- Receive edge fluxes from edges along partition boundary
  - `irecv/request`
- `wait_all` on the requests
- Compute edge fluxes for edges on partition boundary
- Compute the time derivative for all elements

Results