# Parallel & Concurrent Programming:
# Message-Passing

Emery Berger

CMPSCI 691W
Spring 2006

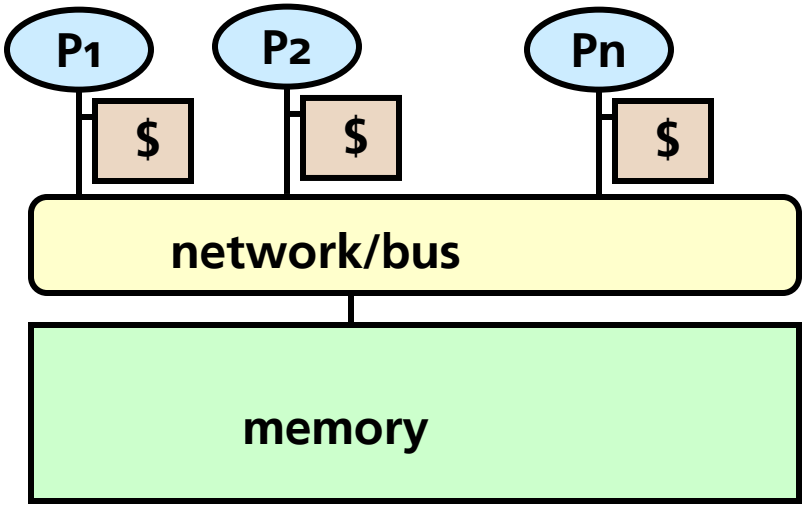
# Outline

- Today:
  - Distributed parallel programming via **message-passing**
  - **MPI** – **library** approach

*some material adapted from slides by Kathy Yelick*

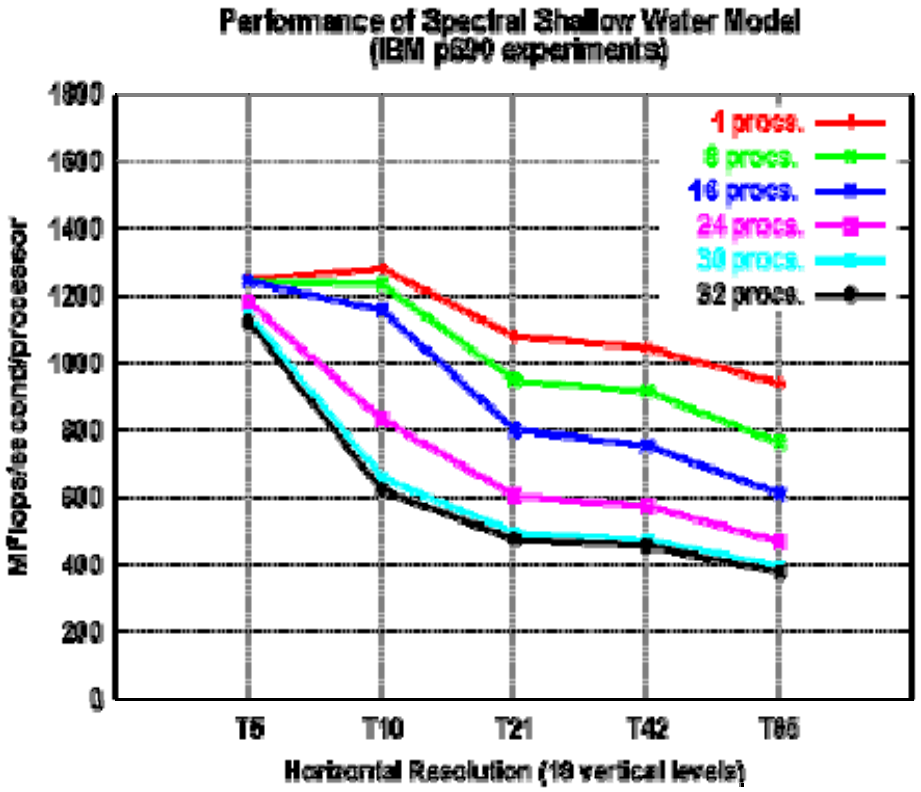
# Why Distribute?

- SMP: easy to program but limited
  - Typically < 32 processors
  - **Bus becomes bottleneck** when processors not operating locally




# Scaling Limits

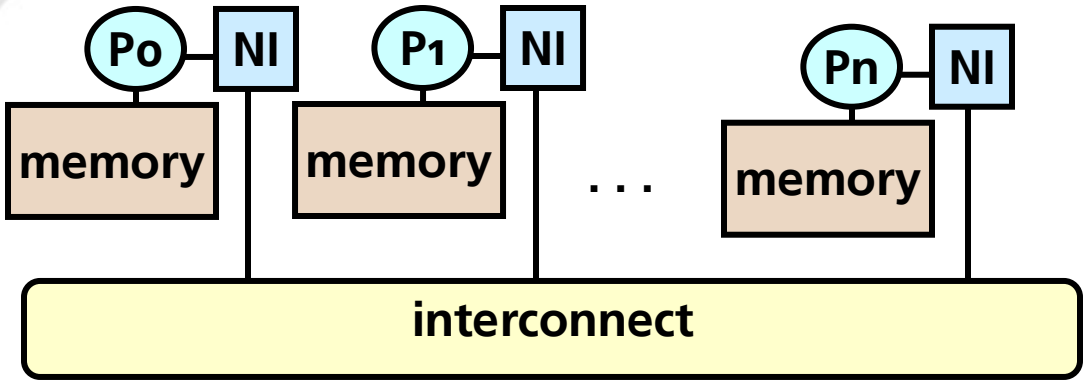
- **Kernel** used in atmospheric models
  - 99% floating point ops; multiplies/adds
  - Sweeps through memory with little reuse
  - One "copy" of code running independently on varying numbers of procs




*From Pat Worley, ORNL*

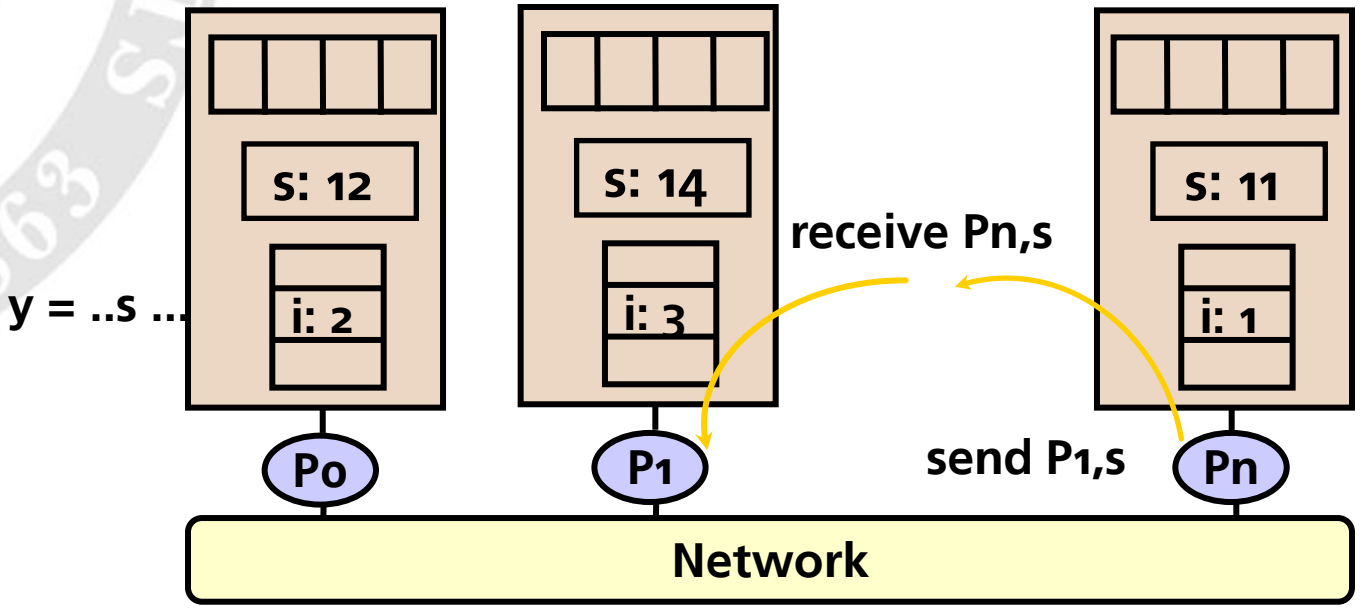
# Distributed Memory

- Distributed memory machines: **local memory** but **no global memory**
  - Individual **nodes** often SMPs
  - **Network interface** for all interprocessor communication




# Message Passing

- Program: # independent communicating processes
  - Thread + **local address space only**
  - Shared data: **partitioned**
- Communicate by **send & receive** events



# Message Passing

- Pros: **efficient**
  - Makes data sharing explicit
  - Can communicate only what is strictly necessary for computation
    - No coherence protocols, etc.
- Cons: **difficult**
  - Requires **manual partitioning**
  - Unnatural model
  - Deadlock-prone
  - Not portable (previously)

# Portability

- *Bad old days:* each vendor had own message-passing solution = no portability
  - Tied to different **network topologies**
    - **Bus, star, hypercube**, …
  - Vastly different **platforms**
    - **SMP** boxes
    - **Beowulf** clusters
    - Supercomputers
- **Goal: write once, run everywhere**

# Message Passing Interface

- **Library** approach to message-passing
- Supports most common architectural abstractions
  - **Vendors** supply optimized versions
  - ⇒ programs run on different machine, but with (somewhat) different performance
- **Bindings** for popular languages
  - Especially Fortran, C
  - Also C++, Java

# MPI execution model

- Spawns multiple copies of **same** program (**SPMD**)
  - Each = different "process" (different local memory)
- Can act differently by determining which processor "self" corresponds to

# An Example

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

int main(int argc, char * argv[]) {
  int rank, size;
  MPI_Init(&argc, &argv );
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from process %d of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

**% mpirun –np 10 exampleProgram**

# An Example

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

int main(int argc, char * ar
  int rank, size;
  MPI_Init(&argc, &argv );
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from process %d of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

initializes MPI (passes arguments in)

```
% mpirun –np 10 exampleProgram
```

# An Example

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

int main(int argc, char * argv[]) {
  int rank, size;
  MPI_Init(&argc, &argv );
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from process %d of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

returns # of processors in "world"

**% mpirun –np 10 exampleProgram**

# An Example

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

int main(int argc, char * argv[]) {
  int rank, size;
  MPI_Init(&argc, &argv );
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from process %d of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

which processor am I?

**% mpirun –np 10 exampleProgram**

# An Example

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

int main(int argc, char * argv[]) {
  int rank, size;
  MPI_Init(&argc, &argv );
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_C                ank);
  printf("Hello world                %d of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

we're done sending messages

**% mpirun –np 10 exampleProgram**

# Message Passing

- Messages can be sent directly to another processor
  - **MPI_Send, MPI_Recv**
- Or to all processors
  - **MPI_Bcast** (does send **or** receive)

# Broadcast

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

int main(int argc, char * argv[]) {
  int rank, value;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  do {
    if (rank == 0)
      scanf( "%d", &value );
    MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf( "Process %d got %d\n", rank, value );
  } while (value >= 0);
  MPI_Finalize( );
  return 0;
}
```

- Repeatedly broadcast input (one integer) to all

# Broadcast

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

int main(int argc, char * argv[]) {
  int rank, value;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  do {
    if (rank == 0)
      scanf( "%d", &value );
    MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf( "Process %d got %d\n", rank, value );
  } while (value >= 0);
  MPI_Finalize( );
  return 0;
}
```

send or receive value

- Repeatedly broadcast input (one integer) to all

# Broadcast

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

int main(int argc, char * argv[]) {
  int rank, value;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  do {
    if (rank == 0)
      scanf( "%d", &value );
    MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf( "Process %d got %d\n", rank, value );
  } while (value >= 0);
  MPI_Finalize( );
  return 0;
}
```

how many to send/receive?

- Repeatedly broadcast input (one integer) to all

# Broadcast

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

int main(int argc, char * argv[]) {
  int rank, value;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  do {
    if (rank == 0)
      scanf( "%d", &value );
    MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf( "Process %d got %d\n", rank, value );
  } while (value >= 0);
  MPI_Finalize( );
  return 0;
}
```

what's the datatype?

- Repeatedly broadcast input (one integer) to all

# Broadcast

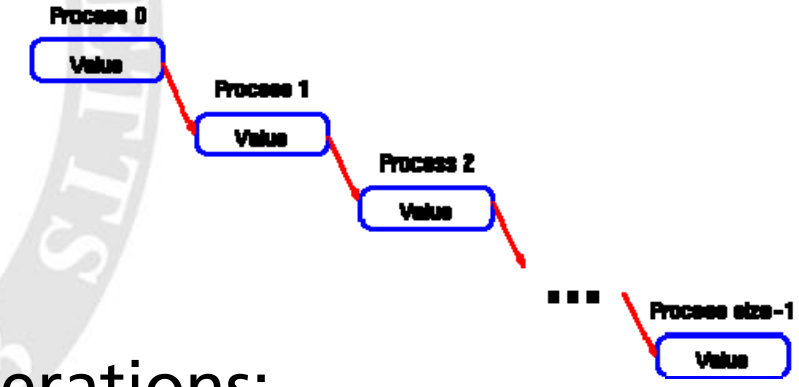
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char * argv[]) {
  int rank, value;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  do {
    if (rank == 0)
      scanf( "%d", &value );
    MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf( "Process %d got %d\n", rank, value );
  } while (value >= 0);
  MPI_Finalize( );
  return 0;
}
```

who's "root" for broadcast?

- Repeatedly broadcast input (one integer) to all

# Send/Recv Example

- Send data from process o to all
- "Pass it along" communication




- Operations:
  - **MPI_Send** (data *, count, **MPI_INT**, dest, o, **MPI_COMM_WORLD** );
  - **MPI_Recv** (data *, count, **MPI_INT**, source, o, **MPI_COMM_WORLD** );

# Send & Receive

```
int main(int argc, char * argv[]) {
  int rank, value, size;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  do {
    if (rank == 0) {
      scanf( "%d", &value );
      MPI_Send(&value, 1, MPI_INT, rank + 1,
               0, MPI_COMM_WORLD );
    } else {
      MPI_Recv(&value, 1, MPI_INT, rank - 1,
               0, MPI_COMM_WORLD, &status );
      if (rank < size - 1)
        MPI_Send( &value, 1, MPI_INT, rank + 1,
                  0, MPI_COMM_WORLD );
    }
    printf("Process %d got %d\n", rank, value);
  } while (value >= 0);
  MPI_Finalize();
  return 0;
}
```

- Send integer input in a ring

# Send & Receive

```
int main(int argc, char * argv[]) {
  int rank, value, size;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  do {
    if (rank == 0) {
      scanf( "%d", &value );
      MPI_Send(&value, 1, MPI_INT, rank + 1,
               0, MPI_COMM_WORLD );
    } else {
      MPI_Recv(&value, 1, MPI_INT, rank - 1,
               0, MPI_COMM_WORLD, &status );
      if (rank < size - 1)
        MPI_Send( &value, 1, MPI_INT, rank + 1,
                  0, MPI_COMM_WORLD );
    }
    printf("Process %d got %d\n", rank, value);
  } while (value >= 0);
  MPI_Finalize();
  return 0;
}
```

send destination?

- Send integer input in a ring

# Send & Receive

```
int main(int argc, char * argv[]) {
  int rank, value, size;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  do {
    if (rank == 0) {
      scanf( "%d", &value );
      MPI_Send(&value, 1, MPI_INT, rank + 1,
               0, MPI_COMM_WORLD );
    } else {
      MPI_Recv(&value, 1, MPI_INT, rank - 1,
               0, MPI_COMM_WORLD, &status );
      if (rank < size - 1)
        MPI_Send( &value, 1, MPI_INT, rank + 1,
                  0, MPI_COMM_WORLD );
    }
    printf("Process %d got %d\n", rank, value);
  } while (value >= 0);
  MPI_Finalize();
  return 0;
}
```

receive from?

- Send integer input in a ring

# Send & Receive

```
int main(int argc, char * argv[]) {
  int rank, value, size;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  do {
    if (rank == 0) {
      scanf( "%d", &value );
      MPI_Send(&value, 1, MPI_INT, rank + 1,
               0, MPI_COMM_WORLD );
    } else {
      MPI_Recv(&value, 1, MPI_INT, rank - 1,
               0, MPI_COMM_WORLD, &status );
      if (rank < size - 1)
        MPI_Send( &value, 1, MPI_INT, rank + 1,
                  0, MPI_COMM_WORLD );
    }
    printf("Process %d got %d\n", rank, value);
  } while (value >= 0);
  MPI_Finalize();
  return 0;
}
```

message tag

message tag

message tag

# Communication Flavors

- In addition to basic, **blocking** messages & point-to-point:
  - **Non-blocking**
    - **MPI_ISend, MPI_IRecv**
    - **MPI_Wait, MPI_Waitall, MPI_Test**
  - **Buffered**

# The End

- Next time:
  - Collective communication
- Due soon (after 1st bi-weekly mtg):
  - **Project report**: describe your proposed work and implementation plan, including division of responsibilities if appropriate, and timeline with milestones.