Supercomputing and Science

An Introduction to High Performance Computing

Part VI: Distributed Parallelism

Henry Neeman, Director
OU Supercomputing Center
for Education & Research

Outline

- The “Huts on Islands” Analogy
- Distributed Parallelism
- MPI

The Huts on Islands Analogy

An Island Hut

- Imagine Henry is on an island in a little hut.
- Inside the hut is a desk.
- On the desk is a phone, a pencil, a calculator, a piece of paper with numbers, and a piece of paper with instructions.

Instructions

The instructions are split into two kinds:

- Arithmetic/Logical: e.g.,
  - Add the 27th number to the 239th number
  - Determine whether two Boolean flags are both true
- Communication: e.g.,
  - dial 555-0127 and leave a voicemail containing the 962nd number
  - call your voicemail box and collect a voicemail from 555-0063

Is There Anybody Out There?

If Henry's in a hut on an island, he isn't specifically aware of anyone else.

Especially, he doesn't know whether anyone else is working on the same problem as he is, and he doesn't know who's at the other end of the phone line.

All he knows is what to do with the voicemails he gets, and what phone numbers to send voicemails to.

Someone Might Be Out There

Now suppose that Chenmei is on another island somewhere, in the same kind of hut, with the same kind of equipment.

Suppose that she has the same list of instructions as Henry, but a different set of numbers.

Like Henry, she doesn’t know whether there’s anyone else working on her problem.

Even More People Out There

Now suppose that Noel and Suresh are also in huts on islands.

Suppose that each of the four has the exact same list of instructions, but different lists of numbers.

And suppose that the phone numbers that people call are each others'. That is, Henry instructions have him call Chenmei's, Noel's and Suresh's numbers.

Then they might all be working together on the same problem.

All Data Are Private

Notice that Henry can’t see Chenmei’s or Noel’s or Suresh’s numbers, nor can they see his or each other’s.

Thus, everyone’s numbers are private: there’s no way for anyone to share numbers, except by leaving them in voicemails.

Long Distance Calls: 2 Costs

When you make a long distance phone call, you typically have to pay two costs:

- Connection charge: the fixed cost of connecting your phone to someone else's, even if you're only connected for a second
- Per-minute charge: the cost per minute of talking, once you're connected

If the connection charge is large, then you want to make as few calls as possible.

Distributed Parallelism

Like Huts on Islands

Distributed parallelism is very much like the Huts on Islands analogy:

- Processors are independent of each other.
- All data are private.
- Processes communicate by passing messages (like voicemails).
- The cost of passing a message is split into the latency (connection time) and the bandwidth (time per byte).

Multiprocessing Jargon

- Threads: execution sequences that share a memory area
- Processes: execution sequences with their own independent, private memory areas

As a general rule, Shared Memory Parallelism is concerned with threads, and Distributed Parallelism is concerned with processes.

Load Balancing

Suppose you have a distributed parallel code, but one processor does 90% of the work, and all the other processors share 10% of the work.

Is it a big win to run on 1000 processors?

Now suppose that each processor gets exactly $1/N_p$ of the work, where N_p is the number of processors.

Now is it a big win to run on 1000 processors?

Load Balancing Is Good

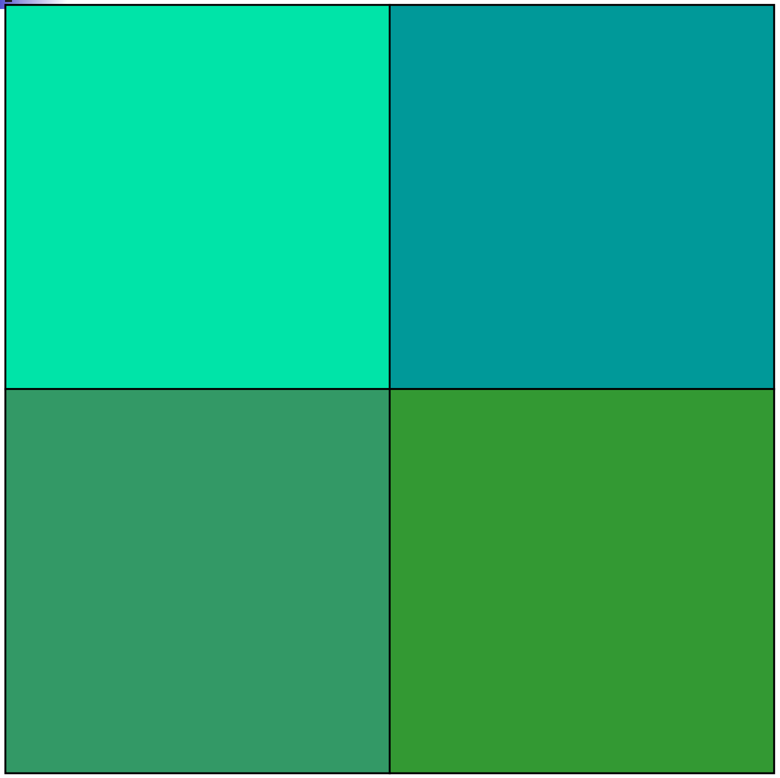
When every processor gets the same amount of work, the job is load balanced.

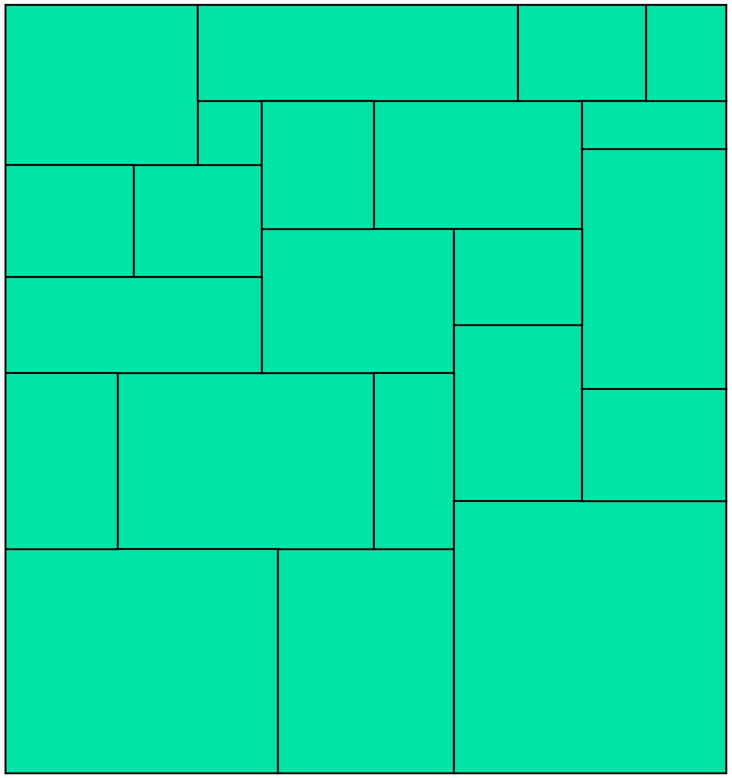
We like load balancing, because it means that our speedup can potentially be linear.

For some codes, figuring out how to balance the load is trivial (e.g., breaking a big unchanging array into sub-arrays).

For others, load balancing is very tricky (e.g., a dynamically evolving collection of arbitrarily many blocks of arbitrary size).

Load Balancing Examples


Easy


Hard

MPI: The Message-Passing Interface

Most of this discussion is from [1] and [2].

What Is MPI?

- The Message-Passing Interface (MPI) is a standard for expressing distributed parallelism via message passing.
- MPI consists of a header file, a library of routines and a runtime environment.
- When you compile a program that has MPI calls in it, your compiler links to a local implementation of MPI, and then you get parallelism; if the MPI library isn't available, then the compile will fail.
- MPI can be used in Fortran, C and C++.

MPI Calls

MPI calls in Fortran look like this:

```
CALL MPI_Funcname(…, errcode)
```

In C, MPI calls look like:

```
errval = MPI_Funcname(…)
```

In C++, MPI calls look like:

```
errval = MPI::Funcname(…)
```

Notice that `errval` is returned by the MPI routine `MPI_Funcname`, with a value of `MPI_SUCCESS` indicating that `MPI_Funcname` has worked correctly.

MPI Is an API

- MPI is actually just an <u>Application Programming Interface</u> (API).
- An API specifies what a call to each routine should look like, and how each routine should behave.
- An API does not specify how each routine should be implemented, and sometimes is intentionally vague about certain aspects of a routine’s behavior.
- Each platform has its own MPI implementation.

Example MPI Routines

- `MPI_Init` starts up the MPI runtime environment at the beginning of a run.
- `MPI_Finalize` shuts down the MPI runtime environment at the end of a run.
- `MPI_Comm_size` gets the number of processors in a run, N_p (typically called just after `MPI_Init`).
- `MPI_Comm_rank` gets the processor ID that the current process uses, which is between 0 and N_p-1 inclusive (typically called just after `MPI_Init`).

More Example MPI Routines

- `MPI_Send` sends a message from the current processor to some other processor (the <u>destination</u>).
- `MPI_Recv` receives a message on the current processor from some other processor (the <u>source</u>).
- `MPI_Bcast` broadcasts a message from one processor to all of the others.
- `MPI_Reduce` performs a reduction (e.g., sum) of a variable on all processors.

MPI Program Structure (F90)

```
PROGRAM my_mpi_program
  USE mpi
  IMPLICIT NONE
  INTEGER :: my_rank, num_procs, mpi_error_code
  [other declarations]
  CALL MPI_Init(mpi_error_code)      !! Start up MPI
  CALL MPI_Comm_Rank(my_rank,   mpi_error_code)
  CALL MPI_Comm_size(num_procs, mpi_error_code)
  [actual work goes here]
  CALL MPI_Finalize(mpi_error_code) !! Shut down MPI
END PROGRAM my_mpi_program
```

Note that MPI uses the term “rank” to indicate process identifier.

MPI Program Structure (in C)

```
#include <stdio.h>
[other header includes go here]
#include "mpi.h"

int main (int argc, char* argv[])
{ /* main */
  int my_rank, num_procs, mpi_error;
  [other declarations go here]
  mpi_error = MPI_Init(&argc, &argv); /* Start up MPI  */
  mpi_error = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  [actual work goes here]
  mpi_error = MPI_Finalize();         /* Shut down MPI */
} /* main */
```

Example: Hello World

1. Start the MPI system.
2. Get the rank and number of processors.
3. If you're not the master process:
    1. Create a "hello world" string.
    2. Send it to the master process.
4. If you are the master process:
    1. For each of the other processes:
        1. Receive its "hello world" string.
        2. Print its "hello world" string.
5. Shut down the MPI system.

hello_world_mpi.c

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

int main (int argc, char* argv[])
{ /* main */
  const int  maximum_message_length = 100;
  const int  master_rank            =   0;
  char       message[maximum_message_length+1];
  MPI_Status status;      /* Info about receive status  */
  int        my_rank;     /* This process ID            */
  int        num_procs;   /* Number of processes in run */
  int        source;      /* Process ID to receive from */
  int        destination; /* Process ID to send to      */
  int        tag = 0;     /* Message ID                 */
  int        mpi_error;   /* Error code for MPI calls   */
  [work goes here]
} /* main */
```

Hello World Startup/Shut Down

```
[header file includes]
int main (int argc, char* argv[])
{ /* main */
  [declarations]
  mpi_error = MPI_Init(&argc, &argv);
  mpi_error = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  if (my_rank != master_rank) {
    [work of each non-master process]
  } /* if (my_rank != master_rank) */
  else {
    [work of master process]
  } /* if (my_rank != master_rank)…else */
  mpi_error = MPI_Finalize();
} /* main */
```

Hello World Non-master’s Work

```
[header file includes]
int main (int argc, char* argv[])
{ /* main */
  [declarations]
  [MPI startup (MPI_Init etc)]
  if (my_rank != master_rank) {
    sprintf(message, "Greetings from process #%d!",
        my_rank);
    destination = master_rank;
    mpi_error =
      MPI_Send(message, strlen(message) + 1, MPI_CHAR,
        destination, tag, MPI_COMM_WORLD);
  } /* if (my_rank != master_rank) */
  else {
    [work of master process]
  } /* if (my_rank != master_rank)…else */
  mpi_error = MPI_Finalize();
} /* main */
```

Hello World Master's Work

```
[header file includes]
int main (int argc, char* argv[])
{ /* main */
  [declarations, MPI startup]
  if (my_rank != master_rank) {
    [work of each non-master process]
  } /* if (my_rank != master_rank) */
  else {
    for (source = 0; source < num_procs; source++) {
      if (source != master_rank) {
        mpi_error =
          MPI_Recv(message, maximum_message_length + 1,
            MPI_CHAR, source, tag, MPI_COMM_WORLD,
            &status);
        fprintf(stderr, "%s\n", message);
      } /* if (source != master_rank) */
    } /* for source */
  } /* if (my_rank != master_rank)…else */
  mpi_error = MPI_Finalize();
} /* main */
```

Compiling and Running

```
% cc  -o  hello_world_mpi  hello_world_mpi.c  -lmpi
% mpirun  -np  1  hello_world_mpi
% mpirun  -np  2  hello_world_mpi
Greetings from process #1!
% mpirun  -np  3  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
% mpirun  -np  4  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
Greetings from process #3!
```

Note: the compile command and the run command vary from platform to platform.

Why is Rank #0 the Master?

```
const int master_rank = 0;
```

- By convention, the master process has rank (process ID) #0. Why?
- A run must use at least one process but can use multiple processes.
- Process ranks are 0 through N_p-1, $N_p \geq 1$.
- Therefore, every MPI run has a process with rank #0.
- Note: every MPI run also has a process with rank N_p-1, so you could use N_p-1 as the master instead of 0 … but no one does.

Why "Rank?"

Why does MPI use the term rank to refer to process ID?

In general, a process has an identifier that is assigned by the operating system (e.g., Unix), and that is unrelated to MPI:

```
% ps
       PID TTY        TIME CMD
  52170812 ttyq57   0:01 tcsh
```

Also, each processor has an identifier, but an MPI run that uses fewer than all processors will use an arbitrary subset.

The rank of an MPI process is neither of these.

Compiling and Running

Recall:

```
% cc  -o  hello_world_mpi  hello_world_mpi.c  -lmpi
% mpirun  -np  1  hello_world_mpi
% mpirun  -np  2  hello_world_mpi
Greetings from process #1!
% mpirun  -np  3  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
% mpirun  -np  4  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
Greetings from process #3!
```

Deterministic Operation?

```
% mpirun  -np  4  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
Greetings from process #3!
```

The order in which the greetings are printed is deterministic. Why?

```
for (source = 0; source < num_procs; source++) {
  if (source != master_rank) {
    mpi_error =
      MPI_Recv(message, maximum_message_length + 1,
        MPI_CHAR, source, tag, MPI_COMM_WORLD,
        &status);
    fprintf(stderr, "%s\n", message);
  } /* if (source != master_rank) */
} /* for source */
```

This loop ignores the receive order.

Message = Envelope+Contents

```
MPI_Send(message, strlen(message) + 1,
  MPI_CHAR, destination, tag, MPI_COMM_WORLD);
```

When MPI sends a message, it doesn’t just send the contents; it also sends an “envelope” describing the contents:

- Size (number of elements of data type)
- Data type
- Rank of sending process (source)
- Rank of process to receive (destination)
- Tag (message ID)
- Communicator (e.g., `MPI_COMM_WORLD`)

MPI Data Types

MPI	C/C++	Fortran
MPI_CHAR	**char**	**CHARACTER**
MPI_INT	**int**	**INTEGER**
MPI_FLOAT	**float**	**REAL**
MPI_DOUBLE	**double**	**DOUBLE PRECISION**

MPI supports several other data types, but most are variations of these, and probably these are all you'll use.

Message Tags

```
for (source = 0; source < num_procs; source++) {
  if (source != master_rank) {
    mpi_error =
      MPI_Recv(message, maximum_message_length + 1,
        MPI_CHAR, source, tag, MPI_COMM_WORLD,
        &status);
    fprintf(stderr, "%s\n", message);
  } /* if (source != master_rank) */
} /* for source */
```

The greetings are printed in deterministic order not because messages are sent and received in order, but because each has a tag (message identifier), and **`MPI_Recv`** asks for a specific message (by tag) from a specific source (by rank).

Communicators

An MPI communicator is a collection of processes that can send messages to each other.

`MPI_COMM_WORLD` is the default communicator; it contains all of the processes. It's probably the only one you'll need.

Some libraries (e.g., PETSc) create special library-only communicators, which can simplify keeping track of message tags.

Broadcasting

What happens if one processor has data that everyone else needs to know?

For example, what if the master processor needs to send an input value to the others?

```
 CALL MPI_Bcast(length, 1, MPI_INTEGER, &
&        source, MPI_COMM_WORLD, error_code)
```

Note that **`MPI_Bcast`** doesn't use a tag, and that the call is the same for both the sender and all of the receivers.

Broadcast Example: Setup

```
PROGRAM broadcast
  USE mpi
  IMPLICIT NONE
  INTEGER,PARAMETER :: master = 0
  INTEGER,PARAMETER :: source = master
  INTEGER,DIMENSION(:),ALLOCATABLE :: array
  INTEGER :: length, memory_status
  INTEGER :: num_procs, my_rank, mpi_error_code

  CALL MPI_Init(mpi_error_code)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank,    &
 &       mpi_error_code)
  CALL MPI_Comm_size(MPI_COMM_WORLD, num_procs, &
 &       mpi_error_code)
  [input]
  [broadcast]
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM broadcast
```

Broadcast Example: Input

```
PROGRAM broadcast
  USE mpi
  IMPLICIT NONE
  INTEGER,PARAMETER :: master = 0
  INTEGER,PARAMETER :: source = master
  INTEGER,DIMENSION(:),ALLOCATABLE :: array
  INTEGER :: length, memory_status
  INTEGER :: num_procs, my_rank, mpi_error_code
```

[MPI startup]

```
  IF (my_rank == master) THEN
    OPEN (UNIT=99,FILE="broadcast_in.txt")
    READ (99,*) length
    CLOSE (UNIT=99)
    ALLOCATE(array(length), STAT=memory_status)
    array(1:length) = 0
  END IF !! (my_rank == master)...ELSE
```

[broadcast]

```
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM broadcast
```

Broadcast Example: Broadcast

```
PROGRAM broadcast
  USE mpi
  IMPLICIT NONE
  INTEGER,PARAMETER :: master = 0
  INTEGER,PARAMETER :: source = master
  [other declarations]

  [MPI startup and input]
  IF (num_procs > 1) THEN
    CALL MPI_Bcast(length, 1, MPI_INTEGER, source, &
 &          MPI_COMM_WORLD, mpi_error_code)
    IF (my_rank /= master) THEN
      ALLOCATE(array(length), STAT=memory_status)
    END IF !! (my_rank /= master)
    CALL MPI_Bcast(array, length, MPI_INTEGER, source, &
            MPI_COMM_WORLD, mpi_error_code)
    WRITE (0,*) my_rank, ": broadcast length = ", length
  END IF !! (num_procs > 1)
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM broadcast
```

Broadcast Compile & Run

```
% f90 -o broadcast broadcast.f90 -lmpi
% mpirun -np 4 broadcast
 0 : broadcast length =  16777216
 1 : broadcast length =  16777216
 2 : broadcast length =  16777216
 3 : broadcast length =  16777216
```

Reductions

A <u>reduction</u> converts an array to a scalar: sum, product, minimum value, maximum value, Boolean AND, Boolean OR, etc.

Reductions are so common, and so important, that MPI has two routines to handle them:

- **`MPI_Reduce`**: sends result to a single specified processor
- **`MPI_Allreduce`**: sends result to all processors

Reduction Example

```
PROGRAM reduce
  USE mpi
  IMPLICIT NONE
  INTEGER,PARAMETER :: master = 0
  INTEGER :: value, value_sum
  INTEGER :: num_procs, my_rank, mpi_error_code

  CALL MPI_Init(mpi_error_code)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank,   mpi_error_code)
  CALL MPI_Comm_size(MPI_COMM_WORLD, num_procs, mpi_error_code)
  value_sum = 0
  value     = my_rank * num_procs
  CALL MPI_Reduce(value, value_sum, 1, MPI_INT, MPI_SUM, &
 &      master, MPI_COMM_WORLD, mpi_error_code)
  WRITE (0,*) my_rank, ": reduce  value_sum = ", value_sum
  CALL MPI_Allreduce(value, value_sum, 1, MPI_INT, MPI_SUM, &
 &      MPI_COMM_WORLD, mpi_error_code)
  WRITE (0,*) my_rank, ": allreduce value_sum = ", value_sum
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM reduce
```

Compiling and Running (SGI)

```
% f90 -o reduce reduce.f90 -lmpi
% mpirun -np 4 reduce
 3 : reduce  value_sum =  0
 1 : reduce  value_sum =  0
 2 : reduce  value_sum =  0
 0 : reduce  value_sum =  24
 0 : allreduce value_sum =  24
 1 : allreduce value_sum =  24
 2 : allreduce value_sum =  24
 3 : allreduce value_sum =  24
```

Why Two Reduction Routines?

MPI has two reduction routines because of the high cost of each communication.

If only one processor needs the result, then it doesn’t make sense to pay the cost of sending the result to all processors.

But if all processors need the result, then it may be cheaper to reduce to all processors than to reduce to a single processor and then broadcast to all.

Example: Monte Carlo

Monte Carlo methods are approximation methods that randomly generate a large number of examples (realizations) of a phenomenon, and then take the average of the examples' properties.

When the realizations' average converges (i.e., doesn't change substantially if new realizations are generated), then the Monte Carlo simulation stops.

Monte Carlo simulations typically are embarassingly parallel.

Serial Monte Carlo

Suppose you have an existing serial Monte Carlo simulation:

```
PROGRAM monte_carlo
  CALL read_input(…)
  DO WHILE (average_properties_havent_converged(…))
    CALL generate_random_realization(…)
    CALL calculate_properties(…)
    CALL calculate_average(…)
  END DO !! WHILE (average_properties_havent_converged(…))
END PROGRAM monte_carlo
```

How would you parallelize this?

Parallel Monte Carlo

```
PROGRAM monte_carlo
  [MPI startup]
  IF (my_rank == master_rank) THEN
    CALL read_input(…)
  END IF !! (my_rank == master_rank)
  CALL MPI_Bcast(…)
  DO WHILE (average_properties_havent_converged(…))
    CALL generate_random_realization(…)
    CALL calculate_properties(…)
    IF (my_rank == master_rank) THEN
      [collect properties]
    ELSE   !! (my_rank == master_rank)
      [send properties]
    END IF !! (my_rank == master_rank)…ELSE
    CALL calculate_average(…)
  END DO !! WHILE (average_properties_havent_converged(…))
  [MPI shutdown]
END PROGRAM monte_carlo
```

Asynchronous Communication

MPI allows a processor to start a send, then go on and do work while the message is in transit.

This is called asynchronous or non-blocking or immediate communication. (Here, “immediate” refers to the fact that the call to the MPI routine returns immediately rather than waiting for the send to complete.)

Immediate Send

```
CALL MPI_Isend(array, size, MPI_FLOAT,              &
 &     destination, tag, communicator, request, &
 &     mpi_error_code)
```

Likewise:

```
CALL MPI_Irecv(array, size, MPI_FLOAT,         &
 &     source, tag, communicator, request, &
 &     mpi_error_code)
```

This call starts the send/receive, but the send/receive won't be complete until:

```
CALL MPI_Wait(request, status)
```

What's the advantage of this?

Communication Hiding

- In between the call to `MPI_Isend/Irecv` and the call to `MPI_Wait`, both processors can do work!
- If that work takes at least as much time as the communication, then the cost of the communication is effectively zero, since the communication won't affect how much work gets done.
- This is called communication hiding.

Communication Hiding in MC

In our Monte Carlo example, we could use communication hiding by, for instance, sending the properties of each realization asynchronously.

That way, the sending processor can start generating a new realization while the old realization's properties are in transit.

The master processor can collect the other processors' data when it's done with its realization.

Rule of Thumb for Hiding

When you want to hide communication:

- as soon as you calculate the data, send it;
- don't receive it until you need it.

That way, the communication has the maximal amount of time to happen in <u>background</u> (behind the scenes).

Next Time

Part VII:
Grab Bag:
I/O, Visualization,
Grid Computing, etc

References

[1] P.S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, 1997.

[2] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. MIT Press, 1999.