Message Passing between Processes

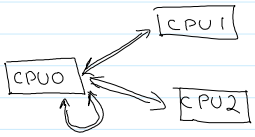
January 14, 2014 9:43 PM

Passing Data

We will look at 2 different "modes":

i) One-to-all (broadcast model)

ii) All-to-one (reverse of broadcast, a "reduce")



We will use the broadcast model in our 1st example.

eg. $\ln 2 = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{1}{n} = \sum_{n=0}^{N} (-1)^n \frac{1}{n+1}$ ← assume "big"

shift sum to start with n=0

divide up sum amongst processors

$$= \sum_{nid=0}^{Nprocs-1} \left(\sum_{c=nid, Nprocs}^{N} (-1)^c \frac{1}{c+1} \right)$$

increment for sum

So cpu 0 computes $(-1)^{\text{original order}} \frac{1}{0+1} + (-1)^{Nprocs} \frac{1}{Nprocs+1} + (-1)^{2Nprocs} \frac{1}{2Nprocs+1} \cdots$

cpu 1 computes $(-1)^{1} \frac{1}{1+1} + (-1)^{Nprocs+1} \frac{1}{(Nprocs+1)+1} + (-1)^{2Nprocs+1} \frac{1}{(2Nprocs+1)+1} + \cdots$

⋮

cpu (Nprocs-1) computes $(-1)^{Nprocs-1} \frac{1}{(Nprocs-1)+1} + (-1)^{2Nprocs-1} \frac{1}{(2Nprocs-1)+1}$

- Let's use a Boss (typically node or cpu 0) - worker (all the rest) model
- Boss does I/O, everyone works on a different set of terms in the sum n=nid, nid+nprocs, etc
- Results from each processor need to be collected and added together to get final result.

Here is some code that does this:

```
#include <iostream>
#include "mpi.h"

class MPI_stuff
{
public:
  int NProcs;
  int MyID;

  MPI_stuff(int &argc, char** &argv)
  {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &NProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &MyID);
  }

  ~MPI_stuff()
  {
    MPI_Finalize();
  }
};

float compute_lnsum(const MPI_stuff &the_mpi, const int N)
{
  float sum = 0, Gsum;

  for(int i = the_mpi.MyID; i < N; i += the_mpi.NProcs)
    if(the_mpi.MyID % 2)
      sum -= (float) 1 / (i + 1);
    else
      sum += (float) 1 / (i + 1);

  MPI_Reduce(&sum,&Gsum,1,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
  return Gsum;
}


int main(int argc, char** argv)
{
  MPI_stuff the_mpi(argc, argv);

  int N;
  if(the_mpi.MyID == 0){
    std::cout << "Please enter the number of terms N -> ";
    std::cin >> N;
  }
  MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);

  float Gsum = compute_lnsum(the_mpi, N);
  if(the_mpi.MyID == 0)
    std::cout << "An estimate of ln(2) is " << Gsum << std::endl;

  return 0;
}
```

same class introduced for helloworld code last lecture

sum different terms based on My ID

collect & sum result from each processor

setup MPI

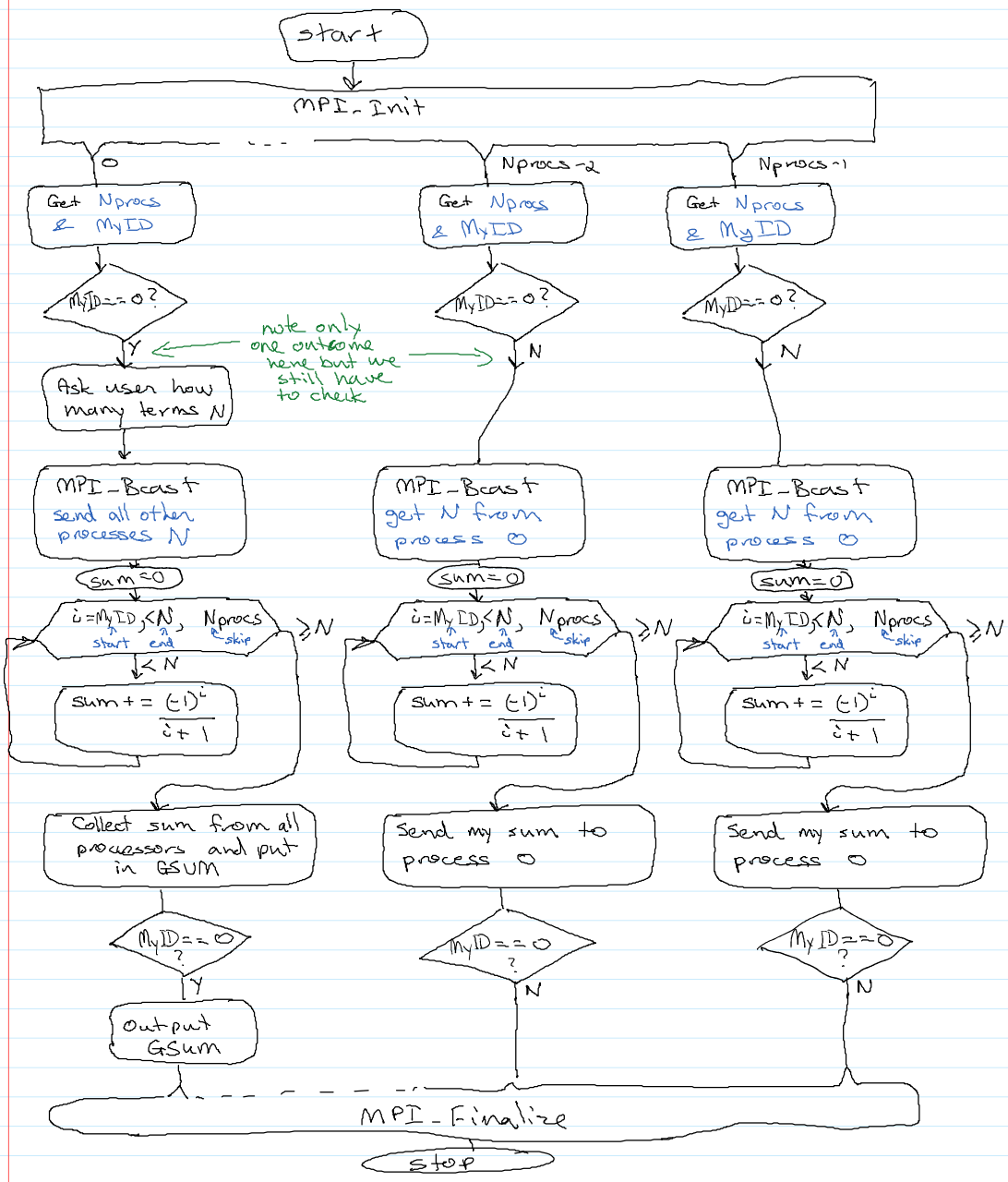
only one process (Boss) should interact with user

send/recieve N via a broadcast

compute sum in parallel (each worker & Boss calls this)

only Boss outputs result (to avoid duplication)

And here is a flowchart of what happens in the code



The new commands that take care of the communication are:

data size being sent is integer

```
MPI_BCast(&N , 1 , MPI_INT , 0 , MPI_COMM_WORLD)
```

- `&N` — pointer to beginning of data buffer to send
- `1` — # of this size to send (i.e. buffer size)
- `MPI_INT` — this size
- `0` — sending processor
- `MPI_COMM_WORLD` — who to send to (everyone here)

```
MPI_Reduce(&sum , &Gsum , 1 , MPI_FLOAT , MPI_SUM , 0 , MPI_COMM_WORLD)
```

- `&sum` — data being collected location
- `&Gsum` — where to put result
- `1` — number of this size in buffer
- `MPI_FLOAT` — this size
- `MPI_SUM` — operation to perform
- `0` — recieving processor
- `MPI_COMM_WORLD` — where info coming from

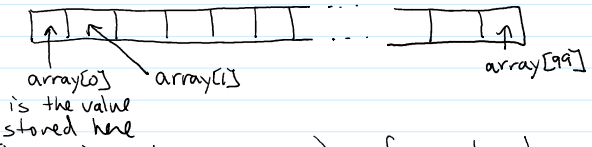
Note: N does not have to be a multiple of Nprocs so some processors will have slightly less work.

So far we have just passed data of a simple fundamental type. To pass larger blocks of data we first need to consider how they are stored.

C-style 1D arrays

```
double array[100];
```
creates a C-style array (also in C++)

This is stored in consecutive spaces in memory which we can illustrate as



The size (in memory) of each box is

```
sizeof(double)
```

which is 64 bits.
In order to send an array from one process to another we need to know where it starts in memory (its address) and its size.
To get the address of the start of the array we can use the address operator & or the array name

eg
```
#include <iostream>

int main()
{
    int array[5] = { 9, 7, 5, 3, 1 };

    // print the value of the array variable
    std::cout << "The array has address: " << array << '\n';

    // print address of the array elements
    std::cout << "Element 0 has address: " << &array[0] << '\n';

    return 0;
}
```

will produce something like

```
The array has address: 0042FD5C   <- array
Element 0 has address: 0042FD5C   <- &array[0]
```
↕ note, same address

To ensure compatible sizes across a potentially heterogeneous environment, MPI introduces its own fundamental data size
```
MPI_INT, MPI_FLOAT, MPI_DOUBLE
```
← use in MPI functions

For dynamic allocation of arrays in C we can use pointers and malloc/calloc. The second is preferable as it also initializes the elements (to zero).

```
float *a;
a = (float *) calloc(n, sizeof(float));
```
- (float *): data type of a
- n: number of elements in array
- sizeof(float): size of each element (in memory)

As with a statically declared array, this creates an array (of size n, which is an integer) stored in a contiguous block of memory. Its elements can be accessed via a[0], a[1], etc as before.

Note: In C (and C++) there is no bounds checking on arrays of this type. So if you try to access array[10] in the first example, or a[n+2] in the second, the compiler will NOT notice. This is a common, and often difficult to spot, source of errors. We will discuss alternatives in later lectures which avoid this, but for now we just need to be careful.

In C++ we can create dynamically allocated arrays using new[] and delete [] but this is no longer recommended (will be made obselete in future versions of C++). The recommended C++ option is std::vector

eg.
```
std::vector<float> a;
a.reserve(n);  a.resize(n);
```
↑ set capacity ↑ set size

will create an array similar to what we did in C. In particular, the elements a[0], a[1], ... are stored consecutively in memory, which is important for MPI (and performance in general).
A vector has a capacity (total amount of space it might use) and a size (amount of space it is currently using). These can be set (and changed) using the reserve and resize functions (for capacity and size respectively), with size ≤ capacity.
For scientific computing applications we are often working at the machine limits or need to know the space our code is going to need to run (required by queuing systems on compute canada systems). As such, it is best to manage the capacity manually (e.g. set it when you declare the vector, as in the example above. Also, do NOT use push_back to fill the vector).
vector comes with bounds checking, but only if you use a.at(j) rather than a[j]. There is a performance hit (small) for using the .at function so we don't typically use it. Again, we will introduce a better option later.

One-to-One Communication

eg. Dot Product
Compute dot product of two vectors "a" and "b" of length n

$$\vec{a}\cdot\vec{b} = \sum_{i=1}^{n} a_i b_i = \sum_{i=0}^{n-1} a[i]*b[i]$$

Let's assume we have Nprocs processors and that n is exactly divisible by Nprocs. We can then break up the task into parts of size $= \frac{n}{Nprocs}$ as a SIMD problem:

$$\vec{a}\cdot\vec{b} = \sum_{i=0}^{size-1} a[i]*b[i] + \sum_{i=size}^{2*size-1} a[i]*b[i] + \cdots + \sum_{i=(Nprocs-1)size}^{n-1} a[i]*b[i]$$

We will again use the Boss-worker model


```
#include <iostream>
#include <fstream>
#include <vector>
#include "mpi.h"

class MPI_stuff
{
public:
  int NProcs;
  int MyID;

  MPI_stuff(int &argc, char** &argv)
  {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &NProcs);
```

← regular I/O
← file stream header
← needed for std::vector

} MPI class from

```
  {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &NProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &MyID);
  }

  ~MPI_stuff()
  {
    MPI_Finalize();
  }
};
```

MPI class from before

```
// Get the number of rows of data in the file from first row of file
int GetNumberElements(std::ifstream &fin)
{
  int n;

  if (fin.is_open())
    fin >> n;
  else { // no file to read from
    std::cout << "Input File not found\n";
    MPI_Abort(MPI_COMM_WORLD,-1);
  }
  return n;
}
```

function to get n from first entry of a file

```
// Read n elemnts of two column data from a file, store in arrays a,b
void ReadArrays(std::ifstream &fin, std::vector<float> &a, std::vector<float> &b, int n)
{
  if(fin.is_open()){
    for(int i = 0; i < n; i++)
      fin >> a[i] >> b[i];
  }
  else{ // fp null means no file to read from
    std::cout << "Input File not found\n";
    MPI_Abort(MPI_COMM_WORLD,-1);
  }
}
```

read arrays from a file

```
// Dot product of two vectors owned and fully stored on Boss node
float DotProduct(std::vector<float> &a, std::vector<float> &b,
      int size, const MPI_stuff &the_mpi)
{
  MPI_Status status;

  // Distribute or Scatter the array to the workers
  if(the_mpi.MyID == 0){
    for(int i = 1; i < the_mpi.NProcs; i++){
      MPI_Send(&a[size*i],size,MPI_FLOAT,i,10,MPI_COMM_WORLD);
      MPI_Send(&b[size*i],size,MPI_FLOAT,i,20,MPI_COMM_WORLD);
    }
  }
  else{
    MPI_Recv(&a[0],size,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);
    MPI_Recv(&b[0],size,MPI_FLOAT,0,20,MPI_COMM_WORLD,&status);
  }

  // Work out my part of the sum
  float sum=0, Gsum=0;
  for(int i = 0; i < size; i++)
    sum += a[i] * b[i];

  // Collect resuls in Gsum
  MPI_Reduce(&sum,&Gsum,1,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);

  return Gsum;
}
```

BOSS sends parts of a,b to workers

matching calls

WORKERS recieve part of a,b

work out dot product of vector fragments

collect and sum results

actual $\vec{a} \cdot \vec{b}$ done here

```
/* See lecture notes for comments */
int main(int argc, char** argv)
{
  MPI_stuff the_mpi(argc, argv);

  std::ifstream fin;
  int n = 0;

  if(the_mpi.MyID == 0) {
    fin.open("DotData.txt");
    n = GetNumberElements(fin);
  }
  MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);

  if(n % the_mpi.NProcs){
    std::cout << "Number of processes is not a multiple of n.\n";
    MPI_Abort(MPI_COMM_WORLD,-2);
  }

  int size = n / the_mpi.NProcs;

  std::vector<float> a,b;
  if(the_mpi.MyID == 0){
    a.reserve(n);  a.resize(n); // try to avoid std::vector reallocations
    b.reserve(n);  b.resize(n);
    ReadArrays(fin,a,b,n);
  }
  else {
    a.reserve(size);  a.resize(size);
    b.reserve(size);  b.resize(size);
  }

  float adotb = DotProduct(a,b,size,the_mpi);

  if(the_mpi.MyID == 0)
    std::cout << "The inner product is " << adotb << std::endl;

  return 0;
}
```

only Boss node reads from the file (multiple processes cannot access file simultaneously)

← send n to everyone

← is n divisible by NProcs, If not exit program

kills program gracefully on all nodes

node 0 stores full arrays

← node 0 reads from file

worker nodes only need an array that fits the part they are working on

← compute $\vec{a} \cdot \vec{b}$

← node 0 (Boss) does output of result

This program introduces some new MPI commands:

pointer to beginning of buffer to be sent

processor to send to

"MPI communicator"

```
MPI_Send(&a[size*i],size,MPI_FLOAT,i,10,MPI_COMM_WORLD);
```

number of size in buffer

tag for this message (to match Recv)

these need to match (& given different tag)

pointer to beginning of receive buffer

where to get message from

tag

```
MPI_Recv(&a[0],size,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);
```

Comments

1. We could fix the n%nprocs condition several ways. An easy fix would be zero padding the end of the array.
2. Boss node uses more memory than other nodes. We could fix this by reading segments of the file and then sending before reading next segment.

If at the same time you got the Boss to work on the end of a&b (rather than begining) you could solve problem ① by having Boss do the

you could solve problem ① by having Boss do the "special" array of smaller size at the end.

3. Potential crash points should be noted and have program exit gracefully with MPI_Abort. This will ensure all MPI processes are stopped.

Summary of MPI Commands encountered so far

```
MPI_Init(&argc, &argv);
MPI_Finalize();
MPI_Abort(MPI_COMM_WORLD,-1);

MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Reduce(&sum,&Gsum,1,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);

MPI_Send(&a[size*i], size, MPI_FLOAT, i, 10, MPI_COMM_WORLD);
MPI_Recv(&a[0], size, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, &status);
```

Another variant of MPI_Reduce is MPI_Allreduce, eg

```
MPI_Allreduce(&maxerr, &maxerrG,1,MPI_FLOAT,MPI_MAX,MPI_COMM_WORLD);
```

In this case maxerr gets reduced to ALL processors, not only to one specific node (result in "maxerrG"). The operations (of type MPI_Op) are varied, in this case it finds the maximum of all maxerr and puts it into maxerrG. (This is All-to-All Communication).

The following predefined operations are supplied for MPI_Reduce and related functions MPI_Allreduce, MPI_Reduce_scatter, and MPI_Scan. These operations are invoked by placing the following in op:

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately below (MINLOC and MAXLOC).

From <https://www.open-mpi.org/doc/v1.8/man3/MPI_Reduce.3.php>

See the source above for more (OpenMPI docs).

Before going on let's discuss the "status" variable in MPI_Recv

MPI_Status is a data structure type with fields:

- MPI_SOURCE - id of processor sending message
- MPI_TAG - the message tag

MPI_Status is a data structure type with fields:
- MPI_SOURCE - id of processor sending message
- MPI_TAG - the message tag
- MPI_ERROR - error status (often not modified as would be the same as what function returns)

other fields are possible but implementation dependent

So status.MPI_SOURCE above could be used to get sender id.

Controlling Sycronization - Blocking & Non Blocking Communication

MPI_Send is what is known as a "blocking" routine: The sending processor waits for a "received" confirmation before working on the next line of code. Similarly MPI_Recv causes the receiving processor to wait for the message before moving on.

eg. Suppose I had 10 tasks, numbered from 1 to 10, to distribute to nprocs = 4 so nid $\in \{0, 1, 2, 3\}$

all integers

```
for (task=1; task <= 10; task++) {
   node=task-nprocs*(task/nprocs);
...
 }
```

← everything here is an integer so this expression is done using integer arithmetic

task	(task/nprocs)	node
1	0	1
2	0	2
3	0	3
4	1	0
5	1	1
6	1	2
7	1	3
8	2	0
9	2	1
10	2	2

So now consider the following code:

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

...
```

```
main(int argc, char** argv)
{
   struct mpi_vars this_mpi = mpi_start(argc, argv);
   MPI_Status status;
   int task, taskx, node;


   for (task=1; task <= 10; task++) {
      node=task-this_mpi.NProcs*(task/this_mpi.NProcs);

      if (node == this_mpi.MyID && node != 0) {   //Check to see if this is my task to do
          if (node==2) waste_your_time();
          MPI_Send(&task,1,MPI_INT,0,10,MPI_COMM_WORLD); //Tell Boss I am the one
      }
      if (this_mpi.MyID == 0) {
         if (node == 0)
           printf("Processor %d will compute %d\n", 0, task);
         else {
           MPI_Recv(&taskx,1,MPI_INT,MPI_ANY_SOURCE,10,MPI_COMM_WORLD,&status);
           printf("Processor %d will compute %d\n", status.MPI_SOURCE, taskx);
         }
      }
   }

   for (task=11; task <= 20; task++) {
      node=task-this_mpi.NProcs*(task/this_mpi.NProcs);

      if (node == this_mpi.MyID && node != 0) {  //Check to see if this is my task to do
          if (node==2) waste_your_time();
          MPI_Send(&task,1,MPI_INT,0,10,MPI_COMM_WORLD); //Tell Boss I am the one
      }
      if (this_mpi.MyID == 0) {
         if (node == 0)
           printf("Processor %d will compute %d\n", 0, task);
         else {
           MPI_Recv(&taskx,1,MPI_INT,MPI_ANY_SOURCE,10,MPI_COMM_WORLD,&status);
           printf("Processor %d will compute %d\n", status.MPI_SOURCE, taskx);
         }
      }
   }
   MPI_Finalize();
}
```

see which node should do the task

some function that takes up some time

take a message from anyone

node the message actually came from

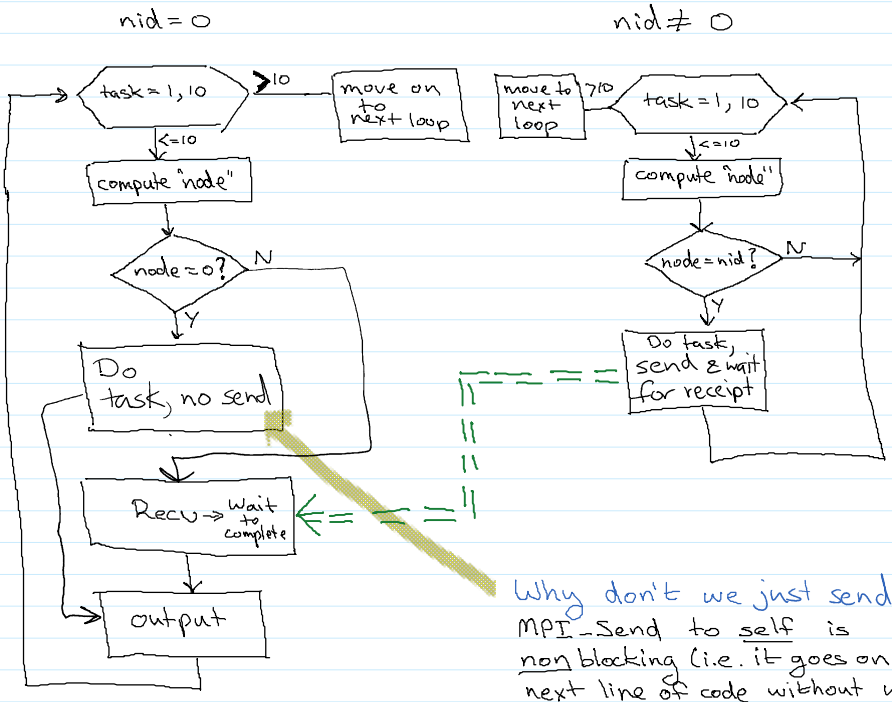
The output from such a code looks something like:

```
Processor        1 will compute        1
Processor        1 will compute        5
Processor        1 will compute        9
Processor        1 will compute       13
Processor        1 will compute       17
Processor        3 will compute        3
Processor        3 will compute        7
Processor        3 will compute       11
Processor        3 will compute       15
Processor        3 will compute       19
Processor        0 will compute        4
Processor        0 will compute        8
Processor        0 will compute       12
Processor        2 will compute        2
Processor        2 will compute        6
Processor        0 will compute       16
Processor        2 will compute       10
Processor        2 will compute       14
Processor        2 will compute       18
Processor        0 will compute       20
```

Comments:

1. The tasks are not completed in order (and order may change if you rerun it)
2. Node 2 is slower to complete tasks (as expected from waste_your_time() call.
3. node 0 is slower to complete tasks because it is busy getting results from other processors and outputting results.
4. node 1 & 3 are finished all tasks before node 0 & 2 are done their first task (they complete both loops before 0&2 finish 1st loop).

Let's create an old-style flowchart for this code's 1st loop:



Why don't we just send to self here?
MPI_Send to self is non blocking (i.e. it goes on with next line of code without waiting for a "received" confirmation.) However, some implementations have a bug on this and block.

If we need to have all of tasks 1-10 complete before going on to the 11-20 tasks, we need to syncronize the threads by adding an MPI_Barrier line between the loops:

MPI_Barrier (MPI_COMM_WORLD);

- blocks caller from proceeding until all processes in MPI_COMM_WORLD have made a similar call.

Adding this line between the loops results in the following output:

```
Processor     1 will compute     1
Processor     3 will compute     3
Processor     1 will compute     5
Processor     3 will compute     7
Processor     1 will compute     9
Processor     0 will compute     4
Processor     2 will compute     2
Processor     0 will compute     8
Processor     2 will compute     6
Processor     2 will compute    10
Processor     1 will compute    13
Processor     3 will compute    11
Processor     1 will compute    17
Processor     3 will compute    15
Processor     3 will compute    19
Processor     0 will compute    12
Processor     0 will compute    16
Processor     2 will compute    14
Processor     2 will compute    18
Processor     0 will compute    20
```

As MPI_Send is a blocking function it is possible to force the tasks to be performed "in order" by making the "tag" more specific. i.e: change loop to:

```
for (task=1; task <= 10; task++) {
      node=task-this_mpi.NProcs*(task/this_mpi.NProcs);

      if (node == this_mpi.MyID && node != 0) {  //Check to see if this is my task to do
          if (node==2) waste_your_time();
          MPI_Send(&task,1,MPI_INT,0,10+node,MPI_COMM_WORLD); //Tell Boss I am the one
      }
      if (this_mpi.MyID == 0) {
         if (node == 0)
           printf("Processor %d will compute %d\n", 0, task);
         else {
           MPI_Recv(&taskx,1,MPI_INT,MPI_ANY_SOURCE,10+node,MPI_COMM_WORLD,&status);
           printf("Processor %d will compute %d\n", status.MPI_SOURCE, taskx);
         }
      }
   }
```

Note however that this will slow things down as a processor that finishes early will now have to wait for node 0 to recieve the previous task before moving on the the next task ⇒ effectively no parallelization as tasks are done sequentially in time, just on different processors.

Non-blocking Communication

```
MPI_Isend(&old[size-1][1],N-1,MPI_FLOAT,nid+1,10,MPI_COMM_WORLD,&req_send10)
MPI_Irecv(&old[0][1],N-1,MPI_FLOAT,nid+1,10,MPI_COMM_WORLD,&req_recv10)
MPI_Request
MPI_REQUEST_NULL
MPI_Wait(&req_recv10,&status)
```

An immediate send (MPI_Isend) implies that the sending processor does not have to wait for a "received" confirmation before working on the next line of code. Such confirmation would be required for the regular MPI_Send. MPI_Isend is a nonblocking routine, while MPI_Send is a blocking routine.

In MPI_Isend note in particular the pointer to the variable req_send10. This is an object of the type MPI_Request. This object can be accessed again later to see whether the message has been delivered.

Similarly, MPI_Irecv contains a request handle, namely req_recv10 so that completion can be checked later.

`MPI_REQUEST_NULL` is the "default" value with which any variable/object of the request type should be initialized.

Lastly, `MPI_Wait` makes the executing node wait until the task associated with `req_recv10` is successfully finished.

Send requests often do not need to have confirmation, whereas if you are receiving data you need to check whether you got something before working with the data. Note that MPI_IRecv + MPI_Wait is effectively the same as MPI_Recv.

eg. Changing our loop from previous example to:

```
for (task=1; task <= 10; task++) {
      node=task-this_mpi.NProcs*(task/this_mpi.NProcs);

      if (node == this_mpi.MyID) {   //Check to see if this is my task to do
          if (node==2) waste_your_time();
          MPI_Isend(&task,1,MPI_INT,0,10,MPI_COMM_WORLD,&req_send10); //Tell Boss I am the one
      }
      if (this_mpi.MyID == 0) {
          req_recv10 = MPI_REQUEST_NULL;
          MPI_Irecv(&taskx,1,MPI_INT,MPI_ANY_SOURCE,10,MPI_COMM_WORLD,&req_recv10);
          MPI_Wait(&req_recv10,&status);
          printf("Processor %d will compute %d\n", status.MPI_SOURCE, taskx);
      }
   }
```

need to set here so not prematurely passed

previously declared of type MPI_Request

This removes the blocking effect of MPI_Send and allows all processes (including MYID=0) to proceed through the entire list of tasks without waiting for BOSS processor to confirm receipt.

To avoid having BOSS node waiting for others to complete before doing its task, we should move the recieves to a seperate statement:

```
for (task=1; task <= 10; task++) {
      node=task-this_mpi.NProcs*(task/this_mpi.NProcs);

      if (node == this_mpi.MyID) {        //Check to see if this is my task to do
          if (node==2) waste_your_time();
          MPI_Isend(&task,1,MPI_INT,0,10,MPI_COMM_WORLD,&req_send10); //Tell Boss I am the one
      }
}

if (this_mpi.MyID == 0) {
  for (task=1; task <= 10; task++) {
          req_recv10 = MPI_REQUEST_NULL;
          MPI_Irecv(&taskx,1,MPI_INT,MPI_ANY_SOURCE,10,MPI_COMM_WORLD,&req_recv10);
          MPI_Wait(&req_recv10,&status);
          printf("Processor %d will compute %d\n", status.MPI_SOURCE, taskx);
  }
}
MPI_Barrier(MPI_COMM_WORLD);
```

← a good idea to put this here to prevent more messages being sent to nid=0 by other processors and getting confused with these tasks by nid=0 as the IRecv is very open (i.e. non-specific, it is taking messages from anyone).