Exercise 1: Hello Word

Problem description:

This application gets the name of each processor. The process with **pid 0** sends "Hello word" message to the rest of the processes.

Student tasks :

The student has to visualize the execution of "hello.c" program with four processes by using Jumpshot.

1. **Compile hello.c for generating the clog2 file:**
    a. **$ mpicc -g –c hello.c**
    b. **$ mpicc -o hello hello.o -llmpe -lmpe –lm**
2. Run hello with 4 processes:
    a. qsub run.sh
3. Convert log format from clog2 to slog2:
    a. /share/apps/jumpshot/bin/clog2TOslog2 hello.clog2
4. Open a new terminal and connect to the cluster with –X
5. Start the jumpshot
    a. /share/apps/jumpshot/bin/jumpshot
    b. Save configuration file
6. Select hello.slog2 in the Jumpshot

Use the following script to submit the job to the cluster (run.sh)

```
#!/bin/bash
## Especifica l,Äôint√®rpret de comandes pel treball (Bash)
#$ -S /bin/bash
## Exporta totes les variables d,Äôentorn al context del treball
#$ -V
## Executa el treball desde el directori de treball actual.
#$ -cwd
## La cua sobre la que es vol llan√ßar (high.q, low.q, all.q)
#$ -q high.q
## Entorn de programaci√≥ paral¬∑lel pel treball i numero de processos
#$ -pe mpich 4
## El nom del treball (opcional)
#$ -N treball
MPICH_MACHINES=$TMPDIR/mpich_machines
cat $PE_HOSTFILE | awk '{print $1":"$2}' > $MPICH_MACHINES
## Aquesta l√≠nia es la que realment executa l'aplicaci√≥

mpiexec -f $MPICH_MACHINES -n $NSLOTS ./hello

rm -rf $MPICH_MACHINES
```

Exercise 2: Understanding Blocking and Non-blocking operations

Basic information before starting the exercise: Blocking Operations

MPI_Send (`void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm`)

The message buffer is described by (buf, `count`, `datatype`).

The target process is specified by `dest`, which is the rank of the target process in the communicator specified by `comm`.

When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI_Recv(`void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status`)

Waits until a matching (on `source` and `tag`) message is received from the system, and the buffer can be used. `source` is rank in communicator specified by `comm`, or `MPI_ANY_SOURCE`. `Status` contains further information. Receiving fewer than `count` occurrences of `datatype` is OK, but receiving more is an error

Part I (blocking.c)

Problem description:

This assignment implements a simple parallel data structure. This structure is a two dimension regular mesh of points, divided into slabs, with each slab allocated to a different processor. In the simplest C form, the full data structure is

```
double x[maxn][maxn];
```

and we want to arrange it so that each processor has a local piece:

```
double xlocal[maxn][maxn/size];
```

where size is the size of the communicator (e.g., the number of processors).

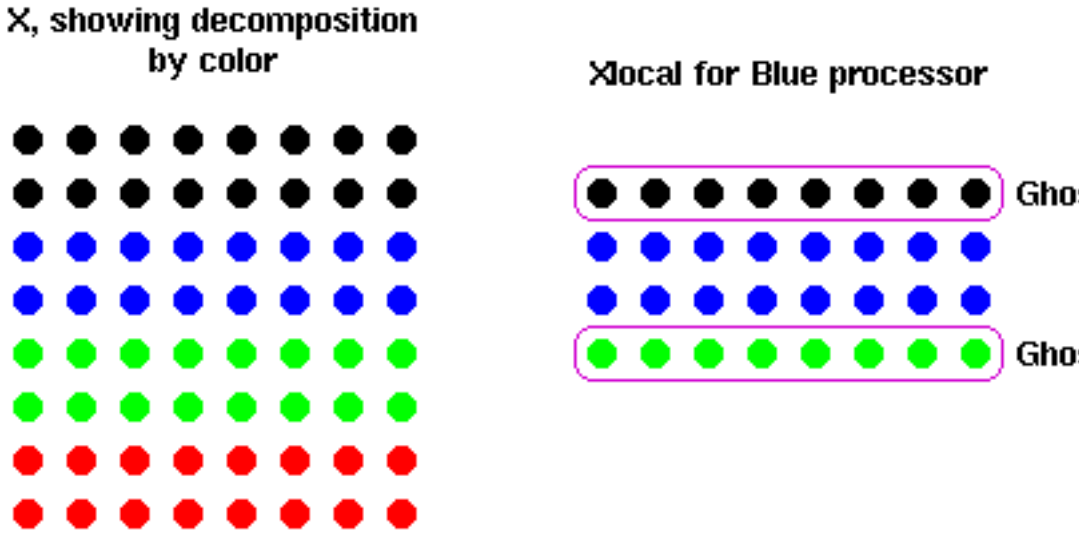
If that was all that there was to it, there wouldn't be anything to do. However, for the computation that we're going to perform on this data structure, we'll need the adjacent values. That is, to compute a new x[i][j], we will need

x[i][j+1] x[i][j-1] x[i+1][j] x[i-1][j]

The last two of these could be a problem if they are not in xlocal but are instead on the adjacent processors. To handle this difficulty, we define ghost points that we will contain the values of these adjacent points.

Write code to copy divide the array x into equal-sized strips and to copy the adjacent edges to the neighboring processors. Assume that x is maxn by maxn, and that maxn is evenly divided by the number of processors. For simplicity, You may assume a fixed size array and a fixed (or minimum) number of processors.

To test the routine, have each processor fill its section with the rank of the process, and the ghostpoints with -1. After the exchange takes place, test to make sure that the ghostpoints have the proper value. Assume that the domain is not periodic; that is, the top process (rank = size - 1) only sends and receives data from the one under it (rank = size - 2) and the bottom process (rank = 0) only sends and receives data from the one above it (rank = 1). Consider a maxn of 12 and use 4 processors to start with.



For this exercise, use MPI_Send and MPI_Recv. See the related exercises for alternatives that use the nonblocking operations or MPI_SendRecv.
A more detailed description of this operation may be found in Chapter 4 of "Using MPI".
You may want to use these MPI routines in your solution:
MPI_Send MPI_Recv

Student tasks :

The student has to visualize the execution of "blocking.c" program with four processes by using Jumpshot.

1. Compile the blocking.c for generating clog2 file
2. Run "blocking" with 4 processes
3. Generate the blocking.slog2 trace
4. Visualize blocking.slog2 with Jumpshot

Basic information before starting the second part of the exercise: Non blocking Operations.

MPI_Isend (**`void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request`**
)

The message buffer is described by (buf, `count, datatype`).

The target process is specified by `dest`, which is the rank of the target process in the communicator specified by `comm`.

This function returns before the communication is completed.

MPI_Irecv(**`int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source int tag, MPI_Comm comm, MPI_Request *request)`**

Nonblocking calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion. A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

A receive request can be determined being completed by calling the MPI_Wait, MPI_Waitany, MPI_Test, or MPI_Testany with request returned will be used to throw an MPI:Exception object.

`MPI_Wait(MPI_Request *request, MPI_Status *status)`

This call will make your process hang until the operation identified by the request is complete. The call to MPI_Wait deallocates the request and sets the request handle to MPI_REQUEST_NULL.

Part II (nonblocking.c)

Problem description

This code uses the non-blocking point-to-point routines instead of the blocking routines. It replaces the MPI_Send and MPI_Recv routines with MPI_Isend and MPI_Irecv, and uses MPI_Wait and MPI_Waitall to test for completion of the nonblocking operations.

Student tasks

The student has to visualize the execution of "nonblocking.c" program with four processes by using Jumpshot and compare it with blocking version.

1. Try to understand the difference between blocking.c and nonblocking.c
2. Compile nonblocking.c for generating clog2
3. Run "nonblocking.c" with 4 processes
4. Generate the nonblocking.slog2 trace
5. Visualize with Jumpshot and study the differences with the previous visualizatiom

Exercise 3: Performance of Blocking and Non-Blocking operations by using Jumpshot and TAU tools.

Problem description

The applications "buffer_blocking.c" and "buffer_non_blocking.c" demonstrate the difference between blocking and non-blocking point-to-point communication. The application buffer_blocking uses MPI_Send/MPI_Recv (blocking) and buffer_non_blocking uses MPI_Isend/MPI_Irecv (non-blocking).

Each task transfers a vector of random numbers (sendbuff) to the next task (taskid+1). The last task transfers it to task 0. Consequently, each task receives a vector from the preceding task and puts it in recvbuff.

The buffer_blocking application shows that MPI_Send and MPI_Recv are not the most efficient functions to perform this work. Since they work in blocking mode (i.e. waits for the transfer to finish before continuing its execution). In order to receive their vector, each task must post an MPI_Recv corresponding to the MPI_Send of the sender, and so wait for the reception to complete before sending sendbuff to the next task. In order to avoid a deadlock, one of the task must initiate the communication and post its MPI_Recv after the MPI_Send. In this example, the last task initiate this cascading process where each consecutive task is waiting for the complete reception from the preceding task before starting to send "sendbuff" to the next. The whole process completes when the last task have finished its reception.

Before the communication, task 0 gather the sum of all the vectors to sent from each tasks, and prints them out. Similarly after the communication, task 0 gathers all the sum of the vectors received by each task and prints them out along with the communication times. The "buffer_non_blocking" show how to use non-blocking communication (MPI_Isend andMPI_Irecv) to accomplish the same work is much less time. **The size of the vector (buffsize) is given as an argument to the program at run time:**

```
mpiexec -f $MPICH_MACHINES -n $NSLOTS
./buffer_blocking 10024
```

Student tasks

The student has to visualize the execution of "buffer_blocking.c" and "buffer_non_blocking.c" programs with 8 and 16 processes by using Jumpshot and TAU. With both tools the student has to compare both version, and understand the performance of both of them.

1. Compile the buffer_blocking.c and buffer_non_blocking.c for generating clog2 files
2. Run buffer_blocking.c with 8 processes and buffer size 10024
3. Run buffer_blocking.c with 8 processes and buffer size 10024
4. Generate the slogs2 for the two applications
5. Visualize both slogs in Jumpshot and compare them.
6. Realize the steps 2-3-4-5 but using 16 processes, and buffer size 20048
7. Compile again buffer_blocking.c and buffer_non_blocking.c (8 processes and 10024 buffer size) to generate the profiles and traces for TAU application. Recommendation: First compile, run and and visualize the profile of buffer_blocking.c. Later, compile, run and visualize the profile of buffer_non_blocking.c
    a. Profiles and Traces:
        i. **Compile buffer_blocking.c:**
            1. **mpicc –o buffer_blocking buffer_blocking.c**
        ii. Run buffer_blocking.c for generating traces and profiles:
            1. export TAU_TRACE=1 # This line is needed only for tracing.
            2. export TAU_PROFILE=1 # This line is needed only for profiling.
            3. mpiexec -f $MPICH_MACHINES -n $NSLOTS **tau_exec -T PROFILE** ./buffer_blocking
        iii. Pack all the profiles.* files in a unique one:
            1. paraprof --pack buffer_blocking.ppk
        iv. Visualize buffer_blocking.ppk using paraprof tool:
            1. Open a new terminal by using –X
            2. /share/apps/tau/i386_linux/bin/paraprof
        v. Realise the same with buffer_non_blocking.c

The student will visualize in TAU the next screen at the end of this exercise.