

Assignment 3

MPI Tutorial

Compiling and running MPI programs

B. Wilkinson and Clayton Ferner: Modification date: Sept 16, 2014

This assignment is a tutorial to learn how to execute MPI programs and explore their characteristics. First you will test your programs on your own computer using the provided virtual machine (or with a native Linux installation). All the programs are provided. Both command line (Part 1) and using Eclipse (Part 2) will be explored. Then, you will test the programs on a remote cluster (Part 3) and measure the speedup. Part 4 is for graduate students (extra credit for undergraduates)

Preliminaries – Software Environment

MPI is process-based where individual processes can execute at the same time on local or distributed computers and explicit message-passing routines are used to pass data between the processes.

You will need MPI software to compile and execute MPI programs. The provided VM has OpenMPI and Eclipse-PTP already installed. The sample MPI programs and data files are in `~/ParallelProg/MPI`. Eclipse-PTP is needed for Part 2.

If you are using your own Linux installation, you will need to install MPI software such as OpenMPI or MPICH. How to install OpenMPI is described under the link “Installing OpenMPI” found at from the link “VM software” on the course home page (Pre-assignment). The sample MPI programs and data files can also be found there. Installing Eclipse-PTP can be found in link “Installing Java and Eclipse-PTP”.

Cd to the directory where the sample MPI programs reside (`~/ParallelProg/MPI`).

Part 1 Using Command Line (Graduates 30%, Undergraduates 35%)

Hello World Program

A simple hello world program, called `hello.c`, is given below demonstrating MPI sends and receives. The program is provided within the MPI directory.

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

main(int argc, char **argv ) {
    char message[256];
    int i,rank, size, tag=99;
    char machine_name[256];

    MPI_Status status;
```

```

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

gethostname(machine_name, 255);

if(rank == 0) {
    printf ("Hello world from master process %d running on %s\n",rank,machine_name);
    for (i = 1; i < size; i++) {
        MPI_Recv(message, 256, MPI_CHAR, i, tag, MPI_COMM_WORLD, &status);
        printf("Message from process = %d : %s\n", i, message);
    }
} else {
    sprintf(message, "Hello world from process %d running on %s",rank,machine_name);
    MPI_Send(message, 256, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
}

MPI_Finalize();
return(0);
}

```

MPI routines are shown in red.

Task 1 Compiling and Executing Hello World Program

Compile and execute the hello world program. To compile, issue the command:

```
mpicc hello.c -o hello
```

from the **MPI** directory. Execute with the command:

```
mpiexec -n 4 ./hello
```

which will use 4 processes. So far, the four instances of the program will execute just on one computer. You should get the output:

```

Hello world from master process 0 running on ...
Message from process = 1 : Hello world from process 1 running on ...
Message from process = 2 : Hello world from process 2 running on ...
Message from process = 3 : Hello world from process 3 running on ...

```

Comment on the how MPI processes map to processors/cores. Try 16 processes and see the CPU usage (in Windows, the Task Manager).

Experiments with Code

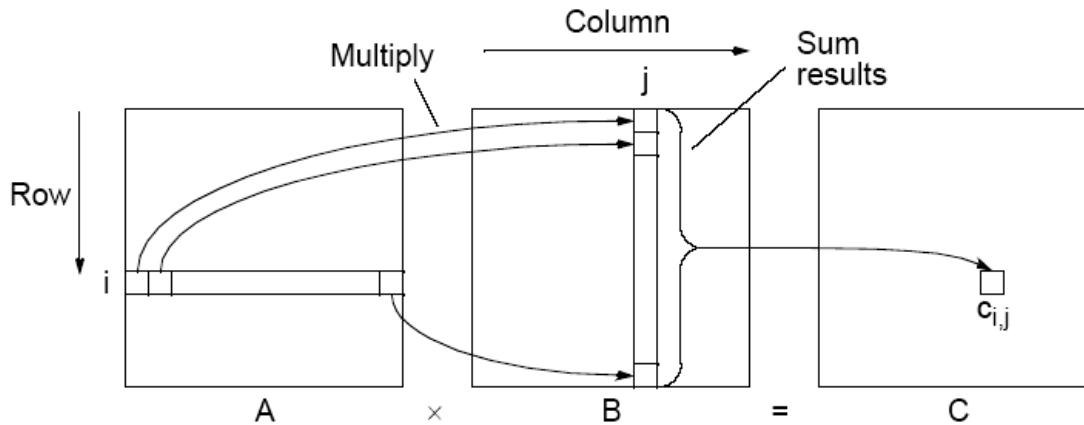
Modify the hello world program by specifying the rank and tag of the receive operation to `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively. Recompile the program and execute. Is the output in order of process number? Why did the first version of hello world sort the output by process number but not the second?

Include in your submission document for Task 1:

1. A screenshot or screenshots showing:
 - a. Compilation of the hello world program
 - b. Executing the program with its output
2. The effects of changing the number of processes
3. The effects of using wild cards (MPI_ANY_SOURCE and MPI_ANY_TAG)
4. Answer to the question about process order.

Matrix Multiplication

Multiplication of two matrices, **A** and **B**, produces matrix **C** whose elements, $c_{i,j}$ ($0 \leq i < n$, $0 \leq j < m$), computed as follows: $c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$ where **A** is an $n \times l$ matrix and **B** is a $l \times m$ matrix:

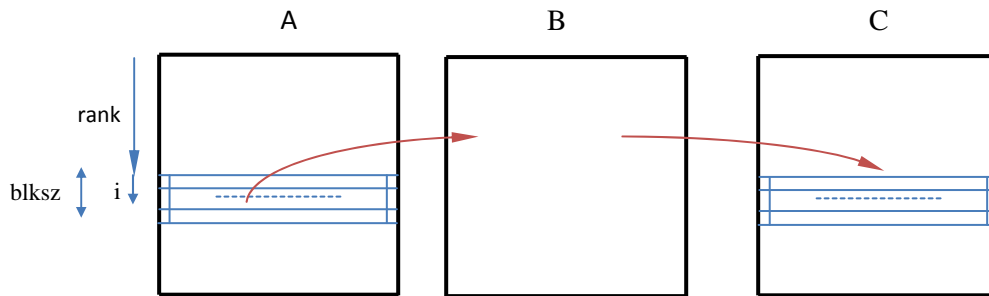


The sequential code to compute **A** x **B** (assumed square $N \times N$) could simply be:

```
for (i = 0; i < N; i++)           // for each row of A
  for (j = 0; j < N; j++) {       // for each column of B
    c[i][j] = 0;
    for (k = 0; k < N; k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
  }
```

It requires N^3 multiplications and N^3 additions with a sequential time complexity of $O(N^3)$. It is very easy to parallelize as each result is independent. Often the size of matrices (N) is much larger than number of processes (P)¹, so rather than having one process for each result, we can have each process compute a group of result elements. In MPI, a convenient arrangement is to take a group of rows of **A** and multiply that with **B** to create a groups of rows of **C**, which can then be gathered using MPI_Gather():

¹ Usually we map one process onto each processor so process and processor are the same, but not necessarily. *It is important not to confuse process with processor.*



This does require B to be broadcast to all processes. A simple MPI matrix multiplication program, called **matrixmult.c**, is given below and provided within the MPI directory:

```

#define N 512
#include <stdio.h>
#include <sys/time.h>
#include "mpi.h"

void print_results(char *prompt, double a[N][N]) {
    int i, j;
    printf ("\n\n%s\n", prompt);
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf(" %.2lf", a[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n");
}

int main(int argc, char *argv[]) {
    int i, j, k, error = 0;
    double a[N][N], b[N][N], c[N][N];

    char *usage = "Usage: %s file\n";
    FILE *fd;

    double elapsed_time;
    struct timeval tv1, tv2;

    MPI_Status status; // MPI variables
    int rank, P, blksz;

    MPI_Init(&argc, &argv); // Start MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &P);

    if ((rank == 0) && (N % P != 0)) { // Program only works if P divides evenly into N
        error = -1;
        printf("Error -- N/P must be an integer\n");
    } else blksz = N/P;

    if(rank == 0) { // open file
        if (argc < 2) {
            fprintf (stderr, usage, argv[0]);
            error = -1;
        }
        if ((fd = fopen (argv[1], "r")) == NULL) {

```

```

        fprintf(stderr, "%s: Cannot open file %s for reading.\n", argv[0], argv[1]);
        fprintf(stderr, usage, argv[0]);
        error = -1;
    }
}

if(rank == 0) { // broadcast any error and close down if error
    for (i = 1 ; i < P; x++) {
        MPI_Send(&error, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
} else {
    MPI_Recv(&error, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}

if (error != 0) { //terminate the process
    printf("This is process %d... An error occurred...I am shutting down...\n", rank);
    MPI_Finalize();

return 0;
}

if (rank == 0) { // read file
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            fscanf(fd, "%lf", &a[i][j]);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            fscanf(fd, "%lf", &b[i][j]);
}

MPI_Barrier(MPI_COMM_WORLD); // Add a barrier prior to the time stamp.

if (rank == 0) gettimeofday(&tv1, NULL);

MPI_Scatter(a, blkksz*N, MPI_DOUBLE, a, blkksz*N, MPI_DOUBLE, 0, MPI_COMM_WORLD); // Scatter a
MPI_Bcast(b, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD); // Broadcast the input matrix b

for(i = 0 ; i < blkksz; i++) {
    for(j = 0 ; j < N ; j++) {
        c[i][j] = 0;
        for(k = 0 ; k < N ; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}

MPI_Gather(c, blkksz*N, MPI_DOUBLE, &c, blkksz*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

if(rank == 0) {
    gettimeofday(&tv2, NULL);
    elapsed_time = (tv2.tv_sec - tv1.tv_sec) + ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
    printf ("elapsed_time=%t%lf (seconds)\n", elapsed_time);

//    print_results("C = ", c); // used to check output
}
MPI_Finalize();
return 0;
}

```

The program reads a data file that contains the two floating point $N \times N$ matrices to multiply together and prints out the resultant matrix. N is defined as 512 (i.e. 512 x 512 matrices). The

name of the input file is given as a command line argument. An input test file **input2x512x512Doubles** is provided and also the answer, **output512x512mult**, which is used for checking results.

Task 2 Compiling and Executing Matrix Multiplication Program

Compile and execute the matrix multiplication program. To compile, issue the command:

```
mpicc matrixmult.c -o matrixmult
```

from the **MPI** directory.

The program reads the input file. To execute the program, you will need to add the name of that file (**input2x512x512Doubles**) as the final command line argument after your program on the **mpixec** command, for example:

```
mpixec -n 4 ./matrixmult input2x512x512Doubles
```

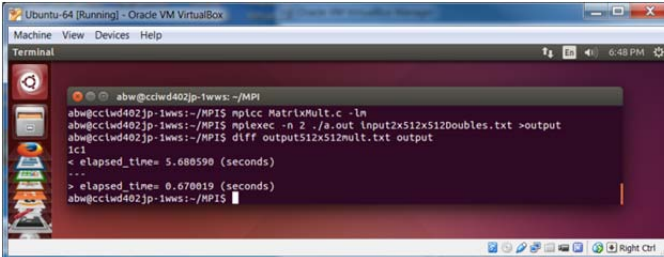
One can experiment with a different number of processes. Unfortunately generally one will not see a particular increase in speed on a personal computer because of the message passing overhead. (VirtualBox limits the number of cores available. Go to **Machines > Settings > System > Processors** to alter and reboot the OS.)

To check answers are correct, remove the **//** comments in front of the **print_results** routine, recompile, execute with re-directing the printout to a file, and use the **diff** command:

```
mpixec -n 4 ./matrixmult input2x512x512Doubles > output.txt
```

```
diff output512x512Mult output.txt
```

Typical output is:



```
abw@cclwd402jp-1rws: ~/MPI
abw@cclwd402jp-1rws:~/MPI$ mpicc MatrixMult.c -ln
abw@cclwd402jp-1rws:~/MPI$ mpixec -n 2 ./a.out input2x512x512Doubles.txt >output
abw@cclwd402jp-1rws:~/MPI$ diff output512x512Mult.txt output
.txt
< elapsed_time= 5.688590 (seconds)
...
> elapsed_time= 0.678019 (seconds)
abw@cclwd402jp-1rws:~/MPI$
```

Include in your submission document for Task 2:

A screenshot or screenshots showing:

- Compilation of the multiplication program
- Executing the program with its output, with different numbers of processes
- Use of the diff command

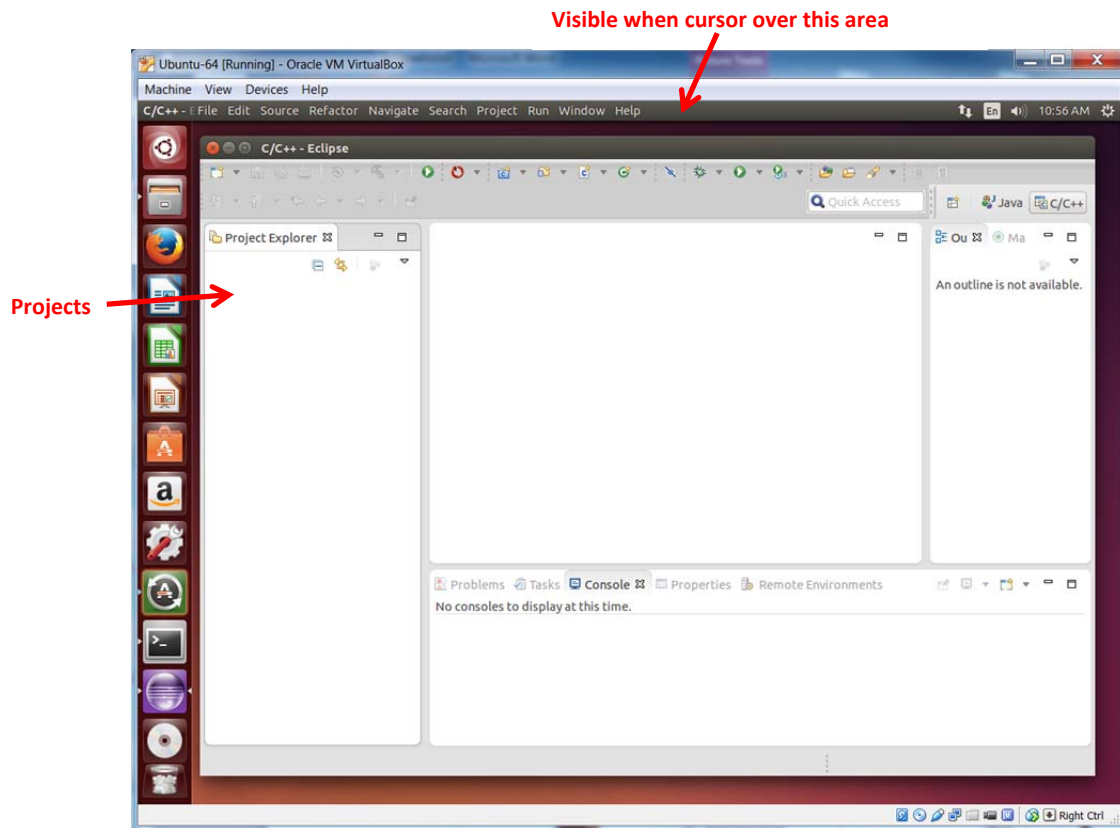
Part 2 Eclipse-PTP (Graduates 30%, Undergraduates 35%)

Eclipse-PTP (Eclipse with the tools for Parallel Applications Developers) can be used to compile and execute MPI programs. In Eclipse, there are different project types for different environments. MPI programs are done as C/C++ projects with build/compilation for MPI pre-configured in Eclipse-PTP. Programs here will be C projects.

Start Eclipse on the command line by typing:

eclipse

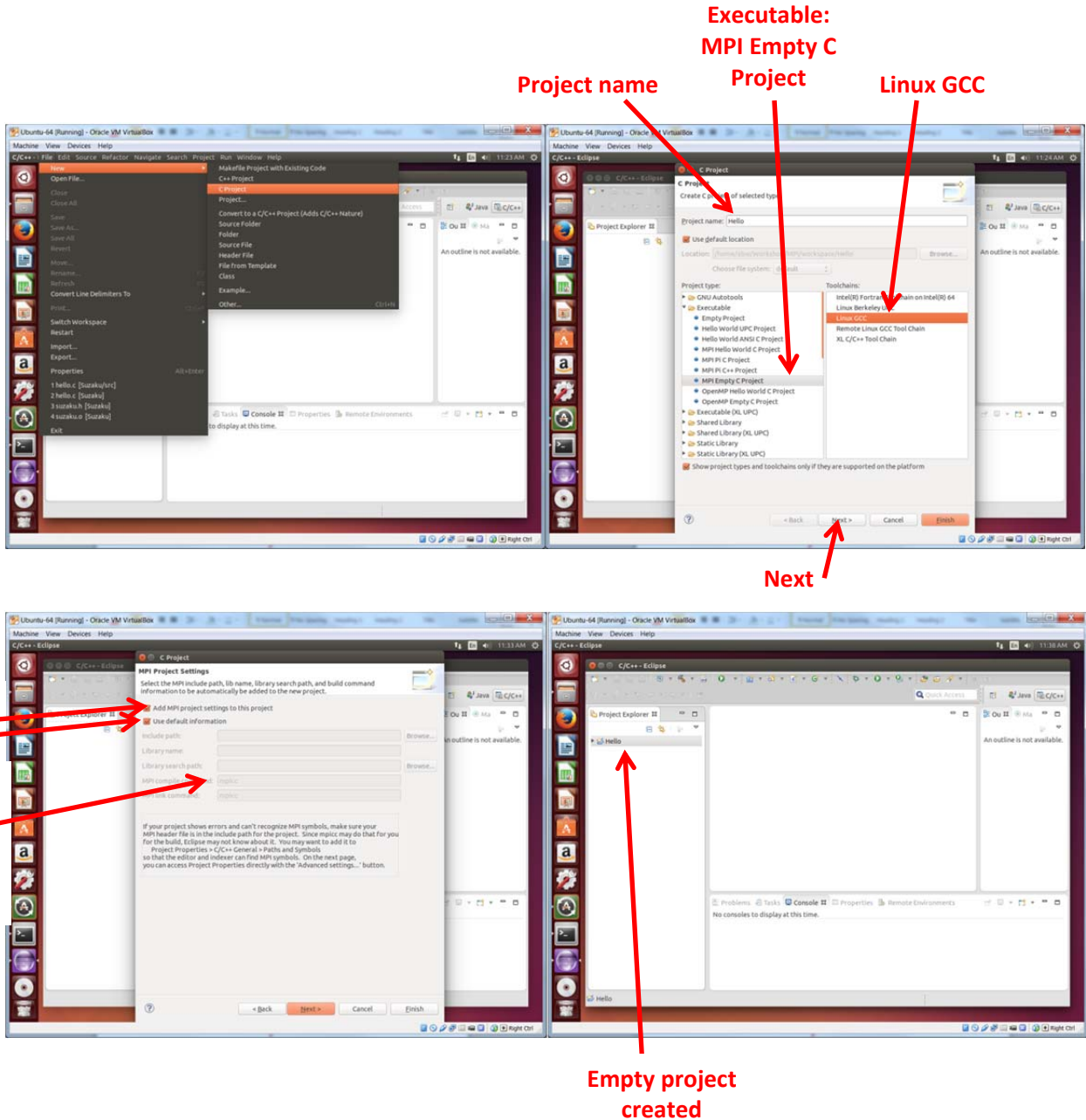
Create/select a workbench location within the MPI directory (i.e. **~/ParallelProg/MPI/workspace**) and go to the workbench. This workspace will be empty until we create projects for the programs we want to execute:



Task 1 Creating and executing Hello world program through Eclipse

(a) Creating project and adding source files

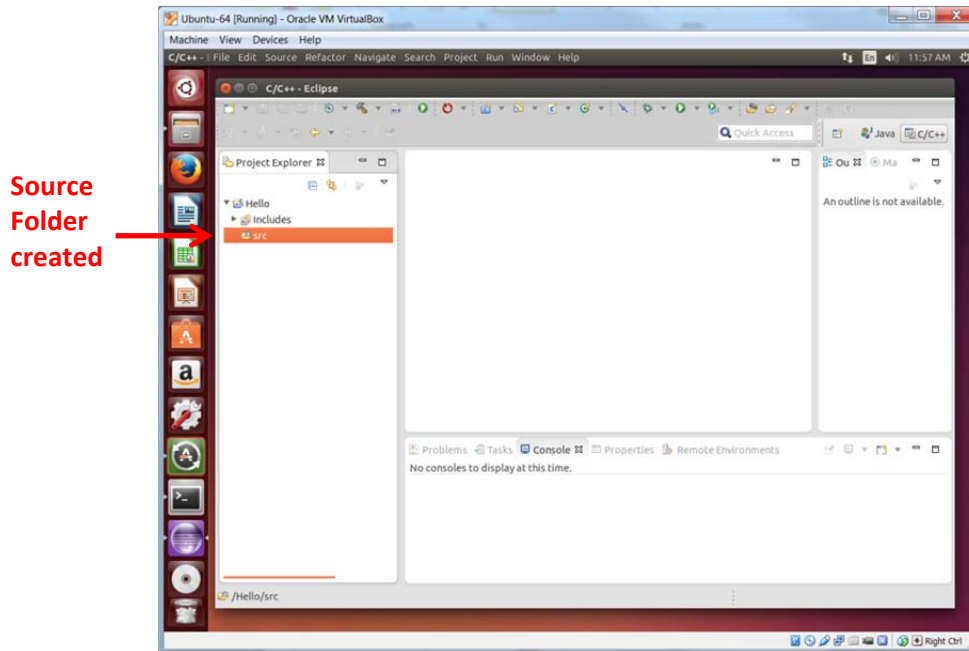
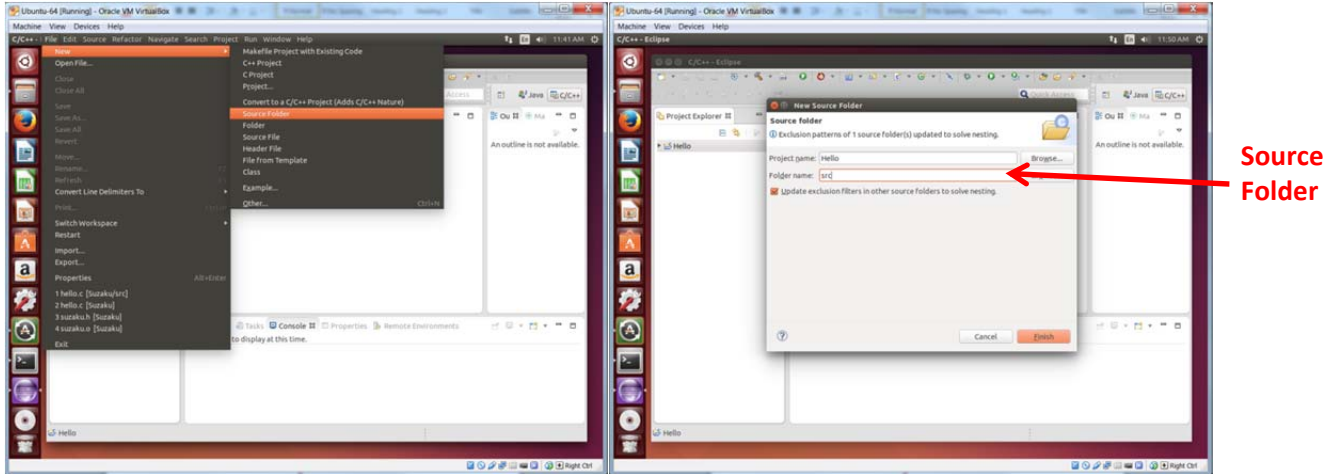
Create new C project (**File > New > C project**) called **Hello** of type “**MPI Empty C project**”² with default settings:



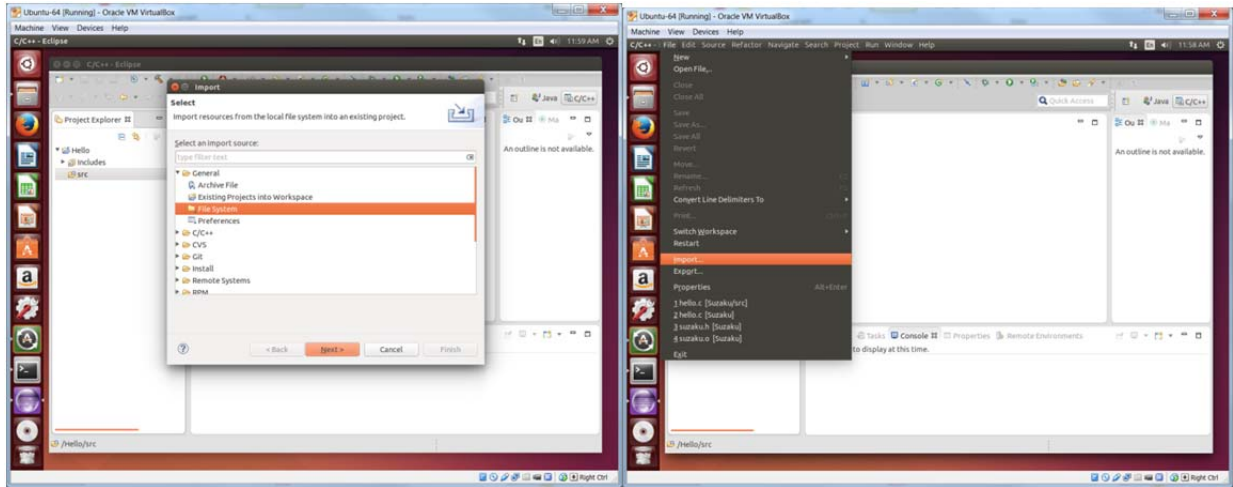
² Eclipse comes with sample programs pre-installed for C, OpenMP, and MPI (“Hello World” and MPI Pi), which are useful for testing the environment.

Adding program source file

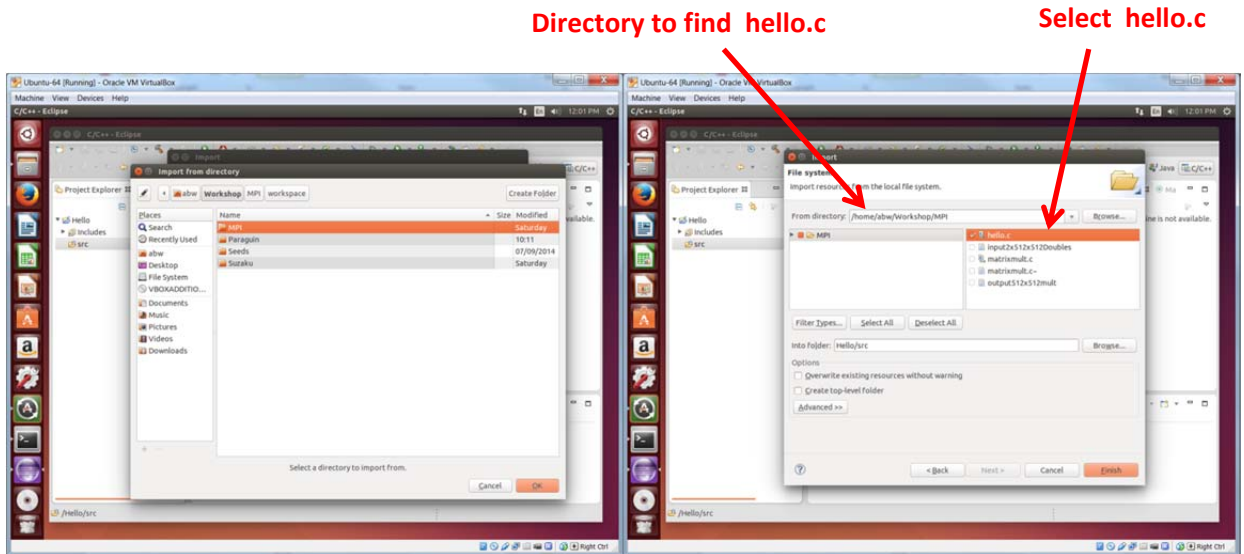
Select Hello project and create a source folder called **src** (**File > New >Source Folder** or **right click project > New >Source Folder**):



Expand the **Hello** project and select the newly generated **src** folder. Select **File > Import > General > File System**:



From “**Import from directory**”, browse for the directory that holds your **hello.c** files (**~/ParallelProg/MPI**). Click **OK**. Select the **hello.c** file to copy into the **MPI/src** project folder.³

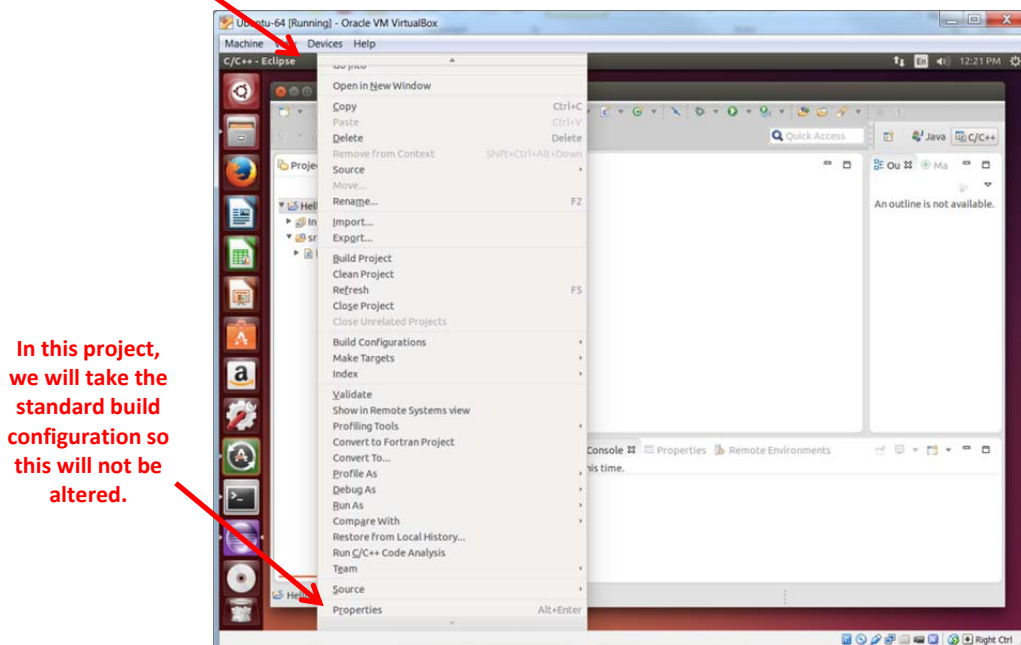
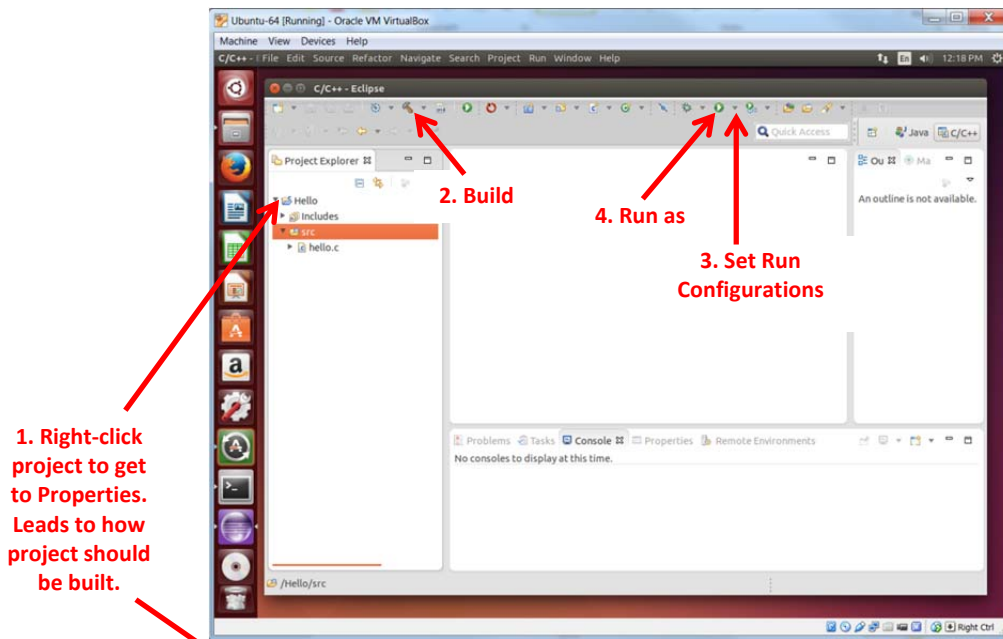


³In a later example, we will demonstrate using a link to the source file at their original location rather than copying it, which is usually better.

(b) Build and Compile

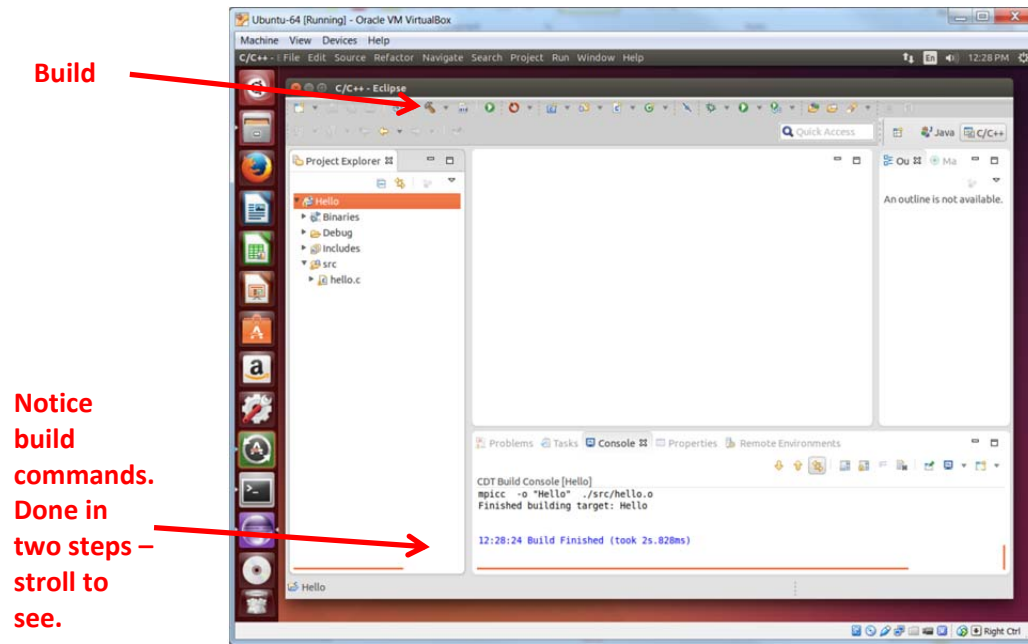
Basic steps:

1. Set how to build (compile) project in **Properties > ... Build**
2. Build project (compile to create executable)
3. Set how to execute compiled program in **Run Configurations**
4. **Run** (execute) using the specified run configurations



Build project

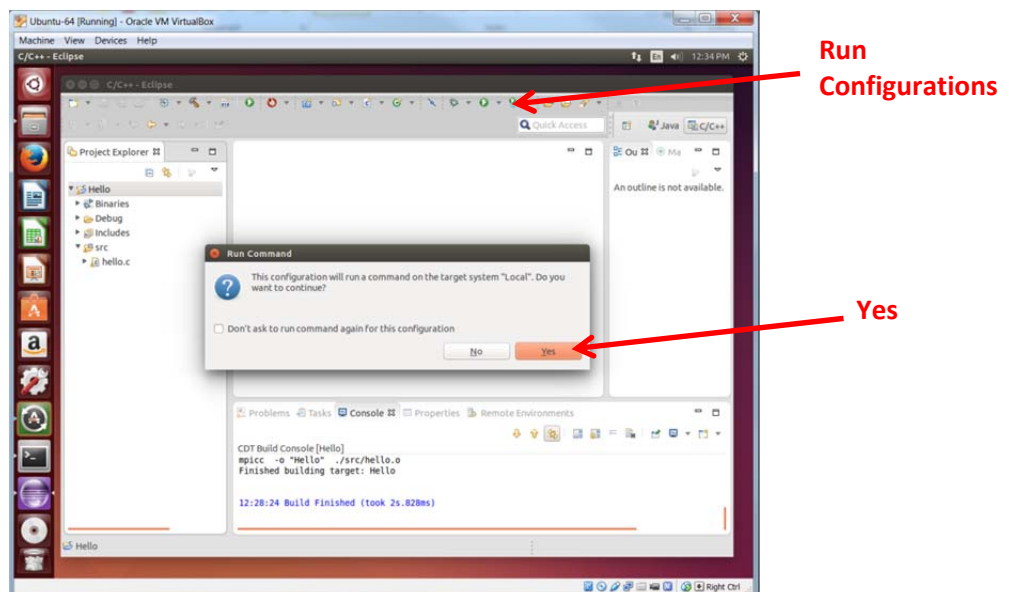
Click **Build** icon (Hammer) to build the project (default “Debug” option):



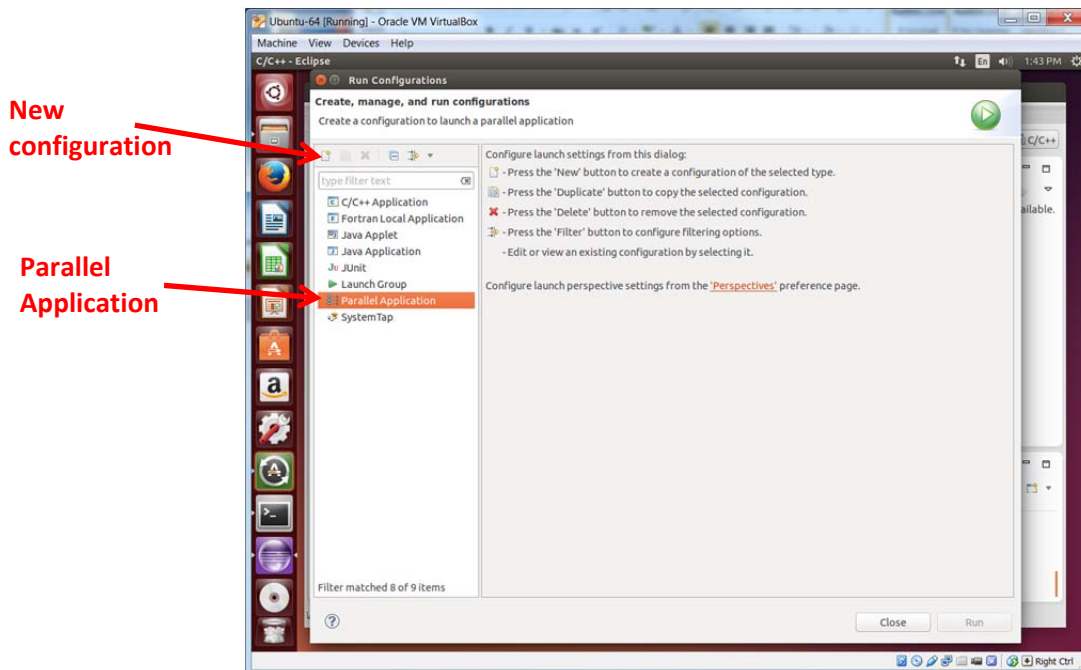
Although building will happen automatically when a project is executed next, sometimes it is handy to know if there are any build errors first.

Execution

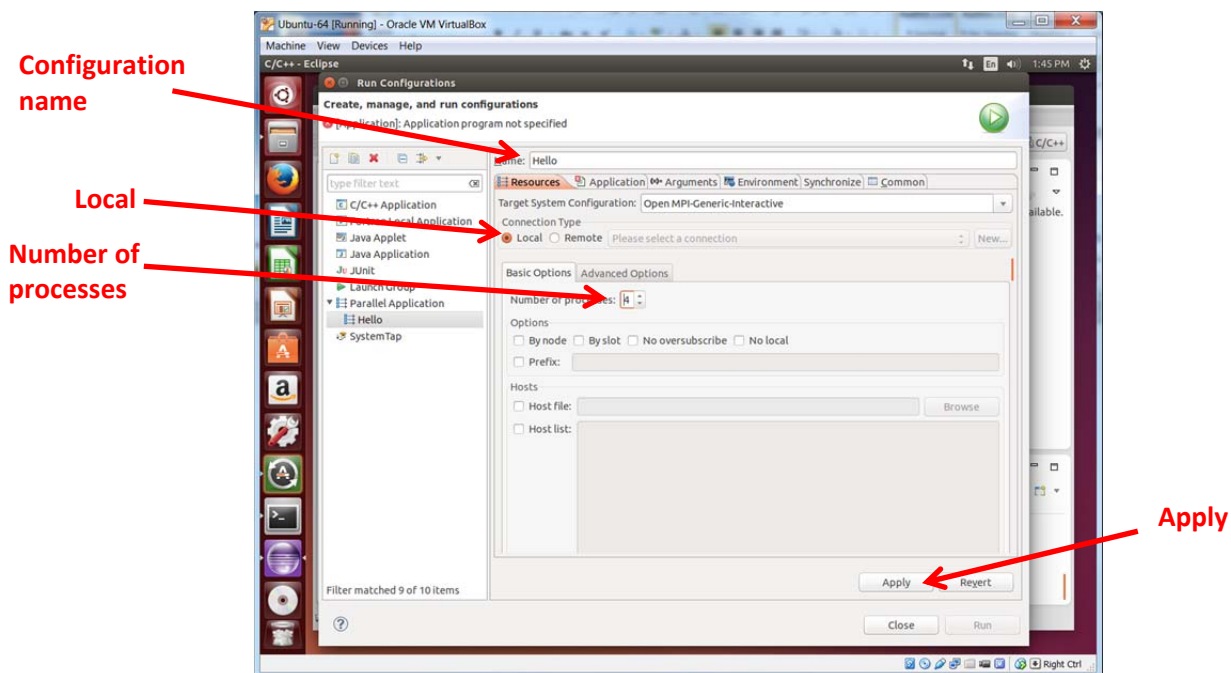
To execute the program, first the **Run Configurations** need to be set up that specify local execution, the software environment, etc. Select **Run Configurations**



Select “Parallel Application” and click the new configuration button:

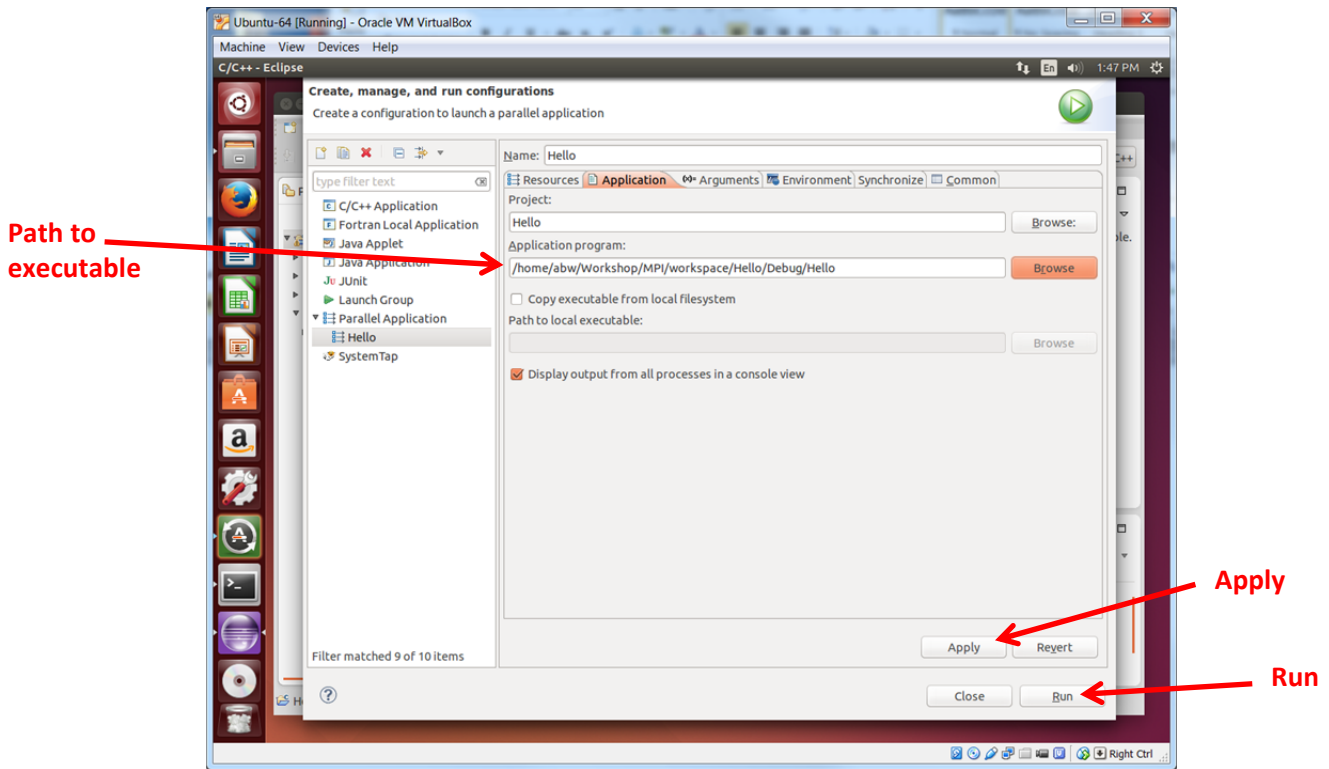


Create a new run configuration called **Hello**. In the *Resource* tab, select the Target Type as “**Open-MPI-Generic-Interactive**”, the connection type as “**Local**”.⁴ Set the number of processes to say 4. Apply.

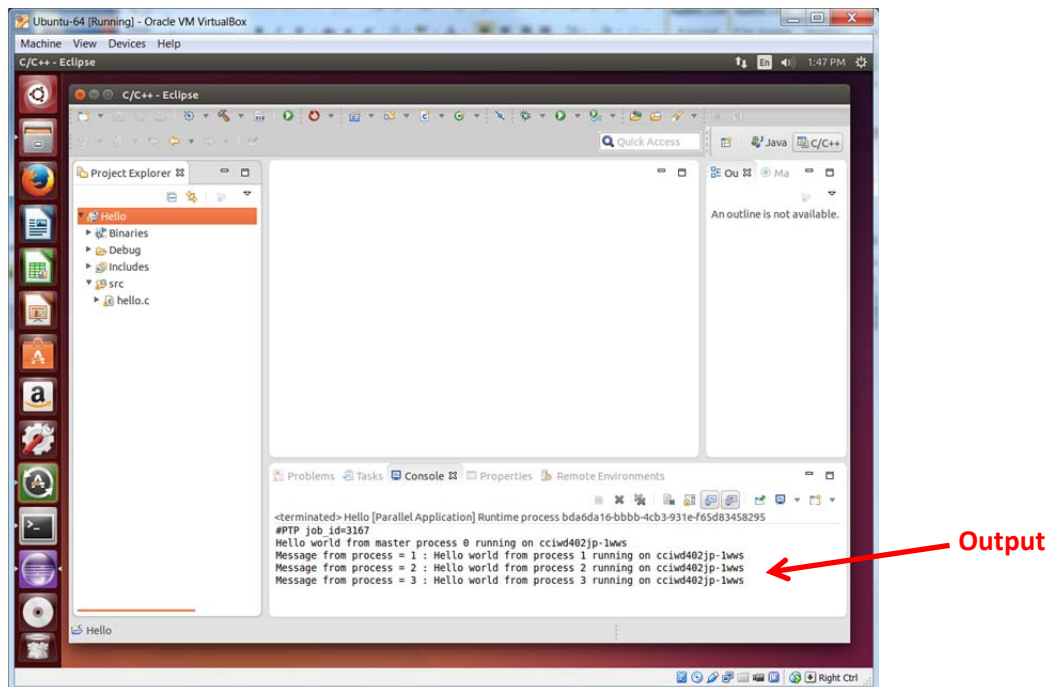


⁴ Selecting “Local” will generate a message confirming you want this and create local resources to do this. Select “Don’t ask to run command again for this configuration” in the Run Command message when it appears to stop the message. The message is most relevant when doing both local and remote executions.

In the *Application* tab, set the Project name to “Hello” and browse for the path to the executable (... /workspace/MPI/Debug/Hello).



Click **RUN**. (If **RUN** is grayed out, there are build or compile errors that prevent execution.)



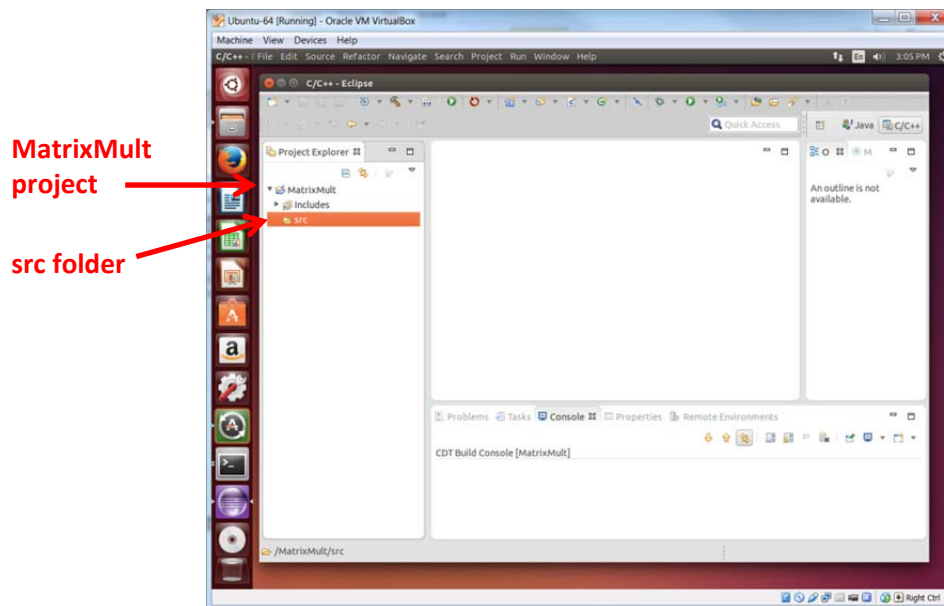
Include in your submission document for Task 1:

A screenshot or screenshots showing executing the hello world program with its output through Eclipse

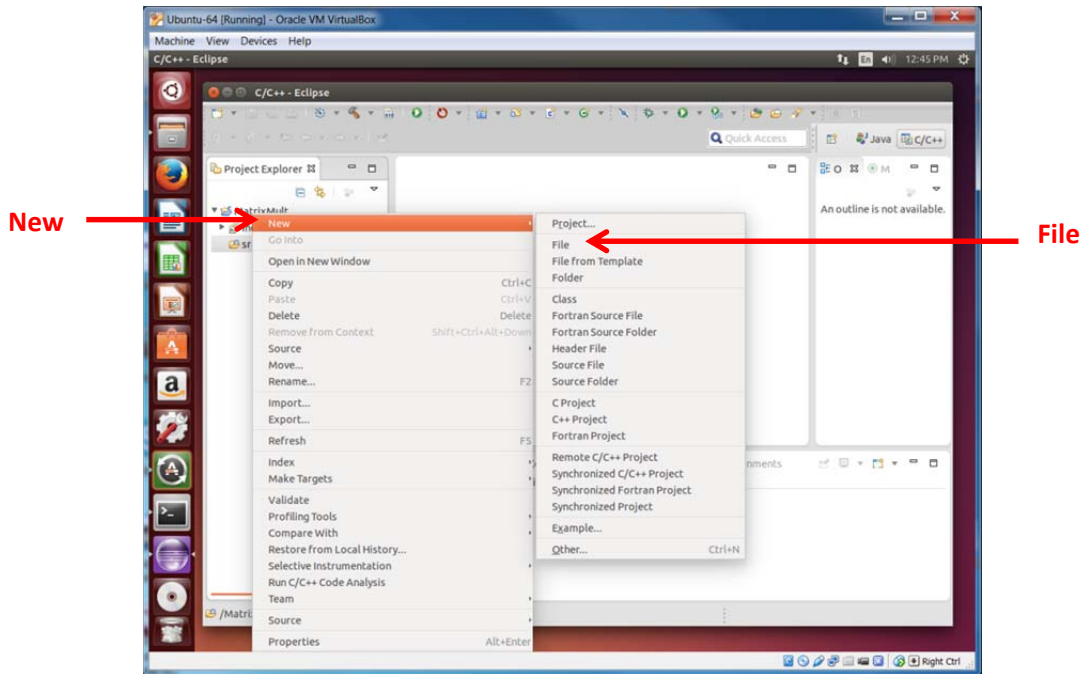
Task 2 Creating and executing Matrix Multiplication program through Eclipse

The process for the matrix multiplication code is similar. We will go through the steps, this time not copying the source file but referring to it in the original location, which generally is a better approach. One additional step for the matrix multiplication program is to add an argument specifying the input file **input2x512x512Doubles** in the Run Configuration.

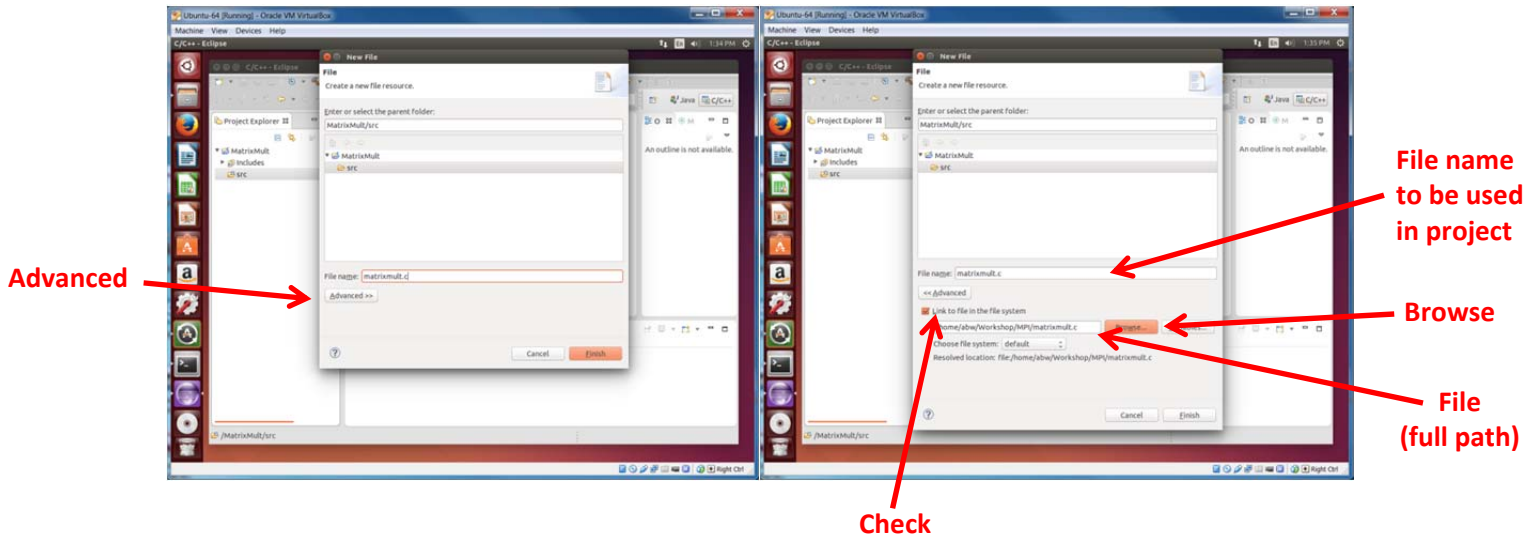
Create an MPI project called **MatrixMult** and a source directory called **src**:



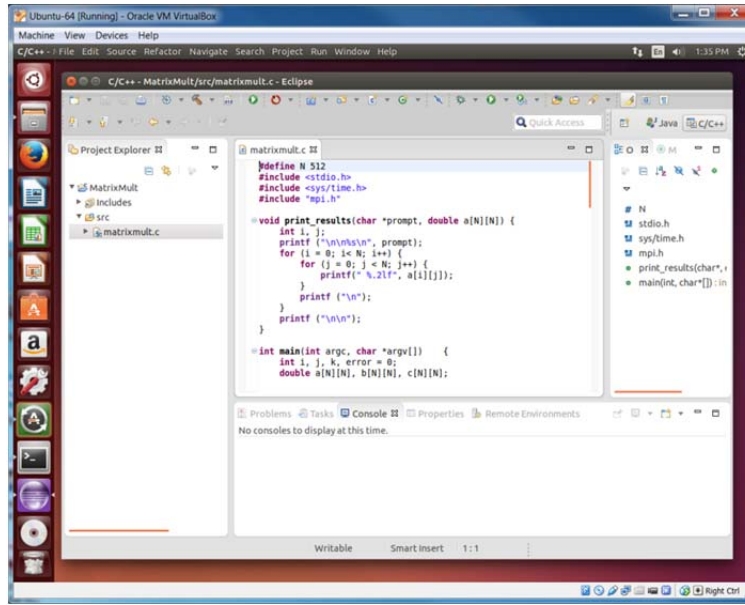
Now we will link source file **matrixmult.c** rather than copy it. Select the source folder **src**, right click, and select **NEW > FILE**:



Click on **Advanced**>> and enter the file name (**matrixmult.c**), check **“Link to file in the file system”** box and enter the full path to the source file (browse for file) and **Finish**:

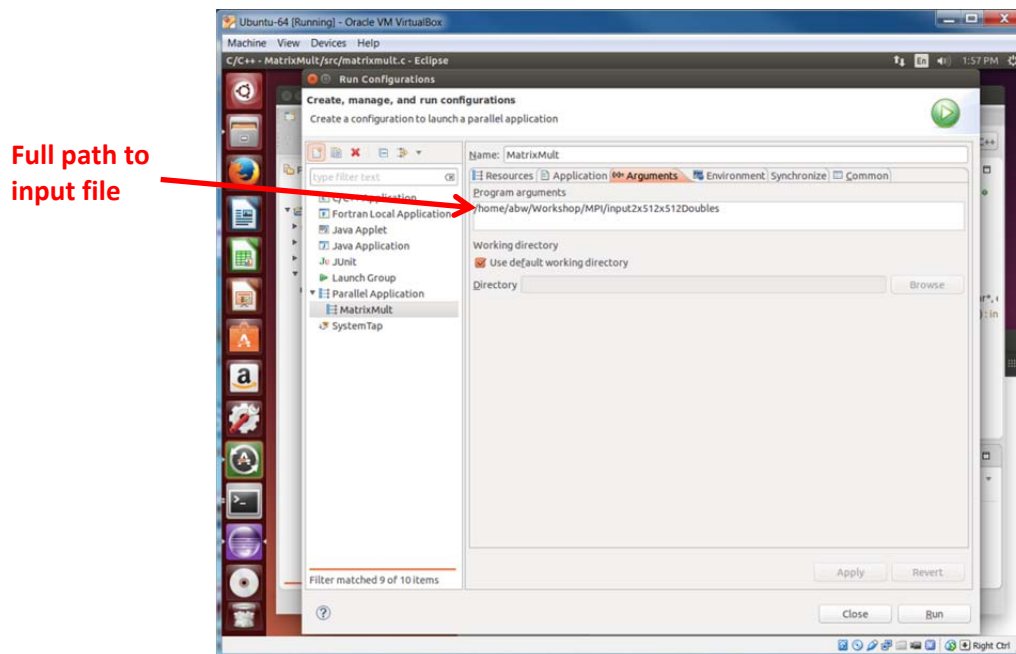


You should now see the source file:

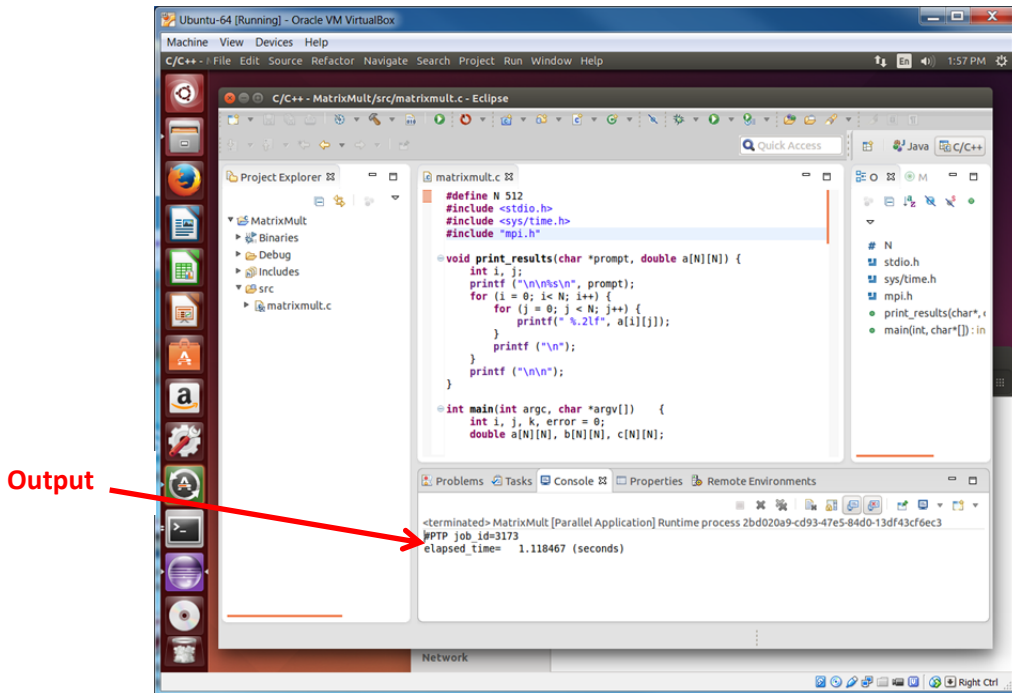


Run Configuration. Now create a run configurations (**MatrixMult**). As before, in the *Resource* tab, select the Target Type as “**Open-MPI-Generic-Interactive**”, the connection type as “**Local**”. Set the number of processes.

In the *Arguments* tab, add the full path to the input file **input2x512x512Doubles** as an argument:

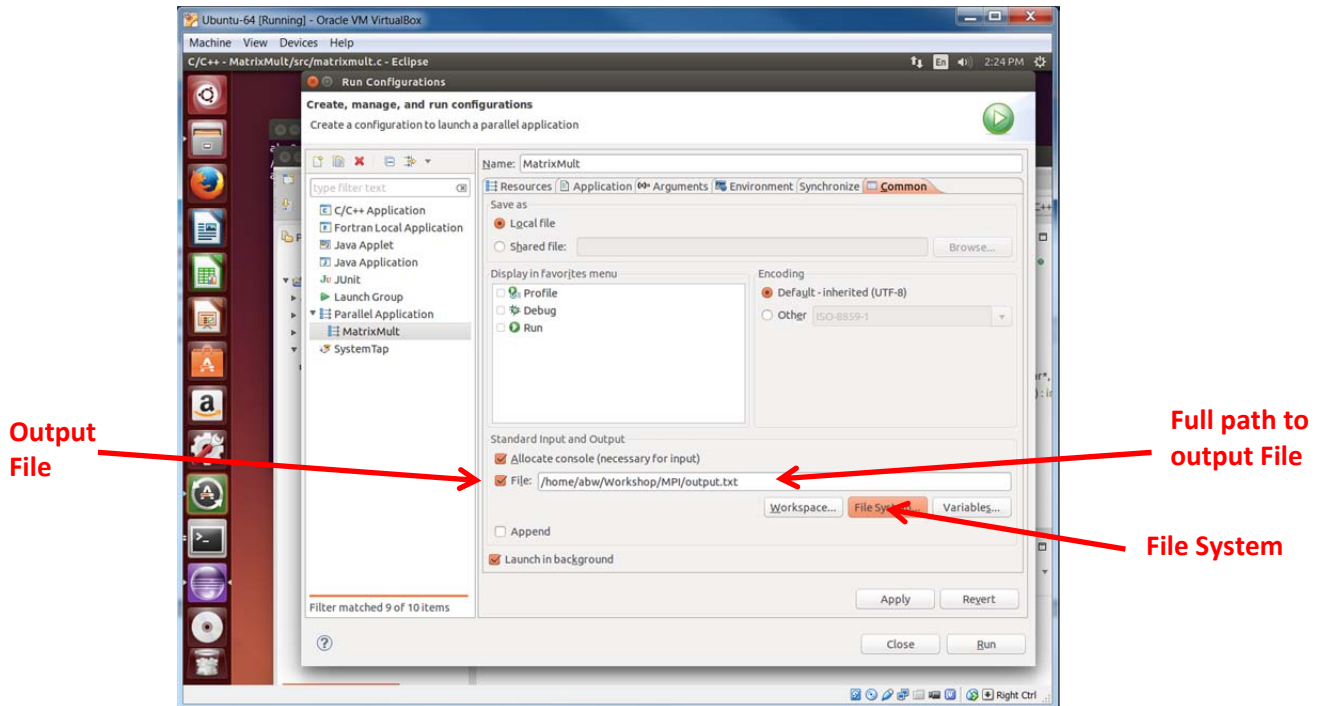


Make be sure to comment out the **print_result()** function call so that the output of the resulting matrix is not printed to the console. Then run the project:



Re-directing Output

To check that the code produces the correct answer, the print statement is uncommented and the output re-directed to a file, which can then be compared with the provided output file, `output512x512Mult`. Re-directing output in Eclipse is done, *not as an argument*, but in the *Common* tab. Check **File** and **File System**. Browse for the file and select:



Uncomment the `print_result()` function call to output the result matrix and run the project. Using the `diff` command will have to be done on the comment line as before.

Include in your submission document for Task 2:

1. A screenshot or screenshots showing executing matrix multiplication program with its output through Eclipse
2. Screenshot or screenshots using the `diff` command showing output is correct.

Part 3 Using a remote cluster (Graduates 30%, Undergraduates 30%)

Now we will use the UNC-C cluster, as specified on the course home page. Details on using the cluster can be found on the course home page. *Carefully review these notes.*

Task 1 Executing MPI programs

Connect to the remote cluster and make a directory in your home directory called `MPI` that will be used for the MPI programs, and `cd` into that directory. Transfer all your MPI source programs `hello.c` and `matrixmult.c` from the previous parts to that directory.

Compile and execute the MPI programs:

- Hello world program, `hello.c`
- Matrix multiplication program, `matrixmult.c`

on the remote cluster with 1, 8, 16, and 32 processors, using four servers, `cci-gridgw.uncc.edu`, `cci-grid05`, `cci-grid7`, and `cci-grid08`. Establish the results are correct.

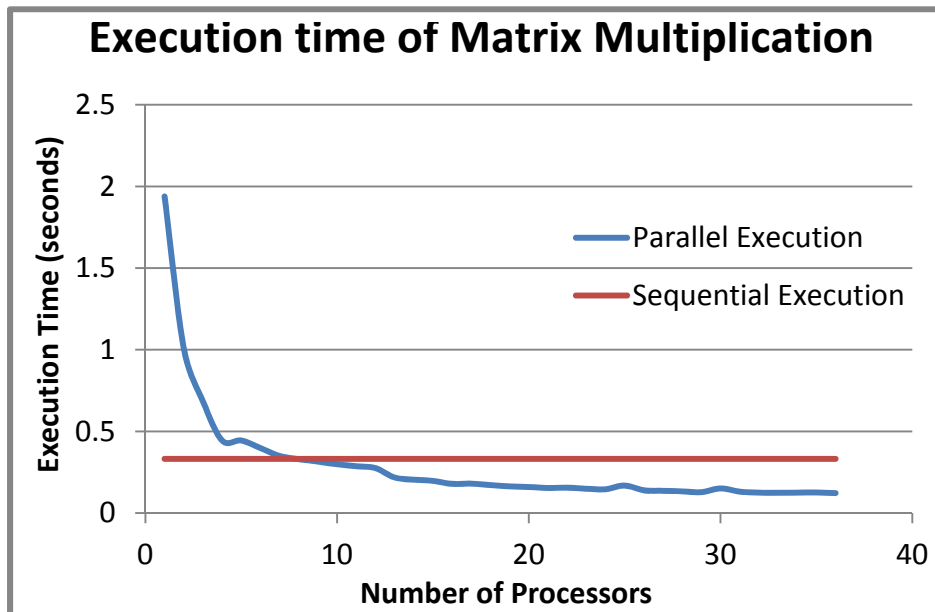
Include in your submission document for Task 1:

1. A screenshot showing the execution of the `hello.c` program on the remote cluster from your account
2. A screenshot showing the execution of the `matrixmult.c` program on the remote cluster from your account
3. For matrix multiplication, show the results of running the `diff` command comparing the parallel output with the sequential outputs

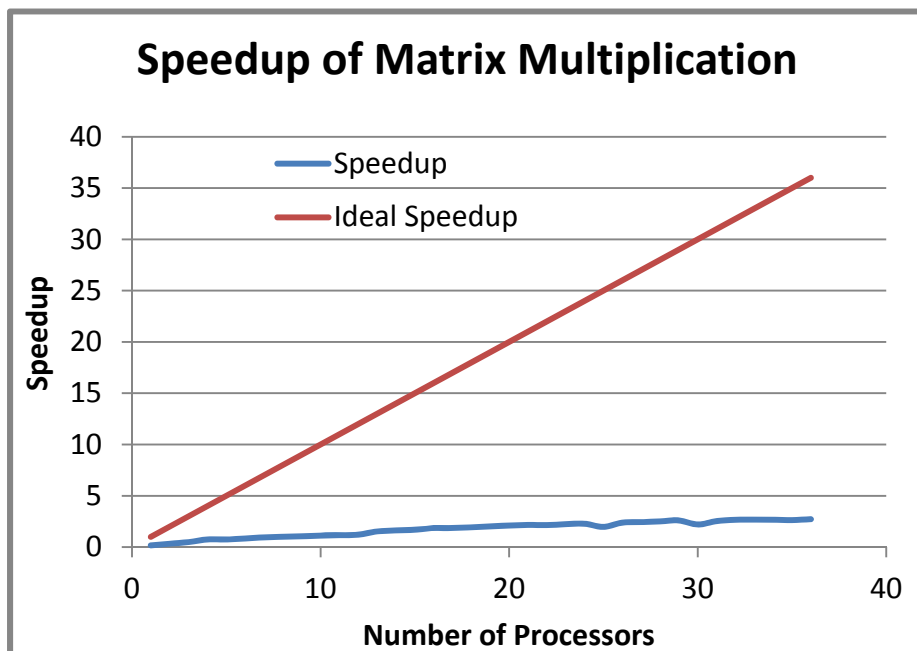
Task 2: Record and analyze results

Record the elapsed times for the parallelized program running on the different number of processors as well as the elapse time for the sequential version. Create a graph of these results using a graphing program, such as a spreadsheet. You have to create a graph of the execution times compared with sequential execution and the speedup curve with linear speedup. These graphs should look something like the figures but the shape of the curves do not. Your curves

may show something entirely different. The figures below are just examples. Make sure that you provide axes labels, a legend (of there is more than one line) and a title to the graphs. Include copies of the graphs in your submission document.



Example Execution Time Graph



Example Speedup Graph

Include in your submission document for Task 2:

1. A copy of your matrix multiplication program
2. A copy of your execution time and speedup graphs
3. A screenshot or screenshots showing:
 - a. Compilation of the program using **mpicc**
 - b. Results of running the **diff** command comparing the parallel output with the sequential outputs
4. Copies of your graphs

Part 4 Changing the matrix multiplication program to handle any value of N. For graduates students (10%) Extra credit for undergraduates (+10%)

The matrix multiplication code, as written, only works if **P** divides evenly into **N**. Modify the program to handle any value of **N**. The stored matrices and messages should not be any larger than necessary, i.e. simply padding out arrays with zeros is not acceptable.

Include in your submission document for Part 4:

1. Matrix multiplication program code that handles any value of **N** and screenshot showing it executing correctly.

Grading

Every task and subtask specified will be allocated a score so make sure you clearly identify each part/task you did. Make sure you include everything that is specified in the “Include in your submission document” section at the end of each part. **Include all code, not as screen shots of the listings but complete properly documented code listing in the report.**

Assignment Submission

Produce a *single pdf* document that show that you successfully followed the instructions and performs all tasks by taking screenshots and include these screenshots in the document. Submit by the due date as described on the course home page.