Assignment 1: Matrix Multiplication using MPI

Problem Description

In this assignment, you are supposed to calculate the product of two matrices A (of size N*32) and B (of size 32*N), which should be an N*N matrix.

Specifically, you are supposed to

- Design a parallel scheme for computing matrix multiplication, including how to:
  - Separate the task into divisions and let each process finish one division
  - Transfer data between processes
  - Form the output matrix using the result of each process.
- Implement the parallel mechanism with 3 different types of communications in MPI RESPECTIVELY (that's to say you have to write 3 programs that differ in communications between processes):
  - Blocking P2P (point-to-point) communication
  - Collective communication
  - Non-blocking P2P communication.
- Observe the running time of your programs; change some of the parameters to see how it is associated with N and communication type (and number of processes, if available).
  - Try to prove that the theoretical time complexity is $\mathrm{O}(N^2)$.
- Finish a report that meets the requirements.

Deadline

23:59 Feb 9 (Friday)

Programming Tasks

Step 1: Install MPI libraries on your computer and make sure it works (for example, make your computer successfully compile and run the "MPI hello world" program on Page 15 of MPI-part1.pdf). Please find the tutorials for both Windows and Mac users (please find posted files).

- For Visual Studio/other IDE users, please create a new C/C++ project, configure it following the tutorials and create a C code file inside it.
- For Mac users without IDE, please create a C code file (using vim in Terminal or text editors like Sublime).

Step 2:

Create a function, Multiply_serial() to perform multiplication without parallelism. Example code (assuming that matrix elements are row-major stored in an array):

```
void Matrix_Multiply(float *A, float *B, float *C, int m, int n, int p)

{
    int i, j, k;
    for (i = 0; i < m; i++)
        for (j = 0; j < p; j++)
        {
            C[i*p + j] = 0;
            for (k = 0; k < n; k++)
                C[i*p + j] += A[i*n + k] * B[k*p + j];
        }
}
```

Create a function, IsEqual(), that examines if two matrices are exactly the same. The function should have the form:

```
int IsEqual(float *A, float *B, int m, int n)
```

In the function, we need to check if every pair of corresponding elements in A and B are the same. Once a discrepancy is found, the program can instantly return 0 (false). Return 1(true) if A and B are equal (no discrepancy is found). Later the function will be used to verify the results of your MPI programs.

- Tip: you can write a simple main() function to make sure the IsEqual() function works fine.

Step 3: Implement your parallel algorithm in main() function, using blocking P2P communication (MPI_Send/MPI_Recv) between processes. main() should at least:

- Initialize and finalize MPI environment
- Let Process #0 generate A (of size N*32) and B (of size 32*N) using random numbers in [0, 1].
- Implement communications between Process 0 and other processes
- Compute A*B using your parallel scheme (assuming the result is stored at C)

- Let Process #0 compute A*B by Multiply_serial() function in Assignment 1 (assuming the result is stored at C_serial)
- Let Process #0 check if C and C_serial are equal (using IsEqual() function) so as to verify your parallel algorithm
- Let Process #0 get the running time of the two computations above

Step 4: Compile and run your program.

- For Windows users: use Visual Studio to compile your program, and use “mpiexec” in command console (Press Win+R, type “cmd” and press Enter) to run it.
- For Mac users: open terminal, use “mpicc” to compile your program, and use “mpirun” to run it.
- Please refer to appendix for more details on how use mpicc/mpiexec/mpirun command.
- Be patient: it is quite common to fail in your first attempts, especially in performing MPI communications. Use breakpoints and printf() wisely for debugging.
- About number of processes: you can use 2 or 4 processes (according to how many cores your computer CPU has) in the assignment. Most i5 CPUs on laptops have 2 cores, hence there might be errors when running 4 processes on them (especially on Mac). Mac users can follow the tutorial to fix the problem.

Step 4.5: Back-up your program. **(Please submit the code files at this step)**

Step 5: Take down the running time.

- If your parallel algorithm gives correct result, you can then store the result of serial computation at C instead of `C_serial` because we do not need more verifications. The definition of `C_serial` can be removed/commented out, and the memory cost will be thus reduced.
- For Windows users: you are recommended to compile and run programs using “Release Mode” after ensuring your program works well.

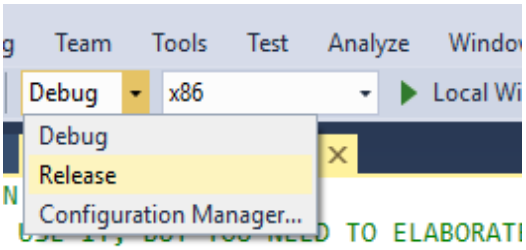


Figure 2: how to switch between Debug/Release modes in Visual Studio

- Try different values of *N* (and number of processes, if applicable) and take down the running times of both serial and parallel methods respectively.

Step 6: Copy your code at Step 4.5 and implement parallelism with other types of communications: collective communications and non-blocking communications. Repeat Step 3 to Step 5 (except that you no longer need to take down the running time of the serial method again).

Report

Please submit a report that at least includes:

- Elaboration of the parallel algorithm you have implemented in terms of design principle and implementation.
    - A good algorithm description can make readers understand how your algorithm works even without reading/running the codes.
- Summary of the differences among your 3 versions of parallel algorithm in terms of principle and implementation.
- Discussion on restrictions of N in your program, if there is any (for example, the sample code assumes N to be a multiple of number of processes). Explain why there should be such restriction.
- Lists of practical running times (together with the time-N curves) of the four methods provided different values of N: serial (Assignment 1), MPI with blocking P2P communications, MPI with collective communications, MPI with non-blocking P2P communications.
- Your findings on the running times you have observed, like if they agree with theoretical time complexity, if being parallel can accelerate the computation and so on. Try to explain your findings.

Submission

Please submit a zip package including:

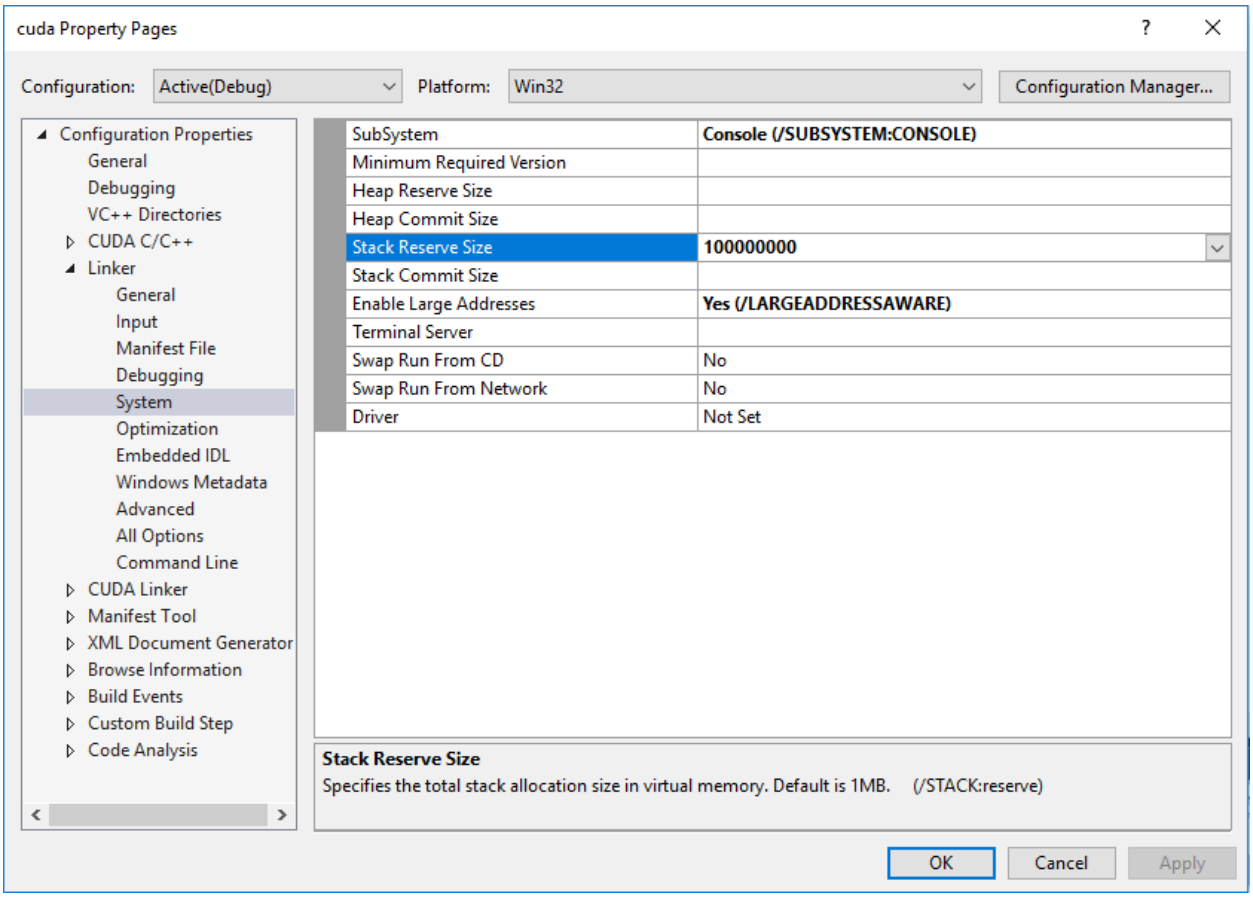
- 3 **C code files**, each corresponding to an MPI communication method.
    - Please attach necessary comments to your code
- A **report** (.doc/.docx/.pdf) that meets the requirement above.

Please finish all the work independently.

Appendix: to expand stack size in Visual Studio

The program will crash if N is too big. That is because the program limits the memory it can use, called "stack". A very large matrix may overflow the stack (namely "stack overflow error").

You can adjust stack size in Visual Studio. Click "Project -> project properties", and select "Linker->System" on the left. The dialog is shown below. Then on the right, change "stack reserve size" to the stack size you want (in bytes), for example, 100000000 (about 100MB), and set "enable large addresses" as "Yes".


cuda Property Pages
Configuration: Active(Debug)
Platform: Win32
Configuration Manager...
Configuration Properties
General
Debugging
VC++ Directories
CUDA C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Windows Metadata
Advanced
All Options
Command Line
CUDA Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Code Analysis
SubSystem
Console (/SUBSYSTEM:CONSOLE)
Minimum Required Version
Heap Reserve Size
Heap Commit Size
Stack Reserve Size
100000000
Stack Commit Size
Enable Large Addresses
Yes (/LARGEADDRESSAWARE)
Terminal Server
Swap Run From CD
No
Swap Run From Network
No
Driver
Not Set
Stack Reserve Size
Specifies the total stack allocation size in virtual memory. Default is 1MB. (/STACK:reserve)
OK
Cancel
Apply