

# CS442/ECE432: Homework 4

March 25, 2008

## General

- The answers to this homework are **due April 1, 2008**.
- Submit via e-mail to `riesen@cs.unm.edu` (Mail it before class on the 1st.) Your subject line must say: “Homework 4 Submission”.
- Keep your answers (and programs) succinct.
- Obey the University rules on **plagiarism**. In particular, do use libraries and the web to find information you need to answer the questions, but do not copy whole answers or programs. Reference your sources. The work you turn in must be **your** work. Ask me for help if you are stuck.

The whole homework will receive **zero points**, if I detect cheating. Any offenses after that will result in dismissal from class and a **report** to the University.

- You are allowed to use the example programs on the class web site as a starting point.

## Grading

With so many students and programs, we want to make the testing and grading as automated as possible. It is therefore very important that you follow the rules below exactly. If we have to manually inspect your program and modify it to work with our scripts, we will deduct points.

### One File Per Program

For each of the exercises below, create a single file name `e_xx.c`, where the `xx` is 01 for the first exercise, 02 for the second, etc.: “`ex_01.c`”, “`ex_02.c`”, ...

For C++ files use “`ex_01.cc`”, “`ex_02.cc`”, ...

### Clean, Readable Code

Your code must be clean and readable. Be consistent with you indentation, use descriptive names for your variables and constants, etc.

Your program must compile without warnings (we use the `-Wall` flag for `gcc`). Your program must be valid MPI; e.g., there must be no pending messages when `MPI_Finalize()` is called.

Some of this is subjective, but we will subtract up to 10 points for code we cannot read or understand, even if it works.

## Languages

Write your programs in C or C++. They must compile with the GNU C compiler and link against an MPI-2 library. Even when using C++, you must use the C bindings for MPI; i.e., use `MPI_Send()` not `MPI::Comm::Send()`.

## Tar File

Put your source code, no executables or object files, into a single tar file and compress it. Name your archive “Hwk4.tgz” It must untar into the current directory; i.e., no sub directories. You can use zip instead of tar. In that case name your file “Hwk4.zip”

The archive must contain a file named “AUTHOR” which contains your name. Thus, a complete archive for this homework would contain the following files **only**: ‘ex\_01.pdf’, “ex\_02.c”, ‘ex\_02.pdf”, “ex\_03.c”, “ex\_04.c”, ‘ex\_04.pdf”, and “ex\_05.pdf”, and “AUTHOR”.

## Documentation

Any documentation and solutions that are not programs must be in pdf or plain ASCII text.

## Assignments

### Exercise 1: Matrix Multiply Performance 1

The matrix multiplication program we developed in Section 5.1.3 produces the correct result, but we do not know whether it is efficient. For this exercise, evaluate that program’s performance. Does it speed up when more threads are assigned to work on it? You can use the Unix `time` command to measure the execution time of the program:

```
./ex_pmatmul -p 4
```

The above command uses the default number of four threads to calculate the result. Measure how long it takes with 1, 4, 9, 25, and 36 threads. Ideally, you will run this experiment on a machine with at least two CPU cores.

**20 points** For the measured results with explanation

### Exercise 2: Matrix Multiply Performance 2

Exercise 1 may not be accurate enough because we are also measuring the time it takes to create and initialize the matrices. Modify the program from Section 5.1.3 and measure only the time after the matrices have been allocated and initialized, until all threads have completed.

Compare your results with the ones you have obtained using the `time` method described in Exercise 1. Is there a difference?

**10 points** For the modified version of the matrix multiply program

**10 points** For the comparison and explanation with the method used in the first exercise

**Exercise 3: Matrix Multiply with Fewer Restrictions**

Modify the program from Section 5.1.3 such that the number of threads ( $p$ ) and dimension of the matrices ( $n$ ) have the restrictions:  $p = i^2$  and  $n/i$  being an integer value, removed. You may still assume all three matrices are of size  $n \times n$ , and that  $p \leq n$ .

Test your program using the following number of threads and dimensions. Of course, it must work with much larger numbers as well. The numbers below are suggested because they will test your thread assignment strategy, but are small enough to print detailed results that can be easily verified.

$p$ threads	$n \times n$
9	$3 \times 3$
2	$4 \times 4$
4	$4 \times 4$
5	$3 \times 3$
5	$5 \times 5$
12	$6 \times 6$

Your program should print out how many elements of matrix  $C$  have been assigned to each thread.

**20 points** For a working program

**Exercise 4: Mutex Overhead**

In the program `Code/ex_pmaxfind.c` from Example 5.6 we found the local maximum first and at the end updated the global value when necessary. Create a second version of `Code/ex_pmaxfind.c` that does not use a private variable to hold the maximum; i.e., update the global maximum every time you find a larger value.

Measure and compare the two versions. Is the frequent locking of the global variable really that much more expensive? Does the cost change with the number of threads?

**10 points** For a modified version of the program `Code/ex_pmaxfind.c` that does not use a private variable to store the current maximum

**10 points** For the results and explanation of the comparison.

**Exercise 5: Matrix**

In program `ex_pmaxfind.c` of Example 5.6 we acquire the mutex lock before we do the if-test on line 177. Could we move the `pthread_mutex_lock()` and `pthread_mutex_unlock()` instructions inside the if clause? This would help with performance, since the thread would only try to acquire the lock when it actually intends to update the global value.

Explain your answer.

**10 points** For a written explanation.