MVP Assignment 1

Due date: 8/3/2010

1 Question

Exercise 1: Exercise in the book.

Q: *Exercise 1, chapter 1.*

A: 1.Thread: When dividing a program, threads can be used to run the program in parallel.

2.Race condition: A race condition occurs when two threads race to influence the output first.

3.Mutex: A mutex is used to lock some lines of code so that only one thread of a time is allowed to execute the lines.

4.Lock Contention: This occurs when two threads tries to acquire a lock at the same time.

5.Grandularity: Is the size of the subproblems.

6.False Sharing: If two threads shares a cache line, they still need to reload the cache line every time they try to access their value in case that the other thread might have changed something in the cache line.

2 Matrix Multiplication

Exercise 2: In this first assignment we will get warmed-up with C and experiment with the influence of caches on an apparently simple algorithm. The standard formula for computing the matrix multiplication $C = A * B$ between two matrices of size $n * n$[1] is:

$$(c_{ij})_{1 \leq i,j \leq n} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

Q: *Write down the straight-forward algorithm for computing this formula.*

Q: *Implement this algorithm in matrix_fibo.c.* This program serves as a test program to help you develop and test your function. It is computing Fibonacci numbers using two techniques and it compares the results for correctness. Keen students are invited to look at the slides that explain the example (given in the source code). You do not need to look at this to complete the assignment.
Notes: `const` means you are not supposed to write the data (see the source code). To address element (i, j) in a matrix of dimention n declared as `int* a`, use `a[i*dim+j]`.

```
1
2  void matrix_mult(const int * a, const int * b, int * c, size_t n)
3  {
4      int i, j, k;                /* You will need at least these. */
5      assert (a != c && b != c); /* Check precondition. */
6
7      /* Write here your matrix multiplication . */
```

[1] We stick to square matrices for simplicity here.

```
        for (int i = 0;i<n;i++)
        {
                for (int j = 0;j<n;j++)
                {
                        int temp = 0;
                        for (int k = 0;k<n;k++)
                        {
                                temp += a[i*n+k]*b[k*n+j];
                        }
                        c[i*n+j] = temp;
                }
        }
}
```

Listing 1: Matrix multiplication function.

3 Re-arranged Matrix-Multiplication

Exercise 3: You will test subsequent matrix-multiplication algorithms using another test program. It is basically generating a random matrix, inverting it, then it multiplies the original with its inverse and checks that it is the identity.

Q: *Transfer your matrix multiplication implementation to pmatrix.c in the mat_mult1 function.* You will need to change the data type to double.

Q: *Why is the identity check not using simple equality tests like* $x == 0$ *or* $x == 1.0$*?*

A: The datatype is double, we use the constant PRECISION to determine if the double is as close to 1 as we want.

Q: *Change the PIVOT parameter to 0. Explain the loss in precision.* Turn back the parameter to 1 after that question.

A: When PIVOT is set to 0, we don't use pivoting in the Gaussian elimination, therefore we have a risk of numerical instability e.g. when dividing with small values.

Q: *What is the memory access pattern of your matrix multiplication implementation?*

A: The a matrix is read sequential and the b matrix is read column wise.

Q: *Why is it bad for the cache?*

A: When reading the b matrix it makes strides, which is bad for the cache.

Q: *Modify your matrix multiplication implementation to make it more cache friendly.* This modification should only concern the memory access pattern. One very simple change in the loops is enough.

```

void mat_mult2(const int* a, const int* b, int* c, size_t n)
{
    int i, j, k;                /* You will need at least these. */
    assert (a != c && b != c); /* Check precondition. */
        for (int k = 0;k<n;k++)
        {
                for (int i = 0;i<n;i++)
                {
                        int temp = 0;
                        for (int j = 0;j<n;j++)
                        {
```

```
13                                  temp += a[i*n+k]*b[k*n+j];
14                              }
15                              c[i*n+j] = temp;
16                          }
17                      }
18
19  }
```

Listing 2: Matrix multiplication function.

Q: *Test your program with sizes 100, 200, ... 1000* and report the relative improvement compared to the previous version.

A:

- With 100

  Multiplying1...done! real: 0.008409s, user: 0.01s, sys: 0s, idle: 0%

  Multiplying2...done! real: 0.007974s, user: 0.01s, sys: 0s, idle: 0%

- With 200

  Multiplying1...done! real: 0.061398s, user: 0.06s, sys: 0s, idle: 2.27695%

  Multiplying2...done! real: 0.062984s, user: 0.06s, sys: 0s, idle: 4.73771%

- With 1000

  Multiplying1...done! real: 11.7579s, user: 11.7s, sys: 0.02s, idle: 0.32243%

  Multiplying2...done! real: 7.76633s, user: 7.73s, sys: 0s, idle: 0.46784%

4 Block-Matrix Multiplication

Exercise 4: We want to improve our implementation to make it more cache friendly. The idea of the block-matrix multiplication is to compute the multiplication not by element-wise multiplications but by blocks-wise multiplications. Thus the formula becomes:

$$(C_{ij})_{1\leq i,j\leq n} = \sum_{k=1}^{n} A_{ik} * B_{kj}$$

where A_{ij}, B_{ij}, and C_{ij} are block sub-matrices of respectively A, B, and C. The multiplication between blocks is the standard element-wise multiplication. We note that thanks to the associativity of additions this formulation is equivalent to the original algorithm.

Q: *What is the point of this reformulation?*

A: ..

Q: *Write down the corresponding algorithm.*

Q: *Implement this algorithm.* Use the BLOCK parameter to tune the size of the blocks.

```
1
2  void mat_mult3(const int* a, const int* b, int* c, size_t n)
3  {
4      int i, j, k;                /* You will need at least these. */
5      assert (a != c && b != c); /* Check precondition. */
6
7      /* Write here your matrix multiplication . */
8
9  }
```

Listing 3: Matrix multiplication function.

Q: *Experiment with the BLOCK parameter and find a good value for your machine. Why is it a good value for your machine?*

A: ..

Q: *Test your program with sizes 100, 200, ... 1000* and report the relative improvement compared to the previous version.

A: **Exercise** 5: [Optional] Micro benchmarks are relatively small benchmark programs whose purpose is to stress few aspects of a system, here memory. The file `bench.c` is a simple micro benchmark test file.

Q: [Optional] Study the file, execute it, and deduce the cache hierarchy of your system from the output. You should execute it several times to get decent results.

A: ..

5 Authors

I/We have solved these exercises independently, and each of us has actively participated in the development of all of the exercise solutions.

Name 1	Name 2
.....	.....
Signature	Signature
Name 3	Name 4
.....	.....
Signature	Signature
Name 5	Name 6
.....	.....
Signature	Signature
Name 7	Name 8
.....	.....
Signature	Signature