PAR – 2^{nd} In-Term Exam – Course 2014/15-Q1

December 17th, 2014

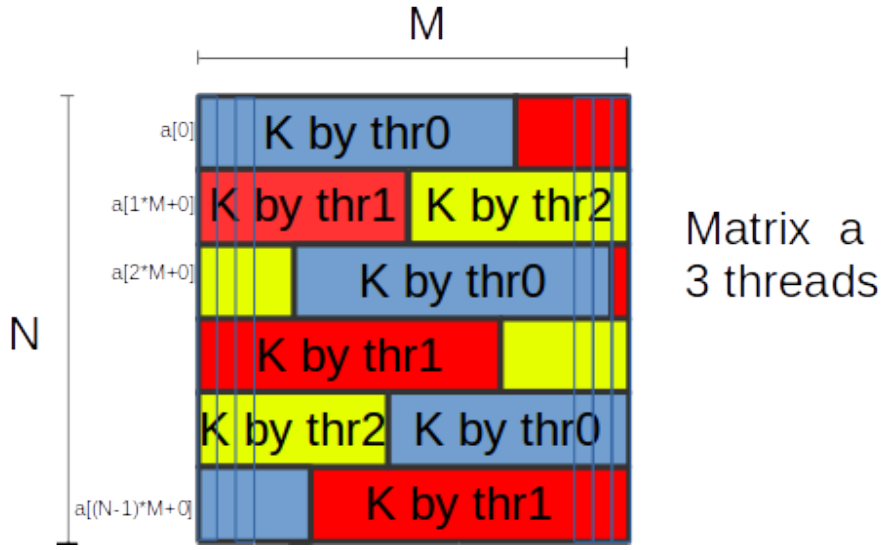
Question 1 (4 points)

Given a $N \times M$ matrix a that is updated by the following code:

```
...
for(i=0; i<N; i++)
  for (j=0; j<M; j++)
  {
     a[i*M+j] += foo(i,j);
  }
...
```

Assume that `foo(i,j)` does not have any memory access. `foo(i,j)` only computes an expression with no side effects. `NT` is the number of threads to be used in the parallel regions. Answer the following questions:

1. (1 point) Implement an OpenMP version of the code using Iterative Task Decomposition Strategy so that iterations are distributed as it is shown in the following figure for an example for $NT = 3$ threads: chunks of K consecutive elements of matrix `a` in memory, in the same row or not, are processed by one thread T (`thrT` in the figure). In this strategy, each thread is STATICALLY assigned chunks of K statements "`a[i*M+j] += foo(i,j)`" as it can be seen in the figure.



Solution:

```
...

#pragma omp parallel num_threads(NT)
#pragma omp for collapse(2) schedule(static,K)
for(i=0; i<N; i++)
  for (j=0; j<M; j++)
  {
     a[i*M+j] += foo(i,j);
  }

...
```

2. (1 point) Implement an OpenMP version of the code using Cyclic Data Geometric Decomposition by rows of the output matrix `a`. **Do not use `omp for`**.

   Solution:

```
...

#pragma omp parallel num_threads(NT) private(i,j)
{

int id = omp_get_thread_num();
for(i=id; i<N; i+=NT)
  for (j=0; j<M; j++)
  {
     a[i*M+j] += foo(i,j);
  }

}

...
```

3. (1 point) Implement an OpenMP version of the code that performs a Data Geometric Decomposition of matrix `a` in blocks of $K \times M$ elements of matrix `a` (a block of K rows of matrix `a`) in such a way that threads are dynamically assigned a new block of data of matrix `a` as soon as they finish their work, and there is still a block pending of being processed. **Do not use `omp for`**.

   Solution:

```
...

#pragma omp parallel num_threads(NT)
#pragma omp single
{



for(ii=0; ii<N; ii+=K)
{
 #pragma omp task firstprivate(ii) private(j,i)
 for(i=ii; i<min(ii+K,N); i++)
 {
  for (j=0; j<M; j++)
  {
     a[i*M+j] += foo(i,j);
  }

}

...
```

4. (1 point) Implement an OpenMP version of the code that performs a Data Geometric Decomposition of matrix `a` in blocks of $K \times K$ elements of matrix `a`. Blocks should be statically assigned between **ONLY TWO THREADS** in a chess board fashion. Assume that $N = M$ and $N\%K$ is zero. **Do not use `omp for`**.

Solution:

```
...

#pragma omp parallel num_threads(2) private(ii,jj,i,j)
{

int id= omp_get_thread_num();

for(ii=0; ii<N/K; ii++)
{
 for(jj=(ii%2)?(id+1)%2:id; jj<M/K; jj+=2)
 {
   for(i=ii*K; i<(ii+1)*K; i++)
   {
    for (j=jj*K; j<(jj+1)*K; j++)
    {
     a[i*M+j] += foo(i,j);
    }
   }
 }
}

...
```

Question 2 (2 points)
Assume you get this piece of OpenMP code:

```
void work(int w)
 {
 \\ Some real work here
 ...
 }

main() {
  omp_lock_t lock;
  omp_init_lock(&lock);
  #pragma omp parallel
  {
    int k;
    int nt = omp_get_num_threads();

    #pragma omp for
    for (k = 0; k < 32; k++) {
       #pragma omp task
       {
        int w = k%nt;
        omp_set_lock(&lock);
        work( w );
        omp_unset_lock(&lock);
       }
    }
  }
  omp_destroy_lock(&lock);

}
```

1. If OMP_NUM_THREADS=4, how many threads and tasks are launched during execution?

   Solution: The number of created threads is 4, as the environment variable OMP_NUM_THREADS states. Tasks are created at the K loop, which is splitted among the 4 existing threads, and each one creates 8 tasks. So 32 tasks total. Threads are responsible to run the tasks. As soon as they finish with one task they look for more and pick the next available one.

2. Is the code deadlock safe?

   Solution: Yes, the code is deadlock safe. All threads that run tasks have to wait for the same lock so they call work() one after the other.

3. If the call to `work ( w )` has no dependences among calls for different values of `w`, could you improve the code in order to allow more parallelism?

   Solution:

   As calls to `work(w)` with different `w` value do not have conflicts, we could create an array of locks so tasks that generate a different `w` value can run the call to `work(w)` in parallel.

```
void work(int w)
 {
 \\ Some real work here
 ...
 }

int main() {
  int i;
  omp_lock_t lock[32];
  for (i = 0; i < 32; i++) omp_init_lock(&lock[i]);
  #pragma omp parallel
  {
    int k;
    int nt = omp_get_num_threads();

    #pragma omp for
    for (k = 0; k < 32; k++) {
       #pragma omp task
       {
        int w = k%nt;
        omp_set_lock(&lock[w]);
        work( w );
        omp_unset_lock(&lock[w]);
       }
    }
  }
  for (i = 0; i < 32; i++) omp_destroy_lock(&lock[i]);

}
```

Question 3 (4 points)

Matrix multiplication can be expressed as multiplications and additions of submatrices. For example, the multiplication $C = A * B$ can be seen as the calculation of their elements or submatrices $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} * \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$
$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$
$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$
$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Note, also, that it can be implemented in a recursive fashion. We have an application that stores matrices as quadtrees, with pointer matrices at the upper levels, and data submatrices (storing numerical values) at the leaves of the quadtree. We have recursive codes implementing matrix multiplication and addition using such quadtrees. We want to provide an efficient parallel implementation of the matrix multiplication based on the sequential versions shown below. The following excerpt of code shows the multiplication $C = A * B$ of matrices stored as quadtrees using routine `QTMat_RecMul`. Symbols a11, a12, ..., d21, d22, have been defined to ease the access to the 4 quadrants in a matrix stored as a quadtree (`qtmatrix` in the code).

```
/*
 * Square matrices of size <= BS are handled with the "classical
 * algorithms".  The shape of almost all functions is something like
 *
 *      if ( n <= BS )
 *          classical algorithms
 *      else
 *          n/= 2
 *          recursive call for 4 half-size submatrices
 *
 */

#define N  1024    /* Problem Size, i.e. multiply matrices of size NxN. */
#define BS   64    /* Data submatix Size. */

typedef union _qtmatrix {
    double **d;
    union _qtmatrix **p;
} *qtmatrix;

/* Let us define symbols a11, a12, ... , d21, d22, to ease the access to
   the 4 quadrants in a qtmatrix, i.e. QTM[i][j], with i,j in [1,2]. */

#define a11 A->p[0]
#define a12 A->p[1]
...
#define d22 D->p[3]

void matmul_submatrix(int N, double *A, double *B, double *C);
void matadd_submatrix(int N, double *A, double *B, double *C);

/* C = A + B */
```

```
void QTMat_RecAdd(int n, qtmatrix A, qtmatrix B, qtmatrix C) {
    if (n <= BS)
        matadd_submatrix(n, A, B, C); /* Add data submatrices. */
    else {
        n /= 2;
        QTMat_RecAdd(n, a11, b11, c11);      /* c11 = a11 + b11 */
        QTMat_RecAdd(n, a12, b12, c12);      /* c12 = a12 + b12 */
        QTMat_RecAdd(n, a21, b21, c21);      /* c21 = a21 + b21 */
        QTMat_RecAdd(n, a22, b22, c22);      /* c22 = a22 + b22 */
    }
}
```

```
/* C = A * B */
void QTMat_RecMul(int n, qtmatrix A, qtmatrix B, qtmatrix C)
{
    qtmatrix D; /* Auxiliary QuadTree Matrix. */

    if (n <= BS)
        matmul_submatrix(n, A, B, C); /* Multiply data submatrices. */
    else {
        D=new_qtmatrix(n); /* Dynamically allocate a QuadTree Matrix
                              for a problem of size n. */
        n2 = n/2;
        QTMat_RecMul(n2, a11, b11, d11);     /* d11 = a11 * b11 */
        QTMat_RecMul(n2, a12, b21, c11);     /* c11 = a12 * b21 */
        QTMat_RecAdd(n2, d11, c11, c11);     /* c11 += d11      */
        QTMat_RecMul(n2, a11, b12, d12);     /* d12 = a11 * b12 */
        QTMat_RecMul(n2, a12, b22, c12);     /* c12 = a12 * b22 */
        QTMat_RecAdd(n2, d12, c12, c12);     /* c12 += d12      */
        QTMat_RecMul(n2, a21, b11, d21);     /* d21 = a21 * b11 */
        QTMat_RecMul(n2, a22, b21, c21);     /* c21 = a22 * b21 */
        QTMat_RecAdd(n2, d21, c21, c21);     /* c21 += d21      */
        QTMat_RecMul(n2, a21, b12, d22);     /* d22 = a21 * b12 */
        QTMat_RecMul(n2, a22, b22, c22);     /* c22 = a22 * b22 */
        QTMat_RecAdd(n2, d22, c22, c22);     /* c22 += d22      */
        free_qtmatrix(D,n); /* Free space of Auxiliary QuadTree Matrix D. */
    }
}

void main() {
  ...
  /* Allocate and initialize quadtree matrices for a problem of size N. */
  ...
  QTMat_RecMul(N,A,B,C);    /* Compute C = A * B */
  ...
}
```

being `matmul_submatrix` a routine used to multiply two input data submatrices (A and B), updating the result matrix (C). Similarly, `matadd_submatrix` performs the addition of submatrices.

We ask you to provide pseudocode for the representative part of an efficient parallelization of the code above, using a *Divide and Conquer* approach and OpenMP explicit tasks.

1. (1.5 points) Write a parallel version using the `Tree` strategy. Do not use `OpenMP dependences`.
   Solution:

```
#define CUTOFF ...

...

/* C = A + B */
void QTMat_RecAdd(int n, qtmatrix A, qtmatrix B, qtmatrix C) {
    if (n <= BS)
        matadd_submatrix(n, A, B, C); /* Add data submatrices. */
    else {
        n /= 2;
        #pragma omp task final( n < CUTOFF ) mergeable
        QTMat_RecAdd(n, a11, b11, c11);      /* c11 = a11 + b11 */

        ... /* Similarly for the other two quadrants. */

        #pragma omp task final( n < CUTOFF ) mergeable
        QTMat_RecAdd(n, a22, b22, c22);      /* c22 = a22 + b22 */
    }
}


/* C = A * B */
void QTMat_RecMul(int n, qtmatrix A, qtmatrix B, qtmatrix C)
{
    qtmatrix D; /* Auxiliary QuadTree Matrix. */

    if (n <= BS)
        matmul_submatrix(n, A, B, C); /* Multiply data submatrices. */
    else {
        D=new_qtmatrix(n); /* Dynamically allocate a QuadTree Matrix
                              for a problem of size n. */
        n2 = n/2;
        /* 1st quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a11, b11, d11);     /* d11 = a11 * b11 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a12, b21, c11);     /* c11 = a12 * b21 */
        #pragma omp taskwait
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecAdd(n2, d11, c11, c11);     /* c11 += d11      */

        /* 2nd quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a11, b12, d12);     /* d12 = a11 * b12 */

        ... /* Similarly for the other two quadrants. */
```

```
        /* 4th quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a21, b12, d22);     /* d22 = a21 * b12 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a22, b22, c22);     /* c22 = a22 * b22 */
        #pragma omp taskwait
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecAdd(n2, d22, c22, c22);     /* c22 += d22      */

        #pragma omp taskwait
        free_qtmatrix(D,n); /* Free space of Auxiliary QuadTree Matrix D. */
    }
}

void main() {
  ...
  #pragma omp parallel
  #pragma omp single
  QTMat_RecMul(N,A,B,C);    /* Compute C = A * B */
  ...
}
```

2. (1 point) Can the function calls in routine `QTMat_RecMul` above be rescheduled so that the `Tree` strategy in the previous exercise achieves better parallel performance? How would you change the code? Show the new parallelization of the resulting excerpt of code.

Solution:

We can move calls to `QTMat_RecAdd` to the end of the basic block which corresponds to the `else`. In this way, the number of synchronizations can be drastically reduced.

```
/* C = A * B */
void QTMat_RecMul(int n, qtmatrix A, qtmatrix B, qtmatrix C)
{
    qtmatrix D; /* Auxiliary QuadTree Matrix. */

    if (n <= BS)
        matmul_submatrix(n, A, B, C); /* Multiply data submatrices. */
    else {
        D=new_qtmatrix(n); /* Dynamically allocate a QuadTree Matrix
                              for a problem of size n. */
        n2 = n/2;
        /* Compute products. */
        /* 1st quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a11, b11, d11);     /* d11 = a11 * b11 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a12, b21, c11);     /* c11 = a12 * b21 */

        /* 2nd quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a11, b12, d12);     /* d12 = a11 * b12 */

        ... /* Similarly for the other two quadrants. */

        /* 4th quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a21, b12, d22);     /* d22 = a21 * b12 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecMul(n2, a22, b22, c22);     /* c22 = a22 * b22 */

        #pragma omp taskwait

        /* Compute additions. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecAdd(n2, d11, c11, c11);     /* c11 += d11      */

        ... /* Similarly for the other two quadrants. */

        #pragma omp task final( n2 < CUTOFF ) mergeable
        QTMat_RecAdd(n2, d22, c22, c22);     /* c22 += d22      */

        #pragma omp taskwait
        free_qtmatrix(D,n); /* Free space of Auxiliary QuadTree Matrix D. */
    }
}
```

3. (1.5 points) Write a parallel version using `OpenMP dependences`.

Solution:

The recursive calls within routine `QTMat_RecAdd` are fully independent. Thus, we do not need to change that routine. The `main` function does not change either. With respect to routine QTMat_RecMul and for the sake of brevity we only include the data references which cause dependencies.

```
/* C = A * B */
void QTMat_RecMul(int n, qtmatrix A, qtmatrix B, qtmatrix C)
{
    qtmatrix D; /* Auxiliary QuadTree Matrix. */

    if (n <= BS)
        matmul_submatrix(n, A, B, C); /* Multiply data submatrices. */
    else {
        D=new_qtmatrix(n); /* Dynamically allocate a QuadTree Matrix
                              for a problem of size n. */
        n2 = n/2;
        /* 1st quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: d11)
        QTMat_RecMul(n2, a11, b11, d11);     /* d11 = a11 * b11 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: c11)
        QTMat_RecMul(n2, a12, b21, c11);     /* c11 = a12 * b21 */

        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(in: d11) depend(inout: c11)
        QTMat_RecAdd(n2, d11, c11, c11);     /* c11 += d11      */

        /* 2nd quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: d12)
        QTMat_RecMul(n2, a11, b12, d12);     /* d12 = a11 * b12 */

        ... /* Similarly for the other two quadrants. */

        /* 4th quadrant. */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: d22)
        QTMat_RecMul(n2, a21, b12, d22);     /* d22 = a21 * b12 */
        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(out: c22)
        QTMat_RecMul(n2, a22, b22, c22);     /* c22 = a22 * b22 */

        #pragma omp task final( n2 < CUTOFF ) mergeable
                         depend(in: d22) depend(inout: c22)
        QTMat_RecAdd(n2, d22, c22, c22);     /* c22 += d22      */

        #pragma omp taskwait
        free_qtmatrix(D,n); /* Free space of Auxiliary QuadTree Matrix D. */
    }
}
```