CSE-613: Parallel Programming, Spring 2015 Homework #1
By Aashray Arora (SBU ID: 109940382, NET ID: aaarora) &
Ajay Lakshminarayanarao (SBU ID: 109898256, NETID: alakshminara)

Task 1: Multiplying Matrices

a) We ran each algorithm given in Figure 1 of the homework on $2^{12} \times 2^{12}$ size matrices generated randomly by the program. The results are tabulated below.

Iterative matrix multiplication variant	Running Time (of algorithm only)
ITER-MM-IJK	1266.85 seconds/ 21.11minutes
ITER-MM-IKJ	370 seconds / 6.16 min
ITER-MM-JIK	1212.89 seconds / 20.21 min
ITER-MM-JKI	2370.34 seconds / 39.50
ITER-MM-KIJ	372.06 s/ 6.20 min
ITER-MM-KJI	2401.08/ 40.018 min

The reason the ITER-MM-IKJ and ITER-MM-KIJ algorithms run significantly faster than others is ***locality of reference***.

Locality of reference shows related storage locations being frequently accessed.
The reason for this speed up is that in the IJK algorithm, the reads of X[i][k] are in cache (since the k index is the contiguous, last dimension), but Y[k][j] is not, so there is a cache miss. Z[i][j] is irrelevant, because it can be factored out of the inner loop. In the IKJ algorithm, the reads and writes of Z[i][j] are both in cache, the reads of Y[k][j] are in cache, and the read of X[i][k] can be factored out of the inner loop. Thus, the IKJ algorithm has no cache miss penalty in the inner loop while the first example has a cache penalty.

Reference: http://en.wikipedia.org/wiki/Locality_of_reference

b) The two fastest implementations are ITER-MM-IKJ and ITER-MM-KIJ. The number of ways to parallelize the 3 for loops is by parallelizing either any one of these 3 loops or any combination of 2 of them or all 3 of them. This gives us 3 + 3 + 1 = 7 ways. We report result of ITER-MM-IKJ here, similar results can be found for ITER-MM-KIJ in the source and script files submitted.

Parallel ITER-MM_IKJ with a parallel *i* loop.

Size of matrices	Running Time(seconds)
16	0
32	0
64	0
128	0
256	0
512	0

1024	6
2048	52
4096	450
8192	3005 seconds

Parallel ITER-MM_IKJ with a parallel *j* loop.

Size of matrices	**Running Time**
16	0
32	0
64	0
128	0
256	2
512	5
1024	26
2048	124 seconds
4096	539 seconds
8192	2649

Parallel ITER-MM_IKJ with a parallel *k* loop.

Size of matrices	**Running Time**
16	0 seconds
32	0 seconds
64	0 seconds
128	0 seconds
256	1 second
512	11 seconds
1024	82
2048	489 seconds
4096	5584 seconds

Parallel ITER-MM_IKJ with a parallel *i and j* loops.

Size of matrices	Running Time
16	0
32	0
64	0
128	0
256	0
512	0
1024	1
2048	4
4096	21
8192	115

Parallel ITER-MM_IKJ with a parallel *j and k* loops.

Size of matrices	Running Time
16	0
32	0
64	0
128	1
256	1
512	1
1024	4
2048	15
4096	65
8192	278

Parallel ITER-MM_IKJ with a parallel *i and k* loops.

Size of matrices	Running Time
16	0

32	0
64	0
128	0
256	0
512	0
1024	0
2048	1
4096	5
8192	50

Parallel ITER-MM_IKJ with a parallel *i, j and k* loops.

Size of matrices	Running Time
16	0
32	0
64	0
128	0
256	1
512	0
1024	1
2048	4
4096	21
8192	115

c) Vary number of cores.

Parallel ITER-MM-IKJ with all three loops in parallel

Number of cores	**Running time in seconds**
1	54
2	55
3	54
4	53

5	51
6	50
7	48
8	46
9	45
10	48
11	50
12	50
13	47
14	51
15	49
16	51

Parallel ITER-MM-IKJ with only *I* loop in parallel.

1	1204
2	3867
3	3414
4	3551
5	4826
6	4504
7	4915
8	3786
9	3711
10	3909
11	4487
12	4770
13	4409
14	3774
15	3660

16	3322

Parallel ITER-MM-IKJ with *I and K* loop in parallel.

16	54
15	58
14	51
13	56
12	57
11	62
10	69
9	74
8	85
7	99
6	115
5	134
4	164
3	227
2	329
1	644

d) **For varying the size of the input matrix**: we see a general trend that larger matrices take longer time, which is obvious. We see that parallelizing the i and k loops for Parallel ITER-MM-IKJ gives us the fastest running time for all inputs. Parallelizing one loop is not so efficient and parallelizing all three loops although good in theory, in practice does not give best result. Parallelizing 2 loops works best and especially loop i and k as it is cache effective.

For varying the number of cores used: we observe a general trend that the more the cores, the faster the algorithm runs. In case of parallelizing all three loops, number of cores does not affect the running time, this maybe because of the overhead of too much communication between parallel processors. If we parallelize only i loop, just having 1 core gives better results due to similar reasons. Parallelizing i and k loops gives good results and running time decreases as we increase the number of cores.

e) We implemented the PAR-Rec-MM algorithm to find the base case to avoid overhead of recursion empirically. The table below shows that 2^8 is the optimal base case that gives us smallest running time. Source code is in file *aashray_rec_mm.c*

Base Case (M)	**Running time (in seconds)**
2^4	883.97

2^5	582.94
2^6	432.67
2^7	386.81
2^8	**307.78**
2^9	373.25
2^{10}	635.84

f) Varying the size of the input (n) from 2^4 $to 2^{13}$ for PAR-REC-MM with optimized base case and running on all cores we get the following results. Source code file *aashray_rec_mm_vary_size.c*

Input Size (n)	**Running time (in seconds)**
2^4	0
2^5	0
2^6	0
2^7	0
2^8	0.01
2^9	0.69
2^{10}	2.93
2^{11}	5.62
2^{12}	40.15
2^{13}	316.40

We observe that as the input size increase the running time increases.

Varying the number of cores we get the following. Source code file: *aashray_rec_mm_vary_cores.c*

Number of cores	**Running time (in seconds)**
16	312.38
15	310.96
14	311.07
13	311.51
12	309.42
11	312.68
10	306.32

9	303.02
8	298.67
7	298.44
6	296.61
5	294.44
4	292.35
3	291.92
2	290.51
1	290.14

We observe that the algorithm runs faster on 1 core than more, increasing the number of cores slowly increases the running time. This could be because recursive algorithms run slower on multiple cores due to communication overhead.

Task 2: The Parenthesis Problem

a) We cannot Parallelize the for loops in the given serial implementation directly because updates to c[i,j] depend on the order of the algorithm i.e on previous updates and parallelizing them would render the algorithm incorrect. We can modify the serial algorithm to make sure we move along the diagonal of the matrix c in such a way the the updates to c[i,j] depend only on the outmost loop which we will not parallelize. Find the parallel algorithm below :

```
for (int p = 2 to n-1)
  Parallel : for (int i = 1 to n-p)
        j = p + i
        Parallel : for (k = i to j)
              c[i,j] = min (c[i][j], c[i,k]+c[k,j])
```

- We tried this implementation and it can be found with the source code. It gives us the correct results.
- The implementation is similar to the serial implementation given with the question. The main difference is the in this we go along the diagonal and not access elements of the same row in the same loop and are able to parallelize it.
- Work : 3 nested for loops. Each running ~n times in worst case. $\Theta(n^3)$
- Span : Θ (n + log n + log n) ~ Θ (n)
- Parallelism : $\Theta(n^2)$

In practice, only parallelizing the i loop gives better results for a 4096 by 4096 matrix.

```
for (int p = 2 to n-1)
  Parallel : for (int i = 1 to n-p)
        j = p + i
        for (k = i to j)
              c[i,j] = min (c[i][j], c[i,k]+c[k,j])
```

Parallel Implementation

Step 1: We do not change the update statement, which is
c[i,j] = min{ c[i,j], c[i,k] + c[k,j]} and we ensure it is for the same values of i, j and k as observed in serial implementation.

Step 2: As k is dependent on value of i and j, we cannot parallelize the earlier loops. To make this possible we introduce another variable (say p) and parallelize the loop for i.

Step 3: We update the elements in a diagonal manner (top left to bottom right). This is done such that the elements are updated only from the previous diagonal elements. This update happens from the equation seen in Step 1 and by initializing j to p + i

Step 4: Step 3 is repeated in a loop by incrementing p and i (in parallel) until the required elements of the matrix are filled and the required result is obtained.

To prove the algorithm produces correct results

In the serial algorithm we were updating the elements in a row wise manner. We updated the entire row 'i' using the inner loops for j and k. These values were updated initially from the diagonal elements of the matrix as mentioned in the problem.

If we instead update in a diagonal manner, we can parallelize the i loop. And instead of updating in a row manner, we can update diagonally. The equation does not change (as seen in Step 1 of parallel implementation). Only the order of update is different in our case.

As we are updating the value of a particular cell only in the k loop for a given i and j. We do not change the results of any of the earlier calculated cells.

The previously calculated cells were calculated using the initial diagonal elements in the problem using equation in Step 1 (same used in serial, and for same values of i, j and k).

As future values are dependent on these previous values which are correct, the previous values do not change with the value of i and j, and the initial values are calculated using the given diagonal values.

Hence we can ensure that our algorithm behaves correctly.

To prove the algorithm by method of contradiction'
Let us assume the diagonals are named from 1 to n. where the first diagonal is from [1,1] to [n,n]. The last diagonal n is [1,n]

From the problem we have diagonal 1 is infinity. And diagonal 2 is given.

Let us assume the top right element (diagonal n) of the matrix is incorrectly calculated by our algorithm.
This element on the top right is calculated using the two diagonal elements on its left.

Using the same equation as observed in serial. And for the same values of i, j and k

Hence the two diagonal elements [diagonal (n-1)] have to be incorrect.

But the elements in diagonal (n-1) are calculated based on the elements in diagonal (n-2) using the same equation and for the same values of i, j and k as observed in serial implementation.

Hence elements in diagonal in n-2 have to be incorrect.

As we follow the above pattern, there is some i at which n-i = 2. Which is the given diagonal elements. As we follow the above algorithm we show that diagonal 2 is incorrect. But elements in this diagonal are given and they are correct. Therefore the assumed element in diagonal n has to be correct and all the diagonals between.

Hence the algorithm is correct.

Work: 3 nested for loops. Each running ~n times in worst case. Θ (n^3)

Span: Θ (n^2 + log n) ~ Θ (n^2)

Parallelism: Θ (n)

b) The algorithm first breaks the matrix into three parts as shown in the Figure 4 of assignment. Solves each part recursively. The base is when the X part (square) of the matrix is a 1 by 1 matrix it takes the minimum of U and V which are diagonal elements. Thus for each X it takes the minimum of U and V in the base case and splits it recursively otherwise. Hence the algorithm follows recurrence 2. Hence it is correct.

Work (T_1):

We can calculate the work of the algorithm using Masters Theorem.
We need to find the running time of the function A_{par}. But that depends on B_{par} , which depends on C_{par} so we will calculate them first.

$$T_C(n) = \begin{cases} 1, & if\ n \leq 1 \\ 8T_C(\frac{n}{2}), & otherwise \end{cases}$$

Using Masters theorem we get $T_C = \Theta\left(n^{\log_2 8}\right) = \Theta\,(n^3)$

$$T_B(n) = \begin{cases} 1, & if\ n = 1 \\ 4\,T_B\left(\frac{n}{2}\right) + 4\,T_C(\frac{n}{2}), & otherwise \end{cases}$$

We know the running time of T_C so using Masters Theorem we can find the running time T_B :

$$T_B = \Theta\,(n^3)$$

$$T_A(n) = \begin{cases} 1, & if\ n = 1 \\ 2\,T_A\left(\frac{n}{2}\right) + T_B(\frac{n}{2}), & otherwise \end{cases}$$

$$T_A = \Theta\,(n^3)$$

Hence **Work (T_1) = Θ (n^3)**

Span (T_∞):

$$T_{\infty C}(n) = \begin{cases} 1, \ if\ n \leq 1 \\ 2T_{\infty C}(\frac{n}{2}), otherwise \end{cases}$$ Using Masters theorem we get $T_{\infty C} = \Theta\left(n^{\log_2 2}\right) = \Theta$ (n)

$$T_{\infty B}(n) = \begin{cases} 1, \ if\ n = 1 \\ 3\ T_{\infty B}\left(\frac{n}{2}\right) + 3\ T_{\infty C}(\frac{n}{2}), \ \ otherwise \end{cases}$$ Using $T_{\infty c}$ and Masters theorem we get $T_{\infty B} = \Theta\left(n^{\log_2 3}\right)$

$$T_{\infty A}(n) = \begin{cases} 1, \ if\ n = 1 \\ T_{\infty A}\left(\frac{n}{2}\right) + T_{\infty B}(\frac{n}{2}), \ \ otherwise \end{cases}$$ Using $T_{\infty B}$ and Masters theorem we get $T_{\infty A} = \Theta\left(n^{\log_2 3}\right)$

Hence Span ($\boldsymbol{T}_\infty$) = $\Theta\left(n^{\log_2 3}\right)$

Parallelism $\left(\frac{T_1(n)}{T_\infty(n)}\right) = \Theta\left(\frac{n^3}{n^{\log_2 3}}\right)$

c) We implemented the algorithm given in Figure 3. It can be found in the source code in file *new_recur_parenthesis.c*. We empirically found the value of m that gives us the best running time by calling the algorithm for an input matrix of $n = 2^{13}$ in a loop for m = $2^1 to\ 2^{10}$. We tabulate the results below.

$\boldsymbol{m}$	**Running Time (seconds)**
2^1	985.27
2^2	5421.39
2^3	3492.07
2^4	1596.78
2^5	1099.23
2^6	645.30
$\mathbf{2^7}$	***575.99***
2^8	757.39
2^9	1121.06
2^{10}	1629.65

d) Varying size of matrix in implementation from part b (base case 1x1).

2) $\boldsymbol{Size\ of\ matrix}$	**Running Time (seconds)**
2^4	0.01
2^5	0.24
2^6	0.27
2^7	0.12
2^8	0.49
2^9	2.19
2^{10}	11.07
2^{11}	66.26
2^{12}	458.82
2^{13}	3460.76

We ran the parallel algorithm (source : *new_recur_parenthesis.c*) with our optimal base of 2^7 for n varying from 2^4 to 2^{13}. Results are tabulated below.

$Size\ of\ matrix$	Running Time (seconds)
2^4	0
2^5	0
2^6	0.54
2^7	0.01
2^8	0.12
2^9	0.32
2^{10}	1.97
2^{11}	12.99
2^{12}	53.32
2^{13}	297.43

e) Varying cores: with optimal base case of size = 128

$Number\ of\ Cores$	Running Time (seconds)
1	80.59
2	127.95
3	116.47
4	150.36
5	141.58
6	149.61
7	150.68
8	165.52
9	179.41
10	192.75
11	206.00
12	221.10
13	219.03
14	232.94
15	245.69
16	253.41

Varying cores : base case of 1 x 1 matrix

$Number\ of\ Cores$	Running Time (seconds)
1	3361.00
2	3365.00
3	3376.55
4	3375.78
5	3386.53
6	3392.02
7	3403.66
8	3453.62
9	3414.79
10	3425.93
11	3484.89
12	3487.56
13	3440.11
14	3448.87
15	3461.80
16	3460.55

f) We observe that the algorithm takes much longer when the base case is a 1x1 matrix and significantly lesser when we reduce to a serial implementation for an optimal base case m. The running time increases as we increase the size of the matrix. The difference is significant only for larger matrices and not the smaller ones. Varying the number of cores only increased the running time of the algorithm for the one with base case of 1x1 matrix. This could be because of the communication overhead of a recursive divide and conquer parallel algorithm. For the one with optimal base we see that number of cores is not making a significant difference on such a large matrix.

Task 3
Sieve of Eratosthenes

(a) Yes the algorithm is still correct. We implemented and found the solution to be correct. We observe that the initial loop marks an element to be a prime number for all values greater than 3 till n. At the end of every cilk_for there is a sync of operations. So unless this initial loop will not complete, the execution will not move further.

The following cilk_for loop executes for a particular value of i. If the 'if' condition is true, there is another cilk_for which is executed. This loop marks all the values in the numbers array to be false, if it is a multiple of i. This continues further till the main loop reached square root of n. We know that all numbers till n if not prime, will be a multiple till all values till square root of n.

In this case we are updating the value to be always false in the array for these elements. Therefore there will be no data corruption observed as if it is updated repeatedly, it will always be false, if updated at least once.

In the end we just pick those numbers who are still marked as true. And these numbers we can safely say as parallel and hence the algorithm works.

We can prove this algorithm by proof of contradiction:

Step 1: Assume at the end of the program, there exists a number which is a multiple from 0 to square root of n but has been showed that it is prime. (The array element is marked true)

Step 2: The main cilk_for goes from all the values of i in range (0, square root of n). Even though it is executed in parallel, the body is executed for all values of i in some processor core.

Step 3: The body of the main cilk_for initially checks whether the given element is true. Which indicates it to be currently marked a prime number. Therefore the condition will be true for the number considered in Step 1. We note that these are independent units of execution in different cores.

Step 4: The condition in Step 3 is always true. The inner cilk_for check for all multiples of i starting from 2i till the last multiple of i less than n. The number assumed is some multiple of i. And for that i in the main cilk_for it always passes the condition (Step 3). This will take place even if it is executed on an independent core and they are operating on the same array. There is no problem of synchronization as it will sync after the inner loop before the next iteration of the main cilk_for.

Step 5: From Step 4 we observe that the assumed number will be marked as false in some processor computation. Therefore it is shown as not prime and our assumption is false.

The computed work for this algorithm is **theta (n * log (log n))**. This is because the number of iterations carries out is **n * (1/2 + 1/3+ 1/5+….+1/ (last parallel < sqrt (n))) = n * log (log (n)).** The work for serial

and parallel implementations is the same. And we can infer from the problem statement the work for serial which is theta (n * log (log n)) due to the above summation.

The computed span is obtained as follows:

Step 1: The initial cilk_for executes in log (n) time in parallel. So our final span is sum of log (n) and time taken by the following cilk_for loop.

Step 2: The main cilk_for executes for (sqrt of n) operations. When executed in parallel, this will be done in ½ * log (n) operations.

The time taken for executing the inner loop for the critical length is approximately log (log (n)).

The inner operation when run in parallel, execute in log (log (n)) time. Therefore the time taken in this step is ½ * log (n) + log (log (n)) which is approximated to log (n) time.

Taking the sum of above steps,

Span = log (n) + log (n)

Therefore the span is **theta log (n)**.

(b) Optimized version of Sieve of Eratosthenes:

```
Cilk_for i from 1 to n
        Add i to Set
Remove 1 from Set.
For (Set is not empty)
        Index = first element of Set
        Remove index From Set
        Add Index to result
        Cilk_for (element in Set)
                If element % index == 0
                        Remove element from set
```

The computed work for this algorithm is **theta (n * log (log n))**. This is because the number of iterations carries out is **n * (1/2 + 1/3+ 1/5+….+1/ (last parallel < sqrt(n))) = n * log (log (n)).**

Hence the computer work is

T_1 : Θ n * log (log (n)).

The computed span is obtained as follows:

Step 1: The initial cilk_for executes in log (n) time in parallel. So our final span is sum of log (n) and time taken by the following cilk_for loop.
Step 2: The main for executes for (sqrt of n) operations. When executed in parallel, this will be done in log (n) operations.
The time taken for executing the inner loop for the critical length is approximately log (log (n)).

The inner operation when run in parallel, execute in log (log (n)) time. Therefore the time taken in this step is log (n) + log (log (n)) which is approximated to log (n) time.
Taking the sum of above steps,
Span = log (n) + log (n)

Therefore the span is

$\boldsymbol{T_\infty}$ **: Θ log (n)**.