CSE613: Parallel Programming, Spring 2019 **Date:** March 4

Homework #1[1]

(Due: March 22)

Task 1. [100 Points] Multiplying Matrices.

(*a*) [**10 Points**] Figure 1 shows the standard iterative matrix multiplication algorithm and its 5 variants obtained by permuting the three nested ***for*** loops in all possible ways. Assume that all three matrices (X, Y and Z) are given in row-major order. Run each implementation on matrices of size $2^r \times 2^r$, where r is the largest integer such that none of the implementations takes more than five minutes to perform the multiplication. Tabulate all running times[2].

(*b*) [**10 Points**] Use PAPI[3] to find the L1/L2/L3 misses incurred by each implementation from part 1(a) when run on matrices of size $2^r \times 2^r$, where r is as defined in part 1(a). Tabulate your results.

(*c*) [**5 Points**] Compare and explain your findings from parts 1(a) and 1(b).

(*d*) [**10 Points**] Take the two fastest implementations from part 1(a), and try to parallelize each simply by replacing one or more serial ***for*** loops with parallel ***for*** loops. In how many ways can you correctly parallelize each implementation? Now run each such parallel implementation on all cores by varying n from 2^4 to 2^s (consider only powers of 2), where s is the largest integer such that none of the parallel implementations takes more than a minute to perform the multiplication. Tabulate or plot the running times.

(*e*) [**10 Points**] Run each parallel implementation from part 1(d) on a $2^r \times 2^r$ input matrix by varying p from 1 to the maximum number of cores available on the machine[4], where r is as defined in part 1(a). Tabulate or plot the running times.

(*f*) [**5 Points**] Explain your findings from parts 1(d) and 1(e).

(*g*) [**30 Points**] Implement the in-place parallel recursive matrix multiplication algorithm shown in Figure 2 (Par-Rec-MM). But do not recurse down to matrices of size 1×1. Instead stop at a base case of size $m \times m$ for some $m > 0$ in order to reduce the overhead of recursion. When you reach a base case of size $m \times m$ use one of the fastest serial algorithms from part 1(a) for multiplying the two subatrices. For simplicity let's assume that both n and m are powers of 2. For $n = 2^r$ (where r is as defined in part 1(a)), empirically find the value of

[1] We will not use hyperthreading in this assignment.

[2] When you implement and compile these you must make sure that the compiler does not change the order in which the loops are nested in each of them

[3] PAPI is installed on both Stampede2 and Comet (please check by running "module avail"). You will have to first load PAPI (by running "module load") in order to use it.

[4] `Cilk` allows you to vary the number of worker threads (available cores) from the command line of your own program.

```
Iter-MM-ijk( $Z$, $X$, $Y$, $n$ )
(Inputs are three $n \times n$ matrices $X$, $Y$ and $Z$. For each $i, j \in [1, n]$, $Z[i, j]$ is set to $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        for $k \leftarrow 1$ to $n$ do
            $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$
```

```
Iter-MM-ikj( $Z$, $X$, $Y$, $n$ )
(Inputs are three $n \times n$ matrices $X$, $Y$ and $Z$. For each $i, j \in [1, n]$, $Z[i, j]$ is set to $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)
for $i \leftarrow 1$ to $n$ do
    for $k \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $n$ do
            $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$
```

```
Iter-MM-jik( $Z$, $X$, $Y$, $n$ )
(Inputs are three $n \times n$ matrices $X$, $Y$ and $Z$. For each $i, j \in [1, n]$, $Z[i, j]$ is set to $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)
for $j \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n$ do
        for $k \leftarrow 1$ to $n$ do
            $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$
```

```
Iter-MM-jki( $Z$, $X$, $Y$, $n$ )
(Inputs are three $n \times n$ matrices $X$, $Y$ and $Z$. For each $i, j \in [1, n]$, $Z[i, j]$ is set to $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)
for $j \leftarrow 1$ to $n$ do
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$
```

```
Iter-MM-kij( $Z$, $X$, $Y$, $n$ )
(Inputs are three $n \times n$ matrices $X$, $Y$ and $Z$. For each $i, j \in [1, n]$, $Z[i, j]$ is set to $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)
for $k \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $n$ do
            $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$
```

```
Iter-MM-kji( $Z$, $X$, $Y$, $n$ )
(Inputs are three $n \times n$ matrices $X$, $Y$ and $Z$. For each $i, j \in [1, n]$, $Z[i, j]$ is set to $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)
for $k \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$
```

Figure 1: Standard iterative matrix multiplication algorithm and its variants.

m that gives you the smallest running time for Par-Rec-MM. Produce a table or a graph showing how the running time varies as you change m.

(h) [**10 Points**] Produce tables / plots similar to those in parts 1(d) and 1(e) for the algorithm in part 1(g) (with optimized base case). Compare the results with parts 1(d) and 1(e), and explain what you observe.

(i) [**10 Points**] Use PAPI to find the total L1/L2/L3 misses incurred (across all cores) by the fastest implementation from part 1(d) as well as the Par-Rec-MM implementation from part 1(g) when run on matrices of size $2^r \times 2^r$, where r is as defined in part 1(a). Tabulate and explain your results.

```
Par-Rec-MM( $Z$, $X$, $Y$ )
(Inputs are three $n \times n$ matrices $X$, $Y$ and $Z$. For each $i, j \in [1, n]$, $Z[i, j]$ is set to $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$. We assume $n$ to be a power of 2.)

1. if $X$ is a $1 \times 1$ matrix then
2.     $Z \leftarrow Z + X \times Y$
   else
3.     parallel : Par-Rec-MM( $Z_{11}$, $X_{11}$, $Y_{11}$ ), Par-Rec-MM( $Z_{12}$, $X_{11}$, $Y_{12}$ )
                  Par-Rec-MM( $Z_{21}$, $X_{21}$, $Y_{11}$ ), Par-Rec-MM( $Z_{22}$, $X_{21}$, $Y_{12}$ )
4.     parallel : Par-Rec-MM( $Z_{11}$, $X_{12}$, $Y_{21}$ ), Par-Rec-MM( $Z_{12}$, $X_{12}$, $Y_{22}$ )
                  Par-Rec-MM( $Z_{21}$, $X_{22}$, $Y_{21}$ ), Par-Rec-MM( $Z_{22}$, $X_{22}$, $Y_{22}$ )
```

Figure 2: Parallel recursive divide-and-conquer algorithm that multiplies two $n \times n$ matrices X and Y and adds the result to another $n \times n$ matrix Z, i.e., for each $i, j \in [1, n]$, $Z[i, j]$ is set to $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$. All entries of Z are initially set to zero. By Z_{11}, Z_{12}, Z_{21} and Z_{22} we denote the top-left, top-right, bottom-left and bottom-right quadrants of Z, respectively. Similarly for X and Y. We assume that n is a power of 2.

Task 2. [100 Points] Schedulers. This task asks you to implement the following three schedulers and compare their performance by running Par-Rec-MM from Figure 2 under each:

- distributed randomized work-stealing (DR-Steal),
- distributed randomized work-sharing (DR-Share), and
- centralized work-sharing (C-Share).

For the purposes of this task you can simply tailor each scheduler implementation to run Par-Rec-MM only instead of implementing their standalone general-purpose versions. To keep things simple use locks or atomic instructions to control accesses to shared data structures, and maintain a global flag *empty* without locks to keep track of when the entire system runs out of work. The flag is initially set to False. The thread that completes the second *sync* of Par-Rec-MM (after Line 4 of Figure 2) sets that variable to True.

In case of the DR-Steal scheduler, each thread will have its own task deque. Whenever a thread runs out of work it first checks its own deque for tasks. If the deque is nonempty it extracts the bottommost task from it and starts working on that. If the deque is empty the thread becomes a thief and tries to steal the topmost task from the deque of a thread chosen uniformly at random. The thread will continue with its steal attempts until either one succeeds or it is sure (or reasonably sure) that the entire system has run out of work.

Under the DR-Share scheduler, too, each thread will have its own task deque. Whenever a thread spawns new tasks it puts each of them (except the one it wants to execute itself) at the top of the deque of a thread chosen uniformly at random. Whenever a thread runs out of work it only looks for tasks in its own deque. It keeps checking its own deque until either someone else puts a new task in it or the thread becomes sure (or reasonably sure) that there is no work in the entire system.

The C-Share scheduler maintains a centralized task queue. Whenever a thread spawns new tasks it keeps one of them for immediate execution and puts the rest of them in that centralized queue. Whenever a thread becomes idle it deques a task (if exists) from the centralized queue and starts executing it.

(a) [**25 Points**] Run Par-Rec-MM under each of the three schedulers by varying n from 2^{10} to 2^s (consider only powers of 2), where s is the largest integer such that none of the three implementations takes more than five minutes to terminate. Use all available cores. For each run compute its rate of execution in GFLOPS[5]. Create a single plot showing the GFLOPS for all three programs. Explain your findings.

(b) [**25 Points**] Repeat part 2(a) but instead of plotting GFLOPS plot cache miss rates[6] for L1, L2 and L3 caches. Explain your findings.

(c) [**25 Points**] Repeat part 2(a) but instead of varying n keep n fixed at 2^s (where s is as defined in part 2(a)) and vary p from 1 to the maximum number of cores available on the machine. For each run compute its *efficiency*[7]. Create a single plot showing how 1−*efficiency* (giving the fraction of total work spent in overheads) varies with p for all three programs. Explain your findings.

(d) [**25 Points**] For this part we modify DR-Steal and DR-Share to DR-Steal-Mod and DR-Share-Mod, respectively, to keep track of the total amount of work the tasks in each deque represent. We assume that the task of multiplying two $m \times m$ matrices requires m^3 units of work. So, when such a task is added to a deque it increases the amount of work in that deque by m^3 and when it is removed it decreases the work by the same amount.

[5] To obtain the rate of execution of a program in FLOPS (FLoating point Operations Per Second) divide the total number of floating point operations it performs by its running time in seconds. GFLOPS (Giga FLOPS) is obtained by dividing FLOPS by 10^9. The total number of floating point operations performed by Par-Rec-MM for multiplying two $n \times n$ matrices is $2n^3$.

[6] For this task we will compute the cache miss rate of a particular run of Par-Rec-MM by dividing the total number of cache misses (across all cores) it incurs by the total number of cache accesses in the worst case which is $3n^3$ when multiplying two $n \times n$ matrices.

[7] Efficiency E_p of a program run on p processing cores is defined as $\frac{T_1}{pT_p}$, where T_p is its running time on p cores.

When a thread runs out of work under DR-Steal-Mod it chooses two deques uniformly at random and attempts to steal work from the one that has more work (breaking ties arbitrarily). On the other hand, when a thread wants to put a newly spawned task in another deque under DR-Steal-Mod it chooses two deques uniformly at random and inserts the task into the one that has less work (again breaking ties arbitrarily).

Now repeat parts $2(a)$ and $2(c)$ with DR-Steal-Mod and DR-Share-Mod, and compare the results with what you got with DR-Steal and DR-Share, respectively. Explain your findings.

Task 3. [**50 Points**] **Some Analyses.** This task asks you to analyze some aspects of the schedulers from Task 2.

(a) [**10 Points**] Suppose that in DR-Steal a thread stops looking for work and terminates after $2p \ln p$ consecutive failed steal attempts. Prove that during those $2p \ln p$ attempts, the thread has checked all deques in the system w.h.p. in p (and found each of them empty).

(b) [**5 Points**] Argue that even if a thread finds every deque in the system empty in consecutive failed steal attempts that does not guarantee that the entire system has run out of work.

(c) [**5 Points**] Argue that even if threads follow the strategy in part $3(a)$ to terminate prematurely (as suggested by part $3(b)$), all work in the system will still be completed.

(d) [**15 Points**] Suppose in DR-Share from Task 2 no two enqueues (i.e., enqueuing a new task into a deque) in the entire system happen at exactly the same time. Then prove that during any sequence of p consecutive enqueues each deque undergoes $\mathcal{O}\left(\frac{\ln p}{\ln \ln p}\right)$ enqueue operations w.h.p. in p.

(e) [**15 Points**] Consider the following simplified version of DR-Share-Mod from Task 2. Whenever a thread needs to put a newly spawned task in another deque it chooses two deques unformly at random and enqueues the task into the deque with the fewer tasks (breaking ties arbitrarily). This part asks for a simple intuitive (not rigorous at all) proof that such a strategy leads to a more balanced distribution of tasks among deques compared to what happens under DR-Share (part $3(d)$).

Suppose that in the version of DR-Share-Mod given above no two enqueue attempts (i.e., enqueuing a new task into a deque) in the entire system happen at exactly the same time. Consider p such consecutive enqueue attempts. Let f_i be the fraction of the p deques that have received at least i tasks during that time. We will say that a task has rank i in a deque provided it is the i-th task landing in that deque during those p enqueue operations.

i. Argue that one can expect $f_{i+1} \leq (f_i)^2$.

ii. Use your result from above to show that for $i \geq 2$, $f_i \leq \frac{1}{2^{2^{i-2}}}$.

iii. Use your result from above to show that no task is likely to have a rank larger than $\log \log p$ during those p consecutive enqueue attempts.

APPENDIX 1: What to Turn in

One compressed archive file (e.g., zip, tar.gz) containing the following items.

- Source code, makefiles and job scripts.
- A PDF document containing all answers and plots.

APPENDIX 2: Things to Remember

- **Please never run anything that takes more than a minute or uses multiple cores on login nodes.** Doing so may result in account suspension. All runs must be submitted as jobs to compute nodes (even when you use Cilkview or PAPI).
- Please store all data in your work folder ($WORK), and not in your home folder ($HOME).
- When measuring running times please exclude the time needed for reading the input and writing the output. Measure only the time needed by the algorithm.