**CSE638: Advanced Algorithms, Spring 2013** **Date:** April 23

# Homework #3

**( Due: May 10 )**

**Task 1. [ 130 Points ] Matrix Multiplication.**

($a$) [ **20 Points** ] Figure 4 shows the standard iterative matrix multiplication algorithm and its 5 variants obtained by permuting the three nested ***for*** loops in all possible ways. Assuming that all three matrices ($X$, $Y$ and $Z$) are given in row-major order, analyze the cache complexity of each of the 6 versions of the algorithm[1].

($b$) [ **30 Points** ] Implement the cache-oblivious recursive matrix multiplication algorithm discussed in the class (Rec-MM in Lecture 15). But do not recurse down to matrices of size $1 \times 1$. 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 most cache-efficient iterative algorithms for part 1($a$) for multiplying the two subatrices. For simplicity let's assume that both $n$ and $m$ are powers of 2. For $n = 2^{13}$, empirically find the value of $m$ that gives you the smallest running time for Rec-MM. Produce a table or a graph showing how the running time varies as you change $m$.

($c$) [ **10 Points** ] Analyze the running time and cache-complexity of the recursive divide-and-conquer algorithm (discussed in the class) that converts a matrix from row-major layout to $Z$-Morton layout. Assume that $n$ is a power of 2. Argue that conversion from $Z$-Morton layout to row-major layout has the same complexity.

($d$) [ **30 Points** ] Consider the divide-and-conquer algorithm from part 1($c$) that converts from row-major to $Z$-Morton layout. Suppose instead of recursing down to matrices of size $1 \times 1$, we stop when we reach matrices of size $m \times m$ for some $m > 0$, and copy the entries from the input submatrix of size $m \times m$ to the output submatrix in row-major order. Similarly, for the algorithm that converts from $Z$-Morton to row-major. Let's call these two functions R2Z and Z2R, respectively. Now modify Rec-MM from part 1($b$) to Rec-MM-2 so that Rec-MM-2 works correctly on a matrix in the hybrid row-major-$Z$-Morton layout described above. Note that the base case size of Rec-MM-2 and the conversion routines must match. We assume for simplicity that both $n$ and $m$ are powers of 2. For $n = 2^{13}$, empirically optimize the value of $m$ for Rec-MM-2 as you did for Rec-MM in part 1($b$). However, this time include the running times of R2Z that puts the input matrices in the $Z$-Morton layout, and of Z2R that converts the output matrix back to row-major layout. Produce a table or a graph showing how the running time varies as you change $m$.

($e$) [ **40 Points** ] Run all your matrix multiplication implementations from parts 1($a$), 1($b$) and 1($d$) (parts 1($b$) and 1($d$) with optimized base) to produce four tables: running times,

[1] when you implement and compile these later in this task you must make sure that the compiler does not change the order in which the loops are nested in each of them

instructions excuted, L1 misses and L2 misses. In each table create a row for each $n = 2^t$, $8 \leq t \leq 15$, and in each row include the corresponding figures for each of your implementations. Create two columns for REC-MM-2: one including the cost of layout conversion and the other without that cost. Use `PAPI`[2] to find the number of L1 and L2 misses as well as the number of instructions executed. Explain your findings. Do you think the limited associativity of the caches had an impact on the running times? Why or why not?

**Task 2. [ 70 Points ] Protein Accordion Folding.**

A protein can be viewed as a string $\mathcal{P}[1:n]$ over the alphabet { *A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V* } of amino acids[3]. A protein sequence is never straight, and instead it folds itself in a way that minimizes the potential energy. Some of the amino acids (e.g., *A, I, L, F, G, P, V*)[4] are called *hydrophobic* as they do not like to be in contact with water (solvent). A desire to minimize the total hydrophobic area exposed to water is a major driving force behind the folding process. In a folded protein hydrophobic amino acids tend to clump together in order to reduce water-exposed hydrophobic area.

For simplicity, in this problem, we assume that a protein is folded into a 2D square lattice in such a way that the number of pairs of hydrophobic amino acids that are next to each other in the grid (vertically or horizontally) without being next to each other in the protein sequence is maximized. We also assume that the fold is always an *accordion fold* where the sequence first goes straight down, then straight up, then again straight down, and so on (see Figure 1).

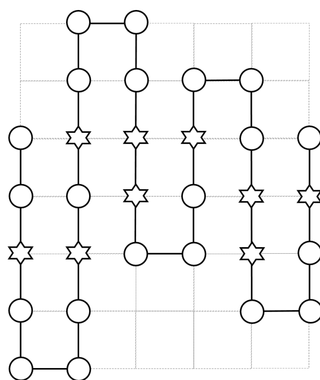

Figure 1: A protein accordion fold where each star represents a hydrophobic amino acid and each circle a hydrophilic one. The accordion score of this folded sequence is 4 which is not the maximum possible accordion score for this sequence.

The recurrence below shows how to compute the optimal *accordion score* of the protein segment $\mathcal{P}[i:j]$. The optimal accordion score is given by $\max_{1<j\leq n}\{\text{SCORE}(1,j)\}$.

[2]To load `PAPI` use "module load papi/4.1.2.1" on `Loanstar`. Check `http://icl.cs.utk.edu/papi/` for usage.

[3]Amino acids: Alanine (A), Arginine (R), Asparagine (N), Aspartic acid (D), Cysteine (C), Glutamic acid (E), Glutamine (Q), Glycine (G), Histidine (H), Isoleucine (I), Leucine (L), Lysine (K), Methionine (M), Phenylalanine (F), Proline (P), Serine (S), Threonine (T), Tryptophan (W), Tyrosine (Y), Valine (V).

[4]Alanine (A), Isoleucine (I), Leucine (L), Phenylalanine (F), Glycine (G), Proline (P), Valine (V).

$$\textsc{Score}(i,j) = \begin{cases} 0 & \text{if } j \geq n-1, \\ \max_{j+1<k\leq n} \{\textsc{Score-One-Fold}(i,j,k) + \textsc{Score}(j+1,k)\} & \text{otherwise.} \end{cases} \quad (1)$$

The function SCORE-ONE-FOLD$(i, j, k)$ counts the number of aligned hydrophobic amino acids when the protein segment $\mathcal{P}[i : k]$ is folded only once at indices $(j, j + 1)$. The function is given in Figure 3, and illustrated graphically in Figure 2.

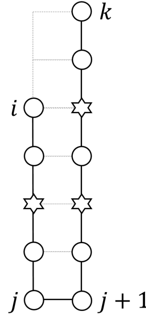


Figure 2: SCORE-ONE-FOLD( i, j, k ) counts the number of aligned hydrophobic amino acids when the protein segment $\mathcal{P}[i : k]$ is folded only once at indices $(j, j + 1)$. In this figure, each star represents a hydrophobic amino acid and each circle a hydrophilic one.

(*a*) [ **5 Points** ] Show that a straightforward iterative algorithm that solves recurrence 1 runs in $\mathcal{O}(n^4)$ time and uses $\Theta(n^2)$ space.

(*b*) [ **5 Points** ] Explain how to modify the algorithm from part 2($a$) to run in $\mathcal{O}(n^3)$ time at the cost of increasing the space complexity to $\Theta(n^3)$. Analyze its cache-complexity.

(*c*) [ **10 Points** ] Show that the algorithm from part 2($b$) can be modified to obtain a simple iterative algorithm that uses only $\mathcal{O}(n^2)$ space, but still runs in $\mathcal{O}(n^3)$ time. Analyze its cache-complexity.

(*d*) [ **50 Points** ] Convert your iterative algorithm from part 2($c$) into a recursive divide-and-conquer algorithm that runs in $\mathcal{O}(n^3)$ time, uses $\mathcal{O}(n^2)$ space, and incurs only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses, where $M$ is the size of the cache and $B$ is the block transfer size. Explain why the algorithm from part 2($b$) cannot be converted in the same way[5] to achieve the same aymptotic cache-complexity.

**Task 3.** [ **Optional, No Point** ] **Parallelizing Task 2.**

(*a*) [ **No Point** ] Parallelize your algorithms from parts 2($a$) and 2($b$). Compute the span and parallelism of each.

[5] convert to a recursive divide-and-conquer algorithm without changing the time and space complexities of the iterative algorithm

SCORE-ONE-FOLD( $i$, $j$, $k$, $\mathcal{P}$ )

($\mathcal{P}[1:n]$ is a sequence of amino acids, where $n > k-1 > j > i > 0$. This function counts the number of aligned hydrophobic pairs when the segment $\mathcal{P}[i:k]$ is folded at locations $j$ and $j+1$.)

1. $c \leftarrow 0$
2. ***for*** $l \leftarrow 1$ ***to*** $\min(\ j-i,\ k-j-1\ )$ ***do***
3.     ***if*** HYDROPHOBIC( $\mathcal{P}[j-l]$ ) ***and*** HYDROPHOBIC( $\mathcal{P}[j+1+l]$ ) ***then*** $c \leftarrow c+1$
4. ***return*** $c$

Figure 3: Count the number of aligned hydrophobic amino acids when a protein sequence is folded once.

# 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 TACC login nodes.** TACC policy strictly prohibits such usage. They reserve the right to suspend your account if you do so. 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. Do the same thing when you use PAPI.

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 $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)

1. ***for*** $i \leftarrow 1$ ***to*** $n$ ***do***
2. ***for*** $j \leftarrow 1$ ***to*** $n$ ***do***
3. ***for*** $k \leftarrow 1$ ***to*** $n$ ***do***
4. $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$

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 $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)

1. ***for*** $i \leftarrow 1$ ***to*** $n$ ***do***
2. ***for*** $k \leftarrow 1$ ***to*** $n$ ***do***
3. ***for*** $j \leftarrow 1$ ***to*** $n$ ***do***
4. $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$

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 $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)

1. ***for*** $j \leftarrow 1$ ***to*** $n$ ***do***
2. ***for*** $i \leftarrow 1$ ***to*** $n$ ***do***
3. ***for*** $k \leftarrow 1$ ***to*** $n$ ***do***
4. $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$

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 $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)

1. ***for*** $j \leftarrow 1$ ***to*** $n$ ***do***
2. ***for*** $k \leftarrow 1$ ***to*** $n$ ***do***
3. ***for*** $i \leftarrow 1$ ***to*** $n$ ***do***
4. $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$

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 $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)

1. ***for*** $k \leftarrow 1$ ***to*** $n$ ***do***
2. ***for*** $i \leftarrow 1$ ***to*** $n$ ***do***
3. ***for*** $j \leftarrow 1$ ***to*** $n$ ***do***
4. $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$

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 $Z[i, j] + \sum_{k=1}^{n} X[i, k] \times Y[k, j]$.)

1. ***for*** $k \leftarrow 1$ ***to*** $n$ ***do***
2. ***for*** $j \leftarrow 1$ ***to*** $n$ ***do***
3. ***for*** $i \leftarrow 1$ ***to*** $n$ ***do***
4. $Z[i, j] \leftarrow Z[i, j] + X[i, k] \times Y[k, j]$

Figure 4: Standard iterative matrix multiplication algorithm and its variants.