**CSE613: Parallel Programming, Spring 2012** **Date:** May 11

# Final Exam

**( 11:15 AM – 1:45 PM : 150 Minutes )**

- This exam will account for either 10% or 20% of your overall grade depending on your relative performance in the midterm and the final. The higher of the two scores (midterm and final) will be worth 20% of your grade, and the lower one 10%.
- There are three (3) questions, worth 120 points in total. Please answer all of them in the spaces provided.
- There are 14 pages including two (2) blank pages. Please use the blank pages if you need additional space for your answers.
- Page 14 contains some useful bounds. No additional cheatsheets are allowed.

## Good Luck!

| Question | Score | Maximum |
|---|---|---|
| 1. Maximal Independent Set ( MIS ) | | 40 |
| 2. Array-based Nonblocking Concurrent Queue | | 40 |
| 3. Distributed-Memory Algorithms | | 40 |
| Total | | 120 |

Name: _____

**Question 1. [ 40 Points ] Maximal Independent Set ( MIS ).** We are already familiar with a randmozied parallel algorithm (Luby's Algorithm) for solving the following problem.

Maximal Independent Set ( MIS )

**Input.** An undirected graph $G = (V, E)$. Vertex indices are unique integers from 1 to $|V|$.

**Output.** A set $MIS$ that includes all vertices in an MIS of $G$, and no other vertices.

In this question we will examine a simpler parallel randomized MIS algorithm as given below.

```
Par-Simple-Randomized-MIS( G = ( V, E ) )
 1. array M[ 1 : |V| ], r[ 1 : |V| ]
 2. MIS ← ∅
 3. while |V| > 0 do
 4.     parallel for each u ∈ V do
 5.         M[u] ← 1                                              {mark u}
 6.         r[u] ← RANDOM[ 0, 1 ]                  {r[u] gets a random number between 0 and 1}
 7.     parallel for each ( u, v ) ∈ E do
 8.         if r[u] ≥ r[v] then M[u] ← 0        {unmark u if it has a neighbor v with r[v] ≤ r[u]}
 9.         if r[v] ≥ r[u] then M[v] ← 0        {unmark v if it has a neighbor u with r[u] ≤ r[v]}
10.     parallel for each u ∈ V do
11.         if M[u] = 1 then
12.             MIS ← MIS ∪ {u}      {u is in MIS provided r[u] < r[v] for each neighbor v of u}
13.             V ← V \ {u}                          {remove vertices included in the MIS from V}
14.     parallel for each ( u, v ) ∈ E do
15.         if M[u] = 1 or M[v] = 1 then
16.             E ← E \ {( u, v )}    {remove edges from E with endpoint(s) included in the MIS}
17. return MIS
```

1(a) **[ 3 Points ]** Suppose in any specific iteration of the ***while*** loop in lines 3–16, a vertex $u$ is included in the $MIS$. Then $r[u] < r[v]$ for each neighbor $v$ of $u$ in the current graph. Additionally, for any given neighbor $v$ of $u$, if we also have $r[u] < r[x]$ for each neighbor $x$ ($\neq u$) of $v$, we call this event $(u \rightarrow v)$. Argue that $Pr[(u \rightarrow v)] \geq \frac{1}{d(u)+d(v)}$, where $d(u)$ (resp. $d(v)$) is the number of neighbors of $u$ (resp. $v$).

1($b$) [ **5 Points** ] Consider any specific iteration of the ***while*** loop in lines 3–16. Let $X_{(u \to v)}$ be a random variable indicating the number of edges connected to $v$ removed in the event of $(u \to v)$, and let $X = \sum_{(u,v) \in E} \left[ X_{(u \to v)} + X_{(v \to u)} \right]$, where $E$ is the set of edges in the current iteration. Prove that $\mathbb{E}[X] \geq |E|$.

[Hint: Use *linearity of expectation*: $\mathbb{E}[X] = \sum_{(u,v) \in E} \left[ \mathbb{E}[X_{(u \to v)}] + \mathbb{E}[X_{(v \to u)}] \right]$.]

1($c$) [ **5 Points** ] Argue that we overcounted in part 1($b$), and each edge was counted at most twice. Hence, the expected number of edges removed in any iteration of the ***while*** loop is at least $\frac{|E|}{2}$, where $E$ is the set of edges in the current iteration.

1($d$) [ **5 Points** ] An iteration of the ***while*** loop is *good* if it removes at least a quarter of the current set of edges. Show that an iteration is good with probability at least $\frac{1}{3}$.

[Hint: Show that, otherwise, it will contradict part 1($c$).]

1($e$) [ **5 Points** ] Show that the algorithm will terminate after at most $3 \log_{4/3} m + 1$ iterations of the ***while*** loop in expectation, where $m$ is the number of edges in the original input graph.

1($f$) [ **10 Points** ] Show that w.h.p. the algorithm will terminate after at most $24 \ln m$ iterations of the ***while*** loop.

[Hint: For each of the $24 \ln m$ iterations create an indicator random variable $X_i$ indicating whether the iteration is "good" (see part 1($d$)) or not, and then use an appropriate Chernoff bound from the appendix to show that w.h.p. at least $4 \ln m$ ($> \log_{4/3} m$) of those $24 \ln m$ iterations are good.]

1($g$) [ **7 Points** ] Given an undirected graph $G = (V, E)$, a *matching* $M$ is a subset of $E$ such that no two edges in $M$ share a common vertex. A matching is *maximal* if no edge can be added without violating the constraint above. Explain how you will transform the given graph $G$ into another undirected graph $G'$ so that a maximal independent set in $G'$ can be transformed into a maximal matching in $G$.

[Hint: $G'$ has $|E|$ vertices, and each vertex of $G'$ corresponds to a distinct edge of $G$.]

Use this page if you need additional space for your answers.

**Question 2. [ 40 Points ] Array-based Nonblocking Concurrent Queue.** We saw a nonblocking concurrent queue based on linked lists in the class. That queue used atomic *compare-and-store* or *CAS* operations instead of locks. CAS( *loc*, *x*, *y* ) is an atomic operation which checks if memory location *loc* contains *x*, and if so, it stores *y* at that location and returns TRUE, otherwise it fails and returns FALSE.

In this question we will examine a nonblocking concurrent queue based on circular arrays. The implementation is almost similar to the one we saw in the class.

```
1. array Q[ 0 : L − 1 ] of ⟨ VAL_TYPE val ⟩                          {the circular queue of size L}
2. integer front, rear                                               {front and rear pointers}
3. INITIALIZATION: front ← 0, rear ← 0          {queue is empty if front = rear, and full if rear = front + L}
                   parallel for i ← 0 to L − 1 do Q[ i ] ← ⟨ NULL ⟩          {empty locations contain NULL}
```

```
ENQUEUE( v )                                                         {enqueue item v into Q}
 1.  done ← FALSE                                                    {flag signalling if enqueue is successful}
 2.  while ( done ≠ TRUE )                                           {keep trying until enqueue is successful}
 3.      r ← rear                                                    {read rear}
 4.      x ← Q[r mod L]                                              {read the rear of the list}
 5.      if r = rear then                                            {if some other thread has not already changed rear}
 6.          if r ≠ front + L then                                   {if queue is not full}
 7.              if x.val = NULL then                                {if the rear of the list is empty}
 8.                  if CAS( Q[r mod L], x, ⟨ v ⟩ ) then              {try to store the item}
 9.                      CAS( rear, r, r + 1 )                       {increment rear if someone else has not already done so}
10.                      done ← TRUE                                 {enqueue is done}
11.              else      {rear of the list is not empty (meaning some other ENQUEUE is between lines 8 and 9)}
12.                  CAS( rear, r, r + 1 )                           {help that ENQUEUE increment rear}
```

```
DEQUEUE( )                                                           {dequeue an item from Q}
 1.  done ← FALSE                                                    {flag signalling if dequeue is successful}
 2.  while ( done ≠ TRUE )                                           {keep trying until dequeue is successful}
 3.      f ← front                                                   {read front}
 4.      x ← Q[f mod L]                                              {read the front of the list}
 5.      if f = front then                                           {if some other thread has not already changed front}
 6.          if f ≠ rear then                                        {if queue is not empty}
 7.              if x.val ≠ NULL then                                {if the front of the list is nonempty}
 8.                  if CAS( Q[f mod L], x, ⟨ NULL ⟩ ) then           {try to remove an item}
 9.                      CAS( front, f, f + 1 )                      {increment front if someone else has not already done so}
10.                      done ← TRUE                                 {dequeue is done}
11.              else      {front of the list is empty (meaning some other DEQUEUE is between lines 8 and 9)}
12.                  CAS( front, f, f + 1 )                          {help that DEQUEUE increment front}
13.  return x.val                                                    {return the dequeued item}
```

2($a$) **[ 10 Points ]** Show that the data structure may behave incorrectly at line 12 of both ENQUEUE and DEQUEUE.

[Hint: For DEQUEUE the incorrect behavior occurs if the queue was empty at line 4, but becomes nonempty again before line 6.]

2($b$) **[ 10 Points ]** Explain how you will fix the code to avoid the incorrect behavior in part 2($a$).

2($c$) [ **10 Points** ] Suppose we want to use the corrected ENQUEUE and DEQUEUE functions from part 2($b$) for the following memory management task.

We have an application that frequently allocates and deallocates (in no predictable order) memory locations to store and manipulate values of some compound data type (i.e., structure/record) DATA. We know that the application uses fewer than $L$ such DATA nodes simultaneously. So in order to reduce the overhead of memory allocation/deallocation we pre-allocate an array of size $L-1$, say, NODE[ $1 : L-1$ ] each entry of which is of type DATA.

Initially, all locations of NODE are free, and we enqueue the index of each such free location into the queue from part 2($b$) by calling ENQUEUE( $i$ ) for each $i \in [1, L-1]$.

Whenever a thread needs a new node it executes $i \leftarrow$ DEQUEUE( ) to grab the index $i$ of a free node, and whenever it no longer needs a node NODE[ $i$ ] it frees it by calling ENQUEUE( $i$ ).

We modify DEQUEUE so that it initializes the new node NODE[ $x.val$ ] between lines 7 and 8 by executing NODE[ $x.val$ ] $\leftarrow \Phi$, where $\Phi$ is the initial value.

Explain why even after DEQUEUE successfully executes the initialization statement as above, the calling thread may find that the index returned by DEQUEUE does not point to an initialized node.

2($d$) [ **10 Points** ] Explain how you will fix the initialization problem in part 2($c$). You will lose 5 points if you perform the initialization elsewhere (except between lines 7 and 8 of DEQUEUE). Performing the initialization completely outside of ENQUEUE and DEQUEUE is not an option.

[Hint: One approach is to attach a counter to each location of $Q$ in order to keep track of the number of times that location has been modified (i.e., undergone enqueue/dequeue). This will of course limit the number of times you can modify a queue location, but suppose that is not a problem for our application. ]

Use this page if you need additional space for your answers.

**Question 3. [ 40 Points ] Distributed-Memory Algorithms.** This question asks you to analyze two distributed-memory message-passing algorithms.

ODD-EVEN-TRANSPOSITION-SORT( $A[\,1 : n\,],\ p$ )

(Input is an array $A$ of $n$ distinct numbers to be sorted using $p$ processes (numbered from 1 to $p$). The output is $A$ in sorted order.)

1. the master process distributes the $n$ keys among all $p$ processes (including itself) as evenly as possible
2. each process sorts its local keys using quicksort
3. ***for*** $k \leftarrow 1$ ***to*** $p$ ***do***
4. ***if*** $k = odd$ ***then***
5. ***parallel for*** $i \leftarrow 1$ ***to*** $\frac{p-1}{2}$ ***do***
6. process $2i$ sends its sorted sequence to process $2i - 1$
7. process $2i - 1$ merges the two sorted sequences,
   keeps the smaller half, and
   sends the larger half to process $2i$
8. ***else***
9. ***parallel for*** $i \leftarrow 1$ ***to*** $\frac{p}{2} - 1$ ***do***
10. process $2i$ sends its sorted sequence to process $2i + 1$
11. process $2i + 1$ merges the two sorted sequences,
    keeps the larger half, and
    sends the smaller half to process $2i$
12. the master process collects the sorted keys from all processes

3($a$) [ **15 Points** ] Analyze the computation time ($t_{comp}$) and communication time ($t_{comm}$) of ODD-EVEN-TRANSPOSITION-SORT. Use the communication bounds given in the appendix.

MATRIX-VECTOR-MULTIPLICATION( $A[\,1:n,\ 1:n\,],\ x[\,1:n\,],\ p$ )

(Input is an $n \times n$ matrix $A$, and a vector $x$ of length $n$. The number of processes $p$ is a perfect square, and $\sqrt{p}$ divides $n$. We assume that the processes are arranged in a $\sqrt{p} \times \sqrt{p}$ grid, and $P_{r,c}$ denotes the process on row $r$ and column $c$ of this grid. Also input matrix $A$ is divided into $\sqrt{p} \times \sqrt{p}$ submatrices of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ each, and $A_{r,c}$ denotes the submatrix on row $r$ and column $c$ of this submatrix grid. We assume that process $P_{r,c}$ already holds submatrix $A_{r,c}$. The output is a vector $y[\,1:n\,]$ such that $y[\,i\,] = \sum_{j=1}^{n} A[\,i,\ j\,] \cdot x[\,j\,]$ for $1 \le i \le n$.)

1. input vector $x$ is divided into $\sqrt{p}$ segments of length $\frac{n}{\sqrt{p}}$ each, and $x_c$ denotes its $c$-th segment
2. the master process scatters $x$ so that process $P_{1,c}$ gets $x_c$
3. ***parallel for*** $c \leftarrow 1$ ***to*** $\sqrt{p}$ ***do***
4. process $P_{1,c}$ broadcasts $x_c$ to processes $P_{r,c}$, $1 \le r \le \sqrt{p}$
5. ***parallel for*** $r \leftarrow 1$ ***to*** $\sqrt{p}$ ***do***
6. ***parallel for*** $c \leftarrow 1$ ***to*** $\sqrt{p}$ ***do***
7. process $P_{r,c}$ computes $y_c[\,i\,] = \sum_{j=(c-1)\frac{n}{\sqrt{p}}+1}^{c\left(\frac{n}{\sqrt{p}}\right)} A[\,i,\ j\,] \cdot x[\,j\,]$ for $i \in \left[\ (r-1)\frac{n}{\sqrt{p}} + 1,\ r\left(\frac{n}{\sqrt{p}}\right)\ \right]$
8. ***parallel for*** $r \leftarrow 1$ ***to*** $\sqrt{p}$ ***do***
9. $P_{r,1}$ applies reduction on process row $r$ to compute $y[\,i\,] = \sum_{c=1}^{\sqrt{p}} y_c[\,i\,]$ for $i \in \left[\ (r-1)\frac{n}{\sqrt{p}} + 1,\ r\left(\frac{n}{\sqrt{p}}\right)\ \right]$
10. the master process gathers all $y[\,i\,]$ values from all processes $P_{r,1}$

3($b$) [ **25 Points** ] Analyze the computation time ($t_{comp}$) and communication time ($t_{comm}$) of MATRIX-VECTOR-MULTIPLICATION. Use the communication bounds given in the appendix.

# Some Useful Bounds

**Chernoff Bounds.** Let $X_1, \ldots, X_n$ be independent Poisson trials, that is, each $X_i$ is a 0-1 random variable with $Pr[X_i = 1] = p_i$ for some $p_i$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then the following bounds hold.

(1) For any $\delta > 0$, $Pr\left[X \geq (1+\delta)\mu\right] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$.

(2) For $0 < \delta < 1$, $Pr\left[X \geq (1+\delta)\mu\right] \leq e^{-\frac{\mu\delta^2}{3}}$.

(3) For $0 < \gamma < \mu$, $Pr\left[X \geq \mu + \gamma\right] \leq e^{-\frac{\gamma^2}{3\mu}}$.

(4) For $0 < \delta < 1$, $Pr\left[X \leq (1-\delta)\mu\right] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu}$.

(5) For $0 < \delta < 1$, $Pr\left[X \leq (1-\delta)\mu\right] \leq e^{-\frac{\mu\delta^2}{2}}$.

(6) For $0 < \gamma < \mu$, $Pr\left[X \leq \mu - \gamma\right] \leq e^{-\frac{\gamma^2}{2\mu}}$.

**Bounds on Some Communication Times.** In the bounds given below, $p$ is the number of processes involved, $m$ is the size of the message to send to (or to receive from) each process, $t_s$ is the startup time, and $t_w$ is the per-word transfer time.

(1) one-to-one transfer: $t_s + t_w m$

(2) one-to-all broadcast, all-to-one reduction: $(t_s + t_w m) \log p$

(3) scatter, gather: $t_s \log p + t_w m(p-1)$