CS 133 Midterm Name: Marek Pokorny UID: 204477064

Parallel & Distributed Computing (Spring 2016): Midterm Exam

87

Show all work on this exam. Time: **1 hr 50 mins**

Make sure to write your full name and student ID on the exam! Only class notes are allowed and no other book or electronic device is allowed.

---

1. (42 points, 6 points each) Please give brief (1-2 lines) answers to each of the following questions.

   A. What is Amdahl's Law? List two possible reasons that one may not achieve the speedup predicted by the Amdahl's Law, even when parallelization is possible?

   Amdahl's Law says that $Speedup = \frac{1}{(1-p)+p/n}$ where p = the fraction of code that can be executed in parallel and n = # of processors.

   1. Workload imbalance between threads
   2. Communication overhead between threads.

   B. When parallelizing a loop using OpenMP, under what condition shall one consider using dynamic scheduling? How can dynamic scheduling be specified?

   Use dynamic scheduling when the work done in each iteration of the loop is not equal, or changes at run time.

   #pragma omp parallel for schedule(dynamic, [min_chunk_size])

   C. In the 4-step Foster's methodology for developing parallel programs, why do we have both the partitioning and agglomeration steps? They seem to have opposite goals.

   Partitioning divides, agglomeration conquers. Partitioning lets us decide how we can parallelize in theory, while agglomeration groups these tiny partitions into larger groups for practical performance considerations.

   D. In exploiting pipelining for parallelism, what are the important factors that may limit getting good speedup?

   Data dependencies and limited resources (e.g. limited # of ALU's) may increase the initiation interval.

   E. Why loop exchange can help to improve performance of parallel program in some cases? When it is safe to exchange the order of two loops?

   Loop exchange may change the order of memory accesses to be more cache-friendly. It is safe to exchange two loops if doing so does not create any data-dependency conflicts.

F. Please list two major differences between OpenMP and MPI?

1. OpenMP is based on the fork-join model, while MPI is based on a message passing model.
2. OpenMP hides most of the communication from the programmer, while MPI forces the programmer to explicitly handle all communication

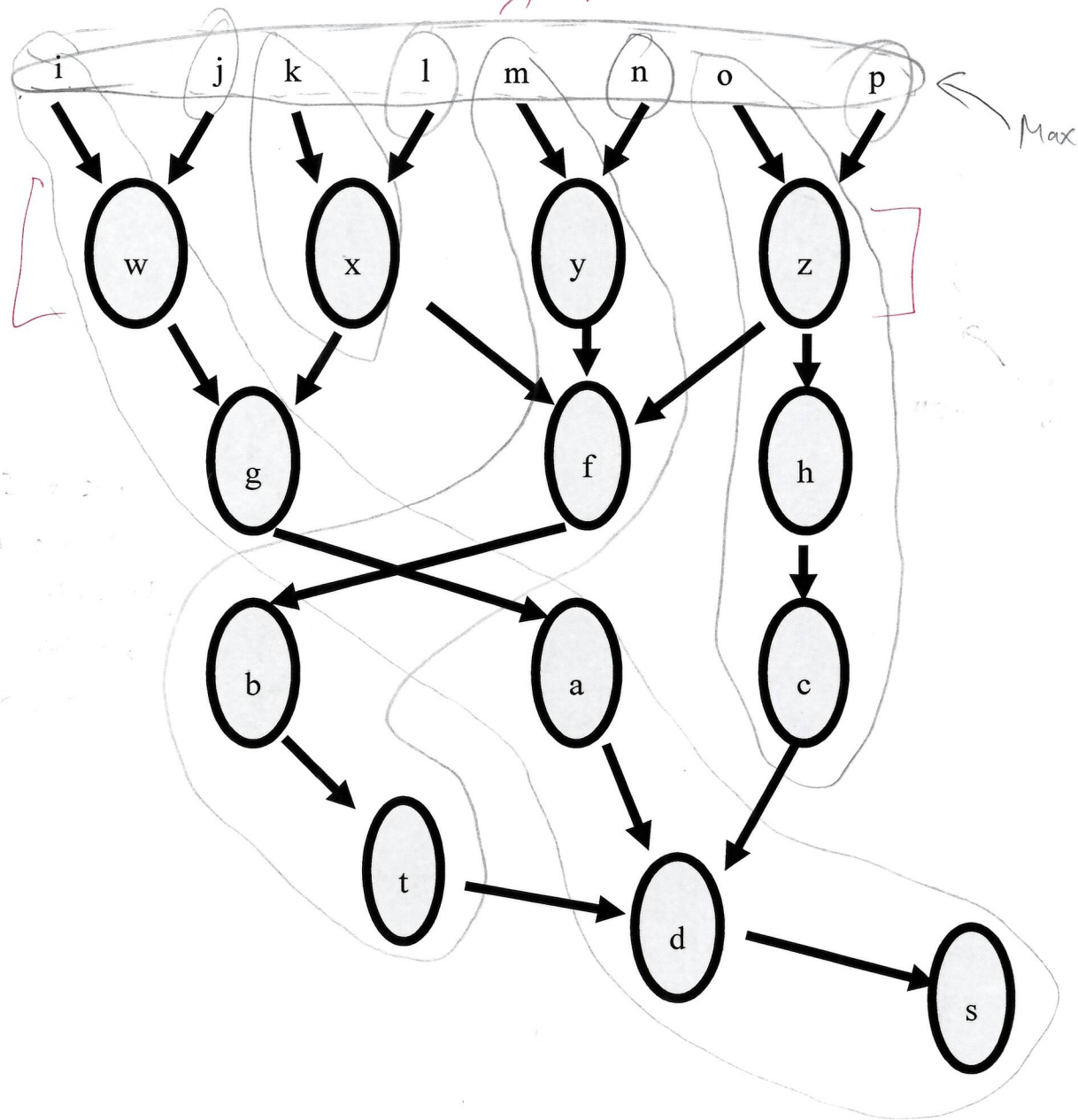
G. Given the following task graph, what is the maximum number of threads (or cores) that one may use so that beyond that, there is no further speedup potential. Explain why.

~~8~~, because the maximum anti-chain has size 8 (Dilworth's theorem)

4

-2

Inputs don't have to be computed.



2. (18 points) For the following sequential program, someone comes up with a parallel implementation using OpenMP on the right, is it correct? If not, please propose two different ways to fix the problem.

```
double a, b, t, output;
int i, n;
t = 0.0;
for (i = 0; i < n; i++) {
    a = (i + 1) / n;
    b = i * i;
    t = t + b / (1.0 + a*a);
}
output = t / n;
```

```
double a, b, t, output;
int i, n;
t = 0.0;
#pragma omp parallel for private(a,b)
for (i = 0; i < n; i++) {
    a = (i + 1) / n;
    b = i * i;
    t = t + b / (1.0 + a*a);
}
output = t / n;
```

No, it is incorrect due to race conditions with t.

Method 1:

add '#pragma omp atomic' before 't = t + b/(1.0+a*a)'

Method 2:

Change the initial #pragma to

#pragma omp parallel for private (a, b) reduction(+:t)

3. (20 points) For the following pseudo-code for computing all-pair shortest paths, (a) please specify all data dependencies; (b) please discuss if each of these loop transformations can be applied. If yes, please explain to which loop(s) it can be applied, if it's going to be beneficial, and why. If not, please explain why as well.
    a. Loop permutation
    b. Loop unrolling
    c. Loop tiling
    d. Loop fusion

```
int main() {
    int i, j, k;
    float **a, **b;
    // initialize a[][], b[][] as the 1-hop distance matrix
    for (k = 0; k < N; k++) {
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++)
            ① a[i][j] = min(a[i][j], b[i][k] + b[k][j]);
        }
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++)
            ② b[i][j] = a[i][j];
        }
    }
}
```

get min a[j]

a) • RAW (flow dependence): a[i][j] in ② depends on a[i][j] in ①
• RAW (flow dependence): b[i][k] and b[k][j] in ① depend on b[i][j] in ②
• ~~WAR~~ (anti dependence): in ①, a[i][j] depends on itself, b[i][k] and b[k][j]
: in ②, b[i][j] depends on a[i][j]

b) Loop permutation can be safely applied to the first i and j loops, which would not be beneficial because it would worsen spacial locality. Loop permutation can also be applied to the second set of i and j loops, but would also not be beneficial because of worsened spacial locality. It cannot be applied with the k loop, as this would break at least one of the data dependencies listed above.

Loop unrolling can always be safely applied to any loop. It would probably have little or no benefit, as the compiler will probably do this for us anyways. It would be most likely to be beneficial on the second j loop, as it is the smallest and simplest loop. Why?

we're talking about the transform itself and ~~not~~ don't focus on who did it (human or compiler)

Loop tiling can be done safely on both of the i-j loop-pairs. This would be beneficial in both cases because it would improve temporal locality, particularly with the i loop.

Loop fusion cannot be done. The first inner loop relies on the values of b remaining constant throughout the loop. Fusing the loops so that the two inner loops are together would change b during the loop.

4. (20 points) Parallelize N-body simulation.
N-body simulation finds the positions/velocities of a collection of interacting particles over a period of time.
The input is
- The particle struct array *Q* of *N* particles, each particle contains
  - The masses *m*,
  - The positions *p (3-dimension)* at time 0,
  - The velocities *v (3-dimension)* at time 0.

The output is
- The same particle struct array *Q* of *N* particles with updated positions/velocities.

The sequential algorithm is shown below (pseudo code):

```
void nbody(Particle *Q) {
  struct Particle new_Q[N]; // N is an external defined constant.
  for timestamp t from 1 to T { // T is an external defined constant.
      for each particle x in Q {
          double force[3] = {0.0}; // Total force of particle x
          for each particle y in Q {
              calcForce(force, Q[x], Q[y]);
              // Compute and accumulate the force between particle x and y
              // in a constant time.
          }
          new_Q[x] = update(Q[x], force);
          // Update the position/velocity for particle x in a constant time.
      }
      swap(Q, new_Q); // Update all positions in a constant time.
  }
}
```

1) Assuming that the array *Q* is stored at processor 0 initially (and the results will be collected at processor 0 as well at the end), please provide MPI-like pseudo code to implement the algorithm using *p* ($p = k^3$ *for some integer k)* processors with distributed memory. Please specify

a. What is your parallelization strategy?
b. How the data will be distributed?
c. What are the MPI communication functions that you are going to use at each step?
d. Please analyze the communication complexity (assuming the network topology is a hypercube), and provide the scalability function of your implementation.

2) Suppose the N articles are all in an L*L*L box and their distribution is roughly uniform. Moreover, forces from particles of distance larger than R can be ignored (assuming that L is a multiple of R). Can you come up with a more efficient solution than the algorithm list above? Please also provide MPI-like pseudo-code.

1) ~~if rank==0:~~ ×

```
broadcast(Q);  // broadcast all of Q to all threads
for timestamp t from 1 to T {
    for each particle x allocated to this thread {   // assume all particles evenly distributed to processors
        double force[3] = {0.0}
        for each particle y in Q {
            CalcForce(force, x, y);
            // compute and accumulate force
        }
        new_Q[x] = update(Q[x], force);
        // update position and velocity in constant time
    }
    Swap(Q, new_Q);
    ~~broadcast(Q, start, end);~~
    // all-to-all broadcast: broadcast this thread's share of particles to all other threads, and receive their shares
    for each other thread:
        recv(Q);
    Barrier();
}
```

you broadcast all...

└ from rank $\frac{N}{\#procs}$ to $(rank+1)\frac{N}{\#procs} - 1$

-3 you cannot use broadcast here unless one-to-all.

a) Data decomposition: tasks were separated by particles ✓

b) Data initially distributed by a one-to-all broadcast. For each timestep, all processors must know the state of all particles, so we must do an all-to-all broadcast each timestep. At the end, thread 0 will already have all results.

-1 c) First broadcast: MPI_Bcast

Each timestep: MPI_Bcast, followed by one MPI_recv for each other thread. ×

d) Communication complexity: dominant communication cost will be the all-to-all broadcast. $T = t_s \log p + t_w m(p-1) \simeq \Theta(p \log p + np)$

Scalability: $\sigma(n) = O(n^2)$ $\varphi(n) = O(n^2)$ $K(n,p) = \log p + n$ ✓

$T(n,p) = n^2/p + \log p + n$ $T(n,1) = O(n^2)$

$T_o(n,p) = cost(n,p) - cost(n,1) = n^2 + p\log p + pn - n^2 = pn + p\log p$

Assuming $n >> p$, $T_o = O(pn)$

$n^2 \geq Cpn \rightarrow n > Cp$ $M(n) = n$ $M(f(n)) = Cp$

2) Basically, only broadcast to all particles within a certain radius

~~if rank == 0:~~ ✗
Broadcast(Q)
bool sendTo[numThreads] = {false}; timestamp?
for each particle x assigned to this processor {
double force[3] = {0}
for each y in Q: still broadcast an entire array.
if dist(x, y) < R {
calcForce(force, x, y)
sendTo[tread that y belongs to] = true; }
new_Q[x] = update(Q[x], force)
}
swap(Q, new_Q)
for each rank i
if sendTo[i]
MPI_Isend(our share of Q)
MPI_recv(all other threads' particles that happened to get sent)
reset sendTo to all false
MPI_barrier
}

-4