Name:______________________________

SS# ________________________________

CS 320

Introduction to Parallel Programming

Midterm Exam

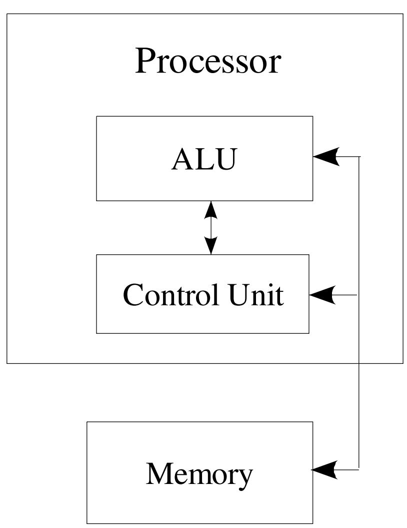
March 10,2000

1. **Short questions** [5 pts.]

(a) State Moore's Law

The number of Processing Elements on a chip doubles every 18 months/year.

(b) List the essential features of the Von Neumann computational model




The Von Neumann model has the following features:

- A unified data and instruction memory.
- A processor which understands instructions which are in the form of commands.
- The processor fetches instructions from the memory and executes them sequentially.

(c) Describe Flynn's taxonomy

SISD – Single Instruction Single Data
SIMD – Single Instruction Multiple Data
MIMD – Multiple Instruction Multiple Data
MISD – Multiple Instruction Single Data

(d) What is SPMD?

Single Program Multiple Data. In this model, there is a single program (or instruction stream) which is executed by all the processors while there are multiple data streams one for each processor.

(e) What are implicit tasks and master tasks?

Master tasks are those which either start the execution of a program or are created via an explicit system call (e.g. fork call in UNIX). Implicit tasks are created (implicitly) either at the start of the program or when the first parallel region is encountered; they remain idle until a parallel region is encountered when they are woken up by the master task to join the execution.

2. **ASCI** [5 pts.] Describe in 200 words or less what you found out about DOE's ASCI project

The Accelerated Strategic Computing Initiative is a result of cooperation between three defense laborataries (Los Alamos, Sandia, and Lawrence Livermore) and technology partners in the industry. Their main objective is to accurately simulate nuclear weapons to verify safety, reliability and performance parameters of the US nuclear stockpile. It uses several high performance parallel computers (with thousands of processors) from Digital Equipment Corporation, IBM and Sun for this. A Numerical Environment for Weapons Simulation (NEWS) has been developed to link the results of weapons codes to stockpile stewardship understanding and judgment.

3. **Loop parallelization** [5 pts.] Convert the following two loops into OpenMPform.

(a)

```
    do i=1,n
           a(i)=b(i)+c(i)
           d(i)=a(i)+x(i)
           s=s+a(i)+b(i)
    end do
```

```
 c$omp parallel do private(i) reduction(+:s)
    do i=1,n
           a(i)=b(i)+c(i)
           d(i)=a(i)+x(i)
           s=s+a(i)+b(i)
    end do
 C$omp end parallel do
```

(b)

```
    do i=1,n
           do j=1,n
                    k=k+2
                    a(i,k)=a(i-1,k)+b(i,k)
                    s=s+a(i,k)
           end do
    end do
```

```
    initk = k
 c$omp parallel do private(i,j) lastprivate(k) reduction(+:s)
    do j=1,n
           do i=1,n
                    k=initk+2*((i-1)*n+j)
                    a(i,k)=a(i-1,k)+b(i,k)
                    s=s+a(i,k)
           end do
    end do
 c$omp end parallel do
```

4. **Parallelization** [5 pts.] Add the necessary directives to execute the following code segment using several threads. The straight–line code should be executed by all threads redundantly whenever possible and the loop iterations should be distributed across the threads.

```
c$omp parallel private(x,t,i)
  x=sin(y)


  t=cos(z)

c$omp do shared(q)
  do i=1,n


          a(i) = b(i) + x * t


          if (i = n) q = a(i)


  end do
c$omp end do        <------  The implicit barrier here is necessary
c$omp psingle
  w = (x+t) * (x+t) + q
c$omp end psingle

c$omp do
  do i=1,n


          c(i) = e(i) + w


  end do
c$omp end do
c$omp end parallel
```

5. **Races conditions** [5 pts.] In the following four code segments, identify whether or not there are race conditions. In each case, explain why or why not and list all race conditions you can detect. Also, indicate what directives need to be inserted in each case to get rid of the race conditions.

(a)

```
c$omp    parallel private(i)
c$omp    pdo
         do i=1,n
              x(i)=x(i)+1
         end do
c$omp   end pdo nowait   <------------- (1)
c$omp   pdo
        do i=1,n
             y(i)=x(i) + 1
        end do
c$omp   end pdo nowait
c$omp   end parallel
```

The nowait marked (1) is the cause of a race condition; in one case it may happen that a thread enters the second loop and uses x(i) which has still not been updated by another thread in the first loop, or in the other case it may happen that the x(i) has been updated – thus it leads to a race condition.

To get rid of the race condition remove the nowait.

(b)

```
c$omp    parallel private(i)
c$omp    pdo
         do i=1,n
               x(i)=x(i)+1
         end do
c$omp    end pdo wait
         x(5) = x(5) + 1  <---------- (1)
c$omp    end parallel
```

The statement x(5)=x(5)+1 marked as (1) is the cause of race condition since multiple threads are reading and writing to the same variable. To remove the race either, (a) move the loop out of the parallel region, or (b) a psingle statement before (1).

(c)

```
c$omp    parallel do
         do i=1,n
                 a(i)=b(i+1)+b(i-1)+a(i)
         end do
```

There are no races in the program. (it is not necessary to have a private(i) – OpenMP takes care of loop variables).

(d)

```
c$omp    parallel private(i,j,k,t)
         k=some_function()
         t=y+x(k)
c$omp    barrier
         j=omp_get_thread_num()
         x(j)=t
c$omp    end parallel
```

There are no races in this segment.

6. **Array algorithms** [5 pts.] Given an n x n matrix A, write a vector algorithm to compute max (A(i,j)) for 1 <= i,j <= n. Assuming an array processor with n/2 PEs, compute speedup and efficiency.

**Vector Algorithm:**

```
max_row(1:n) = maxval(A(1:n,1:n), dim=2)
max = maxval(max_row(1:n))
```

*Speedup*: An efficient implementation of the first step will require 2n steps by assigning 2 rows to each processor. The second step will require $\log_2 n$ steps using the reduction method.
Thus: speedup = $T_S/T_P = n^2/(2n+\log_2 n)$
Efficiency = $T_S/PT_P = 2n/(2n+\log_2 n)$

**Using OpenMP:**

```
#pragma omp parallel private(i,j,thread_num)
{
#pragma omp single
    num_threads = omp_get_num_threads();
    thread_num = omp_get_thread_num();
 #pragma omp for
    for(i=0; i<n; i++)
       for(j=0; j<n; j++)
           if(maxp[thread_num]<A[i][j]) maxp[thread_num]=A[i][j];
}
for(step=1; step<num_threads; step=(step<<1))
#pragma omp parallel for private(i)
    for(i=0; i+step<num_threads; i=i+2*step)
        if(maxp[i]<maxp[i+step]) maxp[i]=maxp[i+step];
```

*Speedup*: The first parallel for loop will require 2n steps with n/2 processors (each processor being assigned 2 rows). The second parallel for loop will require 1 step, and since it is in a loop which has $\log_2 n$ iterations it will require $\log_2 n$ steps. Hence:
speedup = $T_S/T_P = n^2/(2n+\log_2 n)$

Efficiency = $T_S/PT_P = 2n/(2n+\log_2 n)$