CSE 443/543: High Performance Computing & Parallel Programming

Miami University

Exam 2

There are 14 questions for a total of 110 points.

Maximum Possible Score: 100 points

NAME (PRINT IN ALL CAPS): ____

General Instructions

- **Wait until you are instructed to start.**
- First, ensure you have all the 16 pages of this exam booklet before starting.
- This exam is closed notes and closed books. No discussions are permitted.
- Even if your final answers are incorrect, you will get partial credit if intermediate steps are clearly shown to highlight thought process. This applies to program tracing questions as well.
- All work must be written on the exam pages in order to be graded. Any scrap paper used, must be the extra sheets provided during the exam period.
- For programming questions: Be accurate with your C++ syntax. This includes appropriate use of braces, semicolons, and the proper use of upper/lowercase letters. Points will be deducted for syntax errors in programs and C++ statements that you write in this exam.
- You are permitted to use a simple calculator. You calculator cannot be: a graphing calculator, cannot have a QWERTY keyboard, or be integrated with other electronic devices such as: iPods, PDAs, and cellular phones.
- You have about two hours to complete the exam.

GOOD LUCK!

Multiple Choice Questions [30 Points]

1. Clearly circle only the best response for each question below. If your choice is not clearly indicated or if multiple choices seem to be selected then you will not earn any points for that question. If you change your answer, ensure that your final answer is clearly highlighted. **Each question is worth 2 points**.

   i. Each of the following options has a pair of statements. Assume all variables are defined (but you do not know and cannot assume their data type or values). Identify the pair of statements that are clearly concurrent.

      A. `a = b; c = b + a;`

      B. `x[i] = x[j]; x[k] = x[m];`

      [✓ **C.** `a = b; c = b;`]

      D. `*a = b; *c = b + a;`

   ii. Which one of the following is an example of a data parallel application

      A. Searching for a sub-matrix in a large matrix.

      B. Converting a video to different formats using different encoding algorithms.

      C. Searching Internet using Google, Bing, and Yahoo.

      D. None of the above

      E. All of the above

   iii. Typically, as number of threads used for an OpenMP program is increased, the *Ratio* of theoretical time to actual/observed time (see adjacent equation) to run/execute a program is:

      [✓ **A.** Always greater than or equal to 1.]

      B. Always less than or equal to 1.

      C. Always greater than or equal to 2.

      D. Always less than or equal to 2.

$$\mathit{Ratio} = \frac{\text{Actual/Observed time}}{\text{Theoretical time}}$$

iv. The time taken to access data in the memory hierarchy (consisting of: RAM, L1, L2, and L3 caches) of a CPU is different; that is time taken to access data in RAM is the same for different addresses but much higher than the time taken to access data in L1/L2/L3 cache. This architecture falls under the category:

A. NUMA

B. UMA

C. Hybrid UMA

D. Hybrid NUMA

v. A microprocessor has three levels of caches, namely Lα, Lβ, and Lδ. The access times (τ) of the caches follow the relation $\tau\,(L\beta) < \tau\,(L\alpha) < \tau(L\delta)$. Then the tier of cache that is closest to the ALU would typically be:

A. Lα

B. Lβ

C. Lδ

D. The supplied information is insufficient to distinguish the caches.

vi. Cilk Plus array notations of the form `a[0:2000]` improve performance primarily because they use:

A. Multiple threads

B. SSE Instructions

C. SIMD instructions

D. Multiple cores on a CPU

The `/usr/bin/time` command is being used to record the runtime characteristics of the adjacent method with exactly the same inputs but varying number of threads.

```
void digits(std::vector<long>& list) {
#pragma omp parallel for
  for (size_t i = 0; (i < list.size()); i++) {
    list[i] = log10(list[i]) + 1;
  }
}
```

vii. As the number of OpenMP threads are increased, the system time for running the `digits` method will:

A. The system time will increase

B. The system time will decrease

C. The system time will remain unchanged

D. This is not possible to predict the change in system time

viii. As the number of OpenMP threads are increased, the %CPU for running the `digits` method will:

A. The %CPU used will increase

B. The %CPU will decrease

C. The %CPU will remain unchanged

D. This is not possible to predict the change in %CPU

ix. The `/usr/bin/time` utility reports that an OpenMP program used 400% CPU. The number of threads *t* the program used is in the range:

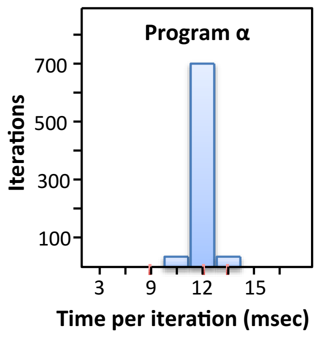
A. $0 < t \leq 4$

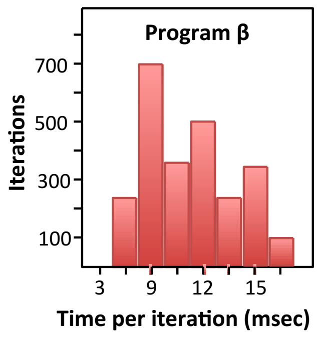
B. $0 < t < 100$

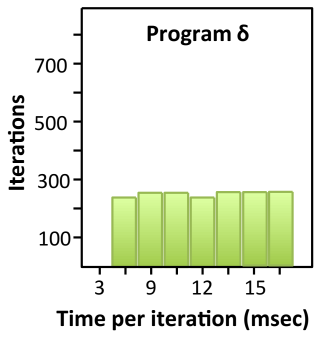
C. $4 < t < 100$

D. $t = 4$

The following charts illustrate histograms of time per iteration of OpenMP parallel `for`-loops from three different programs.







x. Given the charts, the program for which <u>static</u> scheduling would be ideal is:

[✓ **A.** Program-α]

B. Program-β

C. Program-δ

D. None of the above

xi. Given the above charts, the program for which dynamic scheduling would be ideal is:

A. Program-α

B. Program-β

[✓ **C.** Program-δ]

D. None of the above

xii. Given the above charts, the program for which guided scheduling would be ideal is:

A. Program-α

[✓ **B.** Program-β]

C. Program-δ

D. None of the above

xiii. Which one of the following are valid/true facts about using I/O streams (such as: `std::cin` or `std::cout`) in OpenMP parallel blocks:

A. I/O streams cannot be used in OpenMP parallel blocks.

B. I/O streams are never shared and each parallel block always gets its own private I/O streams.

C. I/O streams must always be opened and closed within OpenMP parallel blocks.

[✓ **D.** None of the above.]

xiv. The minimum number of OpenMP threads that are needed to cause a race condition is:

A. 0

B. 1

[✓ **C.** 2]

D. 3

xv. The minimum number of OpenMP threads required to run an OpenMP parallel `for`-loop is:

A. 0

[✓ **B.** 1]

C. 2

D. 3

Short Answer Questions [14 points]

2. Briefly (no more than 2 sentences) describe one advantage and one drawback of developing programs using complier-assisted parallelism approaches, such as OpenMP: [**4 points**]

   Advantage:

   Serial programs can be parallelized using multiple threads using compiler directives or pragmas. The compiler and libraries handle thread management, parallelization of for-loops, and load balancing.

   Disadvantage:

   Since thread management is performed by the system, sophisticated thread pool operations or critical section management cannot be combined with compiler-assisted parallelism approaches such as OpenMP.

3. Tabulate two significant differences between data and task parallelism. [**4 points**]

Data parallelism	Task parallelism
A large number of inputs are transformed using multiple threads and each input item is operated upon by only one thread.	A single input value is processed by all the threads.
All the threads perform the same transformation on a given input item.	Different threads perform different types of transformation on one input item.

4. OpenMP permits critical sections to have names. Using a short code fragment (does not need to be a complete method or a program; but it must be sufficiently complete to clearly convey your thoughts), describe the functional and performance an advantage of using named critical sections. Your descriptions must be sufficiently clear to earn full credit [**6 points**]

   Named critical sections permit multiple threads to concurrently access independent shared values without experiencing race conditions. Performance is improved because multiple threads can operate in parallel in the crtical sections CS1 and CS2 shown in the code fragment below. However, race conditions are avoided because no two threads can simultaneously modify the same-shared variable.

```
void oddEven(std::vector<int>& bigList) {
     int odd = 0, even = 0;
#pragma omp parallel for
     for (size_t i = 0; (i < bigList.size()); i++) {
          if (bigList[i] % 2 == 0) {
#pragma omp critical (CS1)
               even++;
          } else {
#pragma omp critical (CS2)
               odd++;
          }
     }
}
```

Code Analysis Questions [20 points]

5. Consider the following parallelized method called `speedy` in a software library:

```
int speedy(int& p1, int& p2) {
#pragma omp parallel for reduction (+: p1, p2)
   for (int k = 0; (k < 100); k++) {
      (p1 < p2) ? p1 += k : p2 += k;
   }
   return (p1 < p2)  ? p1 : p2;
}
```

The method operated correctly in many tests, but when a customer started using it, the method returned unpredictable results each time it was called. Based on these observations answer the following questions: [**4+2 = 6 points**]

a. Illustrate an example call (in C++) to the `speedy` method and clearly explain where and why the method call would result in an unpredictable output?

If the same variable is passed in as a parameter, then speedy method will have a race condition because the same memory is accessed/modified when the final reduction operation is performed. The following code fragment illustrates a source of race condition:

```
void caller() {
      int i = 10;
      speedy(i, i);
}
```

b. Based on your explanation to the previous sub-question, illustrate an example call (in C++) to the `speedy` method that would produce consistent results (like it allegedly did in many tests).

The race condition will not occur when two separate variables are passed in as shown below (and was most likely how all the testing was accomplished):

```
void caller() {
      int i = 10, j = 10;
      speedy(i, j);
}
```

6. The following code was compiled and run as show further below:

```
bool isPrime(long num);  // Detect if number is prime

int countPrimes(const std::vector<long>& list) {
    int count = 0;
#pragma omp parallel for reduction (+: count)
    for (size_t i = 0; (i < list.size()); i++) {
        count += isPrime(list[i]) ? 0 : 1;
    }
    return count;
}
```

```
$ export OMP_NUM_THREADS=8
$ /usr/bin/time ./CountPrimes > /dev/null
41.70user 0.03system 0:05.23elapsed 125%CPU (0avgtext+0avgdata 3468maxresident)k
0inputs+0outputs (0major+195minor)pagefaults 0swaps
```

a. Based on the above output, briefly describe why the program used only 125% of the CPU and not the full 800% allocated to it? [**4 points**]

If large numbers were all placed together then all threads will finish quickly and only one thread will spend most of the time computing. In other words, there is a load imbalance. Consequently, the average %CPU reported by `time` utility will be much less than 800%.

b. Briefly describe how the net CPU used by the program can be improved? [**2 points**]

Using a load balancing or scheduling strategy, such as `dynamic` or `guided` scheduling, can improve the % CPU.

7. Assume that the adjacent OpenMP program (saved in a file named `mp.cpp`) is compiled and run as shown below. Draw a diagram of the Fork-Join model illustrating each thread (label each thread with this thread number in your diagram) and the value of variable `sum` from each thread. Finally illustrate the overall output from the program. [**8 points**]

```
$ g++ -fopenmp mp.cpp -o mp
$ export OMP_NUM_THREADS=4
$ ./mp 3 4 5 6
```

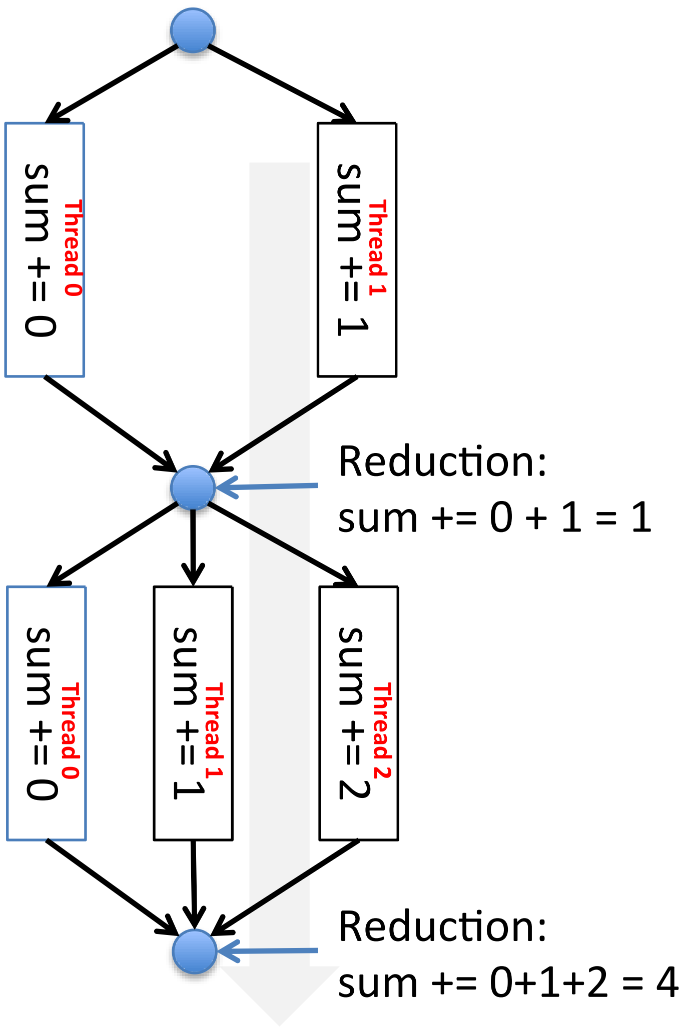
```
int lazy(int id) {
    sleep(1);  // do nothing for 1 second
    return id;
}

int main(int argc, char *argv[]) {
    int sum = 0;

#pragma omp parallel sections \
         reduction(+: sum) num_threads(2)
    {
#pragma omp section
        sum += lazy(omp_get_thread_num());
#pragma omp section
        sum += lazy(omp_get_thread_num());
    }

    std::cout << "sum = " << sum
              << std::endl;

#pragma omp parallel sections \
         reduction(+: sum) num_threads(3)
    {
#pragma omp section
        sum += lazy(omp_get_thread_num());
#pragma omp section
        sum += lazy(omp_get_thread_num());
#pragma omp section
        sum += lazy(omp_get_thread_num());
    }
    std::cout << "sum = " << sum
              << std::endl;
    return 0;
}
```

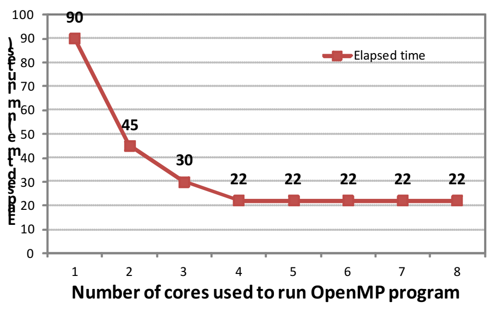


The output from the program is:

```
sum = 1
sum = 4
```

Quantitative Analysis Questions [13 points]

8. The performance profile of a deterministic OpenMP program executed on a workstation with 8 symmetric physical cores is shown in the adjacent graph. Using the adjacent chart answer the following questions:



a. Based on the single threaded runtime, what is the expected theoretical runtime with 3 and 5 threads? [**3 points**]:

i. Expected runtime with 3 threads: 30 sec (90/3=30)

ii. Expected runtime with 5 threads: 16 sec (90/5=16)

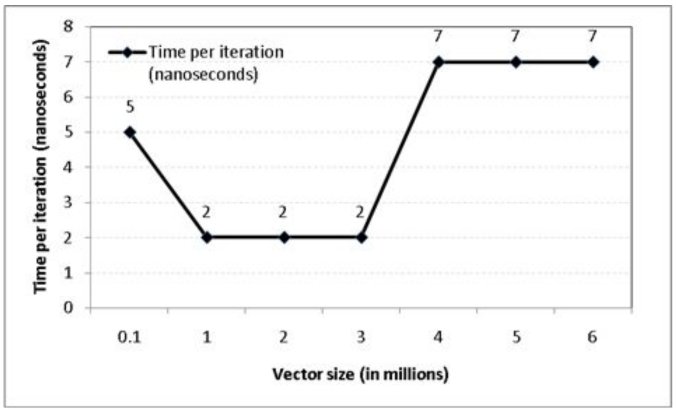
b. Is the application operating effectively as the number of threads was increased? Provide a plausible explanation for the observed performance characteristics [**4 points**].

Initially the application effectively used multiple threads until 4 threads. This part of the curve indicates that the program was able to effectively use 4 threads. However, the application did not use any additional threads because of one of the following reasons:

- Possibly the program is a task parallel application in which only 4 threads are used and other threads are simply idling without effectively contributing to the problem.
- The data is limited to just 4 values and OpenMP is unable to take advantage of the extra threads. Alternatively, there are critical sections in the code that prevent the program from scaling up to effectively use 8 threads.

9. The profile of Time-per-iteration for the `sum()` function (shown below) gathered on a SISD processor is shown in the graph below. Provide a succinct explanation for a plausible cause for the variation in performance characteristics observed for the `sum()` function by answering the sub-questions below:

```
int sum(int vector[],
        const int vectorSize) {
  long sum;
  int i;
  for(i = 0;(i < vectorSize);i++){
    sum += vector[i];
  }
  return sum;
}
```



a. Why did the time-per-iteration initially decrease from 5 nanoseconds to 2 nanoseconds when the vector size was increased? [**3 points**]
This curve is a typical reflection of cache underuse behavior in microprocessor system as pages of memory are fetched a reasonable number of spatial access is needed for caches to perform well. As the size of the vector is increased from a small value to a reasonable size the cache warms up and provides better performance.

b. Why did the time-per-iteration increase from 2 nanoseconds to 7 nanoseconds when the vector size was further increased? [**3 points**]
The change in timing is a classic indicator of increased cache misses. Once the cache is full, subsequent accesses to memory become slow. However, the misses stabilize at a constant rate (because of linear access) providing a consistent increase in time per iteration.

Programming Questions [33 points]

10. Improve performance (while preserving functionality) of the following method (that detects if characters in a string are vowels or consonants) to use Cilk Plus array notations. [**7 points**]

```
std::string detect(const std::string& source) {
    const std::string vowels = "aeiou";
    std::string result;
    for (char c : source) {
        result += (vowels.find(c) != -1UL ? 'V' : 'C');
    }
    return result;
}
```

```
std::string detect(const std::string& source) {
    std::string result(source);
    char *res = &result[0];
    const char* src  = source.data();
    const size_t len = source.size();

    if ((src[0:len] == 'a') || (src[0:len] == 'e') || (src[0:len] == 'i') ||
        (src[0:len] == 'o') || (src[0:len] == 'u')) {
        res[0:len] = 'V';
    } else {
        res[0:len] = 'C';
    }

    return result;
}
```

11. Parallelize the `isGeneticDisorder` method as a task parallel method using OpenMP. [**7 points**]

```
bool isDownsSyndrome(const std::string& genome);
bool isCysticFibrosis(const std::string& genome);
bool isHaemophilia(const std::string& genome);

bool isGeneticDisorder(const std::string& genome) {
       return isDownsSyndrome(genome) || isCysticFibrosis(genome) ||
              isHaemophilia(genome);
}
```

```
bool isGeneticDisorder(const std::string& genome) {
      bool result = false;

#pragma omp parallel reduction(||: result)
      {
#pragma section
            result = isDownsSyndrome(genome);

#pragma section
            result = isCysticFibrosis(genome);

#pragma section
            result = isHaemophilia(genome);
      }
      return result;
}
```

12. The output from the parallelized `enigma` method is almost always garbled as shown. Recode the enigma method such that output is displayed consistently in order from thread #0, thread #1, to thread #*n* ($n \geq 1$). [**8 points**]

```
void compute(int);
void output(int);

void enigma() {
#pragma omp parallel
  {
    int id = omp_get_thread_num();
    compute(id);
    output(id);
  }
}
```

Current output

```
Output from Thr #2
OutOutput from Thr #1
put from Output fr Thr #3
om Thr #0
```

Expected output

```
Output from Thr#0
Output from Thr#1
Output from Thr#2
Output from Thr#3
```

```
void enigma2() {
    int turn = 0;  // shared turn variable
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        compute(id);  // compute first
        int myTurn = 0;
        while (myTurn < id) {  // wait for my turn
#pragma omp critical
            myTurn = turn;  // Avoid race conditions
        }
        output(id);  // print my output
#pragma omp critical
        turn++;  // Let next thread output
    }
}
```

13. Parallelize the following `max` method (which returns the maximum element in a given `list`) using OpenMP. [**11 points**]

```
int max(std::vector<int>& list) {
  return *std::max_element(list.begin(), list.end());
}
```

```
int  max(std::vector<int>& list) {
       std::vector<int> localMax;
#pragma omp parallel {
#pragma omp critical(init)
       localMax.resize(omp_get_num_threads());
       const int thrNum = omp_get_thread_num();
#pragma omp for
       for(int i = 0; (i < list.size()); i++) {
              localMax[thrNum] = std::max(localMax[thrNum], list[i]);
       }
  return *std::max_element(localMax.begin(), localMax.end());
}
```

14. As the names suggest, `veryFastMethod` takes less than a millisecond to run while `slowMethod1` and `slowMethod2` both take about 750 milliseconds to run. I/O overheads have been measured to be negligible. Parallelize the `processFile` method using OpenMP while preserving the exact order of outputs. [**12 points**]

```
std::string veryFastMethod(std::string& s1, std::string& s2);
std::string slowMethod1(std::string& s);
std::string slowMethod2(std::string& s);

void processFile(std::istream& is) {
  std::string s1, s2;
  while ((is >> s1 >> s2)) {
    std::cout << veryFastMethod(slowMethod1(s1), slowMethod2(s2));
  }
}
```

```
void processFile(std::istream& is) {
  std::istream_iterator<std::string> reader, eof;
  std::vector<std:;string> s(reader, eof);

#pragma omp parallel for num_threads(n)
  for(size_t j = 0; (j < s.size()); j++) {
    s[j] = (j % 2 == 0) ? slowMethod1(s[j]) : slowMethod2(s[j]);
  }

  for(int j = 0; (j < s.size()); j += 2) {
    std::cout << veryFastMethod(s[j], s[j+1]);
  }
}
```