15-418/618, Spring 2018
Exercise 3

Assigned:	Mon., Apr. 2
Due:	Fri., Apr. 6, 1:30 pm

You will submit an electronic version of this assignment to Gradescope as a PDF file. For those of you familiar with the LaTeX text formatter, you can download a template file at:

http://www.cs.cmu.edu/∼418/exercises/ex3-answer.tex

Instructions for how to use this template are included as comments in the file. Otherwise, you can use the solution template:

http://www.cs.cmu.edu/∼418/exercises/ex3-answer.pdf

You can either: 1) electronically modify the PDF file, or 2) print it out, write your answers by hand, and scan it. In any case, we expect your solution to follow the formatting of the solution template.

Problem 1: Understanding OpenMP

Figures 2–5 show a number of different OpenMP implementations of a function that will sum all of the elements in an n-element array `d`. The challenge is to aggregate all of the data values into single value `sum`. The different implementations are as follows:

Baseline Sequential implementation

OMP critical Direct OMP to update the value of `sum` atomically

OMP row-wise accumulation Each thread accumulates a partial sum in a separate memory location. These locations form the first row of a 16 × 16 array `accum`.

OMP column-wise accumulation Each thread accumulates a partial sum in a separate memory location. These locations form the first column of a 16 × 16 array `accum`.

OMP threading Use OMP to generate a set of threads, and then (as is typical with a threaded implementation of this task) have each thread compute a partial sum of some portion of the data

OMP reduce Let OMP figure out how to perform a sum reduction on the data

Figure 6 shows the performance of the different implementations on a GHC machine, ranging from one to 16 threads. Performance is measured in giga-FLOPS, with the baseline implementation achieving around 1 GFLOPS.

```
double sum_array(double *d, int n) {
    int i;
    double sum = 0.0;
    for (i = 0; i < n ; i++) {
        double val = d[i];
        sum += val;
    }
    return sum;
}
```

Figure 1: Baseline

```
double sum_array_omp_critical(double *d, int n) {
    double sum = 0.0;
    int i;
    #pragma omp parallel for schedule(static)
    for (i = 0; i < n ; i++) {
        double val = d[i];
        #pragma omp critical
        { sum += val; }
    }
    return sum;
}
```

Figure 2: OMP critical

1A: Performance of OMP Critical

Referring to Figure 2, using the `omp critical` pragma ensured that the sum is computed correctly, but the performance is very poor—ranging from 0.036 GFLOPS for one thread down to 0.006 for 16 threads. To what do you attribute this poor performance? Why does it get worse with more threads?

1B: Thread-specific accumulation

Both versions in Figure 3 achieve speedup with more threads, but their sequential performance (around 0.40 GFLOPS) is worse than that of the baseline. To what do you attribute this performance?

1C: Row- vs. column-wise accumulation

The column-wise accumulation code in Figure 3 has slightly better speedup than the row-wise accumulation code. To what do you attribute this difference?

```
volatile double accum[16][16];

double sum_array_omp_row(double *d, int n) {
    int nthread = omp_get_max_threads();
    double sum = 0.0;
    int i, t;
    for (t = 0; t < nthread; t++)
        accum[0][t] = 0.0;
    #pragma omp parallel for schedule(static)
    for (i = 0; i < n ; i++) {
        double val = d[i];
        int myt = omp_get_thread_num();
        accum[0][myt] += val;
    }
    for (t = 0; t < nthread; t++)
        sum += accum[0][t];
    return sum;
}

double sum_array_omp_col(double *d, int n) {
    int nthread = omp_get_max_threads();
    double sum = 0.0;
    int i, t;
    for (t = 0; t < nthread; t++)
        accum[t][0] = 0.0;
    #pragma omp parallel for schedule(static)
    for (i = 0; i < n ; i++) {
        double val = d[i];
        int myt = omp_get_thread_num();
        accum[myt][0] += val;
    }
    for (t = 0; t < nthread; t++)
        sum += accum[t][0];
    return sum;
}
```

Figure 3: OMP row-wise and OMP column-wise accumulation

```
double sum_array_omp_thread(double *d, int n) {
    int nthread = omp_get_max_threads();
    double laccum[nthread];
    double sum = 0.0;
    int t;
    #pragma omp parallel
    {
        int myt = omp_get_thread_num();
        int istart = myt * n/nthread;
        int iend = (myt == nthread-1) ? n : (myt+1)*n/nthread;
        int i;
        double psum = 0.0;
        for (i = istart; i < iend ; i++) {
            double val = d[i];
            psum += val;
        }
        laccum[myt] = psum;
    }
    for (t = 0; t < nthread; t++)
        sum += laccum[t];
    return sum;
}
```

Figure 4: OMP threading

```
double sum_array_omp_reduce(double *d, int n) {
    double sum = 0.0;
    int i;
    #pragma omp parallel for schedule(static) reduction (+:sum)
    for (i = 0; i < n ; i++) {
        double val = d[i];
        sum += val;
    }
    return sum;
}
```

Figure 5: OMP reduce

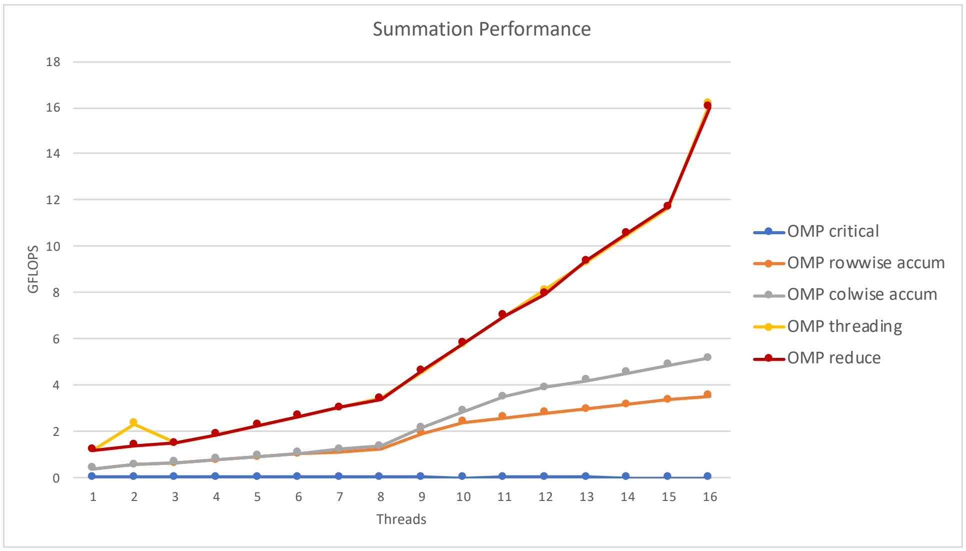

Figure 6: Performance of summation functions

1D: Top performers

The implementation making explicit use of threads (Figure 4) and the one making use of OpenMP reduction (Figure 5) have nearly identical performance. To what do you attribute this similarity?

1E: Top performers vs. others

Why do the two top performers outperform the versions using row-wise and column-wise accumulators?

1F. Bonus Question: Why the upward bends?

The curves of Figure 6 indicate that the performance scales *better* for 8–16 threads than it did for 1–8. To what do you attribute this phenomenon? (Keep in mind that the machine has eight cores with two-way hyperthreading, and that the machine adjusts its clock speed dynamically.)

Problem 2: Scaling Models

This problem explores the two scaling models presented on slides 5–9 in the lecture on Heterogeneous Parallelism:

http://www.cs.cmu.edu/~418/lectures/19_heterogeneity.pdf.

These are based on the paper "Amdahl's Law in the Multicore Era," by Mark D. Hill and Michael R. Marty of the University of Wisconsin, published in *IEEE Computer*, July, 2008.

In both models, we express computing performance and resources (e.g., chip area) relative to a baseline processor. In scaling to a processor with r resource units (scaled such that the baseline processor has $r = 1$), we obtain a processor with single-threaded performance $\mathit{perf}(r)$ (scaled such $\mathit{perf}(1) = 1$.)

The slides show graphs for the case of $\mathit{perf}(r) = \sqrt{r}$. This function is plausible—it captures the idea that increasing processing resources will improve performance, but not in a linear way. Increasing to large values of r yields diminishing returns in terms of single-threaded processor performance. We will generalize this model to one with $\mathit{perf}(r) = r^{\alpha}$, where α is value with $0.0 \leq \alpha \leq 1.0$. (For example, the slides are based on $\alpha = 0.5$.)

For scaling the system design to use n resource units, two options are considered:

[Homogeneous: We create a new processor design that uses r resource units, and populate the system with $\lfloor n/r \rfloor$ identical processors.

Heterogeneous: We create a design that uses r resource units, and then create a system with one of these more powerful processors, plus $\lfloor n - r \rfloor$ baseline processors.

As with the conventional presentation of Amdahl's Law, we assume that the problem to be solved has some fraction f that can use arbitrary levels of parallelism, while the remaining fraction $1 - f$ must be executed sequentially.

With performance normalized to the baseline processor, Amdahl's Law states that the speedup over the baseline is given by the equation

$$S = \frac{1}{T_{\text{seq}} + T_{\text{par}}} \tag{1}$$

Where T_{seq} is the time required to execute the sequential portion of the code and T_{par} is the time required to execute the parallel portion of the code.

2A: Homogeneous Model Speedup

We can refine slide 6 to state that the speedup for the homogeneous model is:

$$S_{\text{ho}} = \frac{1}{\frac{1-f}{\mathit{perf}(r)} + \frac{f}{\mathit{perf}(r) \cdot \lfloor \frac{n}{r} \rfloor}} \tag{2}$$

The factor n/r given in the original model is refined to be $\lfloor n/r \rfloor$, to account for the fact that we may need to let some of the resources go to waste if n/r is not an integer.

Explain how Equation 2 follows the form of Equation 1. What resources are used and what is the duration of T_{seq}? What resources are used and what is the duration of T_{par}?

2B: Heterogeneous Model Speedup

We can refine Slide 8 to state that the speedup for the heterogeneous model is:

$$S_{\mathrm{he}} = \frac{1}{\frac{1-f}{\mathit{perf}(r)} + \frac{f}{\mathit{perf}(r)+\lfloor n-r \rfloor}} \tag{3}$$

In this equation, we replace the term $n - r$ with $\lfloor n - r \rfloor$ to reflect that fact that if r is not an integer, some of the resources will be wasted.

Explain how Equation 3 follows the form of Equation 1. What resources are used and what is the duration of T_{seq}? What resources are used and what is the duration of T_{par}?

2C: Optimizing Parameter r for the Homogeneous Model

For either of these models, we can find a value r^* that achieves maximum speedup as a function of f, n, and α. Keeping n fixed at 256 but varying parameters α and f, fill in the table below giving the values of r^* and n/r^* for the Homogeneous model, and the resulting speedups.

Hints: One can readily see that Equation 2 will be maximized for values of r such that n/r is an integer. For any noninteger, it would be better to increase r to avoid wasting resources. For your convenience, you can download an Excel spreadsheet at

http://www.cs.cmu.edu/∼418/exercises/ex3-model.xlsx

This spreadsheet generates the graphs shown on slide 9 for the case of $n = 256$. You can adapt this spreadsheet to determine the values of r^*.

α	f	r^*	n/r^*	S_{ho}
0.4	0.80			
0.4	0.95			
0.4	0.99			
0.8	0.80			
0.8	0.95			
0.8	0.99			

2D: Understanding 2C

Explain the trends you see in your answer to 2C: Why do changes to α and/or f cause the changes to r^* and S_{ho} that you have computed?

2E: Optimizing Parameter r for the Heterogeneous Model

Fill in the table below giving the values of r^* for the Heterogeneous model, and the resulting speedups.

Hint: One can readily see that Equation 3 will be maximized for values of r such that r is an integer. For any noninteger value of r, it would be better to increase to the next highest integer to avoid wasting resources.

α	f	r^*	S_{he}
0.4	0.80		
0.4	0.95		
0.4	0.99		
0.8	0.80		
0.8	0.95		
0.8	0.99		

2F: Understanding 2E (and 2C)

Explain the trends you see in your answer to 2E: why do changes to α and/or f cause the changes to r^* and S_{he} that you have computed? How do these trends compare to those computed for the Homogeneous model?