CAAM 520, Spring 2020 - Problem Set 1

Eappen Nelluvelil

February 21, 2020

1. *CPU's and caches*

   (a) For a problem of size $n = 1024$, would you expect any difference in the performance of code A and code B? Explain your answer.

   Both pieces of code carry out $2mn$ operations (one multiplication and one addition in the innermost `for` loops), but code A computes the element-wise product of vectors `b` and `c` When $n = 1024$, vectors `a`, `b`, and `c` each take up 8192 bytes (collectively 24576 bytes, or 24 KiB), so all three vectors can be stored in, read from, and written to the L1 cache in code A and code B. Thus, code A and code B are not limited by memory bandwidth, so the performance of both pieces of code should be similar.

   (b) For a problem of size $n = 2048$, would you expect any difference in the performance of code A and code B? Explain your answer.

   When $n = 2048$, vectors `a`, `b`, `c` each take up 16384 bytes (collectively 49152 bytes, or 48 KiB). We see that this exceeds the available L1 cache memory, and at most two vectors can be stored in, read from, and written to the L1 cache in code A and code B. Code A sums the element-wise product of vectors `b` and `c` m times and writes this sum to the vector `a`. This code works with the entirety of vectors `a`, `b`, and `c` in each iteration of the outermost `for` loop. On the other hand, code B does the same task, but computes this running sum and writes it to `a` entry by entry. Because all three vectors cannot be stored in, read from, and written to the L1 cache, we see that code A reads and writes to main memory more often than code B, which implies that code A is much slower than code B. We see that the processor clock rate is not relevant because memory bandwidth is independent of processor clock rate. Thus, we expect code B to be much faster than code A.

   (c) For a problem of size $n = 2048$, would you expect any difference in the performance of code A and code B if the clock frequency was lowered to 2.6 GHz? Explain your answer.

   No, we would not see much of a difference. In the earlier sub-part, we saw that the main performance bottleneck was how fast the processor could read and write to memory, not how many CPU cycles the processor was capable of carrying out per second.

2. *Amdahl's law*

   (a) Compute the speedup and parallel efficiency achieved when using $N = 4$ workers.

   From the lecture slides, we know that the speedup S^N is given by

   $$S^N = \frac{1}{s + \frac{1-s}{N}},$$

   where s is the fraction of work that must be done sequentially and N is the number of workers. Here, we

take $s = 0.1$ and $N = 4$. The speedup S^4 is then

$$\begin{aligned} S^4 &= \frac{1}{0.1 + \frac{1-0.1}{4}} \\ &= \frac{1}{0.1 + \frac{0.9}{4}} \\ &= \frac{1}{0.1 + 0.225} \\ &= \frac{40}{13} \\ &\approx 3.07692307 \end{aligned}$$

From the lecture slides, we know that the parallel efficiency is given by

$$\begin{aligned} E_f^N &= \frac{S^N}{N} \\ &= \frac{\frac{1}{s + \frac{1-s}{N}}}{N} \\ &= \frac{1}{Ns + 1 - s} \end{aligned}$$

The parallel efficiency in this case is given by

$$\begin{aligned} E_f^4 &= \frac{1}{4(0.1) + 1 - 0.1} \\ &= \frac{1}{0.4 + 0.9} \\ &= \frac{10}{13} \\ &\approx 0.76923076923 \end{aligned}$$

(b) Compute the limits of the speedup and parallel efficiency as the number of workers is increased indefinitely ($N \to \infty$).

Fixing the amount of work that can be done sequentially as $s = 0.1$ and taking $N \to \infty$, we see that

$$\begin{aligned} \lim_{N\to\infty} S^N &= \lim_{N\to\infty} \frac{1}{s + \frac{1-s}{N}} \\ &= \frac{1}{s + (1-s)\lim_{N\to\infty} \frac{1}{N}} \\ &= \frac{1}{s} \\ &= 10 \end{aligned}$$

and

$$\begin{aligned} \lim_{N\to\infty} E_f^N &= \lim_{N\to\infty} \frac{S^N}{N} \\ &= 10 \lim_{N\to\infty} \frac{1}{N} \\ &= 0 \end{aligned}$$

(c) Compute the maximum number of workers for which a parallel efficiency is at least 80%, i.e., $E^N \geq 0.8$, can be achieved.

We see that

$$\begin{aligned} E^N \geq 0.8 &\iff \frac{\frac{1}{s+\frac{1-s}{N}}}{N} \geq 0.8 \\ &\iff \frac{1}{s+\frac{1-s}{N}} \geq 0.8N \\ &\iff 1 \geq 0.8Ns + 0.8\,(1-s) \\ &\iff \frac{1-0.8\,(1-s)}{0.8s} \geq N \\ &\iff 3.5 \geq N, \end{aligned}$$

i.e., the largest number of workers we can use to achieve a parallel efficiency of at least 80% is $N = 3$.

3. *Race conditions and thread safety*

```
int counter; // global variable

int increment()
{
    counter++;
    return counter;
}
```

(a) Is the `increment()` function *thread safe*, i.e., does it still exhibit the desired behavior when called concurrently by multiple (OpenMP) threads? Explain your answer.

The above function is not thread safe because the post-increment operator is not thread safe. If two threads call the increment function at the same time, it is possible for one thread to access the counter variable and increment it while the other thread in the process of accessing the counter variable. This can lead to the first thread correctly updating the counter variable by one and the second thread incorrectly updating the counter variable.

(b) If the function is not thread safe, modify the code to make it thread safe.

```
int counter; // global variable

int increment()
{
    #pragma omp parallel
    {
        #pragma omp critical
        {
            counter++;
        }
    }
    return counter;
}
```

This code is now thread safe as the critical directive forces all threads to increment `counter` one at a time.

(c) Add a function `void reset()` which resets the value of the global `counter` variable to zero. Make sure that the `increment()` and `reset()` functions are still thread safe, keeping in mind that both functions access the same *global* variable.

```
void reset()
{
    #pragma omp parallel
    {
        #pragma omp critical
        {
            counter = 0;
        }
    }
}
```

The `reset` function is thread safe, despite `counter` being a global variable, because if multiple threads call both the `increment` and `reset` functions, the critical sections in both functions are mutually exclusive. That is, if thread A is in either critical section, the other threads are blocked from executing code in either critical section until thread A is done.

4. *Sparse matrices and the Jacobi method using OpenMP*

(a) See code.

(b) See code.

(c) Test your implementation using the matrices below. For each test case, report the number of Jacobi iterations needed to achieve a tolerance of $\epsilon = 10^{-3}$.

i. `nasa2146` (file `nasa2146.mtx`) with $\omega = 0.25$
The Jacobi method implementation took 4004 iterations to reach a relative residual of 9.991008×10^{-4}.

ii. `plbuckle` (file `plbuckle.mtx`) with $\omega = 0.6$
The Jacobi method implementation took 21517 iterations to reach a relative residual of 9.997921×10^{-4}.

iii. `bcsstk08` (file `bcsstk08.mtx`) with $\omega = 0.7$
The Jacobi method implementation took 13559 iterations to reach a relative residual of 9.995399×10^{-4}.

(d) Your implementation of the Jacobi method should contain a `for` loop over the rows of the sparse matrix A. Should OpenMP's `static` or `dynamic` schedule be used for this loop, or does the optimal choice depend on the matrix at hand? If so, which properties of A should influence your choice?

A `static` schedule assigns a fixed number of iterations to each thread before a `for` loop is executed, whereas a `dynamic` schedule assigns iterations to each thread while a `for` loop is executing, i.e., once a thread finishes its assigned iterations, it can request more iterations. It is easy to see that the `static` schedule is appropriate if we assume that each iteration of a `for` loop is expected to take the same amount of work to complete, whereas the `dynamic` schedule is appropriate if the amount of work per iteration varies highly across iterations.

We know that the i^{th} entry of the matrix-vector product Ax is given by the dot product of the i^{th} row of A with x. In this code, we work with sparse matrices A, i.e., A is mostly zero, but we do not know ahead of time if A is uniformly sparse or A contains rows that are significantly more dense than other rows. If A is uniformly sparse, it is more appropriate to use a `static` schedule because the computation of each entry of the matrix-vector product Ax will roughly take the same amount of work. If A is not uniformly sparse, i.e., there are certain blocks that are significantly more dense than others, it is more appropriate to use a `dynamic` schedule because the threads that get assigned sparse rows can quickly finish their computations

and assist threads that are working on dense rows. We know that the CSR matrix representation makes no assumptions about the sparsity structure of A, and the Jacobi method assumes only that A is irreducibly or strictly diagonally dominant. Because we cannot make many assumptions about the sparsity structure of A, we would typically use the `dynamic` schedule to perform the `for` loop iterations, but we know that in all three test cases, the A matrix is very sparse and symmetric positive definite, so we use the default (`static`) loop schedule in the implementation.