Homework 2
High Performance Computing

Evan Toler

Mar. 16, 2020

1. The homework repository contains two simple programs that contain bugs. Use valgrind to find these bugs and fix them. Add a short comment to the code describing what was wrong and how you fixed the problem. Add the solutions to your repository using the naming convention `val_test01_solved.cpp`, `val_test02_solved.cpp`, and use the Makefile to compile the example problems.

Solution: Here is a summary of my bug fixes.

- `val_test01.cpp`: The code only allocated enough memory for n Fibonnaci numbers, not $n+1$. It also tried to free its memory with the `delete` keyword, which only applies to objects created with the `new` keyword.
- `val_test02.cpp`: The code tries to read uninitialized values and write them into other memory addresses. This is unsafe, so I changed the code to read only initialized values.

2. In this homework you will optimize the matrix-matrix multiplication code from the last homework using blocking. This increases the computational intensity (i.e., the ratio of flops per access to the slow memory) and thus speed up the implementation substantially. The code you can start with, along with further instructions are in the source file `MMult1.cpp`. Specifying what machine you run on, hand in timings for various matrix sizes obtained with the blocked version and the OpenMP version of the code.

Solution: I ran this code on a standard CIMS desktop. The processor is an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz. The compiler is `g++` version 4.8.5.

After experimenting with different loop orderings in serial programming, I found that the most efficient is to keep the order from `MMult0.cpp`. That is, the outermost loop increments $j = 0 : n-1$, the middle loop $p = 0 : k-1$, and the inner loop $i = 0 : m-1$. This is because the matrices are stored on the heap in column major order. If we want to access the matrix entries sequentially in the order they are stored, then the memory accesses `a[i+p*m]`, `b[p+j*k]`, and `c[i+j*m]` require that i increments before p, p increments before j, and that i increments before j. Thus the loop order we stated is the only one which accesses the memory most efficiently. Below, we compare runtimes, flop rate, and bandwidth with another candidate ordering, which we describe next.

Another candidate for efficient ordering comes from minimizing the number of reads and writes to memory. We achieve this by incrementing p in the inner loop, then i, then j in the outer loop so that all operations involving the pointer `c` can move outside of the innermost loop. We reduce the total number of memory accesses, but as we see in Table 1, performance decreases since we do not take advantage of the memory structure.

$m = n = k$	Loop Structure	Runtime (s)	Gflop/s	GB/s
256	j, p, i	0.003565	9.245712	111.093007
688	j, p, i	0.075147	8.751497	105.068851
1,024	j, p, i	0.316528	7.340057	88.109357
1,504	j, p, i	1.356762	5.029314	60.365144
1,984	j, p, i	3.230298	4.854035	58.258204
256	j, i, p	0.014053	1.517237	12.185309
688	j, i, p	0.298226	2.076229	16.633973
1,024	j, i, p	1.178608	1.299286	10.404436
1,504	j, i, p	6.002820	0.966565	7.737660
1,984	j, i, p	27.613783	0.515544	4.126432

Table 1: Runtime, flop rate, and bandwidth calculations for serial dense matrix multiplication without blocking. All computations use optimization flag `-O3` and the flag `-march=native`.

From here on, we assume that matrix multiplication uses the optimal loop ordering j, p, i. Computational efficiency drops off for n large, beyond about 1,000. We combat this by loading blocks of A, B, and C to the stack. The blocks are sized $K \times K$, with $K \ll n$. Since loading and writing to the blocks does not follow the memory structure of the original existing matrices, we order the code to minimize the number of memory accesses before performing the actual matrix multiplication.

After searching for the optimal block size (in multiples of 4) to maximize efficiency, we summarize some of our computations in Table 2.

$m = n = k$	Block size, K	Runtime (s)	Gflop/s	GB/s
252	12	0.004425	7.233105	99.311679
684	12	0.090432	7.077478	96.891084
1,020	12	0.322679	6.577491	89.995559
1,500	12	1.097119	6.152478	84.149489
1,980	12	2.531762	6.132007	83.853641
240	120	0.002282	12.115524	148.213250
720	120	0.062545	11.935250	145.477435
1,080	120	0.211550	11.909328	145.073262
1,560	120	0.643619	11.797092	143.652276
1,920	120	1.207189	11.726231	142.766863
320	160	0.005594	11.715453	142.635645
640	160	0.045526	11.516149	139.921211
1,120	160	0.243995	11.516057	139.796707
1,600	160	0.711008	11.521672	139.815496
1,920	160	1.235152	11.460761	139.057231

Table 2: Runtime, flop rate, and bandwidth calculations for serial dense matrix multiplication using blocking. All computations use optimization flag `-O3` and the flag `-march=native`.

The optimal block size appears to be around $K = 120$. When the block size is too small, about $K < 40$, there is too much overhead cost of forming the blocks and accessing memory inefficiently, and the efficiency drops for $n < 1000$.

We can further improve our performance by parallelizing the matrix-matrix multiplication. We perform the parallel version on matrices of size $m = n = k = 1920$ with block size $K = 120$ and compare the runtimes and speedup in Figure 1. We can achieve about 55% of the theoretical maximum speedup. However, we are limited by the blocking procedure, which is inefficient to parallelize, and the overhead of managing multiple threads at once.

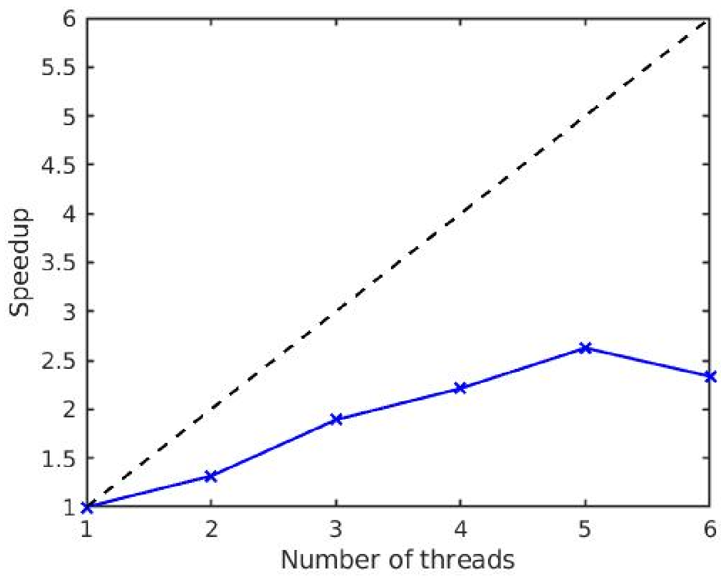

Figure 1: Efficiency scaling for our implementation of matrix multiplication

3. The homework repository contains five OpenMP problems that contain bugs. These files are in C, but they can be compiled with the C++ compiler. Try to find these bugs and fix them. Add a short comment to the code describing what was wrong and how you fixed the problem. Add the solutions to your repository using the naming convention `omp_solved{2,...}.c`, and provide a Makefile to compile the fixed example problems.

Solution: Here is a summary of my bug fixes.

- `omp_bug2.c`: I added private variables to the parallel region to prevent race conditions. I also restructured the summation operation to prevent a race condition on the variable `total`. I could also have resolved this by adding a reduction statement to the parallel for loop. Finally, the computed sum is large and has many terms, so I changed `total` from a `float` to a `double` for more precision.

- `omp_bug3.c`: Execution reaches a deadlock since there is a `barrier` directive inside of a function that only two threads access. Two possible solutions are to demand only two threads at the beginning of the parallel region, or to remove the barrier. I chose the latter, since the `barrier` is unnecessary in the first place.

- `omp_bug4.c`: With $N = 1048$, an $N \times N$ array of `double`s is too large to store on each thread's stack and causes a segmentation fault. Possible solutions are to store the array on the heap or to increase the thread stack size. I chose the former, since this is a more robust solution. If the code is shared with collaborators, they may not be able to support large stack sizes.

- `omp_bug5.c`: It is not clear what the intended purpose of this code is. If the code is meant to compute one quantity through two different sums and check that the results are equal, then it is impossible to accomplish this without radically changing the code structure. On the other hand, the main issue at execution is deadlock because of the ordering of locks on critical statements. I changed the lock order so that one thread "unlocks B" before the other thread tries to lock it.

- `omp_bug6.c`: The code does not compile because the reduction variable in the parallel for loop is private to each thread, when it should be shared. I remedy this by creating a shared pointer to a shared variable that will store the total sum. Each thread then computes its own dot product and adds the result to the shared sum.

4. Implement first a serial and then an OpenMP version of the two-dimensional Jacobi and Gauss-Seidel smoothers. This is similar to the problem on the first homework assignment, but for the unit square domain $\Omega = (0,1) \times (0,1)$.

   Solve the linear system that arises from a five-point Laplacian stencil applied to the Poisson equation

$$\begin{cases} -\Delta u = 1 & \text{on } \Omega \\ u = 0 & \text{on } \partial\Omega. \end{cases}$$

   Report timings for different values of N and different numbers of threads. These timings should be for a fixed number of iterations, since convergence is slow, and slows down even further as N becomes larger.

Solution: I ran this code on a standard CIMS desktop. The processor is an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz. The compiler is `g++` version 4.8.5. Various run times for my implementations are in Table 3. We use a relative tolerance of 10^{-6} so that the listed number of iterations does not satisfy the stopping criterion on the residual $\|Au_k - f\|$.

N	Iterations	Method	Number of threads	Runtime (s)
51	3,000	Jacobi	1 (serial)	0.047459
51	3,000	Jacobi	4	0.061508
51	3,000	Jacobi	6	0.085524
51	3,000	Gauss-Seidel	1 (serial)	0.017643
51	3,000	Gauss-Seidel	4	0.027840
51	3,000	Gauss-Seidel	6	0.030193
101	10,000	Jacobi	1 (serial)	0.684290
101	10,000	Jacobi	4	0.406560
101	10,000	Jacobi	6	0.409229
101	10,000	Gauss-Seidel	1 (serial)	0.229584
101	10,000	Gauss-Seidel	4	0.138325
101	10,000	Gauss-Seidel	6	0.131824
1,001	10,000	Jacobi	1 (serial)	73.438338
1,001	10,000	Jacobi	4	31.234586
1,001	10,000	Jacobi	6	34.601011
1,001	10,000	Gauss-Seidel	1 (serial)	30.342224
1,001	10,000	Gauss-Seidel	4	10.755080
1,001	10,000	Gauss-Seidel	6	11.015953

Table 3: Runtime comparisons for the Jacobi and Gauss-Seidel methods. Timings are averages over 100 identical instances of the problem.

One initially surprising observation is that the Gauss-Seidel method is always faster than the Jacobi method for a fixed problem size and thread count, even though the update is more complicated for Gauss-Seidel. However, this is less surprising when we consider the memory structure for both methods. The Gauss-Seidel method only requires *one* copy of the approximate solution `u` (stored in column major order, `u[j]` for $0 \le j \le N^2 - 1$), and about half of the memory reads to update `u[j]` come from a spatially local place in memory (`u[j-1]` and `u[j+1]`). On the other hand, Jacobi requires another copy of the solution from the previous iteration, `u_old`. Updates always reference `u_old`, but `u_old` is stored in a completely different memory location from `u`! So, the speedup from spatially local memory accesses is destroyed.

As we expect, for low problem instances ($N < 70$), the overhead from parallelization causes a slowdown in runtime. Moreover, the overhead of creating and managing six threads also causes a slight slowdown compared to the optimal performance of four threads.