High Performance Computing – Problem Set 2 (Due Mar. 14) solutions by Frederick Law

1. **Finding Memory bugs**. 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: The bugs were both fixed. Identification of the problems, description, and solution are found in the comments of the code.

2. **Optimizing matrix-matrix multiplication**. 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 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: As mentioned in the comments for the code, we first analyzed the loop order for the matrix-matrix multiplication (mat-mat). Namely, to compute

$$C_{ij} = C_{ij} + \sum_p A_{ip} B_{pj}$$

three loops must be done over i, j, p respectively. For the outer, middle, and inner loops, this gives a total of 6 possible combinations. With the notation (outer, middle, inner), we timed these loop orders for the unblocked mat-mat for various large N, over a total of 10 repeats. The timings were performed on a Intel(R) Core(TM) I7-7700 @ 3.60GHz processor, and the results are in Fig. 1.

	$N = 600$	$N = 700$	$N = 800$	$N = 900$	$N = 1000$
(i, j, p)	2.38	3.81	5.83	8.34	16.42
(i, p, j)	2.70	4.20	24.48	51.26	73.45
(j, i, p)	2.36	3.77	6.15	8.99	12.70
(j, p, i)	0.98	1.56	2.35	3.48	5.18
(p, i, j)	2.83	5.89	12.36	22.46	57.45
(p, j, i)	1.05	1.67	2.59	3.90	5.93

Figure 1: Timings for unblocked mat-mat using different loop orders. All timings are reported in seconds and are for NREPEATS=10, with GNU compiler flags `-march=native` and `-O2`.

Looking Fig. 1 we see that the loop order (j, i, p) is the fastest while (i, p, j) is the slowest. The order (p, j, i) is also relatively fast compared to the others. Moreover, the differences in timing is extreme, with the fastest (j, p, i) being nearly 25x faster than the slowest (i, p, j). One reason behind this is likely that the matrices are stored in column major order. So in the case where C is $m \times n$, A is $m \times k$, and B is $k \times n$, then we get

$$C_{ij} = C[i + j * m], \quad A_{ip} = A[i + p * m], \quad B_{pj} = B[p + j * k]$$

As a result, to get the most out of spatial locality of memory, we want to access i before j for C, then i before p for A, and then p before j for B. This leads to i as the inner loop, p as the middle, and j as the outer, hence (j, p, i) being the fastest loop ordering. Note that this loop ordering has *more* memory accesses $(nk + 3mnk)$ than the intuitive (i, j, p) which has $2mn + 2mnk$. We keep note of this faster loop ordering with respect to column major ordering and use this for the blocked mat-mat as well (in the reads, block multiplications, and writes). We note that we also use this ordering to our advantage in the later implementation of Jacobi iterations and Gauss-Seidel with red black coloring.

Next we implemented our blocked mat-mat. The loop ordering for the blocks does not play a major factor (since regardless with a reasonable block size we must jump across memory for different blocks), but within each loop when we perform a read, block-multiplication, or write, we use the optimal ordering as from above. Namely, we assume that m, n, k are all multiples of a fixed block size b. Then we do $C_{IJ} = C_{IJ} + \sum_P A_{IP} B_{PJ}$ for all blocks I, J, P.

In the code, doing $C_{IJ} = C_{IJ} + \sum_P A_{IP} B_{PJ}$ amounts to loading C_{IJ} from the full C onto the stack, then looping over P where at each P we load A_{IP} from A and B_{PJ} from B onto the stack. At each point in the loop we do the block matrix multiplication, and then add to C_{IJ}. Lastly, at the end we write from C_{IJ} back to the full C. Note that this significantly reduce the number of memory operations. Now we have $2\left(\frac{mn}{b^2}\right) + 2\left(\frac{mnk}{b^3}\right)$ block memory operations, but each block memory operations is b^2. So the total number of memory operations is $2mn + 2\left(\frac{mnk}{b}\right)$.

For a variety of block-sizes we test our blocked implementation, as well as an OpenMP version of the unblocked mat-mat. In this implementation we use 4 threads, putting just an `#pragma omp for` on the outermost loop. Note that at the start of the function call we put that the pointers to the global matrices are shared. Despite this, there is inherently no race condition. This is because we use the (j, p, i) loop ordering. As a result, the only write onto $C[i + j * m]$ never has a conflict, since the outer loop is j and thus the 4 threads never share j so they never access the same location in memory. The following results were obtained on a Intel(R) Core(TM) I5-5200U CPU @ 2.20GHz processor. (Apologies for switching the timing from testing the loop orderings, but had to switch computers due to NYU health policies.). Each timing represents 10 runs. We also plot the flop rates for the parallel unblocked and serial unblocked mat-mat (again over 10 runs).

	$b = 16$	$b = 32$	$b = 64$	$b = 128$	OpenMP unblocked
$N = 128$	0.0217	0.0155	0.01774	0.0190	0.0147
$N = 256$	0.1282	0.1371	0.1185	0.1326	0.0773
$N = 384$	0.4599	0.3947	0.4357	0.4643	0.2328
$N = 512$	1.1208	0.9951	1.0244	1.1717	0.4876
$N = 640$	2.2889	1.8786	2.0853	2.1339	1.2141
$N = 768$	3.7402	3.3055	3.5605	3.6983	2.1788
$N = 896$	5.8652	5.1784	5.5754	5.8809	3.3179
$N = 1024$	10.1334	8.3446	9.0101	8.7646	5.0950

Figure 2: Timings for different N under different block sizes, as well as parallel unblocked. All timings are reported in seconds and are for NREPEATS=10, with GNU compiler flags `-march=native` and `-O2`.

	$N = 128$	$N = 256$	$N = 512$	$N = 1024$
serial unblocked	2.2902	2.4634	2.5377	1.9804
OpenMP unblocked	2.8415	4.3368	5.5052	4.2148
% of optimal rate	31.01%	44.01%	54.23%	53.20%

Figure 3: Flop rates for different N comparing the serial and OpenMP versions of the optimal loop order unblocked mat-mat. All measurements are in Gflop/s and are for NREPEATS=10, with GNU compiler flags `-march=native` and `-O2`.

Looking in Fig. 2 we see that the blocking has a minor effect, but not major. The only real distinction arises when the block size is around $b = 32$, but even then the savings are small. The best guess is that if we continue to increase the block size, we are essentially choosing blocks that are too large to fit into the cache and hence not gaining. We see that the OpenMP unblocked code with 4 threads is significantly faster than all the blocked versions.

Looping at the flop rates in Fig. 3 we see that the OpenMP implementation with 4 threads achieves near half the optimal flop rate, but this is more visible at large matrices ($N = 512, 1024$) which the computational effort per loop iteration is large (since we only parallelize the outer loop). □

3. **Finding OpenMP bugs**. 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`$\{2, \dots\}$`.cpp`, and provide a Makefile to compile the fixed example problems.

   Solution: The bugs were both fixed. Identification of the problems, description, and solution are found in the comments of the code.

4. **OpenMP version of 2D Jacobi/Gauss-Seidel smoothing.** 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 $\Omega = (0,1) \times (0,1)$. For a given function $f : \Omega \to \mathbb{R}$, we aim to find $u : \Omega \to \mathbb{R}$ such that

$$-\Delta u := -(u_{xx} + u_{yy}) = f \text{ in } \Omega \tag{1}$$

and $u(x,y) = 0$ for all boundary points $(x,y) \in \partial\Omega := \{(x,y) : x = 0 \text{ or } y = 0 \text{ or } x = 1 \text{ or } y = 1\}$. We go through analogous arguments as in homework 1, where we used finite differences to discretize the one-dimensional version of (1). In two dimensions, we choose the uniformly spaced points $\{(x_i, y_j) = (ih, jh) : i,j = 0,1,\dots,N+1\} \subset [0,1] \times [0,1]$, with $h = 1/(N+1)$, and approximate $u(x_i, y_j) \approx u_{i,j}$ and $f(x_i, y_j) \approx f_{ij}$ for $i,j = 0,\dots,N+1$. Using Taylor expansions as in the one-dimensional case results in the standard 2D Laplacian stencil:

$$-\Delta u_{ij} \approx \frac{-u_{i-1,j} - u_{i,j-1} + 4u_{ij} - u_{i+1,j} - u_{i,j+1}}{h^2}$$

which yields a linear system $A\mathbf{u} = \mathbf{f}$, where

$$\mathbf{u} = (u_{1,1}, u_{1,2}, \dots, u_{1,N}, u_{2,1}, u_{2,2}, \dots, u_{N,N-1}, u_{N,N})^T$$
$$\mathbf{f} = (f_{1,1}, f_{1,2}, \dots, f_{1,N}, f_{2,1}, f_{2,2}, \dots, f_{N,N-1}, f_{N,N})^T$$

Note that the points at the boundaries are not included, as we know that their values to be zero. Similar to the one-dimensional case, the resulting Jacobi update for solving this linear system is

$$u_{ij}^{k+1} = \frac{1}{4}\left(h^2 f_{ij} + u_{i-1,j}^k + u_{i,j-1}^k + u_{i+1,j}^k + u_{i,j+1}^k\right)$$

and the Gauss-Seidel update is given by

$$u_{ij}^{k+1} = \frac{1}{4}\left(h^2 f_{ij} + u_{i-1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i+1,j}^k + u_{i,j+1}^k\right)$$

where it depends on the order of the unknowns which entries on the right hand side are based on the kth and which on the $(k+1)$st iteration. The above update formula is for lexicographic ordering of the points.

As can be seen, the update at the (i,j)th point in the Gauss-Seidel smoother depends on previously updated points. This dependence makes it difficult to parallelize the Gauss-Seidel algorithm. As a remedy, we consider a variant of Gauss-Seidel, which uses *red-black coloring* of the unknowns. This amounts to coloring the unknowns like a chessboard, and splitting each Gauss-Seidel iteration into two sweeps: first, one updates all the black and then all the red points. The point updates in the red and black sweeps are independent from each other and can be parallelized using OpenMP.

- Write OpenMP implementations of the Jacobi and Gauss-Seidel methods with red-black coloring, and call them `jacobi2D-omp.cpp` and `gs2D-omp.cpp`. Make sure your OpenMP codes also compile with OpenMP compilers using preprocessor commands (`#ifdef_OPENMP`) as shown in class.
- Choose the right hand side $f(x, y) \equiv 1$, and report timings for different values of N and different numbers of threads, specifying the machine you run on. These timings should be for a fixed number of iterations as, similar to the 1D case, the convergence is slow, and slows down even further as N becomes larger.

Solution: We created OpenMP implementations of the Jacobi and Gauss-Seidel(GS) methods with red-black coloring, named under `jacobi2D-omp.cpp` and `gs2D-omp.cpp`. The codes can be run in serial by omitting the `-fopenmp` compiler flag. Both codes have a similar structure: given N (either passed in the command line under `-N`, or by default $N = 100$) we store our solution as u_{ij} with $i, j = 0, 1 \dots, N, N+1$. The reason we manually keep the boundary zeros in our solution is that it makes the loop structure significantly easier which lends itself to better parallelism. Had we omitted storing these extra zeros on the boundary, we would have to run separate loops over $u_{1,\cdot}$, $u_{N,\cdot}$, $u_{\cdot,1}$ and $u_{\cdot,N}$ which is detrimental to parallelism.

In both Jacobi and GS, we store two copies of the solution, `u_curr` and `u_prev`. At each step, we use the values of `u_prev` to update `u_curr` (so as to avoid any read/write hazards) and then deep copy `u_curr` into `u_prev`. The Jacobi iteration amounts to a simple double loop on $i, j = 1, \dots, N$. The GS iteration has two sweeps, first we go through all $i, j = 1, \dots, N$ and check if $i + j$ is even - these are all the red points. Only those points where $i + j$ is even are updated. We then deep copy those red pieces back to `u_prev` again only updating where $i + j$ is even. Then we update the black points, running the same double loop but now only updating when $i + j$ is odd. Again, after this we deep copy the black points to `u_prev` which ends that iteration step.

While we could have manually selected out the red and black points to avoid running a double loop (since in the red update around half of the points will have $i + j$ odd so the loop skips this), the fact that the loop condition is exceedingly simple aids us in parallelism. In fact upon first implementation we tried to selectively choose the indices for all the red and black points, but this approach was significantly slower than just running the above mentioned double loop. Additionally, since we store both `u_curr` and `u_prev` in column major order, we order the double loop so for u_{ij}, then j is the outer loop and i is the inner loop. At the end of each iteration we compute the residual which requires a double loop $i, j = 1, \dots, N$. These loops are also ordered j outer and i inner.

We parallelized using OpenMp using just a simple `#pragma omp for` for all the double loops. For computing the residual, we also need a reduction in order to safely sum all the residual terms together. Initially we believed that a `collapse(2)` collapsed loop would provide speed up, but even for large $N > 1000$ it actually proved to slow down the code. One reason this could be is that the workload inside each term in the double loop is relatively small. Hence parallelizing the outer loop only (N loop iterations but large work each) gives better speed than collapsing (N^2 loop iterations but small work each).

Then we solved the Poisson equation with $f(x, y) \equiv 1$. Before timing for larger N and a fixed number of iterations, we tested both scripts for N small (order of $N \sim 50$). We saw that for small N, both methods were able to reduce the initial residual by at least 6 orders of magnitude in a reasonable number of iterations (on the order of hundreds). This served as a confirmation that both codes converged for small N and were hence implemented correctly.

Next we timed for various N in serial and with 2 and 4 threads in OpenMP. The following results were obtained on a Intel(R) Core(TM) I7-7700 @ 3.60GHz processor.

Jacobi	$N = 128$	$N = 256$	$N = 512$	$N = 1024$	$N = 2048$
serial	0.2460	0.9851	4.2559	20.3186	81.6945
2 threads	0.2799	0.9782	3.8126	18.4662	68.9661
4 threads	0.2845	0.9118	3.4424	16.1743	62.3899
Gauss-Seidel	$N = 128$	$N = 256$	$N = 512$	$N = 1024$	$N = 2048$
serial	0.4304	1.6860	6.6864	29.7844	127.3747
2 threads	0.2879	1.0516	4.0844	19.4459	90.4453
4 threads	0.1590	0.5419	2.0594	14.8245	77.4741

Figure 4: Timings for Jacobi and Gauss-Seidel with black red labeling. All timings are averaged over 10 runs, using the GNU compiler flag `-O2`.

Looking at Fig. 4 we see that the GS method is slower than Jacobi in serial, as is to be expected. However, once parallelized we see that the GS method begins to beat Jacobi until large problem size $N = 2048$. Moreover, we see that moving from serial to 2 threads in GS has a noticeable impact, as does moving from 2 threads to 4 threads. However, serial to 2 threads and then to 4 threads has a minor impact on Jacobi. This suggests that with red-black ordering, GS has much more potential for parallelization than Jacobi. Lastly, we note that in our implementation we stored two copies for GS. Since we only use black points to update red and then red to update black, it would be possible to implement GS with only one copy of the array. So long as we are careful to implement the code, we can achieve a significant speed up as it would nearly double the spatial locality of the memory, since now we are accessing within the same array instead of jumping to the secondary copy of the array which is elsewhere in memory. $\square$