CSCI-GA.2945-002, High Performance Computing February 14, 2022

Solutions to Problem 1 of Homework 1

Name: Anav Prasad (ap7152) *Due: 1:25 PM on Monday, February 14*

Collaborators:

Describe a parallel application and the algorithms used. Find and examine an application problem for which high-performance computing has been used. Pick a problem from your own research, or find a problem elsewhere. Prepare a 1–2 page description of the problem and describe where and how successful high-performance computing has been/is used. Consider to include the following:

(a) What's the application problem being solved?

(b) Why does the problem require large/fast computation?

(c) What are the underlying algorithms?

(d) If the application uses a supercomputer, where is that computer on the Top500 list (http://www.top500.org/)? Say a few words about the kind of architecture.

(e) How well does the algorithm perform? Does it "scale"?

If you are looking for an application, take a look at the papers from one of the previous Supercomputing conferences [1]. Alternatively, take a look at the National Science Foundation (NSF)-funded supercomputing centers. These centers provide computing resources for open research under the Extreme Science and Engineering Discovery Environment (XSEDE)[2] and the website usu- ally has science stories with links to papers. Here are some direct links to US-based computing centers. [3] [4] [5] [6] [7]

Solution:
Application Problem: Sparse Matrix Vector Product.
The high performance algorithm for the improvement of sparse matrix-vector product (SpMV) that is going to be reviewed here comes from the Merge-based Parallel Sparse Matrix-Vector Multiplication[8] paper published by Duane Merril[9] and Michael Garland[10] in 2016 which presents a strictly balanced method for the parallel computation of SpMV.

[1] http://supercomputing.org/history.php and choose "Conference Proceedings".
[2] XSEDE: https://www.xsede.org/
[3] Oark Ridge National Laboratory: https://www.olcf.ornl.gov/
[4] National Energy Research Scientific Computing Center (NERSC): https://www.nersc.gov/
[5] San Diego Supercomputing Center: http://www.sdsc.edu/
[6] Nasa Advanced Supercomputing Division: http://www.nas.nasa.gov/
[7] Texas Advanced Computing Center (TACC): https://www.tacc.utexas.edu/
[8] https://dl.acm.org/doi/10.5555/3014904.3014982
[9] https://developer.nvidia.com/blog/author/dumerrill/
[10] https://mgarland.org/

Large/Fast Computation Requirement:
In its simplest form, a matrix vector product operation computes $y = Ax$. In cases of sparse matrix-vector product (SpMV), the matrix A is sparse and vectors x and y are dense. In the most general case, SpMV has time complexity $O(mn)$, assuming the dimensions of the matrix is $m \times n$ and the dimension of the vector is n. But, in the cases of large sparse matrixes, there is a lot of room of improvement in the time taken to compute the matrix vector product. And, since sparse matrix vector products are important within many application domains (including computational science, graph analytics, and machine learning), high performance algorithms for sparse linear algebra is a very worthy research avenue.

Brief Algorithm Description:
In general[11] [12], high performance algorithms for SpMV use Compressed Sparse Row (CSR) format[13] for in-memory representation of the matrixes. However, contemporary CsrMV strategies that attempt to parallelize SpMV independently often suffer from performance reduction that arises from highly varying row lengths and/or wide aspect ratios. Despite various heuristics for constraining imbalance, they often underperform for small-world or scale-free datasets having a minority of rows that are many orders of magnitude longer than average. Merril's and Garland's Merge-based Parallel SpMV (MSpMV) removes that problem by equitably splitting the aggregate work to be performed. This is done so by the adaptation of a few parallelization strategies [14] [15] [16] that were originally designed for efficient merging of sorted sequences. The crucial idea in MSpMV is the use of the adapted parallelization strategy by framing it around the non-zero row lengths in CSR representation of the sparse matrix. Therefore, the individual processing elements end up assigned with equal-sized shares of the rows in the sparse matrix and, thus neatly dividing the processing load equitably.

Does the application use a supercomputer?
While the MSpMV algorithm was tested on a supercomputer, improvements in SpMV would automatically result in improvements in sparse linear solvers, eigenvalue systems, Krylov subspace methods, large-scale combinatorial graph algorithms [17] [18], and many more besides.

[11] J. L. Greathouse and M. Daga, "Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Piscataway, NJ, USA, 2014, pp. 769–780.

[12] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, New York, NY, USA, 2009, pp. 18:1–18:11.

[13] https://people.eecs.berkeley.edu/ aydin/csb2009.pdf

[14] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, "Merge Path - Parallel Merging Made Simple," in Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Washington, DC, USA, 2012, pp. 1611– 1618.

[15] S. Baxter and D. Merrill, "Efficient Merge, Search, and Set Operations on GPUs," Mar-2013. [Online]. Available: http://on-demand.gputechconf.com/gtc/2013/presentations/S3414-Efficient- Merge-Search-Set-Operations.pdf.

[16] N. Deo, A. Jain, and M. Medidi, "An optimal parallel algorithm for merging using multiselection," Inf. Process. Lett., vol. 50, no. 2, pp. 81– 87, Apr. 1994.

[17] J. Kepner and J. Gilbert, Graph Algorithms in the Language of Linear Algebra. Society for Industrial and Applied Mathematics, 2011.

[18] M. Mohri, "Semiring Frameworks and Algorithms for Shortest-distance Problems," J. Autom. Lang. Comb., vol. 7, no. 3, pp. 321–350, Jan. 2002.

Performance and Scaling:
In its testing, MSpMV demonstrated 1.6x and 1.1x respective average speedup for medium and large-sized datasets, and upto 198x for highly irregular datasets as compared to MKL [19] and cuSPARSE[20]. Apart from that, MSpMV showed performance that was substantially uncorrelated to irregular row-lengths and highly correlated to problem size and, thus, easily scaled, w.r.t matrix sizes, while retaining its performance boost. Also, MSpMV performed favorably against algorithms that leveraged specialized matrix representation formats [21] and expensive per-processor and per-matrix auto tuning [22].

Conclusion:
MSpMV exists as a somewhat distinct example that specifically demonstrates the performance improvements that can be made via *parallelization* which is one of the core tenets of high performance computing and, as such, is an excellent exemplar to the usefulness of the same. □

[19] Intel Math Kernel Library (MKL) v11.3. Intel Corporation, 2015.
[20] NVIDIA cuSPARSE v7.5. NVIDIA Corporation, 2013.
[21] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks," in Proc. SPAA, Calgary, Canada, 2009.
[22] A. Jain, "pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures," Master's Thesis, University of California at Berkeley, 2008.

CSCI-GA.2945-002, High Performance Computing February 14, 2022

Solutions to Problem 2 of Homework 1

Name: Anav Prasad (ap7152) *Due: 1:25 PM on Monday, February 14*

Collaborators:

We will experiment with a simple implementation of a matrix- matrix multiplication, which you can download from the homework1 directory in https:// github.com/pehersto/HPCSpring2022/. We will improve and extend this implementation throughout the semester. For now, let us just assess the performance of this basic function. Report the processor you use for your timings. For code compiled with different optimization flags (`-O0` and `-O3`) and for various (large) matrix sizes, report

- the flop rate,
- and the rate of memory access.

Solution:
Processor: Apple M1 Chip
Max. CPU Clock Rate: 3.2GHz
The results obtained with different optimization flags are as follows:

- with `-O0` flag:

Dimension	Time	Gflop/s	GB/s
20	0.005581	0.143348	2.293564
40	0.035912	0.178214	2.851424
60	0.116200	0.185887	2.974186
80	0.276018	0.185495	2.967924
100	0.538255	0.185786	2.972571
120	0.916442	0.188555	3.016886
140	1.474805	0.186058	2.976936
160	2.225827	0.184021	2.944344
180	3.220300	0.181101	2.897618
200	4.315659	0.185371	2.965944
220	5.840392	0.182317	2.917064
240	7.493779	0.184473	2.951568
260	9.750617	0.180255	2.884084
280	11.990474	0.183079	2.929259

Dimension	**Time**	**Gflop/s**	**GB/s**
300	14.607748	0.184833	2.957335
320	18.025082	0.181791	2.908658
340	22.295347	0.176288	2.820606
360	25.818899	0.180705	2.891277
380	29.995136	0.182936	2.926981
400	35.523323	0.180163	2.882613
420	41.474336	0.178636	2.858172
440	47.383472	0.179776	2.876412
460	54.700724	0.177943	2.847085
480	61.468360	0.179917	2.878671
500	70.284324	0.177849	2.845585
520	79.372736	0.177149	2.834384
540	88.702093	0.177520	2.840321
560	96.477281	0.182028	2.912454
580	109.296740	0.178516	2.856254

- with `-O3` flag:

Dimension	**Time**	**Gflop/s**	**GB/s**
20	0.000729	1.097206	17.555289
40	0.004469	1.431981	22.911691
60	0.013807	1.564476	25.031614
80	0.027820	1.840400	29.446397
100	0.047336	2.112551	33.800823
120	0.082953	2.083104	33.329667
140	0.132262	2.074671	33.194730
160	0.204266	2.005233	32.083728
180	0.297475	1.960501	31.368023
200	0.409611	1.953072	31.249154
220	0.560208	1.900722	30.411548
240	0.756213	1.828057	29.248912
260	0.993283	1.769485	28.311761
280	1.264259	1.736353	27.781645

Dimension	**Time**	**Gflop/s**	**GB/s**
300	1.597971	1.689642	27.034275
320	2.024015	1.618960	25.903360
340	2.403126	1.635536	26.168577
360	2.894210	1.612046	25.792734
380	3.461319	1.585292	25.364668
400	4.090497	1.564602	25.033631
420	4.902313	1.511287	24.180588
440	5.707345	1.492533	23.880526
460	6.496060	1.498385	23.974163
480	7.302939	1.514349	24.229589
500	8.433003	1.482272	23.716345
520	9.494851	1.480887	23.694190
540	11.213300	1.404261	22.468177
560	12.036582	1.459019	23.344301
580	13.480641	1.447350	23.157593

We see that with `-O3` optimization, we achieve a 7-11x speedup. □

CSCI-GA.2945-002, High Performance Computing February 14, 2022

Solutions to Problem 3 of Homework 1

Name: Anav Prasad (ap7152) *Due: 1:25 PM on Monday, February 14*

Collaborators:

Write a program to solve the Laplace equation in one space dimension. For a given function $f : [0,1] \to \mathbb{R}$, we attempt to solve the linear differential equation

$$-u'' = f \text{ in } (0,1), \text{ and } u(0) = 0, u(1) = 0 \tag{1}$$

for a function u. In one space dimension [23] , this so-called boundary value problem can be solved analytically by integrating f twice. In higher dimensions, the analogous problem usually cannot be solved analytically and one must rely on numerical approximations for u. We use a finite number of grid points in $[0,1]$ and finite-difference approximations for the second derivative to approximate the solution to (1). We choose the uniformly spaced points $\{x_i = ih : i = 0, 1, ..., N, N+1\} \subset [0,1]$, with $h = 1/(N+1)$, and approximate $u(x_i) \approx u_i$ and $f(x_i) \approx f_i$, for $i = 0, ..., N+1$. Using Taylor expansions of $u(x_i - h)$ and $u(x_i + h)$ about $u(x_i)$ results in

$$-u''(x_i) = \frac{-u(x_i - h) + 2u(x_i) - u(x_i + h)}{h^2} + \text{h.o.t.},$$

where h.o.t. stands for a remainder term that is of higher order in h, i.e., becomes small as h becomes small. We now approximate the second derivative at the point x_i as follows:

$$-u''(x_i) \approx \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2}.$$

This results in the following finite-dimensional approximation of (1):

$$A\boldsymbol{u} = \boldsymbol{f}, \tag{2}$$

where

$$A = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}, \boldsymbol{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix}, \boldsymbol{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N \end{bmatrix}.$$

[23]The generalization of (1) to two and three-dimensional domains Ω instead of the one-dimensional interval $\Omega = [0,1]$ is the Laplace equation,

$$\begin{aligned} -\Delta u &= f \text{ on } \Omega, \\ u &= 0 \text{ on } \partial\Omega, \end{aligned}$$

which is one of the most important partial differential equations in mathematical physics.

Simple methods to solve (2) are the Jacobi and the Gauss-Seidel method, which start from an initial vector $u^0 \in \mathbb{R}^N$ and compute approximate solution vectors u^k, $k = 1, 2, \ldots$. The component-wise formula for the Jacobi method is

$$u_i^{k+1} = \frac{1}{a_{ii}} \left(f_i - \sum_{j \neq i} a_{ij} u_j^k \right),$$

where a_{ij} are teh entries of the matrix A. The Gauss-Seidel algorithm is given by

$$u_i^{k+1} = \frac{1}{a_{ii}} \left(f_i - \sum_{j<i} a_{ij} u_j^{k+1} - \sum_{j>i} a_{ij} u_j^k \right).$$

If you are unfamiliar with these methods, please take a look at the Section 11.2 in Matrix Computations by Golub and Van Loan

(a) Write a program in C that uses the Jacobi or the Gauss-Seidel method to solve (2), where the number of discretization points N is an input parameter, and $f(x) \equiv 1$, i.e., the right hand side vector f is a vector of all ones.

(b) After each iteration, output the norm of the residual $\|A\boldsymbol{u}^k - \boldsymbol{f}\|$ on a new line, and terminate the iteration when the initial residual is decreased by a factor of 10^6 or after 5000 iterations. Start the iteration with a zero initialization vector, i.e., $\boldsymbol{u}^0$ is the zero vector.

(c) Compare the number of iterations needed for the two different methods for different numbers $N = 100$ and $N = 10,000$. Compare the run times for $N = 10,000$ for 100 iterations using different compiler optimization flags (`-O0` and `-O3`). Report the results and a listing of your program. Specify which computer architecture you used for your runs. Make sure you free all the allocated memory before you exit.

Solution:
Architecture:
Processor: Apple M1 Chip
Max. CPU Clock Rate: 3.2GHz
Cache:
L1 Cache:
192+128 KB per core (performance cores)
128+64 KB per core (efficient cores)
L2 cache:
12 MB (performance cores)
4 MB (efficient cores)
Cores:
8 (4× high-performance + 4× high-efficiency)

Tabulated Results:

Optimization Flag	Method	N	Time(s)	$N_{\text{iterations}}$
`-O0`	Jacobi	100	0.028058	5000
`-O3`	Jacobi	100	0.004981	5000
`-O0`	Jacobi	10000	2.449090	5000
`-O3`	Jacobi	10000	0.553198	5000
`-O0`	Gauss-Seidel	100	0.028680	5000
`-O3`	Gauss-Seidel	100	0.010072	5000
`-O0`	Gauss-Seidel	10000	2.477076	5000
`-O3`	Gauss-Seidel	10000	1.026958	5000

Number of iterations needed for convergence for $N = 100$ and $N = 10,000$ was the same i.e. 5000 iterations (the upper limit).
Comparative run times for $N = 10,000$ for 100 iterations using different compiler optimization flags are as follows:

Optimization Flag	Method	N	Time(s)
`-O0`	Jacobi	10000	0.056494
`-O3`	Jacobi	10000	0.013826
`-O0`	Gauss-Seidel	10000	0.052786
`-O3`	Gauss-Seidel	10000	0.023386

From the output obtained from running of the code, it can be seen that increasing n leads to slower convergence in terms of the decrease in the residual for the same number of iterations.
Also, trivially, increasing the value of n makes the program run overall slower (in the same number of iterations).
The `-O3` flag makes the overall program $\approx$ 2-3x faster.
My code listing is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#include <time.h>
#define ITER_LIMIT 100
#define MAX_RES_DECREASE 1000000.0
typedef enum method{JACOBI, GAUSS_SEIDEL} method;

int getNum(char ch){ return ((int)(ch - '0')); }

int readInt(char ar[]){
    int n = 0, i = 0;
    while(ar[i] != '\0') n = n*10 + getNum(ar[i++]);
    return n;
}

int getIndex(int i, int j, int n){ return (i * n + j); }
```

```
void init(double *A, double *f, double *u, int n){
    for (int i = 0; i < n; ++i){
        f[i] = 1;
        u[i] = 0;
        for (int j = 0; j < n; ++j){
            if (i == j) A[getIndex(i, j, n)] = 2;
            else if (abs(i - j) == 1) A[getIndex(i, j, n)] = -1;
            else A[getIndex(i, j, n)] = 0;
        }
    }
}

void deepCopyVector(double *src, double *dest, int n){
    for (int i = 0; i < n; ++i)
            dest[i] = src[i];
}

double computeResidual(double *A, double *f, double *u, int n){
    double norm = 0, temp;
    for (int i = 0; i < n; ++i){
        temp = A[getIndex(i, i, n)] * u[i];
        if (i > 0) temp += A[getIndex(i, i - 1, n)] * u[i-1];
        if (i < n - 1) temp += A[getIndex(i, i + 1, n)] * u[i+1];
        temp -= f[i];
        norm += temp * temp;
    }
    return sqrt(norm);
}

bool checkIfDone(double initialResidual, double currentResidual, int iterCount){
    if (iterCount >= ITER_LIMIT) return true;
    return (currentResidual <= initialResidual/((double)MAX_RES_DECREASE));
}

void update(double *A, double *f, double *u, int n, method updateMethod){
    double *prevU;
    if (updateMethod == JACOBI){
        prevU = malloc(n * sizeof(double));
        deepCopyVector(u, prevU, n);
    }
    for (int i = 0; i < n; ++i){
        u[i] = 0;
        if (i > 0){
            if (updateMethod == JACOBI)
                u[i] += A[(getIndex(i, i-1, n))] * prevU[i-1];
            else
                u[i] += A[(getIndex(i, i-1, n))] * u[i-1];
        }
        if (i < n - 1) u[i] += A[getIndex(i, i + 1, n)] * u[i+1];
        u[i] = (f[i] - u[i])/A[getIndex(i, i, n)];
    }
    if (updateMethod == JACOBI) free(prevU);
}

void solve(double *A, double *f, double *u, int n, method updateMethod){
    double initialResidual = computeResidual(A, f, u, n), currentResidual;
    currentResidual = initialResidual;
    int iterCt = 0;
    printf("iteration = 0, initial residual = %lf\n", currentResidual);
    while(!checkIfDone(initialResidual, currentResidual, iterCt)){
        iterCt++;
        update(A, f, u, n, updateMethod);
        currentResidual = computeResidual(A, f, u, n);
        printf("iteration = %d, residual = %lf\n", iterCt, currentResidual);
    }
    if (iterCt != ITER_LIMIT)
    printf("Residual decreased by a factor of 10^6 in %d iterations\n", iterCt);
}

method getUpdateMethod(int val){
    switch(val){
        case 0: return JACOBI;
        case 1: return GAUSS_SEIDEL;
    }
    return JACOBI;
}

int main(int argc, char *argv[]){
    int n = readInt(argv[1]);
    method updateMethod = getUpdateMethod(((argc == 3)? (readInt(argv[2])) : 0));
    double *A, *f, *u;
    A = malloc((n*n) * sizeof(double));
    u = malloc((n) * sizeof(double));
    f = malloc((n) * sizeof(double));
    init(A, f, u, n);
    clock_t start = clock();
    solve(A, f, u, n, updateMethod);
    clock_t end = clock();
    double cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time taken: %f\n", cpu_time_used);
    free(A); free(u); free(f);
    return 0;
}
```

□