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
The results obtained with different optimization flags are as follows:

- with `-O0` flag:

Dimension	Time	Gflop/s	GB/s
20	0.005334	0.002999	2.399569
40	0.047960	0.002669	2.135113
60	0.116195	0.003718	2.974314
80	0.267990	0.003821	3.056832
100	0.538572	0.003714	2.970821
120	0.917298	0.003768	3.014071
140	1.458527	0.003763	3.010161
160	2.181550	0.003755	3.004103
180	3.109171	0.003751	3.001185
200	4.271699	0.003746	2.996466
220	5.690734	0.003742	2.993779
240	7.393920	0.003739	2.991431
260	9.416209	0.003733	2.986510
280	11.753274	0.003735	2.988376

Dimension	Time	Gflop/s	GB/s
300	14.477891	0.003730	2.983860
320	17.633859	0.003716	2.973189
340	21.141582	0.003718	2.974536
360	25.091385	0.003719	2.975109
380	29.601478	0.003707	2.965906
400	34.503788	0.003710	2.967790
420	40.772205	0.003634	2.907392
440	46.947766	0.003629	2.903107
460	54.249104	0.003588	2.870787
480	60.972539	0.003628	2.902080
500	70.212102	0.003561	2.848512
520	79.169652	0.003552	2.841655
540	88.773435	0.003548	2.838038
560	96.895735	0.003625	2.899876
580	110.408044	0.003534	2.827504

- with `-O3` flag:

Dimension	Time	Gflop/s	GB/s
20	0.000732	0.021844	17.475405
40	0.004857	0.026355	21.083876
60	0.014802	0.029186	23.348920
80	0.027851	0.036768	29.414017
100	0.047061	0.042498	33.998127
120	0.082460	0.041911	33.528831
140	0.132225	0.041505	33.204113
160	0.200849	0.040787	32.629447
180	0.286816	0.040667	32.533792
200	0.402399	0.039762	31.809214
220	0.559339	0.038073	30.458787
240	0.757871	0.036481	29.184933
260	0.994505	0.035346	28.276989
280	1.263213	0.034756	27.804657

Dimension	Time	Gflop/s	GB/s
300	1.602840	0.033690	26.952155
320	2.025206	0.032360	25.888127
340	2.409130	0.032629	26.103365
360	2.896626	0.032214	25.771224
380	3.493567	0.031413	25.130534
400	4.082232	0.031355	25.084314
420	4.784492	0.030970	24.776050
440	5.565125	0.030614	24.490805
460	6.436452	0.030245	24.196188
480	7.320607	0.030214	24.171109
500	8.433038	0.029645	23.716245
520	9.507764	0.029578	23.662009
540	10.875153	0.028958	23.166792
560	11.990609	0.029292	23.433805
580	13.860926	0.028153	22.522247

We see that with -O3 optimzation, 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,

$$-\Delta u = f \text{ on } \Omega,$$
$$u = 0 \text{ on } \partial\Omega,$$

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:
Processor: Apple M1 Chip
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;
}
```

□