

1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Homework02, put each programming exercise into its own subdirectory of this directory, zip the entire Homework02 directory up into the file Homework02.zip, and then submit this zip file to Homework #2. Make sure all the code and document files are included in the zip file.

2 Programming Exercises

1. When we solve a large linear system, we often use Gaussian elimination followed by backward substitution. An upper triangular system has zeroes below the main diagonal extending from the upper left-hand corner to the lower right-hand corner. For example, the linear system (matrix)

$$\begin{bmatrix} 2 & -3 & 0 & 3 \\ 4 & -5 & 1 & 7 \\ 2 & -1 & -3 & 5 \end{bmatrix}$$

can be reduced to the upper triangular form

$$\begin{bmatrix} 2 & -3 & 0 & 3 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & -5 & 0 \end{bmatrix}$$

If the coefficient matrix is square and none of the diagonal entries after reduction are zero then the system has a unique solution, as above.

This system can be solved by first finding using the last equation, then finding using the second equation, and finally finding using the first equation. We can devise a couple of serial algorithms for back substitution. A row-oriented version is the following where A is the square coefficient matrix, b is the vector of constants $([3, 7, 5])$ in the above example, x is the solution vector, and n is the size of the matrix A , specifically $n \times n$.

An alternative is the following column-oriented algorithm. As before A is the square coefficient matrix, b is the vector of constants $([3, 7, 5])$ in the above example, x is the solution vector, and n is the size of the matrix A , specifically $n \times n$.

- (a) Determine whether the outer loop of the row-oriented algorithm can be parallelized. Determine whether the inner loop of the row-oriented algorithm can be parallelized.
- (b) Determine whether the (second) outer loop of the column-oriented algorithm can be parallelized. Determine whether the inner loop of the column-oriented algorithm can be parallelized.

Algorithm 1 Row-Oriented Backward Substitution

```

1: procedure ROWBACKSUB( $A, b, x, n$ )
2:   for  $i$  from  $n - 1$  down to  $0$  do                                ▷ Inclusive
3:      $x[i] \leftarrow b[i]$ 
4:     for  $j$  from  $i + 1$  to  $n - 1$  do                                ▷ Inclusive
5:        $x[i] \leftarrow x[i] - A[i][j] \cdot x[j]$ 
6:     end for
7:      $x[i] \leftarrow x[i] / A[i][i]$ 
8:   end for
9: end procedure

```

Algorithm 2 Column-Oriented Backward Substitution

```

1: procedure COLBACKSUB( $A, b, x, n$ )
2:   for  $i$  from  $0$  to  $n - 1$  do                                ▷ Inclusive
3:      $x[i] \leftarrow b[i]$ 
4:   end for
5:   for  $i$  from  $n - 1$  down to  $0$  do                                ▷ Inclusive
6:      $x[i] \leftarrow x[i] / A[i][i]$ 
7:     for  $j$  from  $0$  to  $i - 1$  do                                ▷ Inclusive
8:        $x[j] \leftarrow x[j] - A[j][i] \cdot x[i]$ 
9:     end for
10:  end for
11: end procedure

```

Write an OpenMP program that includes serial functions for the above two back-substitution methods and the parallelizations you can do with either or both. Specifically,

- (a) Let the user input the size of the array, and populate the coefficient array and the constants array with random integers. Make sure that the coefficient array is upper-triangular and none of the diagonal entries are 0.
 - (b) Have the program solve the system and time the serial versions and the parallel versions. Make sure that you use the same initial arrays for all the algorithms.
 - (c) Include a function that will test the solutions, have it take in the solution from a serial version and those from the parallel runs and check that all the entries are the same. With round-off and order errors the values may not be exactly the same so we will call them the same if the absolute value of their difference is less than 0.000001. This test will help determine if your parallelizations are working.
 - (d) Have the program display the times of each run. Also have it print out the speedup and efficiency of the run.
 - (e) Experiment with the different clause options and loop scheduling to get the best time out of your parallel code.
 - (f) Run your program on the machines in the Linux lab and in a separate document answer the following questions.
 - i. What from these algorithms could be parallelized? And why?
 - ii. What loop scheduling gave the best times? Why do you think this is the case.
 - iii. From your runs on the Linux lab machines, what was your maximum speedup and how many threads did you use to get it?
 - iv. From your runs on the Linux lab machines, what was your maximum efficiency and how many threads did you use to get it?
 - v. Compare the run-times of the different methods and their parallelizations.
2. Take the Matrix-Vector example from class and revise it into a program that will multiply two matrices together. The process can be found in nearly any introductory linear algebra textbook. The matrices to be multiplied need to be compatible in size. If they are not then have the program halt.
- (a) Let the user input the sizes of the two matrices, and populate the two matrices with random doubles between -10 and 10 .
 - (b) Have the program multiply the matrices using a serial version of the algorithm and the parallel version.
 - (c) Include a function that will test the solutions, have it take in the solution from a serial version and those from the parallel runs and check that all the entries are the same. With round-off and order errors the values may not be exactly the same so we will call them the same if the absolute value of their difference is less than 0.000001. This test will help determine if your parallelizations are working.

- (d) Have the program display the times of each run. Also have it print out the speedup and efficiency of the run.
 - (e) Experiment with the different clause options and loop scheduling to get the best time out of your parallel code.
 - (f) Run your program on the machines in the Linux lab and in a separate document answer the following questions.
 - i. What from the algorithm could be parallelized? And why?
 - ii. What loop scheduling gave the best times? Why do you think this is the case.
 - iii. From your runs on the Linux lab machines, what was your maximum speedup and how many threads did you use to get it?
 - iv. From your runs on the Linux lab machines, what was your maximum efficiency and how many threads did you use to get it?
3. Use the `task` directive (and any other OpenMP directives) to implement a parallel merge sort program. A templated merge sort function is below.

```

template<class T>
void merge(T A[], T Temp[], int startA, int startB, int end) {
    int aptr = startA;
    int bptr = startB;
    int i = startA;

    while (aptr < startB && bptr <= end)
        if (A[aptr] < A[bptr])
            Temp[i++] = A[aptr++];
        else
            Temp[i++] = A[bptr++];

    while (aptr < startB)
        Temp[i++] = A[aptr++];

    while (bptr <= end)
        Temp[i++] = A[bptr++];

    for (i = startA; i <= end; i++)
        A[i] = Temp[i];
}

template<class T>
void mergeSort(T A[], T Temp[], int start, int end) {
    if (start < end) {
        int mid = (start + end) / 2;
        mergeSort(A, Temp, start, mid);
        mergeSort(A, Temp, mid + 1, end);
        merge(A, Temp, start, mid + 1, end);
    }
}

template<class T>
void mergeSort(T A[], int size) {
    T *Temp = new T[size];
    mergeSort(A, Temp, 0, size - 1);
    delete [] Temp;
}

```

- (a) Let the user input the size of the array, and populate the array with random integers.

- (b) Have the program sort the array of integers and time a serial version and then sort and time a parallel version. Make sure that you use the same initial array for both algorithms. You don't want to sort an already sorted array.
- (c) Include a function that will test if an array is sorted. After you sort the arrays always send them through this test do you know if the algorithms and your parallelizations are working.
- (d) Have the program display the times of each run. Also have it print out the speedup and efficiency of the run.
- (e) Experiment with the different clause options and loop scheduling to get the best time out of your parallel code.
- (f) Run your program on the machines in the Linux lab and in a separate document answer the following questions.
 - i. From your runs on the Linux lab machines, what was your maximum speedup and how many threads did you use to get it? What was your maximum efficiency and how many threads did you use to get it?
 - ii. Update the program to include functions for the insertion sort, and for the `sort` function in the algorithm library.
 - iii. Time the 3 sorts, merge, merge parallel, and sort, and compare these.