[4005-735-01] Parallel Computing I
Programming Project 2

Christopher A. Wood
caw4567@rit.edu
4/29/2013

Overview

This report documents my work on the second Parallel Computing I programming project. It includes answers for the required questions, data to support my answers, and a personal reflection describing what I learned throughout the course of this work.

Questions

Question 1

Describe how your parallel program is designed. As part of your description, address these points:

a. Which parallel design patterns did you use?
b. What data structures did you use?
c. How did you partition the computation among the parallel processes?
d. Did you need to synchronize the processes, and if so, how did you do it?
e. Did you need to send messages between the processes, and if so, which message passing operations did you use and what data did you send in the messages?
f. Did you need to do load balancing, and if so, how did you do it?

Answer 1a

Based on similar reasoning discussed in project 1, I used the *result parallelism* design pattern in my program for both the initialization and solution tasks that are responsible for populating **A** and **b** and then solving the system, respectively. The **A** and **b** slices computed by each worker process in the initialization task are dependencies for the subsequent solution task. Therefore, we can say that each process is interested in computing every element of its own slice in order to proceed with solving the system, which is clearly result parallelism.

Similarly, in the solution task, all worker processes run the same block of code to compute a different slice of the **y** vector and determine if the new values for their slice converged below the specified convergence threshold. Once each process is done computing its own **y** slice, the processes perform an allGather and allReduce to distribute the new values and process-local convergence results. Both of these pieces of data are necessary to determine whether another iteration is required and, if necessary, print the solution. Since the output of this task may be viewed as the entire solution vector **x** (i.e. we are interested in every element of this vector), and the process of computing this vector is partitioned among the worker processes for any arbitrary number of iterations, this is clearly result parallelism.

Again, similar to project 1, the solution task must not be construed as agenda parallelism.

While the purpose of the reduction step is to share the convergence results (i.e. a Boolean flag), this is not the output of the task. The convergence flag is merely used to control the number of iterations of the algorithm. Therefore, a convenient way to view the role of the reduction step is that it helps the overall computation of the **x** vector.

Since this is a cluster program, I did make use of other explicit parallel design patterns for collective communication. As already mentioned, I used the all-gather and all-reduce message passing patterns to handle the distribution of **y** slices and convergence results, respectively. In addition to being a particular type of communication pattern for parallel programs, the use of the all-reduce communication procedure is a direct realization of the more general parallel reduction design pattern.

Finally, I *could* have made use of the master-worker design pattern to handle the distribution of row slices to each worker process for load balancing purposes. However, given the communication overhead that would result from the application of this pattern, I made a conscious decision not to implement this technique.

Answer 1b

My choice of data structures was based on the stated project requirements and the nature of the cluster-based implementation of the Jacobi algorithm. In particular, since each process now owns its own data and does not share access to its data structures between threads, the **A** matrix and the **b**, **x**, or **y** vectors did not require any kind of synchronization. Therefore, these are simply represented as double arrays (or a double matrix in the case of **A**).

Interprocess communication at the end of a single iteration of the inner loop of the algorithm is the only time when data is shared between processes. Given my row-based partitioning scheme for dividing the contents of **A**, **b**, and **y**, each process owns and broadcasts a unique partition of the **y** vector during each communication event. To distribute this information, instances of the DoubleBuf class were used as wrappers for the newly computed **y** vector. The Parallel Java library ensures that the contents of these buffers are placed into the appropriate locations in the destination buffers, which meant that I did not have to implement any synchronization code to facilitate the storage of data.

The last important piece of data is the algorithm control flow flag. Again, since this is local to each process, a primitive boolean type is used as its storage container. Also, similar to the solution distribution event, the convergence results for each process must be combined (reduced) across every process and then subsequently stored back in every process (i.e. through the use of an allReduce event). I was able to wrap this flag for distribution using the BooleanBuf class in the Parallel Java library, and then leverage the global communication to distribute the data throughout the set of worker processes. With this approach, I did not have to write any synchronization code to perform the Boolean reduction myself.

Therefore, to summarize, I used primitive double arrays to store computational data for the algorithm, primitive boolean variables to control the flow of the algorithm, and Parallel Java Buf classes to wrap each data structure for distribution throughout the worker processes.

Answer 1c

Following the same approach as in project 1, I partitioned the computation among the set of worker processes by assigning each process a disjoint subset (or slice) of the solution vector **y** to compute. In effect, this meant that each process only needed its own corresponding slice of **A** and **b**. Therefore, each process only allocated enough storage for its own slice of these respective data structures. However, given the nature of the Jacobi algorithm and the need to compute a matrix-vector product between **A** and **x**, each process needed to store the entire solution vector **x** when computing its own slice of **y**. This data structure allocation scheme is summarized in Figure 1.

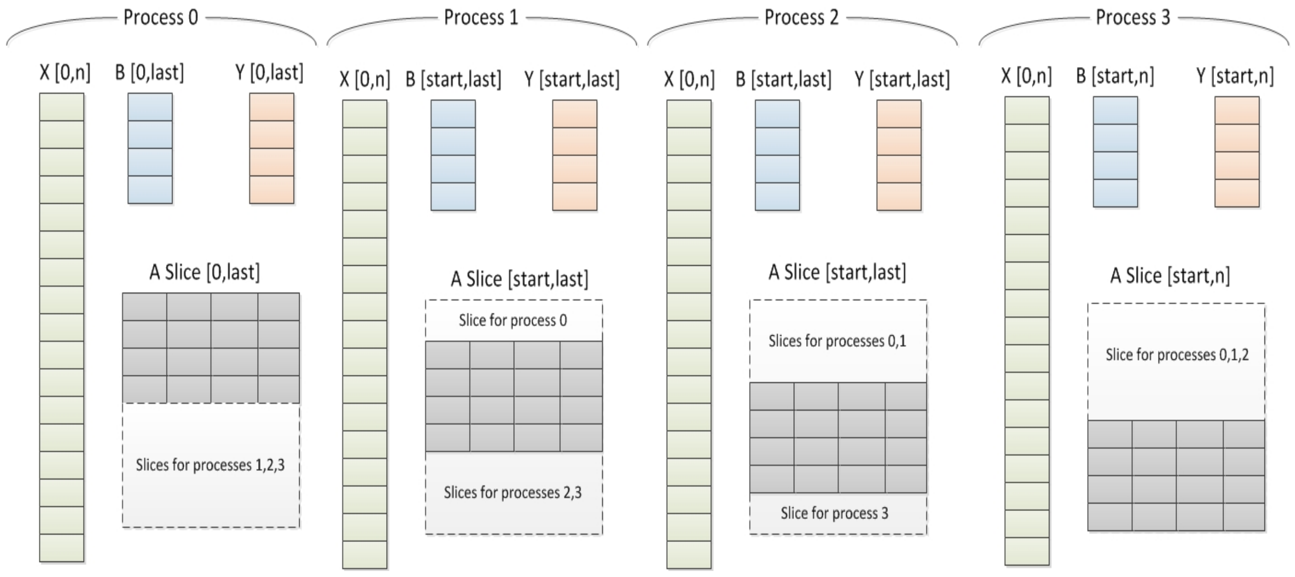

Figure 1. Data structure allocation strategy for a cluster program with K = 4 processes. The same idea extends to other values of K.

Also similar to project 1, there are two main tasks, or blocks of executed code, that implement the cluster-based version of the Jacobi algorithm. The initialization task is responsible for populating **A** and **b** with the data needed to compute the solution vector **x**, and the solution task is responsible for actually computing **x**. Based on my row-based partition of the computation, each process will only populate its own slice of **A** and **b** during the initialization task. However, since the parallel program must produce the same output as the sequential program, the PRNG objects used to populate **A** and **b** skip ahead to the appropriate place in the PRNG sequence of numbers based on the starting index of their own row slice. In particular, since each process starts at index ‘first’ and computes (n+1) random numbers for each row (n for **A** and one for **b**), the skip value for each process is equivalent to (n+1)*first.

Once the initialization task is complete, the program then executes the solver task, where new values of the solution vector **x** are computed until all processes converge on a shared result. During each iteration of this task, each process uses its own slice of **A** and **b**, in addition to the entire **x** vector, to compute the resulting slice of the **y** vector. Each process maintains a Boolean flag indicating whether or not their slice of the new solution converged. This is determined by checking the relative difference between each **y** coordinate and the corresponding **x** coordinate. Once each **y** slice has been fully computed, these partitions are then gathered into the new solution vector of every other process using an allGather operation (emulating a swap of **x** and **y**). Immediately after, the per-process convergence flags are reduced (using allReduce) and the result is stored in every process, which then determines if the algorithm proceeds.

The last important part of the program, which is not necessarily a part of the computation, is printing the result. Since this is a cluster-based program, only one worker process should be responsible for sending console output to the frontend process. To accomplish this task, process 0 is selected as the process that will display the output. All other processes will terminate earlier without displaying any output.

Answer 1d

Given the sequential dependency that exists between every iteration of the inner loop in the Jacobi algorithm, synchronization was needed to control the executions of the inner loop for each process in the solution task. Without this synchronization, we could not guarantee that each process would perform the same number of iterations and yield consistent results for the same input. Process synchronization is performed using two separate techniques. First, the distribution of each **y** partition to every process in the set of worker processes uses a blocking allGather procedure, which synchronized each process at the swapping stage of the algorithm. Then, immediately after, the blocking allReduce procedure is invoked to ensure the convergence result from every process is updated to every other process, which also synchronized the processes. While these two procedures are not explicit barriers, the blocking nature of the message passing operation can be conceptually viewed as an implicit barrier that synchronizes each process before continuing into the next iteration of the algorithm. Figure 2 illustrates how these two communication steps synchronize the processes during the iterations of the algorithm.

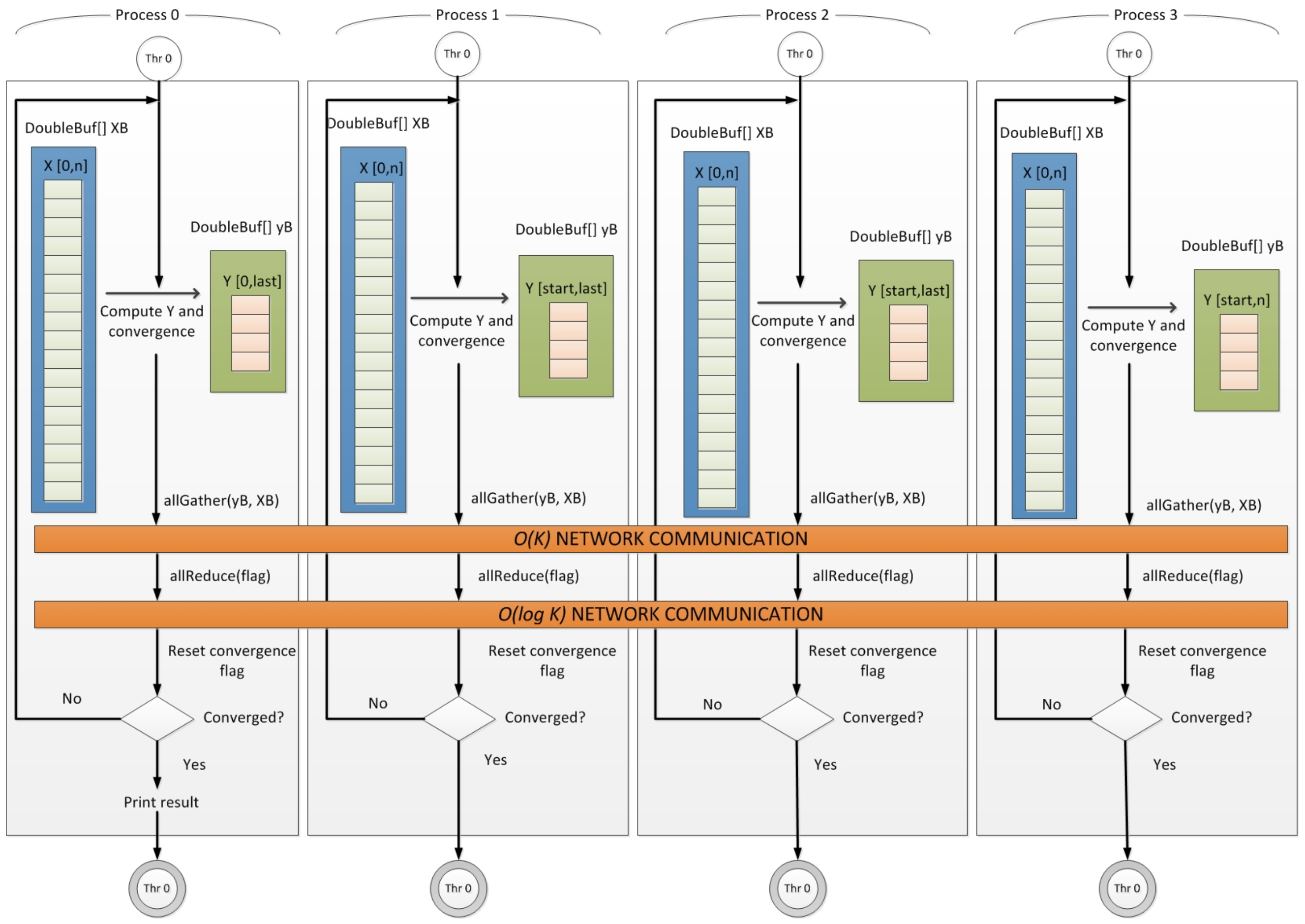

Figure 2. Control flow of the cluster-based Jacobi program with synchronization implemented using the allGather and allReduce operations (shown in the orange color).

Answer 1e

As previously discussed in answers 1c and 1d, message passing was needed to distribute the **y** partitions and the convergence results among the processes. For the **y** vector, an allGather procedure was used in which the source buffer was the unique partition of the **y** vector computed by each process and the destination buffer was the per-process **x** vector that would store the entire newly computed solution. Since both **x** and **y** are of type double[], each of the communication buffers used to distribute and store the **y** partitions were of type DoubleBuf. Also, an allGather operation was used because each process needed to broadcast its computed **y** partition and subsequently receive all other partitions of the entire **y** vector from the other processes, which is exactly what the all-gather pattern operation accomplishes.

With regards to the convergence result, an allReduce operation was used to reduce the results of each process's convergence flag into a single Boolean value, which was then redistributed to each process. The BooleanOp.AND reduction operator was used in the reduction phase. This was because if any process set its convergence flag to false, indicating that it did not converge, then the entire reduction result would be false, indicating that each process must complete at least one more iteration. As a parameter to the allReduce method, a single BooleanBuf object that wraps the per-process convergence flag was used. Each process then resets its own convergence flag when the allReduce procedure returned using the new Boolean value contained in the same buffer.

Answer 1f

Unlike the first project, load balancing was not implemented for the cluster-based version of the Jacobi algorithm. This is because the computation is partitioned across different processes that communicate using message passing, rather than different threads that communicate using shared memory. While it is true that some processes might detect convergence faster than others, I expect that it would have been detrimental to the overall performance to implement a dynamic or guided schedule to distribute row chunks to each process. The communication overhead of transmitting individual elements (smaller chunks) of the rows to processes with a dynamic or guided scheduling algorithm would have almost certainly outweighed the cost of additional relative difference checks in the solution task. Therefore, a conscious decision was made to opt for computation overhead through computing the relative difference with a fixed schedule, as opposed to the communication overhead through distribution many small row chunks.

Question 2

Table 1. Data set 1 results.

NP	T1 (msec)	T2 (msec)	T3 (msec)	T (msec)	Speedup	Efficiency	ESDF
Seq	53115	53169	53147	53115			
1	42870	43362	42825	42825	1.240	1.240	
2	26087	25237	25036	25036	2.122	1.061	0.169
3	17744	17511	17492	17492	3.037	1.012	0.113
4	15114	14147	15434	14147	3.755	0.939	0.107
8	9960	9902	9974	9902	5.364	0.671	0.121

Question 3

Table 2. Data set 2 results.

NP	T1 (msec)	T2 (msec)	T3 (msec)	T (msec)	Speedup	Efficiency	ESDF
Seq	87457	87172	87271	87172			
1	72004	72032	72148	72004	1.211	1.211	
2	44531	40303	41274	40303	2.163	1.081	0.119
3	28670	29022	28912	28670	3.041	1.014	0.097
4	23277	22869	22936	22869	3.812	0.953	0.090
8	14549	13982	14066	13982	6.235	0.779	0.079

Question 4

Table 3. Data set 3 results.

NP	T1 (msec)	T2 (msec)	T3 (msec)	T (msec)	Speedup	Efficiency	ESDF
Seq	128527	128514	128561	128514			
1	98957	99002	98777	98777	1.301	1.301	
2	58058	58019	58488	58019	2.215	1.108	0.175
3	43203	40097	40170	40097	3.205	1.068	0.109
4	31460	31931	31695	31460	4.085	1.021	0.091
8	17413	18488	17675	17413	7.380	0.923	0.059

Question 5

Summarize your measurements from Questions 2-4. As part of your summary, address these points:

a. How close to the ideal speedup and efficiency did your parallel program achieve as the number of parallel processes increased?
b. What is causing the discrepancy if any between the ideal and the measured speedup and efficiency in your parallel program?
c. Is this problem a good candidate for a cluster parallel program? Why or why not?

Answer 5a

Generally speaking, the speedups and efficiencies were very close to the ideal values for $K \leq 3$ processes. For K = 1, 2, and 3, the speedups were above 1.0, 2.0, and 3.0, respectively, and the efficiencies were all above 1.0. However, it is important to note that the efficiencies slowly decreased as the value of K increased. For example, in the last data set, my efficiencies for K =1 and K = 3 were 1.301 and 1.068, respectively. As the number of processes increased beyond K = 3, the speedups and efficiencies decreased and dipped below the ideal values (with the exception of K = 4 for the last data set). Collectively, these results are a direct realization of Amdahl's law, which states that the speedups will approach an asymptotic limit as the number of processors increases.

Answer 5b

The cause for these discrepancies is simple. First, we observe that the majority of the sequential fraction of the program lies in the synchronized communication steps (i.e. the allGather and allReduce communication operations), printing the solution, and controlling the flow of the algorithm. For a fixed problem size, the overhead of synchronizing the processors will increase with the number of processors. Therefore, as K increases, this sequential fraction will never decrease. As predicted by Amdahl's law, this means that the speedups and efficiencies will decrease as the overall execution time approaches 1/F, which is exactly what the data indicates.

Second, note that larger data sets achieve better speedups and efficiencies, causing higher superlinear values for small number of processors as compared to those values achieved with smaller data sets. The cause of this discrepancy is the surface-to-volume effect. The total computational complexity is on the order of $O(n^2)$, accounting for the initialization and solution tasks, whereas the communication complexity is only on the order of O(n). Therefore, the sequential fraction is equivalent to $O(n^{-1})$, which means that the sequential fraction will decrease as the problem size increases, causing the efficiencies to increase, which is exactly what's observed in the data for all three problem sizes.

Other causes for the superlinear speedups and efficiencies in the smaller number of processors are the usage of cache memory to share commonly used variables and early JVM optimizations for program hotspots. The cache helps improve the memory access times for **A**, **b**, and **x** when computing new **y** slices. As for the JVM, the code for the parallel program was structured so that the main hotspot of the solution task (i.e. the computation of the **y** slice) is isolated to a single method. This method is invoked for every iteration of the solution vector, which caused the JVM to compile the Java bytecode down to machine language sooner, and thus attain better speedups than in the sequential version.

Answer 5c

This answer depends on the input data set for the program. Although the computation is effectively distributed among each of the worker processes, the overhead of distributing the **y** vector slices and the convergence results is too large (i.e. on the order of O(n)) to warrant the problem to be run as a cluster parallel program. However, this is only absolutely true for relatively “small” input sizes. As can be seen from the data, the program experiences very good sizeups when the problem size is increased and run among multiple processes. Therefore, if the input data size is to be very large compared to those limited by the SMP parallel program, then a cluster parallel program would be a good candidate. With such large data sizes, the overhead of computation, which is $O(n^2)$, would far outweigh the overhead of communication, which is O(n), and we would be able to achieve better overall sizeups. This is not the case in an SMP machine in which the input size is limited by the amount of main memory available.

Question 6

Write a paragraph telling me what you learned from this project.

Answer

I learned that care is needed when synchronizing and sharing data across worker processes in a cluster parallel program. The network interface and type of messages used to send data need to be taken into account in order to avoid deadlocks and enable efficient communication. This task would be particularly difficult without useful libraries such as Parallel Java to handle the network I/O and corresponding synchronization logic. Also, I learned that clever restructuring of the code can yield vastly superior performance by exploiting the JVM. In particular, by forcing certain blocks of code to execute more often than others, we can ensure that such code is optimized quicker and thus reduces the overall runtime for the program. This observation was critical in speeding up my cluster parallel program.