School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS3008A Course Assignment

Hand-out date: May 4, 2023
Due date: Monday 16:00, June 5, 2023

Contents

1 Introduction

1. You are expected to work in a group of two on this assignment.
2. The assignment consists of three problems, each group is requested to complete all of them.
3. If you decide to work alone, you can choose any two of the three problems.
4. Hand-ins:
   (a) Only ONE submission is required from each group.
   (b) Submit a single compressed file named as `src_<student number>` for all the source codes. When decompressed, the folder `src_<student number>` should include three (or two for an individual project) folders, one for each problem. See at the end of each problem the details on source files to be submitted. (**Do not submit any code which does not compile!**)

(c) Submit a report named as `report_<student number>.pdf` that includes contents such as (also see Section 4)

- Parallelization approaches; you may use high-level pseudo codes or algorithms to describe the parallelization methods used. Do not put the source code in your report. It is a part of your overall methodology for designing a method to test the correctness of the parallelization results. In general, the key elements in the methodology include parallelization method, correctness check, performance evaluation, and very importantly the choice of baseline serial implementation.
- Experiment details and performance evaluation.
- Evaluation results and discussions. Note that the instructions given in this brief are rather high-level. It is the task for you to design and put in place the details and necessary elements to solve the problems fully.

5. Submission of code without report or report without code will result in 0 mark.
6. In your report, proper citations and references must be given where necessary.
7. Using Latex to write the report is required. A template tex file is provided.
8. Problem 1: 33%, Problem 2: 34%, Problem 3: 33%, and the total mark is 100 which makes up 20% course weight.
9. Start early and plan your time effectively. Meet the deadline to avoid late submission penalties (20% to 40% depending on the time for overdue).

2 Due Date

Monday, 16:00, June 5, 2023 — the submission of final report (plus a Turnitin report) and source codes on ulWazi course site. Turnitin report is expected to be generated automatically when you submit the report PDF file on ulwazi. Thus, separate submission links for report and source code will be used in order to generate Turnitin report successfully.

3 Problems

1. **Problem 1: Parallel Scan** Based on the algorithm given in Blelloch 1990, implement parallel scan algorithm using **OpenMP and MPI**, respectively (i.e., separately).

   Scan operation, or all-prefix-sums operation, is one of the simplest and useful building blocks for parallel algorithms (Blelloch 1990). Given a set of elements, $[a_0, a_1, \cdots, a_{n-1}]$, the scan operation associated with addition operator for this input is the output set $[a_0, (a_0 + a_1), \cdots, (a_0 + a_1 + \cdots + a_{n-1})]$. For example, the input set is $[2,1,4,0,3,7,6,3]$, then the scan with addition operator of this input is $[2,3,7,7,10,17,23,26]$.

   It is simple to compute scan operations in serial, see Listing 1.

```
scan(out[N], in[N]) {
  i=0;
  out[0]=in[0];
  for(i=1;i<N;i++){
    out[i]=in[i]+out[i-1];
  }
}
```

Listing 1: Sequential algorithm for computing scan operation with '+' operator

Sometimes it is useful for each element of the output vector to contain the sum of all the previous elements, but does not include the element itself. Such an operation is called prescan. That is, given the input $[a_0, a_1, \cdots, a_{n-1}]$, the output of prescan operation with addition operator is $[0, a_0, (a_0 + a_1), \cdots, (a_0 + a_1 + \cdots + a_{n-2})]$.

The algorithm for scan operation in Listing 1 is inherently sequential, as there is a loop carried dependence in the `for` loop. However, Blelloch 1990 gives an algorithm for calculating the scan operation in parallel (see Blelloch 1990, Pgs. 40 - 44). Based on this prescan parallel algorithm, implement parallel programs for scan operation using OpenMP and MPI, respectively.

Name your source files as `scan.<file extension>` (serial implementation), `scan_omp.<file extension>` (OpenMP implementation), and `scan_mpi.<file extension>` (MPI implementation). Organize the source files into a folder named `scan`. In the same folder, provide `Makefile` and `run.sh` for the compilation and running.

2. **Problem 2: Parallel Bitonic Sort** The bitonic sort is based on the idea of sorting network. The bitonic sorting algorithm is suitable for parallel processing, especially for GPU sorting. **However, in this problem, you are requested to implement parallel bitonic sorting of integers based on the algorithm given in Section 5 using OpenMP and MPI, respectively (i.e., separately).**

   Name your source files as `bitonic.<file extension>` (serial implementation), `bitonic_omp.<file extension>` (OpenMP implementation), and `bitonic_mpi.<file extension>` (MPI implementation). Organize the source files into a folder named `bitonicsort`. In the same folder, provide `Makefile` and `run.sh` for the compilation and running.

3. **Problem 3: Parallel Graph Algorithm** Dijkstra's Single Source Shortest Path (SSSP) Algorithm. Based on the naive parallelization approach given in Section 6, implement parallel Dijkstra's algorithm using **OpenMP and MPI**, respectively (i.e., separately).

   NOTE: To test your serial and parallel implementations of Dijkstra's algorithm, you may consider graphs that are undirected and connected. A number of test graphs are provided in folder `input_graphs`. The first line of each text file in this folder specifies the number of vertices and the number of edges in the graph, followed by a list of edges that are represented as a pair of vertices that are connecting an edge, and its weight. For example, a line that is given as `0 5 31` indicates there is an edge between vertices 0 and 5 with a weight 31. Your program should read from such a text file to obtain the input graph. Files `graph_0.txt` and `graph_1.txt` also include the sample outputs including the minimum cost to each vertex from the source and the paths to all other vertices from the source vertex. Finally a graph read in from a text file should be stored using an adjacency matrix which is a square and symmetric matrix for undirected graphs.

   Name your source files as `sssp.<file extension>` (serial implementation), `sssp_omp.<file extension>` (OpenMP implementation), and `sssp_mpi.<file extension>` (MPI implementation). Organize

the source files into a folder named `sssp_dijkstra`. In the same folder, provide `Makefile` and `run.sh` for the compilation and running.

4 General marking guide

The marks will be distributed approximately as the following.

1. In each implementation of the problems listed above, you are expected to include
   - a (correct) baseline (often a very optimal serial algorithm) that is as efficient as possible; [20%]
   - a parallel implementation with an efficient and relevant correctness assertion (depending on the nature of a problem, this check could be designed in different ways); [40%]
   - (giving a bad choice of serial algorithm or implementation in order to make the speedup look good will be penalized.)
   - no result of running the MPI code on the cluster will be penalized by a small mark (to be decided).
2. Report (only one combined report, a PDF file of 3-6 pages, is required): [30%]
   - The main body of the report should include a section for each problem solved.
   - Adequate writing to describe clearly the approach to solve the problems including parallelization methods, correctness, and testing methods.
   - Performance evaluation by i) comparing the performance (speedup) of the baseline and parallel version; ii) testing the scalability by varying the input size or number of processing elements. (Proper tables, line graphs, or bar graphs are good ways of presenting such results.)
3. Makefiles, run scripts, (job scripts where applicable), readme files are provided, and files submitted are as required. [10%]
4. You may also check out a draft marking sheet (https://docs.google.com/spreadsheets/d/16dQzj9mxhiPnSV23yjbTysjleZq-H7Ed/edit?usp=sharing&ouid=107626726130756614486&rtpof=true&sd=true) to be used, where you can see how the marks are going to be allocated in more details.

   The code quality is evaluated on criteria (wherever relevant) including roughly: the code compiles and run using the submitted Makefile and run scripts; the terminal outputs give very brief essential information including 1) correctness assertion; 2) sequential runtime, parallel runtime, and expected or reasonable speedup; 3) **own effort is evident**, and 4) using the submitted run script file, test runs for different problem sizes and using differing number of processes can be done successfully, and the performance is relatively consistent (some fluctuation is allowed) for different problem sizes. The output (usually in the terminal) after a run gives basic information on correctness assertion, runtime, and speedup. If unnecessary excessive outputs such as the entire output or input array are printed on the terminal (or other bad implementations), which makes it difficult to find the relevant basic information, then **negative** marks will be given even if the implementation is correct.

5 Parallel Bitonic Sort

The following is a brief introduction of bitonic sorting taken from Grama et al. 2003, Chap. 9.

- A sequence of keys $(a_0, a_1, \ldots, a_{n-1})$ is bitonic if
  1. there exists an index m, $0 \leq m \leq n-1$ such that
  $$a_0 \leq a_1 \leq \cdots \leq a_m \geq a_{m+1} \geq \ldots a_{n-1},$$
  2. or there exists a cyclic shift σ of $(0, 1, \ldots, n-1)$ such that the sequence $(a_{\sigma(0)}, a_{\sigma(1)}, \ldots, a_{\sigma(n-1)})$ satisfies condition 1. A cyclic shift sends each index i to $(i+s)$ mod n, for some integers s.
- A bitonic sequence has two tones – increasing and decreasing, or vice versa. Any cyclic rotation of such networks is also considered bitonic.
- $(1,2,4,7,6,0)$ is a bitonic sequence, because it first increases and then decreases. $(8,9,2,1,0,4)$ is another bitonic sequence, because it is a cyclic shift of $(0,4,8,9,2,1)$. Similarly, the sequence $(1,5,6,9,8,7,3,0)$ is bitonic, as is the sequence $(6,9,8,7,3,0,1,5)$, since it can be obtained from the first by a cyclic shift.
- If sequence $A = (a_0, a_1, \ldots, a_{n-1})$ is bitonic, then we can form two bitonic sequences from A as
  $$A_{min} = (min(a_0, a_{n/2}), min(a_1, a_{n/2+1}), \ldots, min(a_{n/2-1}, a_{n-1})),$$
  and
  $$A_{max} = (max(a_0, a_{n/2}), max(a_1, a_{n/2+1}), \ldots, max(a_{n/2-1}, a_{n-1})).$$
  A_{min} and A_{max} are bitonic sequences, and each element of A_{min} is less that every element in A_{max}.
- We can apply the procedure recursively on A_{min} and A_{max} to get the sorted sequence.
- For example, $A = (6,9,8,7,3,0,1,5)$ is a bitonic sequence. We can split it into two bitonic sequence by finding $A_{min} = (min(6,3), min(9,0), min(8,1), min(7,5))$, which is $A_{min} = (3,0,1,5)$ (first decrease, then increase), and $A_{max} = (max(6,3), max(9,0), max(8,1), max(7,5))$, which is $A_{max} = (6,9,8,7)$ (first increase, then decrease).
- The kernel of the network is the rearrangement of a bitonic sequence into a sorted sequence.

Original sequence	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
1st Split	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
2nd Split	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3rd Split	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
4th Split	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

Table 1: Merging a 16-element bitonic sequence through a series of log 16 bitonic splits.

- We can easily build a sorting network to implement this bitonic merge algorithm.
- Such a network is called a *bitonic merging network*. See Table 1 for an example.

- The network contains $\log n$ columns (see Figure 3). Each column contains $n/2$ comparators and performs one step of the bitonic merge.
- We denote a bitonic merging network with n inputs by $\oplus$BM[n].
- Replacing the $\oplus$ comparators by $\ominus$ comparators results in a decreasing output sequence; such a network is denoted by $\ominus$BM[n]. **(Here, a comparator refers to a device with two inputs x and y and two outputs x' and y'. For an increasing comparator, denoted by $\oplus$, $x' = min(x,y)$ and $y' = max(x,y)$, and vice versa for decreasing comparator, denoted by $\ominus$.)**
- The depth of the network is $\Theta(\log^2 n)$. Each stage of the network contains $n/2$ comparators. A serial implementation of the network would have complexity $\Theta(n \log^2 n)$. On the other hand, a parallel bitonic sorting network sorts n elements in $\Theta(\log^2 n)$ time. (The comparators within each stage are independent of one another, i.e., can be done in parallel.)
- How do we sort an unsorted sequence using a bitonic merge?
  - We must first build a single bitonic sequence from the given sequence. See Figure 1 for an illustration of building a bitonic sequence from an input sequence.
    - A sequence of length 2 is a bitonic sequence.
    - A bitonic sequence of length 4 can be built by sorting the first two elements using $\oplus$BM[2] and next two, using $\ominus$BM[2].
    - This process can be repeated to generate larger bitonic sequences.
  - Once we have turned our input into a bitonic sequence, we can apply a bitonic merge process to obtain a sorted list. Figure 2 shows an example.

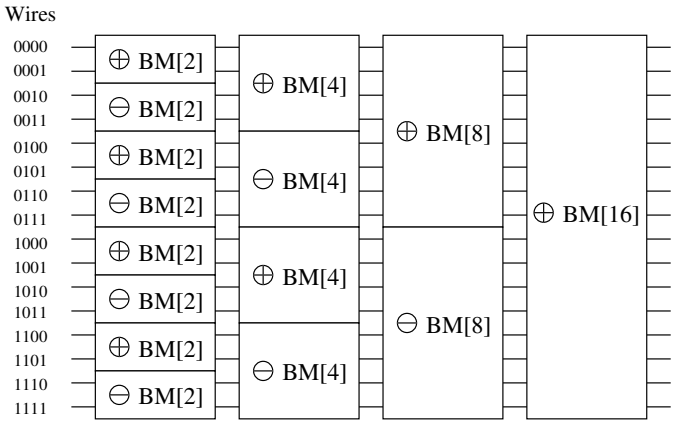

Figure 1: A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example, $\oplus$BM[k] and $\ominus$BM[k] denote bitonic merging networks of input size k that use $\oplus$ and $\ominus$ comparators, respectively. The last merging network ($\oplus$BM[16]) sorts the input. In this example, $n = 16$.

To implement bitonic sorting in MPI, the basic idea is that you have much more elements than the number of PEs. For example sorting one million or even more data elements using a small number of MPI

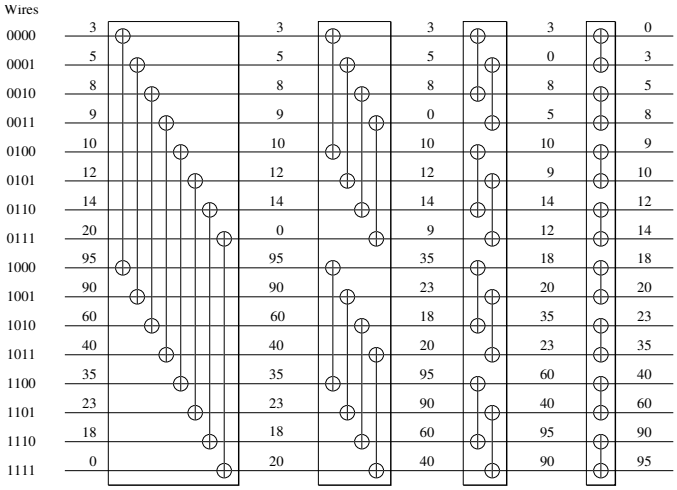

Figure 2: A bitonic merging network for $n = 16$. The input wires are numbered $0, 1 \ldots, n-1$, and the binary representation of these numbers is shown. Each column of comparators is drawn separately; the entire figure represents a $\oplus$BM[16] bitonic merging network. The network takes a bitonic sequence and outputs it in sorted order.

processes in our case, say 4, 8, or 16 processes. So, you need to fit the bitonic sorting idea into this kind of framework, not that you are sorting 16 elements only using 16 PEs. In your implementation, the main element from the perspective of distributed programming model is to figure out how to pair up a PE with its correct partner at each step. The overall idea is to begin by dividing the data elements among the PEs evenly first, then each PE sorts its share of elements in increasing or decreasing orders depending on its MPI process rank. In this way, the goal is to generate a global bitonic sequence owned by the entire MPI processes. For example, if you have 4 MPI processes, then you can have processes 0 and 1 jointly own a monotonic (say increasing) sequence; and processes 2 and 3 jointly own another monotonic (say decreasing) sequence. Then processes 0, 1, 2, and 3 jointly own a bitonic sequence. The remaining work will be then how to carry out the bitonic split steps. **Lastly, note that you need to assume both the number of elements to be sorted and the number of PEs to be powers of two.**

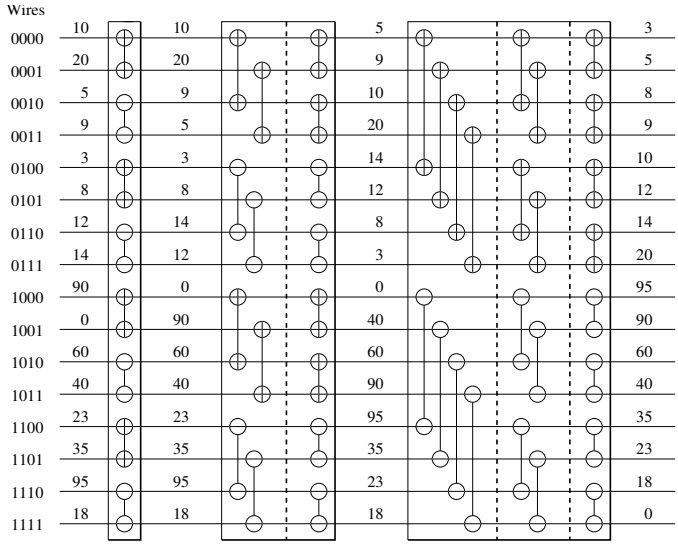

Figure 3: The comparator network that transforms an input sequence of 16 unordered numbers into a bitonic sequence.

6 Parallel Graph Algorithm

The following is a brief introduction of SSSP based on Grama et al. 2003, Chap. 10.

For a weighted graph $G = (V, E, w)$, where V is the set of vertices, E is the set of edges, and w contains the weights, Dijkstra's SSSP algorithm finds the shorted paths from a source vertex $s \in V$ to all other vertices in V. A shortest path is from u to v is a minimum-weight path. Depending on the application, edge weights may represent time, cost, penalty, loss, or any other quantity that accumulates additively along a path and is to be minimized. Dijkstra's SSSP solves the single-source shortest-path problem on both directed and undirected graphs with non-negative weights. It is a greedy algorithm that incrementally finds the shortest paths. Given a weighted graph $G(V, E, w)$, where V is the set of vertices, E is the set of edges, and w contains the weights, Dijkstra's algorithm is shown in Algorithm 1.

Dijkstra's algorithm is iterative (the `while` loop in Line 9). Each iteration adds a new vertex to the shortest paths. Since the value of $l[v]$ for a vertex v may change every time a new vertex u is added in V_T, it is difficult to select more than one vertex to include in the shortest paths. Thus it is not easy to perform different iterations of the `while` loop in parallel. However, each iteration can be performed in parallel as follows.

Let p be the number of processes, n be the number of vertices in the graph, and assume we are using an adjacency matrix representing the graph. Then the set V is partitioned into p subsets using 1D block mapping, that is, each process has n/p (assume $n\%p = 0$) consecutive vertices (or columns in the adjacency matrix), and the work associated with each subset is assigned to a different process. Let V_i be the subset of vertices assigned to process p_i where $i = 0, 1, \ldots, p-1$. Each process p_i stores the part of

Algorithm 1 Dijkstra's algorithm

1: $V_T = \{s\}$; ▷ Initialize V_T with the source vertex
2: **for** all $v \in (V - V_T)$ **do**
3: **if** (s, v) exists **then**
4: set $l[v] = w(s, v)$;
5: **else**
6: set $l[v] = \infty$;
7: **end if**
8: **end for**
9: **while** $V_T \neq V$ **do**
10: find a vertex u such that $l[u] = \min\{l[v] | v \in (V - V_T)\}$; ▷ $l[u]$ stores the minimum cost to reach u from source s by means of vertices in V_T
11: $V_T = V_T \cup u$;
12: **for** all $v \in (V - V_T)$ **do**
13: $l[v] = \min\{l[v], l[u] + w(u, v)\}$;
14: **end for**
15: **end while**

the array l that corresponds to V_i (that is, process p_i stores $l[v]$ such that $v \in V_i$). Each process p_i then computes $l_i[u] = \min\{l[v] | v \in (V_i - V_T)$ for its subset of vertices during each iteration of the `while` loop. The overall minimum is then obtained over all $l_i[u]$ by using all-to-one reduction operation and is stored in one process, say p_0. Process p_0 now holds the new vertex, which will be inserted into V_T. Process p_0 broadcasts u to all processes by using one-to-all broadcast. The process p_i responsible for vertex u marks u as belonging to set V_T. Finally each process updates the values of $l_i[v]$ for its subset of vertices based on this new vertex u.

When a new vertex u is added to V_T, the values of $l[v]$ for $v \in (V - V_T)$ must be updated. The process responsible for v must know the weight of the edge (u, v). Hence, each process p_i needs to store the columns of the weighted adjacency matrix corresponding to set V_i of vertices assigned to it. This corresponds to 1-D block mapping of the matrix.

References

Blelloch, Guy E. (Nov. 1990). *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

Grama, Ananth et al. (2003). *Introduction to Parallel Computing*. Addison Wesley.