Collected SPD Exercises for the Academic Year 2018–19

Dr. M. Coppola

Rev. date 21/03/2019

1 Introduction

This document collects questions and exercises that support the teaching of the SPD course I taught over the years at the University of Pisa within the master course in Computer Science and Networking.

2 Guidelines

2.1 Difficulty Degree of Exercises

We adopt a way of ranking the expected difficulty of exercises that was made popular by the TAOCP series of books . Each exercise will have its difficulty ranked between 0 and 50 in the left margin. The most significant digit is the "class of the exercise", and the least significant digit ranks the exercises within the same class. ***

Symbol	Meaning in TAOCP	Meaning in this collection
00	Immediate	
10	Simple (one minute)	design in one minute, about 5 minutes to code and test
20	Medium (quarter hour)	design in 15 mnutes, about one hour to code and test
30	Moderately Hard	design is not trivial, coding may require a few hours
40	Term Project	Term project
50	Research Problem	Research Problem / Innovation topic
►	recommended	not used yet
M	Mathematically oriented	not used yet
HM	Requiring "higher math"	not used yet

As the convention is adapted to the level and content of the course

- some symbols are not used,
- at the time of initial compilation no exercise is present yet that fits in the two classes of highest difficulty,
- as most if not all exercises include coding, the time required to write, compile and test is longer that what is required to only figure out and design a solution.

2.2 Benchmarking and Evaluating Your Own Solutions

Plan in advance to measure the execution of your programs, and to measure the performance with different values of the input parameters and data. On a single machine, benchmarking performance gives you at least the option to esteem the parallel overhead of the algorithm, and an approximate idea of the degree of parallelism exploitation up to the number of core on the machine; on a parallel platform you shall measure a true parallel speedup.

2.3 Measuring time

system approach employ the time program

```
time mpirun -np 4 myprogram parameter1 parameter2
```

The obvious disadvantage: poor granularity and no deaggregation of the time measurement. Only suitable for quick and dirty checks.

MPI Wtime look up the man of `double MPI_Wtime(void)`, a portable and easy to use function provided by MPI. Its actual accuracy may depend on the implementation, see `MPI_Wtick`.

Plan within your code for repeated measurements in order to assess result reliability. E.g. report average and standard deviation of results. Check that what you are trying to measure is large enough to allow significant measurement.

2.4 Code Reuse, Results Reuse

Some exercises are based on code and results from previous ones. Always design and code allowing for future reuse.

- Can you just change a communicator and plug the farm source code in a different program?
- Stream management should never depend on knowing the stream length in advance
- Can you add grain management to your stream based computation? Can you dynamically vary the grain?
- Store results in organized form, employ scripting tools to run the tests and plot results. Larger projects often require frequent test reruns and comparisons, and producing figures from results should be within this automated process.
- Can you model the execution time for the program, and use it to predict the performance on different (smaller, bigger) computations?

3 MPI exercises

► **[22] Exercise MPI.1** – Ping-pong

Define the classical ping-pong program with 2 processes P_0 and P_1 that send back and fort a data buffer. The first process sends some data, the second process executes a simple operation on the data (e.g. sum 1).

Use a basic datatype for this exercise. Initialize properly the data so that you can actually verify you have received the operation result.

- Write the program so that it can perform a specified number N of iterations if needed.
- Verify after the given number $1 \dots N$ iterations, that the expected result is achieved.
- Add printouts close to communications: does the printout work correctly? are the strings in a recognizable order? Why?

Extension of the exercise:

- Generalize the ping-pong example to N processes. Each process sends to the next one, with some processes being special, e.g. implement
  - a Token ring (a process has to start and stop the communication by receiving back the token from process $N - 1$)
  - a One-way pipeline (one process starts and sends only, the last one only receives)
- Can you devise a communicator structure for these examples that goes beyond a single common communicator?

► **[18] Exercise MPI.2** – MPI derived Datataypes

Build datatypes for

- a square matrix of arbitrary element types and constant size $N \times N$
- a column of the matrix
- a row of the matrix
- a group of k columns of the matrix (e.g. $k = 3$).
- the upward and downward diagonals of the matrix

Perform a test of the datatypes within the code of exercise MPI.1, i.e. define the datatypes and the corresponding C / C++ data structures, emply them in communications and check that they work as expected: initialize the matrix in a known way, perform computation on the part that you pass along (e.g. multiply or increment its elements) and check the result you receive back.

► **[25] Exercise MPI.3** – MPI matrix multiplication

Write a program that can multiply $N \times N$ matrices A, B into a matrix C, that works with a non trivial range of the N parameter, e.g. $N \approx 100$. The program shall distribute the actual computation among the M available processes at runtime.

The multiplication algorithm is the classical one:

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} * B_{k,j}$$

Data of A and B are initially at a single process. The matrix C shall be partitioned among all processes. After the computation, the matrix C must be collected to a single process.

- Ensure that you can repeat the same test if needed. When initializing the data it makes sense either to read it from disk, or to initialize to know values . In order to easily check the results, it will be handy to be able to initialize one of A or B to a scalar multiple of the Identity matrix I , e.g. $A = x \cdot I, x \in \mathbf{R}$.

- for simplicity, you can choose to not implement all combinations of (N, M) *and* bail out in some cases,
- assume your processes are arranged either as a 1D or a 2D array, where each process owns respectively a strip or a square subset of the matrix C
- apply the owner-computes rule, so that each process will need the corresponding input data (rows of A and columns of B) to compute its share of C. The data from A and B shall be distributed at runtime to all the processes that need them. Do that with point to point communications in you first implementation.
- Compute the type and size of the data structures at runtime according to the number of processes (again, you can and should bail out on inappropriate inputs)

► [25] **Exercise MPI.4** – k-asynchronous point-to-point communication

Plain MPI point to point API does not provide communication with assigned, exact degree of asynchrony[1]. How do you implement such communication with given asynchrony k?

- Implement a communication function with asynchrony 1
- Implement a communication function with asynchrony k

Key questions to answer when designing you solution:

- Can you rely on MPI buffering?
- How do you implement a fixed size buffer?

Write a send and/or a receive function(s) wrapper in such a way to ease its reuse in different contexts (different variable values types, size, value of k).

► [25] **Exercise MPI.5** – k-arity tree of processes

Define a complete k-ary tree (a tree where each interior node has exactly k children) of processes, where at each node corresponds to an MPI communicator including the node itself and its children.

Test that the communicator struccture works by performing the following communication pattern: starting from the root, each node broadcasts an initial value (which is determined by the root node) to each son node (using `MPI_Bcast` in the top level communicator). Each interior node will in its turn broadcast the initial value to its own children, get back results from all of them, add a value computed from its own index, return the reply to the parent node. Leaf nodes only compute on the received values and send back the result. [2]

A simple example : local computation is multiplication of the received value v by the node index. All values received by child nodes are summed with the local value, and passed back to the father node. Each node n_i computes

$$f(n_i, v) = v \cdot i + \sum_{h : n_h \in sons(n_i)} f(n_h, v)$$

which also makes quite simple to check the result at the root.

Suggestions:

[1] Assigned asynchrony of degree K: asynchronous communication between a sender and a receiver (the sender normally does not block) which becomes synchronous if more than K messages are pending. So, the receiver can wait/skip at most K receives before the sender blocks on the send operation. We still assume messages must be received in the same order as they are sent.

[2] Level k in the tree is defined as the set of nodes at depth k (e.g. distance k) from the root node. Node indexes are consecutive positive integer values assigned to nodes starting from 0 (the root node), and following increasing levels. Given two nodes with indexes i, j, for all nodes p, q respectively a child of i and j, it holds true that $i < j \rightarrow p < q$. A full 3-ary tree with 3 levels will thus have the following indexes in its levels $\{0\}, \{1, 2, 3\}\{4, 5, 6, 7, 8, 9, 10, 11, 12\}$

- decide if you want to have separate processes for different nodes, or if you want to reuse processes for more tree layers. This impacts the way you assign indexes to nodes and the way you arrange communications.
- choose which collective and point to point operation to use
- if using a general value of k is too complex, start with the fixed value $k = 4$.

► [30] **Exercise MPI.6** – Task-farm skeleton

Build a task farm skeleton program aiming at general reusability of MPI code. Your solution should allow to change the data structures, computing functions and possibly the load distribution policy without changing the MPI implementation code. *(further description and notes are present on the slides for the MPI lab time).*

Simplifying assumptions:
- single emitter and collector
- stream generation and consumption are functions called within the emitter and collector processes
- explicitly manage End-of-stream conditions via messages/tags of you choice

Constraints: in order to remain generic, outside of the stream generation function your code cannot assume that the stream lenght and content is known in advance.

Suggestions: leverage the separation of concerns as much as possible, by having (1) each kind of process code being a C/C++ function, as well as (2) each computing task being a function called by the generic worker/support process.

Experiment with different communication and load balancing strategies:
- simple round-robin,
- load balancing with explicit task request;
- explicit task request, implicit request via Ssend,
- a varying degree of worker buffering

What are the pros and cons in using separate communicators for the farm skeleton and its substructures?

Think of how you could implement some common extensions of the basic farm semantics: initial/periodic worker initialization, workers with status and status collection, work stealing strategies. ***

► [20] **Exercise MPI.7** – Mandelbrot set computation

Background The Mandelbrot set $\mathscr{M}$ is the set of points c in the complex plane $\mathbb{C}$, where a succession $z_0(c), z_1(c), \ldots$ does not diverge. The succession $z_i(c)$ is defined as

$$z_0(c) = c; z_{n+1}(c) = z_n^2(c) + c$$

where $c \in \mathbb{C}$ is the initial point of the succession.

We say that $c \in \mathscr{M}$ if the modulus of the sequence $z_0(c), z_1(c), \ldots$ doesn't diverge to ∞; formally:

$$\mathscr{M} = \left\{ c \mid \exists d \in \mathbb{R} : \forall n, \left| z_n(c) \right| \leq d \right\}$$

Well-known images of $\mathscr{M}$ are obtained by mapping a square of the complex plane to a bitmap ($c(x, y) = \alpha x + \beta i y$) and plotting pixels with a color that reflects how quickly $z_i(c)$ diverges (i.e. the number of iteration needed in order to detect divergence).

The Mandelbrot set is a fractal, with several interesting properties which we here state without proof.

- It is not possible to explicitly compute all the points in $\mathscr{M}$ without using the recursive definition[3].
- If $|c| > 2$, then $c \notin \mathscr{M}$, so the Mandelbrot set is confined in a circle with radius 2; this is paramount to let $d = 2$ in previous definition.
- $\mathscr{M}$ is approximately self-similar; it is roughly composed by a superimposition of three circles (or a circle and a cardiod figure), but it has smaller and smaller ramifications that start from its apparent borders and produce finer structures resembling the larger shape.
- The set is *connected*, and has no holes: which means that, for each couple of points $c_1, c_2 \in \mathscr{M}$, there is a path that connects c_1, c_2 and is entirely composed of points within $\mathscr{M}$; as a consequence, the seemingly separated copies of $\mathscr{M}$ that surround it are actually linked to the main set by a stripe so thin that we cannot observe it at the resolution used to approximate the set.

A common algorithm to approximate $\mathscr{M}$ is the *escape time algorithm*. For each interesting point c we compute iteratively the values $z_i(c)$ until which a critical escape condition is met, which tells us the succession starting from c diverges. Our approximations lies in assuming that $c \in \mathscr{M}$ if i exceeds a given threshold t, and we still have not detected divergence. The output of our algorithm is thus the number of iterations performed $i = escape(c) : 0 \leq i \leq t$.

The common escape conditions $|z_n(c)| > 2$ can be made simpler and quicker to evaluate. First, we observe that $|z_n(c)| > 2 \quad iff \quad |z_n(c)|^2 > 4$, so we can save a square root. As no complex number with a real or imaginary part greater than 2 can have modulo less than 2 a further simplification is to only check that both components of $z_n(c)$ are less than 2 in absolute value. The points that require the greater computational efforts are those in $\mathscr{M}$, followed by those which are very close to the set boundary.

***

Assignment Employ the mandelbrot escape time test on all the values c corresponding to pixels in a rectangular bitmap.

- Program input parameters are the extremes of the rectangle in C, the bitmap density (i.e. the α, β values) and the threshold t in number of iterations.
- Plot the results (or save them to a file for visualization with external tools).
- Insert the mandelbrot computation function and the overall work so obtained in the farm template coded in exercise 6. The input stream will contain at least candidate c values and the output stream the number of iterations reached.
- Test and evaluate the speedup, with respect to changing values of (1) input parameters (i.e. amount of points, iteration threshold). (2) farm size (3) farm distribution and collection policies (4) task stream organization

***

Questions and Notes How do you add task grain management?
Can you dynamically vary the grain? (Aggregation of a square or row of points in a single task is problem-specific : a nice feature, but it is not a general form of farm grain-size control)
Can you model the execution time of the farm from a small execution and try to predict for a longer one?

[3]We can prove that some parts of the complex plane do/do not belong to the mandelbrot set; but we cannot prove it for arbitrary sets of points close to the set boundary.

How do the grid resolution and iteration parameters, as well as choices about communication and load balancing, affect the prototype?

[20] **Exercise MPI.8** – Farm Skeleton with worker status and Reinitialization

Extends exercise MPI.6

Add to the farm skeleton the option for workers to have an internal status (represented by a data structure of your choice) that can affect the computation, as well as a mechanism to reinitialize the workers (i.e. send to all workers a message that changes their status data and influences the computation from that point of the stream of tasks).

The stream computation performed by the workers depends on the farm workers' "status"; each part of the stream (substream) is associated with a specific status that is spread to all workers at the beginning of that substream [4].

Example: the status is the max number of iteration i_{max} in a Mandelbrot computation. As a new image is being computed, a new substream of tasks starts and the value of i_{max} used by the workers needs to be updated.

Constraints:

- You cannot just assume to send the whole status within each job (status updates may be sporadic and quite larger than ordinary tasks).
- The substream computation associated with any status value at the emitter. Your code must deal with varying computation time in the worker in such a way that the above rule is not violated, i.e. all tasks of a substream are computed using the same "status" value.

Suggestions:

- How do you send/receive status updates? Some options are with `MPI_ISSend`, `MPI_IBSend`, `MPI_Ssend` or controlling non-determinism within the workers' `MPI_Recv` operations.
- Should you serialize the communications, and how? E.g. by adding a progressive identifier to the task, to the status messages or both, and how to link these identifiers to the substream.
- Manage substream ordering in the emitter (chose a semantics: no ordering, reordering the results by the task id, reordering the result by substreams but not by the tasks).

[4]The ordinary farm of exercise MPI.6 is the special case where the stream contains only one substream, thus initialization is performed only once at the beginning of the stream, and there is no need to send reinit messages by the emitter.

4 TBB

► [15] **Exercise TBB.1** – Parallel for

Write a parallel for with TBB, with a step-by-step approach, using 1D and 2D ranges.

1. Write the for without any actual computation (leave operator() empty)
2. Perform a simple computation without any actual value passed, just using the indexes
3. Implement passing an initialized array of given type to the for: have the operator() perform some computation on the array data; add another array to store the results.
4. Modify the code to deal with matrices and employ a 2D range in the parallel for.

[20] **Exercise TBB.2** – Simple Mandelbrot

Extends exercise TBB.1

Starting from the 2D version of the parallel for add a Mandelbrot function as the operator and let it compute over a 2D range that spans a rectangle in the complex plane. The per-element result is an integer, representing the number of iterations performed before detecting divergence.

Some code snippets are found here
`http://didawiki.cli.di.unipi.it/doku.php/magistraleinformaticanetworking/spd/2018/mandel`
that include code for the Mandelbrot function and code for saving array data as .ppm files you can view with standard tools.

Further ideas As a preliminary step to test your use of TBB, instead of the Mandelbrot function compute any long, iterative function and write the result somewhere (so that the compiler does not optimize the computation away).

[25] **Exercise TBB.3** – Study Mandelbrot load distribution

Extends exercise TBB.2

Starting from the Mandelbrot example do the following

- Set the investigated area to cover at least part of the Mandelbrot set
- Raise the number of pixels and the number of iterations of a couple of order of magniture
- setup your TBB program to only use one thread — see example there: `https://software.intel.com/en-us/node/506296` where explicit task scheduler initialization allows to control the thread pool size –
- experimentally find values that cause a sequential running time in the range [15s, 60s]
- allow TBB to create $T_N > 1$ threads, measure the difference in execution time with varying T_N
- devise a mechanism for approximating the distribution of computation time over the tasks as related to the function load imbalance. E.g. use an array of buckets where you total how many times a certain number of iterations shows up.
  The data structure to gather this data is a source of thread conflicts: evaluate the use of atomics, locking, or thread-local structures in order to gather the sums efficiently (reduce the performance impairment due to the monitoring)

Can you refine the model of the program computation time using this information?

List of Exercises

MPI.1	Ping-pong	3
MPI.2	MPI derived Datataypes	3
MPI.3	MPI matrix multiplication	3
MPI.4	k-asynchronous point-to-point communication	4
MPI.5	k-arity tree of processes	4
MPI.6	Task-farm skeleton	5
MPI.7	Mandelbrot set computation	5
MPI.8	Farm Skeleton with worker status and Reinitialization	7
TBB.1	Parallel for	8
TBB.2	Simple Mandelbrot	8
TBB.3	Study Mandelbrot load distribution	8

A Lab hints

A.1 Thread Building Blocks

- Download latest TBB either as binary package (for your OS) or as source package (needed to extract in-source documentation with doxygen).
  - Unpack in your working directory
  - If starting from source, check out the readme and compile it, e.g. with `make all` (this will also compile examples)
  - The html docs are only extracted if you explicitly trigger their makefile target. i.e. `make doxygen` and, if it works, check the root page `index.html`; you will need to have doxygen installed, of course.
- Be sure to have a suitable tool chain (e.g. recent C++ compiler and linker version). Check there `https://www.threadingbuildingblocks.org/system-requirements`
- *EITHER* set up your environment for command line usage
  - Look for the script `.../bin/tbbvars.*` (either the `sh` or `csh` version depending on your shell)
  - Edit the script and set up your TBB install directory location there.
  - Call it from a shell every time you need TBB (or launch it from your shell default configuration file)
- *OR* follow the instructions in TBB readme files to configure and use a GUI- based code development environment (DE)
  - Check for your own preferred DE : Eclipse, MS Visual Studio, Xcode ...and follow the instructions
  - Note that many configurations work besides those officially listed - Eclipse with CDT plugins is known to support TBB on Linux and OS X