Parallel and distributed systems: paradigms and models

Final project – Version 0.7

Academic Year 2021—22

M. Danelutto

The final project of the Parallel and distributed systems: paradigms and model course for the A.Y. 21—22 consists in providing a parallel implementation of one of the following subjects. There are two subjects (plus a free choice one) in the "Applications" section and two (plus a free choice one) in the "Run time supports" section. The former consists in the implementation of small applications or kernels in parallel, and requires two implementations, one developed using native C++ threads and one developed using FastFlow. The latter consists in the implementation of a small library supporting the execution in parallel of a particular computation pattern. In this case, it is required to provide again two implementations, one developed using native C++ threads and the other one developed using a limited subset of FastFlow components.

Each student must pick up one of the subjects and send an email to the professor (Subject "SPM22: project choice") with the indication of the choice. After email confirmation the student may start working on the subject agreed.

The project must be considered first from a theoretical viewpoint. Students must consider the problem at hand, figure out viable solutions, implement a sequential version providing useful insights on the weight and kind of different computations involved and then start actual project coding. The wrong way to work consists in trying different solutions directly and then filtering one the "good" one (please consider what stated in this respect during lessons and project discussion).

After implementing the project, students must deliver the project code and a PDF report by email, by the term stated on esami.unipi.it, Subject "SPM22 project submission"). We will take some time to review the submitted code and report and then an agenda will be published with a suitable number of slots to reserve the oral exam date. The oral exam will consist of a discussion of the project, and of some questions related to the more theoretical subjects covered during the lessons.

The report submitted along with the project code:

- Should be at most 10 pages
- Should not include known material, such as project problem description, known performance formulas, and so on

- Should detail the process leading to the final implementation (different alternatives, considered, evaluations used to pick up the final alternative, problems faced in the implementation and particular solutions taken, etc.)
- Should include some discussion relative to the motivation of the differences between achieved performance figures and the ideal/predicted ones, in particular pointing out qualitatively and quantitively, the major sources of overhead.
- Should include experiments performed on the remote virtual machine, with proper plots (B&W only, please do not use colors)
- Should include the instructions needed to re-compile and run the code

Applications

Jacobi (AJ)

The Jacobi iterative method computes the result of a system of equations Ax = B (with x vector of variable of length n, A matrix of coefficients of dimension n by n and B vector of known terms of length n) iteratively computing a new approximation of the values of the different variables according to the formula:

$$x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j>i} a_{ij}x_j^k) \quad i = 1, \cdots n$$

starting from some initial assignment of each x_i (e.g. 0).

We require to implement the Jacobi method with both native C++ threads and FastFlow.

Video motion detect (AV)

Simple motion detection may be implemented by subtracting video frames from some background images. We require to implement a parallel video motion detector processing video such that:

1. The first frame of the video is taken as “background picture”
2. Each frame is turned to greyscale, smoothed, and subtracted to the greyscale smoothed background picture
3. Motion detect flag will be true in case more the *k%* of the pixels differ
4. The output of the program is represented by the number of frames with motion detected

OpenCV may be used to open video file and read frames only (greyscale and smoothing should be programmed explicitly. Converting an image to greyscale can be achieved by substituting each (R,G,B) pixel with a grey pixel with a grey value which is the average of the R, G and B values. Smoothing requires pixel is obtained “averaging” its value with the value of the surrounding pixels. This is achieved by usually multiplying the 3x3 pixel neighborhood by one of the matrixes

$$H1 = \frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix} \quad H2 = \frac{1}{10}\begin{bmatrix}1&1&1\\1&2&1\\1&1&1\end{bmatrix} \quad H3 = \frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix} \quad H4 = \frac{1}{8}\begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}$$

and subsequently taking as the new value of the pixel the central value in the resulting matrix (H1 computes the average, the other Hi computes differently weighted averages.

Free choice (FCA)

In case you have some application or kernel, for instance from your research domain, and you are interested in working on a structured parallel implementation, you may pick up this “free choice” subject. The subject must be agreed with the professor. This means you have to write a few lines (half a page max) describing the kind of application/kernel you want to parallelize and send a message to professor (Subject “SPM22 project choice”). After approval, you can start working on the subject. The free choice application/kernel should be implemented in two versions, C++ native threads and FastFlow, as for the other application subjects.

Run time supports

Divide and conquer (DC)

We require to implement a divide and conquer pattern, accepting as parameters a bool basecase(Tin), a Tout solve(Tin), a vector<Tin> divide(Tin), a Tout conquer(vector<Tout>), and a bool stop(Tin) function and computing the divide and conquer pattern over a generic Tin input dataset. The additional stop function is used to figure out where to stop the parallelism exploitation. If stop(x) return true and basecase(x) returns false, the further divide and conquer steps must be computed sequentially rather than in parallel.

We require the implementation is provided either using STDlib C++ threads or using FF, limited to either the base patterns ff_pipe and ff_farm (with all their variants), or to the FF Buidling block components.

Dag interpreter (DI)

We require to implement in parallel the computation of a graph representing dependencies in between computations identified through void(void) lambdas. The graph must be expressed declaring Node objects (with an integer id, an input arity and an output arity), and dependencies may be added using the Node addDep(Node) method. Code to be computed in a node may be given through an addCompute(function<void(void)>). Graph object may be declared, and Node may be added through addNode(Node) method calls. Graphs may be computed (in parallel) invoking the Graph compute() method.

A simple graph with four nodes can be defined as follows:

```
Node A(1,1,2);  Node B(2,1,1);  Node C(3,1,1);  Node D(4,2,1);
A.addDep(B);  A.addDep(C);  B.addDep(D);  C.addDep(D);
A.addCompute([&]() { yb = x-1; yc = x+1; });
B.addCompute([&]() { zb = yb*2; });
C.addCompute([&]() { zc = yc*3; });
D.addCompute([&]() { res = zb+zc; });
Graph g;
g.addNode(A); g.addNode(B); g.addNode(C); g.addNode(D);
```

and execution invoked through the blocking method call

```
g.compute();
```

We require that Graph compute is implemented using with plain STDlib C++ threads or FastFlow, limited to either the base patterns ff_pipe and ff_farm (with all their variants), or to the FF Buidling block components.

Free choice (FCS)

In case you have some parallel pattern, for instance from your research domain, and you are interested in working on a structured parallel implementation, you may pick up this “free choice” subject. The subject must be agreed with the professor. This means you must write a few lines (half a page max) describing the kind of pattern you want to implement along with an idea of some different cases where the pattern may be effectively exploited, if available and send a message to professor (Subject “SPM22 project choice”). After approval, you can start working on the subject. As for the other pattern subjects, the implementation must be provided either using native C++ threads or FastFlow, limited to either the base patterns ff_pipe and ff_farm (with all their variants), or to the FF Buidling block components.