GrPPI: A Generic and Reusable Parallel Pattern Interface
Hands-On Exercises

Manuel F. Dolz, David del Rio Astorga, Javier Fernández
University Carlos III of Madrid

January 5, 2018

1 Introduction

This document contains a collection of hands-on exercises about the GrPPI interface. GrPPI is an open source generic and reusable parallel pattern programming interface developed at Univ. Carlos III of Madrid. This C++ header only library offers a high-level interface, similar to STL, of parallel patterns, which are implemented using existing parallel programming frameworks, e.g. ISO C++ Threads, OpenMP and Intel TBB. The parallel patterns supported by GrPPI are targeted for both stream processing and data-intensive applications and can be composed among them to build more complex constructions.

2 Building the exercises

The collection of GrPPI exercises has been placed in a git repository which automatically downloads GrPPI and builds the exercises of this collection.

```
$ git clone https://github.com/mdolz/grppi-exercises.git
```

Once cloned the repository, you can compile the collection of exercises by doing:

```
$ cd grppi-exercises
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Note that compiled programs will only be placed in the `build` folder. However, when doing the exercises you should modify the sources placed in the `apps` folder, outside `build`. To try out a new program version, you can go back to your `build` folder and re-execute `make`. This will re-compile your programs, so you will be able to try them again.

3 Getting started

Each exercise in this collection contains two versions of the application: the sequential and the parallel. The sequential versions, with the `_seq.cpp` suffix, are complete and are prepared to run right away. The parallel versions, with `_grppi.cpp` suffix, refer to the exercises that should be implemented using the GrPPI parallel pattern interface. You may also note that each application receives a different number of arguments, depending on their purpose (see function `main`. In this cases, the parallel versions have already been prepared for receiving an additional argument `mode`, which refers to the execution frameworks or back end that should be used for running a GrPPI pattern. In the same program you can find a helper routine `grppi::dynamic_execution execution_mode` that given a string, returns the parallel execution model that should be used. These strings are assigned in the following way: *i)* `seq` = Sequential execution; *ii)* `thr` = C++ Threads; *iii)* `omp` = OpenMP; and *iv)* `tbb` = Intel TBB. Any change in the execution parameters of these policies may be done directly in this helper routine.

Data parallel patterns

Exercise 1. The Mandelbrot set is the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, etc., remains bounded in absolute value. Thanks to these properties, the Mandelbrot set produce complex fractal shapes. Its definition and name are due to Adrien Douady, in tribute to the mathematician Benoît Mandelbrot.

Mandelbrot images may be created by sampling the complex numbers and determining, for each sample point c, whether the result of iterating the above function goes to infinity. Treating the real and imaginary parts of c as image coordinates $(x + yi)$ on the complex plane, pixels may then be colored according to how rapidly the sequence $z_n^2 + c$ diverge. The Mandelbrot set has become popular outside mathematics both for its aesthetic appeal and as an example of a complex structure arising from the application of simple rules. Especially, it became prominent in the mid-1980s as a computer graphics demo, when personal computers became powerful enough to plot and display the set in high resolution.

Given the inherent embarrassingly parallel nature of this problem, where each pixel in the image can be computed independently, this exercise intends to make use of the GrPPI patterns in order to accelerate the computation of the basic Mandelbrot image. Follow these steps and answer the questions:

1. Explore sequential naive code of the Mandelbrot application in `apps/mandelbrot/mandelbrot_seq.cpp`.
2. Compile and run the application to generate a BMP image of size 1,024×720:

```
$ make mandelbrot_seq
$ ./mandelbrot_seq 1024 720 test.bmp
```

3. Check if the image was correctly generated. How long did it take to run?
4. Which pattern(s) do you think the algorithm in function `mandelbrot` can match?
5. Use the code `apps/mandelbrot/mandelbrot_grppi.cpp` as a template to implement a parallel version of the algorithm in function `mandelbrot` using the a GrPPI pattern. Note that the `exec` argument refers to the GrPPI execution model, and can be directly used as the first argument in the pattern call. For more information, you can check the documentation in `https://github.com/arcosuc3m/grppi`. You can use the following trick to implement a pseudo parallel-for using the Map pattern:

```
std::vector<int> values(n), idx(n);
std::iota(idx.begin(), idx.end(), 0); // Fill vector: idx[0]=0; idx[1]=1; idx[2]=2; etc.
grppi::map(exec, begin(idx), end(idx), begin(values),
  [&] (int i) {
     // do something to compute and return values[i]
  }
);
```

6. Once implemented the parallel version, compile and run it. You can use first the sequential back end:

```
$ make mandelbrot_grppi
$ ./mandelbrot_grppi 1024 720 test.bmp seq 1
```

7. Did you observe any major overhead with respect to the naive sequential version?
8. Change to a parallel back end, e.g. C++ threads, OpenMP or Intel TBB, and run the application again using 8 threads.

```
$ make mandelbrot_grppi
$ ./mandelbrot_grppi 1024 720 test.bmp thr 8
```

9. Try configurations with increasing number of threads and check if the application scales.
10. How is the workload divided among the worker threads? Do you see the pixels computation as a homogeneous or a heterogeneous workload? Justify your answer.

Exercise 2. The matrix-vector product multiplication (also known as `gemv` in BLAS) between a matrix A and a vector x can be performed only for the case when the number of columns in A equals the number of rows in x. So, the product is defined as Ax, where A is a $m \times n$ matrix and x is an $n \times 1$ column. The matrix-vector product results in b, a $m \times 1$ column vector. The general formula for a matrix-vector product is given by

$$Ax = b \implies \begin{pmatrix} a_1 & a_2 & a_3 & \cdots & a_n \\ b_1 & b_2 & b_3 & \cdots & b_n \\ c_1 & c_2 & c_3 & \cdots & c_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_1 & m_2 & m_3 & \cdots & m_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a_1x_1 & a_2x_2 & a_3x_3 & \cdots & a_nx_n \\ b_1x_1 & b_2x_2 & a_3x_3 & \cdots & b_nx_n \\ c_1x_1 & c_2x_2 & a_3x_3 & \cdots & c_nx_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_1x_1 & m_2x_2 & a_3x_3 & \cdots & m_nx_n \end{pmatrix}.$$

Given these formulation, you can see the matrix-vector multiplication as a the sequence of the scalar products between the rows of A and the x column vector. Recall that the scalar product in BLAS is defined as the `dot` operation.

As you can see, the matrix-vector (`gemv`) product and their scalar (`dot`) products can be parallelized in different ways. In this exercise you are asked to represent this operations in terms of GrPPI patterns. To do so, follow these steps and answer the questions:

1. Explore sequential naive code of the matrix-vector product in `apps/dgemv/dgemv_seq.cpp`.
2. Compile and run the application to multiply a matrix of size $20,000 \times 20,000$ by a vector of size $20,000$:

```
$ make dgemv_seq
$ ./dgemv_seq 20000 20000
```

3. How long did it take to complete?
4. Which pattern do you think you can better represent the matrix-vector product in function `dgemv`?
5. Use the code `apps/dgemv/dgemv_grppi.cpp` as a template to implement a parallel version of the matrix-product in `dgemv` using a GrPPI pattern. The argument `exec` refers to the GrPPI execution model, and can be directly used as the first argument in the pattern call. As the matrix has been represented as a vector of vectors, you can directly use this pattern to iterate over the matrix rows to perform the `dot` operation.
6. Once implemented the parallel version, compile and run it. You can use first the sequential back end:

```
$ make dgemv_grppi
$ ./dgemv_grppi 20000 20000 seq 1
```

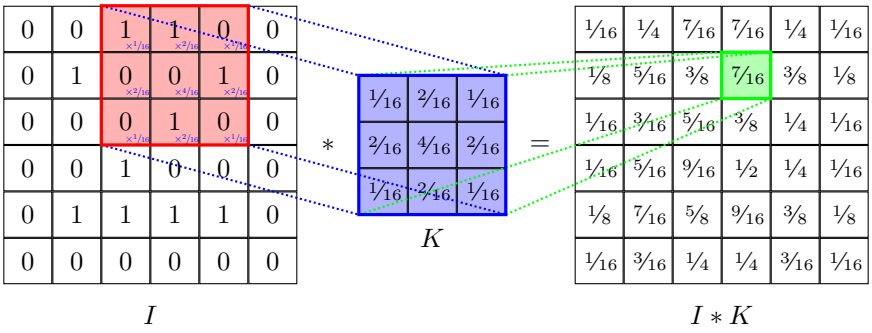
7. Did you observe any major overhead with respect to the naive sequential version?
8. Use a parallel back end, e.g. C++ threads, OpenMP or Intel TBB to run the application using 8 threads.

```
$ make dgemv_grppi
$ ./dgemv_grppi 20000 20000 thr 8
```

9. Do you think there is still room for parallelizing this algorithm? Focus on function `ddot` and think which pattern could represent the scalar product.
10. Use a GrPPI parallel pattern to re-implement the algorithm in function `ddot`.
11. Run again the application using the native (`thr`) back end. Could you speed up the execution time?
12. Check how the threads are created and destroyed in GrPPI each time a new entry in b is computed.
13. Study the best parallelization arrangement for this problem. Do you think that nested parallelism helps in this problem? If not, state why.

Exercise 3. In image processing, a Gaussian blur filter is the result of blurring an image by a Gaussian function. This effect is widely used in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen.

Mathematically, applying a blur filter to an image is the same as convolving the image with a Gaussian function or kernel. In this sense, the convolution in image processing is the process of multiplying each input image pixel and its neighbors by the a given kernel to obtain the output image. This operation is denoted by *. For example, if we have if we have the image I and the Gaussian blur 3×3 kernel K, the process would produce the filtered $I * K$ image:



where the pixel at coordinates [2, 4] is a weighted combination of the corresponding entries of I with the weights of K, i.e.,

$$I[2,4] * K = \big(1 \cdot 1 + 1 \cdot 2 + 0 \cdot 1 + 0 \cdot 2 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 + 1 \cdot 2 + 0 \cdot 1\big)\frac{1}{16} = \frac{7}{16}.$$

As you may see, the convolution is an embarrassingly parallel process, given that each pixel of the resulting image can be computed independently. In this exercise you are asked to parallelize in terms of parallel patterns the convolution for applying the Gaussian blur filter to input images. To do so, follow these steps and answer the questions:

1. Explore sequential naive code of the Gaussian blur filter in `apps/blur/blur_seq.cpp`.
2. Compile and run the application to blur the Lena image `lena.bmp` using a plain blur kernel of 3×3:

```
$ make blur_seq
$ ./blur_seq kernel_avg3.txt lena.bmp out.bmp
```

3. How long did it take to complete?
4. Which pattern do you think you can better represent the convolution operation in function `blur`?
5. Use the code `apps/blue/blur_grppi.cpp` as a template to implement a parallel version of the convolution operation in `blur` using a GrPPI pattern. The argument `exec` refers to the GrPPI execution model, and can be directly used as the first argument in the pattern call. Note that the image has been flattened in a vector. Also, given the BMP format, the image has three color bands (RGB) that have to be processed by the blur filter.

   A possible implementation of the neighborhood function could be to return a C++ vector of tuples, where each tuple in the vector represents a neighbor, containing the image pixel value and the corresponding kernel coordinate (`std::vector<std::tuple<unsigned char,int>>`). The counterpart kernel function may perform the multiplications between image pixels and kernel coordinates, sum up multiplication results and return the output pixel value.
6. Once implemented the parallel version, compile and run it. You can use to parallel back end, e.g. C++ threads to run the application in parallel using 8 threads:

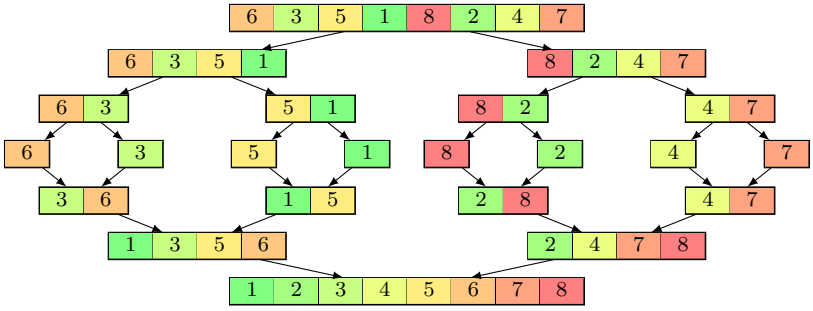
```
$ make blur_grppi
$ ./blur_grppi kernel_avg3.txt lena.bmp out.bmp thr 8
```

7. What was the speed up obtained with respect to the sequential version?
8. **Optional:** Modify the application to accept other types of kernels, e.g., for edge detection, for sharpening images, computing gradients, etc. See `https://en.wikipedia.org/wiki/Kernel_(image_processing)`.

Task-parallel patterns

Exercise 4. Mergesort is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945.

The top-down implementation of mergesort recursively splits an array A into two subarrays of until their size is 1. Afterwards, it merges those subarray to produce a sorted array.



Conceptually, the mergesort algorithm works as follows:

Divider Splits the array A of size n down the middle into two subarrays, each of size roughly $n/2$. This process ends when the subarray have been split down to a single item.

Solver Sorts the base case problem. An array of length one is trivially sorted.

Combiner Merges the two sorted subarrays into a single sorted one.

Given the behavior of this algorithm, where different subarrays in the process can be sorted independently, the sorting process can be parallelized. This exercise is intended to parallelize the mergesort algorithm in terms of GrPPI patterns. Follow these steps and answer the questions:

1. Explore sequential code in `apps/mergesort/mergesort_seq.cpp`.
2. Compile, run and check the execution time for sorting $2,000,000$ elements:

```
$ make mergesort_seq
$ ./mergesort_seq 2000000 no
```

3. Use the code `apps/mergesort/mergesort_grppi.cpp` as a template to implement a pattern version of the algorithm using a GrPPI pattern. As a first approach use the sequential (`parallel_execution_seq`) back end to run the algorithm.
4. Once implemented, compile and run the GrPPI version in parallel using 8 threads:

```
$ make mergesort_grppi
$ ./mergesort_grppi 2000000 no thr 8
```

5. What was the execution time difference with respect to the sequential version?
6. Set the number of threads in the native back end `thr` to the number of cores in the platform.
7. What was the speedup obtained with respect to the sequential version?
8. Change the parallel execution to `omp`. Did you observe any change in the execution time compared with the native back end? Try other back ends, e.g., `tbb`.

Stream parallel patterns

Exercise 5. As studied in Exercise 3, images of the Mandelbrot set exhibit an elaborate and infinitely complicated boundary that reveals progressively ever-finer recursive detail at increasing magnifications of the core function (remember the `zoom` variable). The "style" of this repeating detail depends on the region of the set being examined. The set's boundary also incorporates smaller versions of the main shape, so the fractal property of self-similarity applies to the entire set, and not just to its parts.

As stated, depending on the value of the `zoom` variable, finer details of the Mandelbrot set can be observed. Thus, we can generate an animation of the Mandelbrot images at increasing zoom values. This streaming application consists of the following stages:

Generator returns monotonically linear increasing zoom values, which are passed to the Mandelbrot stage.

Mandelbrot takes the zoom received by the generator stage and computes the Mandelbrot image corresponding to such zoom value and the coordinates of the point of interest.

Printer prints by the standard output the image computed by the mandelbrot stage.

Note that to ease the visualization, the pixels of the images generated are printed in ASCII, so they can easily be observed in a terminal.

Given the aforementioned stages, this streaming application can be parallelized by means of GrPPI patterns. Follow these steps and answer the questions:

1. Explore sequential naive code of the Mandelbrot video application in `apps/mandebrot_video/mandebrot_video_seq.cpp`.
2. Compile and run the application in sequential, change the terminal size to 132×43:

```
$ make mandebrot_video_seq
$ ./mandebrot_video_seq
```

3. Check the statistics on the top: instantaneous and averaged frames per second, frame number and elapsed time. How long did it take to process all the video frames?
4. Which pattern do you think can be used for this streaming application in order to compute the stages in parallel?
5. Use the code `apps/mandebrot_video/mandebrot_video_grppi.cpp` as a template to implement a parallel version of the `while` loop in function `mandelbrot` using a GrPPI pattern. Here again, the argument `exec` refers to the GrPPI execution model, and can be directly used as the first argument in the pattern call. Different to the data parallel Mandelbrot, this streaming application computes a single mandelbrot image in sequential.
6. Once implemented the parallel version, compile and run it on the same size. You can use the native back end:

```
$ make mandebrot_video_grppi
$ ./mandebrot_video_grppi thr
```

7. What was the speed up obtained with respect to the sequential version? How many threads were used? Were constant the instantaneous FPS during all the program execution? If not, state why.
8. Do you think that different mandelbrot frames can be computed in parallel? Which pattern could you use to speed up the throughput of the Mandelbrot stage? Implement a new version of the application using this pattern, and adapt the application to allow specifying the number of worker threads as a parameter.
9. Check the execution model variable `exec` as it can be further configured for streaming patterns. Do you need ordering in this case?
10. **Optional:** Modify the application to accept other points of interest or locations (see `http://www.cuug.ab.ca/dewara/mandelbrot/images.html`).