Handout : parallel simulation of the abelian sandpile model

Alice Lasserre - Raymond Namyst - Pierre-André Wacrenier

Université de Bordeaux

1 About EASYPAP

This assignment uses the EASYPAP [LNW21] framework which is an easy-to-use C programming environment designed to help students to learn HPC. EASYPAP integrates a graphic display window, a real-time activity monitor and an off-line trace analyzer that provide new ways of visualizing tasks together with their associated data. For instance, Figure 1 presents two execution traces of the `ssandPile` kernel over a 2048×2048 sparse configuration. The traces display tasks executed during the same 1000th iteration performed by a lazy OpenMP variant. Here 32×32 tiles are used for the top trace against 64×64 tiles for the bottom one. We can see that, in this case, using 32×32 tiles is more efficient than 64×64: indeed the computed area is much smaller.

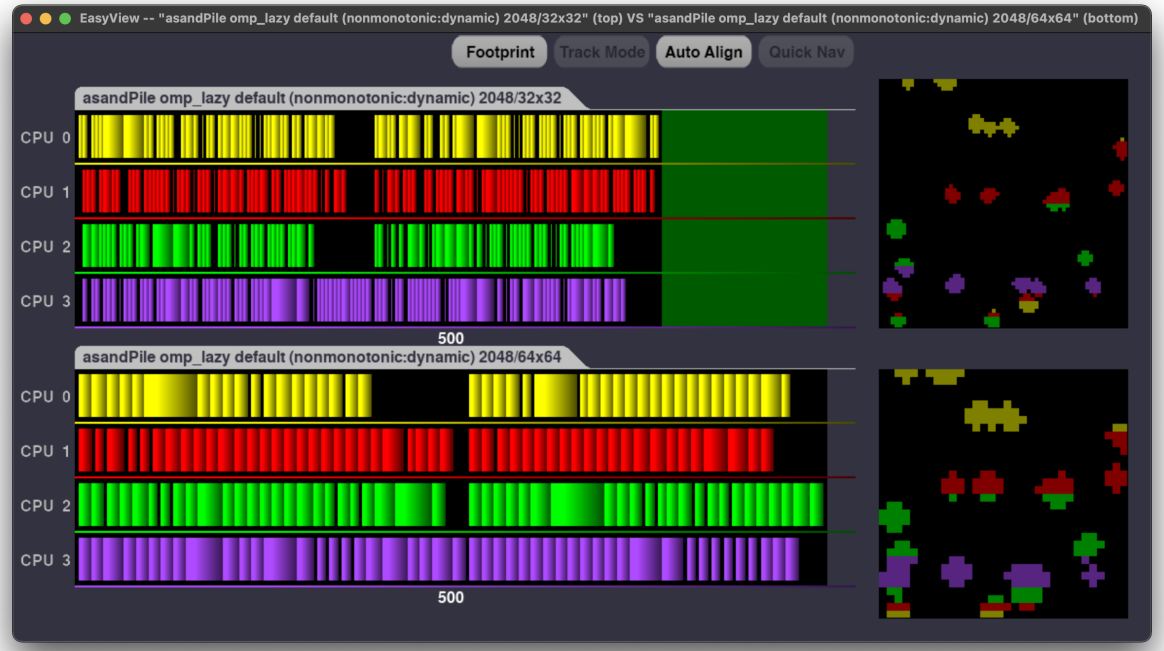

Figure 1: Comparison of two execution traces of the `ssandPile` kernel over a 2048×2048 sparse configuration. Gantt charts display per-CPU sequences of tasks where tiles computed by the same CPU have the same color. On the right side, corresponding tiles are highlighted over a reduced image.

The documentation of EASYPAP is available online: https://gforgeron.gitlab.io/easypap/doc/Getting_Started.pdf

The EASYPAP framework is regularly updated and is available on gitlab:

```
git clone https://gitlab.com/gforgeron/easypap-se.git
```

2 The abelian sandpile model

The Bak-Tang-Wiesenfeld' *Abelian Sandpile Model*[BTW88] is known as the first discovered example of a dynamical system displaying self-organized criticality. This kind of model allows scientists to question nature, from earthquakes to brain activity[1].

In the Abelian Sandpile Model [BTW88], the sandpile is a $N \times M$ 4-connected cellular automaton such that the border cells are connected to a special cell called *sink*. The state of a cell is an integer corresponding to its number of grains of sand. By definition a regular cell is said to be *stable* whenever it contains 3 or less grains, *unstable* otherwise. The rule of the automaton is the following: an unstable cell gives its grains to its 4 neighbors by distributing them equitably. For example, if a cell contains 11 grains, then it will give 2 to each neighbor and keep the remaining 3 grains. Starting from an initial unstable configuration, the simulation of a sandpile consists in applying iteratively the automaton rule until a stable configuration is reached, which indicates that all cells are stable.

Dhar [Dha90] proved that the reached stable configuration is unique regardless of the computation order; therefore the simulation of a sandpile allows a great flexibility for parallelization.

3 About the provided source code

The goal of the assignment is to efficiently compute the final stable configuration. For this, you are given the file `kernel/c/sandPile.c` that contains two kernels (Fig. 2):

- function `ssandPile_do_tile_default()` is a synchronous version of the sandPile's simulation where all cells are treated simultaneously thanks to the use of an auxiliary array.
- function `asandPile_do_tile_default()` is an asynchronous variant where sand slides are computed "in place" and have a direct impact on neighboring cells.

First, let us try these two kernels :

```
--> ./run -k ssandPile -s 32
--> ./run -k asandPile -s 32
```

It seems that the final images are the same, although we can check this thanks to the dump option of EASYPAP:

```
--> ./run -k ssandPile -s 32 -n -du
Using kernel [ssandPile], variant [seq], tiling [default]
Computation completed after 242 iterations
1.595
--> ./run -k asandPile -s 32 -n -du
Using kernel [asandPile], variant [seq], tiling [default]
Computation completed after 138 iterations
0.363
--> diff dump-ssandPile-seq-dim-32-iter-242.png dump-asandPile-seq-dim-32-iter-138.png
--> echo $?
0
```

Please, keep in mind to always check the correctness of your code before making performance measurements.

[1] https://www.quantamagazine.org/brains-may-teeter-near-their-tipping-point-20180614/

```
int ssandPile_do_tile_default(int x, int y, int width, int height)
{
  int diff = 0;

  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
    {
      table(out, i, j) = table(in, i, j) % 4;
      table(out, i, j) += table(in, i + 1, j) / 4;
      table(out, i, j) += table(in, i - 1, j) / 4;
      table(out, i, j) += table(in, i, j + 1) / 4;
      table(out, i, j) += table(in, i, j - 1) / 4;
      if (table(out, i, j) >= 4)
        diff = 1;
    }

  return diff;
}

int asandPile_do_tile_default(int x, int y, int width, int height)
{
  int change = 0;

  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      if (atable(i, j) >= 4)
      {
        atable(i, j - 1) += atable(i, j) / 4;
        atable(i, j + 1) += atable(i, j) / 4;
        atable(i - 1, j) += atable(i, j) / 4;
        atable(i + 1, j) += atable(i, j) / 4;
        atable(i, j) %= 4;
        change = 1;
      }
  return change;
}
```

Figure 2: Synchronous and asynchronous variants of kernel `sandPile`.

4 Assignments

4.1 ILP optimization

The given functions `asandPile_do_tile_default()` and `ssandPile_do_tile_default()` are not optimized. The idea is to simplify their code so that we may rely on compiler auto-vectorization. For this, define two functions `asandPile_do_tile_opt()` and `ssandPile_do_tile_opt()`. Check the correctness and then the performance gain obtained on the size 512 case:

```
--> ./run -k asandPile -wt opt -s 512 -n
--> ./run -k ssandPile -wt opt -s 512 -n
```

Note, that the C keyword **`restrict`** should be useful.

4.2 OpenMP implementation of the synchronous version

First, write an OpenMP version `ssandPile_compute_omp()` that uses neither tiles nor collapse clause.
Use the `schedule(runtime)` clause in order to easily make multiple experiments. After having verified the correctness of the code, produce a speed-up graph. For this, you can refer to the script `plots/run-xp-mandel.py` to produce a CSV file `ssand-omp.csv` and then use `EasyPlot`[2] to produce the graph:

```
--> plots/run-xp-ssand-omp.py       # it's up to you to write this script !
--> plots/easyplot.py --delete iterations -if ssand-omp.csv
--> evince plot.pdf                 # use your favorite viewer
```

Then write a function `ssandPile_compute_omp_tiled()` using the `collapse(2)` clause to distribute the tiles to the threads. Compare the performances according to the geometry of the tiles with the size 512 case. To conduct this experiment you can refer to the script `plots/run-xp-heat-mandel.py`. You can produce a heat map thanks to the command :

```
--> plots/run-xp-heat-ssand.py
--> plots/easyplot.py --delete iterations -if heat-ssand.csv --plottype heatmap \
                  -heatx tilew -heaty tileh -v omp_tiled -C schedule
```

You can also write a function `ssandPile_compute_omp_taskloop()` that use ... tasks and compare its performance to other variants.

4.3 OpenMP implementation of the asynchronous version

Write a function `asandPile_compute_omp()`. You have to take into account that two neighboring tiles cannot be processed simultaneously by two different threads (unless access to shared cells is protected). It is also necessary to avoid unnecessary synchronizations. There are two main ways to do this:

[2] A user guide (in French) for `EasyPlot` is available under https://gforgeron.gitlab.io/pap/td/easyplot/Sujet/tuto.pdf

- either color the tiles so that two neighboring tiles have different colors and then process the tiles of the same color in parallel;
- or use atomic instructions / mutexes to protect shared cells.

Then check the correctness of your code - in this aim we advise that you use small tiles on small configurations:

```
--> OMP_SCHEDULE=dynamic ./run -k asandPile -v omp -s 64 -ts 4 -n -du
```

Then you can compare the performance of all variants on the size 512 case.

4.4 Lazy OpenMP implementations

First look at the following sparse simulation:

```
--> ./run -k ssandPile -wt opt -s 512 -a alea -m
```

The idea is to develop a lazy evaluation algorithm that avoids computing tiles whose neighborhood was in a steady state at the previous iteration (see Fig. 1). Write two sequential functions `asandPile_compute_lazy()` and `ssandPile_compute_lazy()` which implement the lazy evaluation. Note that you can look at the tiling window to make sure that areas where "nothing changes" are not computed.

Then write their OpenMP variants: `asandPile_compute_omp_lazy()` and `ssandPile_compute_omp_lazy()`.

To address the challenge of mitigating the load imbalance introduced by "sparse" configurations, you have to experiment with various scheduling policies and various tile sizes.

Produce a heatmap with the size 2048 case in order to compare the performances of the two lazy OpenMP variants according to the geometry of the tiles.

4.5 AVX implementation

The goal of this assignment is to define a function `ssandPile_do_tile_avx()` that is compatible with the lazy implementations.

Cells located on borders need special attention, because they have no neighbors. As a consequence, explicit vectorization of border tiles is not trivial. In a first approach, we advise that you treat border tiles and inner tiles differently, using a suboptimal implementation for border tiles. Then process a border tile, you can use `ssandPile_do_tile_opt()` to process *thin* border sub-tiles and the AVX for the inner sub-tiles. Note that it is possible to make a SIMD code that is suitable for both inner and outer tiles.

4.5.1 The asynchronous case

The SIMDization of the asynchronous version is more challenging. Remark that it is not possible to faithfully vectorize the given asynchronous algorithm. The idea is to design an algorithm that is locally synchronous but globally asynchronous. Let $\vec{X}$ denotes the vector $(x_k, \ldots, x_0)$ and t_{ji} a cell of the sandpile, we will use the following notations:

$$\vec{X}[i] = x_i$$
$$\vec{X} \bmod p = (x_k \bmod p, \ldots, x_0 \bmod p)$$
$$\vec{X} \text{ div } p = (x_k \text{ div } p, \ldots, x_0 \text{ div } p)$$
$$\vec{X} >> 1 = (0, x_k, \ldots, x_1)$$
$$\vec{X} << 1 = (x_{k-1}, \ldots, x_0, 0)$$

The division by 4 and modulo 4 computations can be performed using `*_srli_*` and `*_and_*` intrinsics. The shift of the vector elements is performed with `*_alignr_*` intrinsic (AVX512).

First load the vectors:
$\vec{T}_{j-1,i} \longleftarrow (t_{j-1,i+k}, \ldots, t_{j-1,i})$
$\vec{T}_{j,i} \longleftarrow (t_{j,i+k}, \ldots, t_{j,i})$
$\vec{T}_{j+1,i} \longleftarrow (t_{j+1,i+k}, \ldots, t_{j+1,i})$

Then let the sand fall synchronously:
$\vec{D} \longleftarrow \vec{T}_{j,i} \text{ div } 4$
$\vec{T}_{j,i} \longleftarrow \vec{T}_{j,i} \bmod 4 + (\vec{D} << 1) + (\vec{D} >> 1)$
$\vec{T}_{j-1,i} \longleftarrow \vec{T}_{j-1,i} + \vec{D}$
$\vec{T}_{j+1,i} \longleftarrow \vec{T}_{j+1,i} + \vec{D}$

Finally store the results :
$t_{j,i-1} \longleftarrow t_{j,i-1} + D[0]$
$t_{j,i+k+1} \longleftarrow t_{j,i+k+1} + D[k]$
$(t_{j-1,i+k}, \ldots, t_{j-1,i}) \longleftarrow \vec{T}_{j-1,i}$
$(t_{j,i+k}, \ldots, t_{j,i}) \longleftarrow \vec{T}_{j,i}$
$(t_{j+1,i+k}, \ldots, t_{j+1,i}) \longleftarrow \vec{T}_{j+1,i}$

Note that to achieve good performance it should be necessary to factorize the vectors loads and stores. Therefore we advise to adapt this algorithm to the processing of a sub-tile (ie. a column of vectors): it consists in storing the vectors in an array and using a second array of vectors to accumulate the grains that fall on the left and right sides of the sub-tile. Finally, remark that it may be interesting to perform the computation phase several times.

4.6 MPI Implementation

For this assignment you can refer to the `spin` kernel's code and to the documentation.

Here we ask you to implement the *Ghost Cell Pattern*[KS10]: every iteration have each pair of neighboring processes exchange a copy of their borders. However, the communication overheads are such that you have to develop a solution that trades redundant computation for less frequent communication.

It is interesting to produce an MPI variant for the asynchronous kernel. Indeed, the advantage of this kernel is that we can trade redondant computation for less frequent communication in a particular way. In addition to the traditional ghost zone we can add an *overflow zone*. This overflow zone have to be empty after each communication step. During the computation phase, the whole domain (overflow zone + ghost zone + process's domain) is computed. At each communication step, ghost zones have to be exchange and the grains that fell in overflow zone must be *added* to the corresponding zone of the neighbor.

4.7 OpenCL Implementation

The goal of this assignment is to produce an OpenCL variant for the synchronous case. A skeleton of the OpenCL kernel (`kernel/ocl/ssandPile.cl`) is provided. Please note that the `ssandPile_update_texture` kernel should not be modified: it is used by EASYPAP to refresh a SDL texture on the GPU before it is displayed on the screen.

There's nothing special to write on the C side at this point: by default, the kernel will be invoked with two parameters (*in* and *out* buffers) and will be executed by a set of `DIM×DIM` workitems.

Juste complete the code of the `ssandPile_ocl` function and test it:

```
./run -k ssandPile -o -i 69191
```

We'll address the problem of detecting termination later.

4.7.1 Generating dumps

In case you need to generate dump files to check the correctness of your code, make sure that the following *refresh* function is defined in your `sandPile.c`.

```
1  // Only called when --dump or --thumbnails is used
2  void ssandPile_refresh_img_ocl ()
3  {
4    cl_int err;
5
6    err = clEnqueueReadBuffer (queue, cur_buffer, CL_TRUE, 0,
7                               sizeof (unsigned) * DIM * DIM, TABLE, 0, NULL, NULL);
8    check (err, "Failed to read buffer from GPU");
9
10   ssandPile_refresh_img ();
11 }
```

The following command should provide you with a nice PNG file:

```
./run -k ssandPile -o -i 69191 -du
```

4.7.2 Detecting termination

You will need to allocate (at least) one additional buffer on the GPU, and transfer the content of this buffer from the GPU to the RAM between iterations, so that the CPU side can detect when to stop. The `scrollup.c` file contains examples illustrating how to allocated additional buffers.

Because these transfers are costly, please set up a mechanism enabling to decrease the frequency at which the transfers are performed.

4.7.3 OpenCL + OpenMP Implementation

One of the goal of this assignment is to implement border transfers between the CPU and the GPU. We advise that you implement a first version where the transfers consist in sending only one pixel line to the CPU and one to the GPU. However these transfers are costly and it is worth to transfer several pixel lines at a time. The technic is, as in the MPI case, to trade some redundant computations for less frequent communications.

This assignment also focuses on the load balancing problem. A version where the distribution adapts to the performance of the computing units instead of fixing an arbitrary division can therefore be implemented.

References

[BTW88] Per Bak, Chao Tang, and Kurt Wiesenfeld. Self-organized criticality. *Physical Review A*, 38:364–374, 07 1988.

[Dha90] Deepak Dhar. Self-organized critical state of sandpile automaton models. *Phys. Rev. Lett.*, 64:1613–1616, Apr 1990.

[KS10] Fredrik Berg Kjolstad and Marc Snir. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, New York, NY, USA, 2010. Association for Computing Machinery.

[LNW21] Alice Lasserre, Raymond Namyst, and Pierre-André Wacrenier. Easypap: A framework for learning parallel programming. *J. Parallel Distributed Comput.*, 158:94–114, 2021.