Parallel Computing - Lab 3: OpenMP

- Benjamín Ocampo: nicolasbenjaminocampo@gmail.com
- Mateo de Mayo: mateodemayo@gmail.com

Changes from E5-2620v3 to E5-2680v4

Before adding a line of code we needed to re-measure our previous results since they will not be comparable with future versions of the project due to the change of architecture from Haswell to Broadwell in zx81 and jupiterace. We also had to corroborate that our best approaches were still obtaining similar outcomes. It was also important to decide if our best approach, which was called *Stream*, would be our *baseline* during this lab. Therefore, it was compared along with *Shload* and our basic vectorization in linsolve. Remember that the three of them contain all the improvements in *advect* and *project* pointed out in the previous laboratory. We got the following conclusions:

- Intel Optimization Manual (2.4.7) and Agner Fog Optimization Manual 3 (10.14):
- Gathers have been improved (./vectortest now tells gathers are better than single loads) (14.16.4)
- `fpmul` from 5 to 3 cycles
- `PCLMULQDQ` is one cycle
- Also
- `fload2x4` is notably worse (~10ns) than `fload8` as expected
- `shload` is now not worth it, it is better just directly read from memory
- `stream` is now slower. (Note that reading and writing the same memory location is now **really** bad, and in haswell it seemed to be *"free"*)

We were taken by surprise when stream was worse for the smallest cases (N = 128, 512, and 2048) but considerably better for the largest ones (N = 4096 and 8192) *Shload* was also worse than without it. And the code without these approaches was the best one for the smaller cases but one of the worst for the largest ones. That was totally confusing. Which of them was the fastest one? Which could be used during the lab 3?

We decided to remove our tricks and aces up to the sleeve, i.e, working without *Shload* and *Stream* but organizing the code in such a way that works with them in order to choose the fastest one at the end of the laboratory.

Tidying up

In order to provide a parallelization which works for the three approaches mentioned above implemented in *intrinsics* and *ispc*(except for of *Stream* which

has not been done in *ispc* yet), a code migration was performed. The file *solver.c* was changed in such a way that *advect*, *project* and *linsolve* were implemented by means of functions that share the same interfaces in intrinsics and ispc. Therefore, it is possible to compile either intrinsics or ispc code by means of Makefile rules. A non-vectorized version was also included as an option if its use is required.

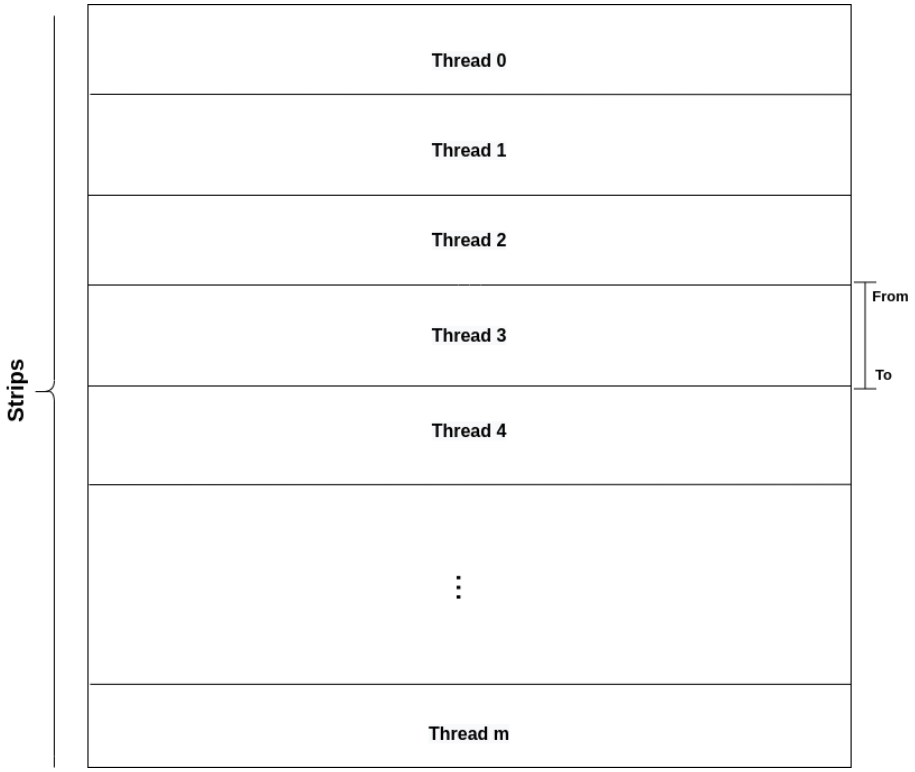
Other things that changed were the functions *dens_step* and *vel_step*. Remember that another version of *advect* (called *vel_advect*) was implemented in the previous lab. Even though this approach increases the performance updating both **u** and **v** instead of doing it separately. It decreases the reusability and mantainability of code. In order to keep just one function that performs the advection, *dens_step* and *vel_step* were merged in just one function called *step*. But now, *advect* will not only update **u** and **v**, but also **d** which stores the density values. We will discover later that these changes will be useful for minimizing the number of omp barriers. Here is the code of *step*.

```
void step(unsigned int n, float *d, float *u, float *v, float *d0, float *u0,
          float *v0, float diff, float visc, float dt) {
  // Density update
  add_source(n, d, d0, dt);
  SWAP(d0, d);
  diffuse(n, NONE, d, d0, diff, dt);
  SWAP(d0, d);
  // density advection will be done afterwards mixed with the velocity advection

  // Velocity update
  add_source(n, u, u0, dt);
  add_source(n, v, v0, dt);
  SWAP(u0, u);
  diffuse(n, VERTICAL, u, u0, visc, dt);
  SWAP(v0, v);
  diffuse(n, HORIZONTAL, v, v0, visc, dt);
  project(n, u, v, u0, v0);
  SWAP(u0, u);
  SWAP(v0, v);
  advect(n, d, u, v, d0, u0, v0, dt);
  project(n, u, v, u0, v0);
}
```

Thinking in strips

In order to divide equally the work of updating the grid (space where the algorithm is running) among several threads, we needed to find the best possible share-out. We found out that a strip-division, i.e, a set of rows of the same

length for each thread it is the one that feets better. So, each thread receives a strip of size ceil(N/threads). The last thread will receive the remaining rows if the number of threads does not divide N.



From-To division

Since we wanted a general implementation which works with intrinsics and ispc (That does not allow the use of omp pragmas), two new parameters were included in the function *step*. The first argument is *from*, that indicates from which row a thread will start updating the grid, and the second one is *to*, that says up to which row will do it. After that, all the functions that are called in the definition of step were changed to the from-to format, that includes *add_source*, *advect*, *project*, *linsolve* (involving the functions implemented in intrinsics and ispc), and *set_bnd*.

Another possibility of work distribution was a square-division of the grid. But in that case, we needed to find w and h (the width and height of the square respectively) in such a way that each thread receives one of the squares. A simple test in linsolve selecting the "best" w and h was performed but unfortunately, it gives not as good results as strip-division. Obtaining these two values could also

be messy and not as straightforward. That is why it was not implemented in this way.

OpenMP

Remember that in zx81 and jupiterace its topology consist of two NUMA nodes, where each node consists of 14 cores. All of them sharing an L3 cache of 35MB. Finally, each core consists of an L2, L1d, and L1i of 256KB, 32KB, and 32KB respectively.

Given a certain number of threads, we needed to think about a placement which maximizes the *affinity* of both nodes. I.e, placing in a way that the latency for a memory accesses of a certain core is minimum.

It was not so hard to do so, just indicating omp directives before running that threads had to be allocated close together, belonging to the same NUMA node (whenever it is possible).

Memory allocation was performed in parallel sections in order to reduce the necessity of accessing memory allocated in another NUMA node (just in border cases). Since malloc allocates memory just when it is written and the function *clear_data* is the first one that initializes arrays u, v, and d we parallelized it.

```
static void clear_data(void) {
  int i, size = (N + 2) * (N + 2);

  #pragma omp parallel for
  for (i = 0; i < size; i++) {
    u[i] = v[i] = u_prev[i] = v_prev[i] = dens[i] = dens_prev[i] = 0.0f;
  }
}
```

React

One main parallel section was in react. Since we needed to compute two maximum values, a reduction over a parallel for was used.

```
  float max_velocity2 = 0.0f;
  float max_density = 0.0f;

  #pragma omp parallel for default(none) private(i) firstprivate(size, uu, vv, d) reduction(
  for (i = 0; i < size; i++) {
    if (max_velocity2 < uu[i] * uu[i] + vv[i] * vv[i]) {
      max_velocity2 = uu[i] * uu[i] + vv[i] * vv[i];
    }
    if (max_density < d[i]) {
```

```
      max_density = d[i];
    }
  }
```

Finally, since these values are used to update velocity and density, a parallel for collapse directive was applied in these cases.

```
  if (max_velocity2 < 0.0000005f) {
    uu[IX(N / 2, N / 2)] = force * 10.0f;
    vv[IX(N / 2, N / 2)] = force * 10.0f;
    #pragma omp parallel for collapse(2)
    for (int y = 64; y < N; y += 64)
      for (int x = 64; x < N; x += 64) {
        uu[IX(x, y)] = force * 1000.0f * (N / 2 - y) / (N / 2);
        vv[IX(x, y)] = force * 1000.0f * (N / 2 - x) / (N / 2);
      }
  }
  if (max_density < 1.0f) {
    d[IX(N / 2, N / 2)] = source * 10.0f;
    #pragma omp parallel for collapse(2)
    for (int y = 64; y < N; y += 64)
      for (int x = 64; x < N; x += 64) d[IX(x, y)] = source * 1000.0f;
  }
```

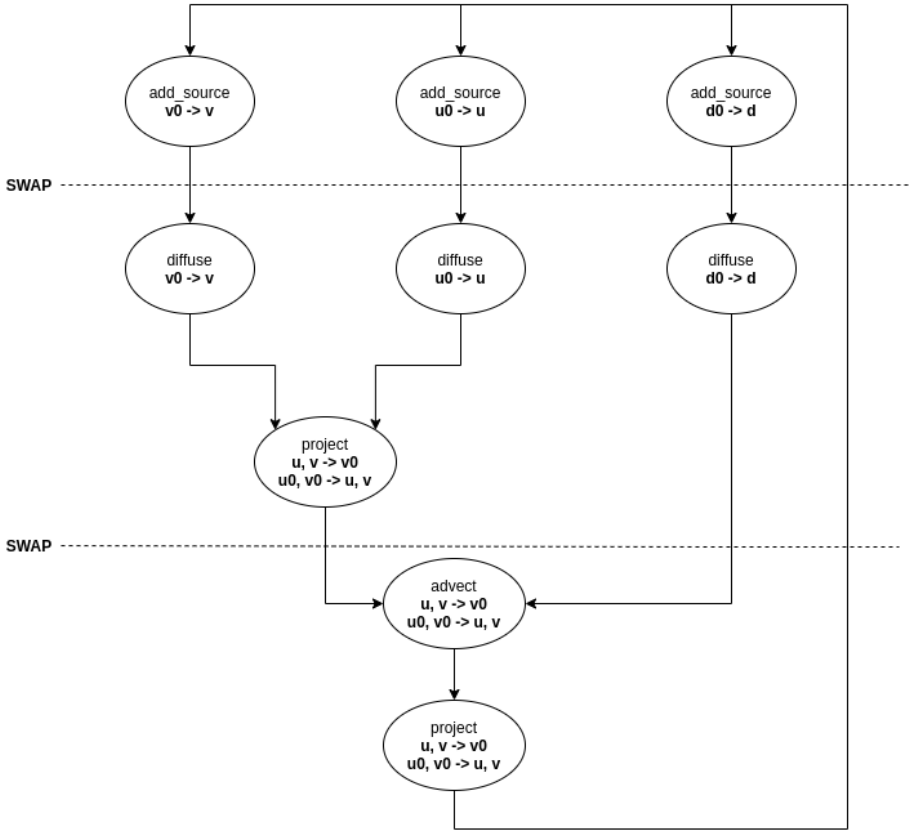
Step

In order to stay in parallel sections most of the time, the other main region is placed in *step* as we said before, so if we have m threads, they will be distributed along the entire grid, computing the function step from a certain region. Here is the code of the work distribution performed at calling the function step.

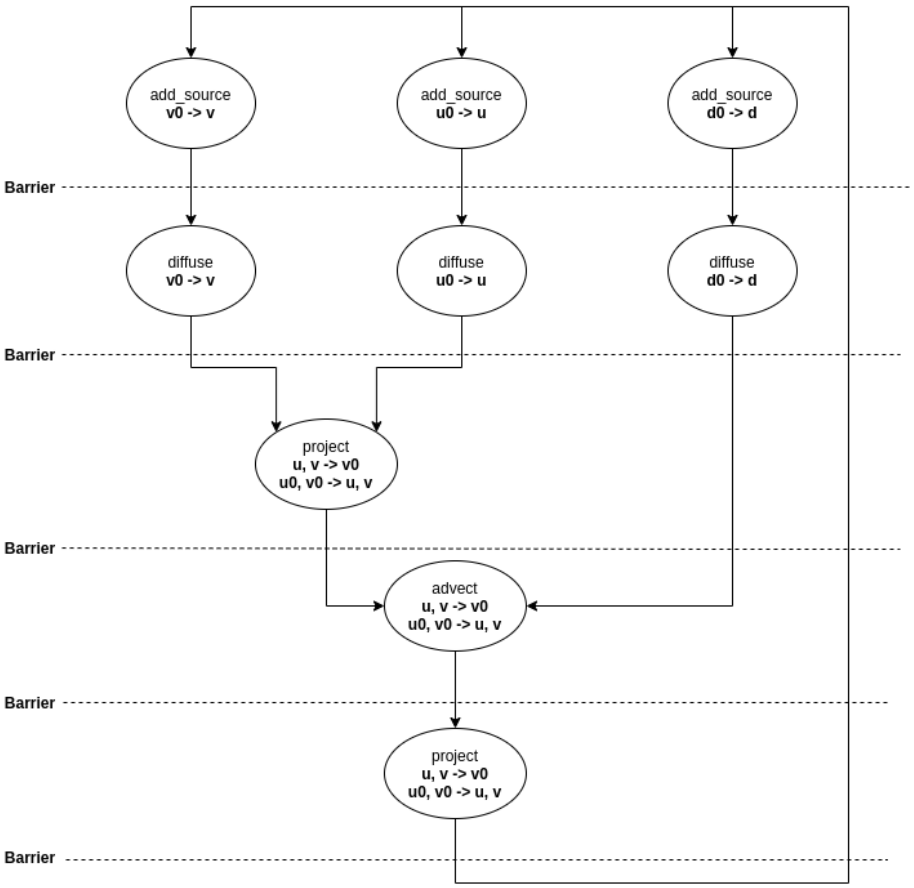
```
#pragma omp parallel firstprivate(dens, u, v, dens_prev, u_prev, v_prev, diff, visc, dt)
  {
    int threads = omp_get_num_threads();
    int strip_size = (N + threads - 1) / threads;
    #pragma omp for
    for(int tid = 0; tid < threads; tid++){
      int from = tid * strip_size + 1;
      int to = MIN((tid + 1) * strip_size + 1, N + 1);
      step(N, dens, u, v, dens_prev, u_prev, v_prev, diff, visc, dt, from, to);
    }
  }
```

Now, each thread has an strip of the grid to update. Nevertheless, some of the functions defined in step depends on certain strips assigned to a different thread, especially in the upper and lower borders (since it is an stencil problem).

So barriers should be added to avoid a race condition. Here is a graph of dependencies between the procedures that performs step.



As we can see, the three add_source can be executed without any problem. Then, diffuse would have a race condition if one thread needs information of a neighbour strip. If it needs so, add_source should had been done. So one barrier must be place after add_source. Applying a similar analysis, we found out where would be necessary synchronization.




```
void step(unsigned int n, float *d, float *u, float *v, float *d0,
          float *u0, float *v0, float diff, float visc, float dt,
          unsigned int from, unsigned int to) {
  add_source(n, d, d0, dt, from, to);
  add_source(n, u, u0, dt, from, to);
  add_source(n, v, v0, dt, from, to);
  #pragma omp barrier

  SWAP(d0, d);
  SWAP(u0, u);
  SWAP(v0, v);
  diffuse(n, NONE, d, d0, diff, dt, from, to);
  diffuse(n, VERTICAL, u, u0, visc, dt, from, to);
  diffuse(n, HORIZONTAL, v, v0, visc, dt, from, to);
  #pragma omp barrier

  project(n, u, v, u0, v0, from, to);
```

```
  #pragma omp barrier

  SWAP(d0, d);
  SWAP(u0, u);
  SWAP(v0, v);
  advect(n, d, u, v, d0, u0, v0, dt, from, to);
  #pragma omp barrier

  project(n, u, v, u0, v0, from, to);
  #pragma omp barrier
}
```

linsolve

In the case of linsolve, 20 iterations of gauss-seidel is performed. It would reasonable that a barrier should be necessary between a call of both lin_solve_rb_step (again for boundary dependencies of neighbour strips). Fortunately, the simulation seems to work fine without it. We could not get rid of the barrier before a set_bnd giving rise to a sincronization every iteration. It also leads to threads with a distance of just one iteration (which improves correctness).

```
static void lin_solve(unsigned int n, boundary b, float *restrict x,
                      const float *restrict x0, const float a, const float c,
                      const unsigned int from, const unsigned int to) {
  unsigned int color_size = (n + 2) * ((n + 2) / 2);
  const float *red0 = x0;
  const float *blk0 = x0 + color_size;
  float *red = x;
  float *blk = x + color_size;

  for (unsigned int k = 0; k < 20; ++k) {
      lin_solve_rb_step(RED, n, a, c, red0, blk, red, from, to);
      lin_solve_rb_step(BLACK, n, a, c, blk0, red, blk, from, to);
      #pragma omp barrier
      set_bnd(n, b, x, from, to);
  }
}
```

advect and project

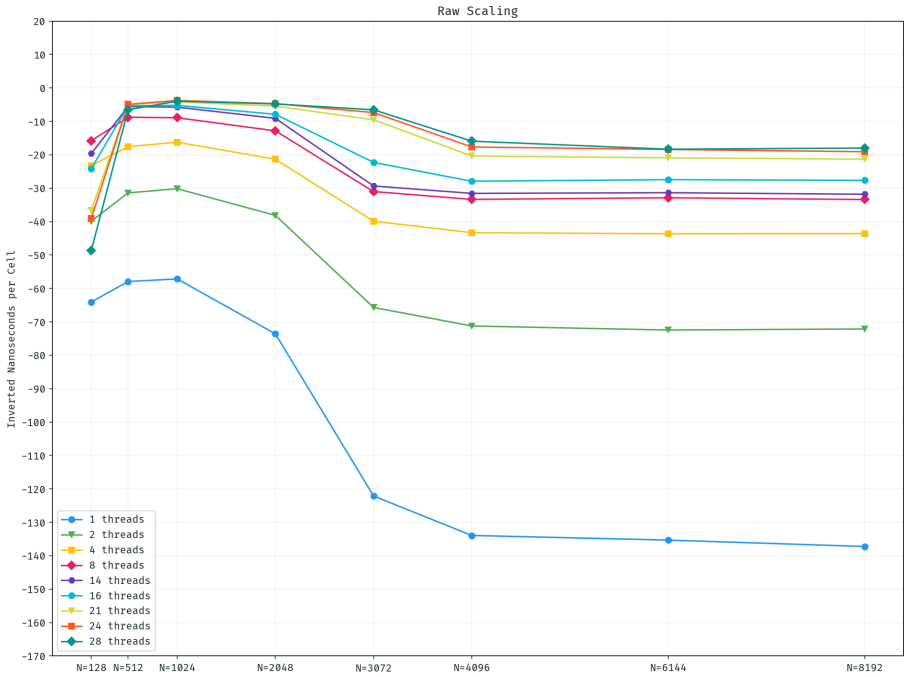
In these functions we used symilar analyisis, keeping in mind dependencies. So, barrier-placement was not so hard to deal with.

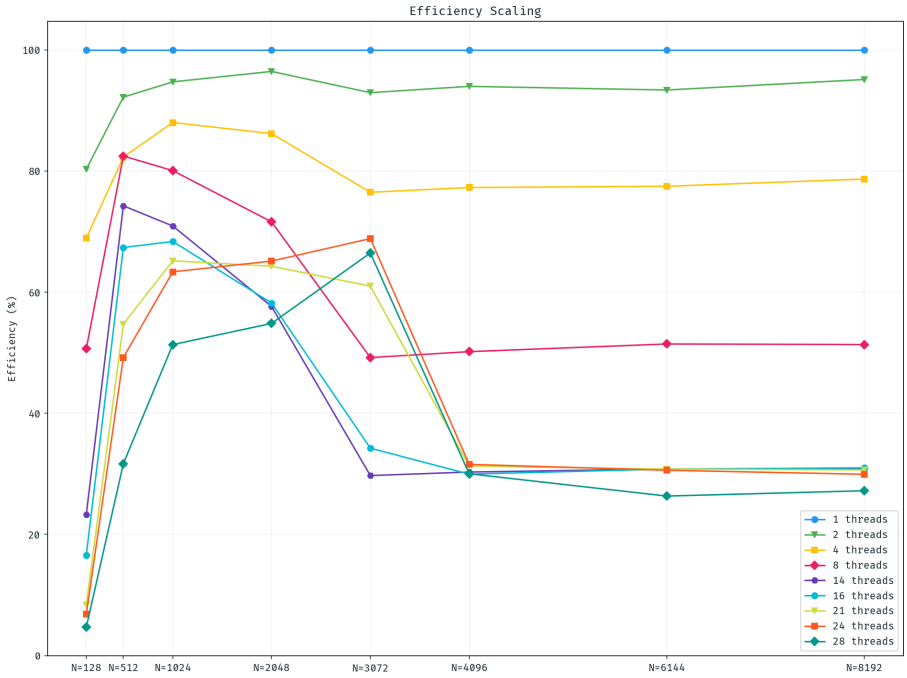
```
static void advect(unsigned int n, float *d, float *u, float *v,
                   const float *d0, const float *u0, const float *v0, float dt,
```

```
                    const unsigned int from, const unsigned int to) {

  ...  // Declaration of variables.

  advect_rb(RED, n, redd, redu, redv, redd0, redu0, redv0, d0, u0, v0, dt, from, to);
  advect_rb(BLACK, n, blkd, blku, blkv, blkd0, blku0, blkv0, d0, u0, v0, dt, from, to);
  #pragma omp barrier
  set_bnd(n, VERTICAL, u, from, to);
  set_bnd(n, HORIZONTAL, v, from, to);
}

static void project(unsigned int n, float *u, float *v, float *u0, float *v0,
                    unsigned int from, unsigned int to) {

  ... // Declaration of variables

  project_rb_step1(n, RED, redu0, redv0, blku, blkv, from, to);
  project_rb_step1(n, BLACK, blku0, blkv0, redu, redv, from, to);
  #pragma omp barrier

  set_bnd(n, NONE, v0, from, to);
  set_bnd(n, NONE, u0, from, to);
  #pragma omp barrier

  lin_solve(n, NONE, u0, v0, 1, 4, from, to);
  #pragma omp barrier

  project_rb_step2(n, RED, redu, redv, blku0, from, to);
  project_rb_step2(n, BLACK, blku, blkv, redu0, from, to);
  #pragma omp barrier

  set_bnd(n, VERTICAL, u, from, to);
  set_bnd(n, HORIZONTAL, v, from, to);
}
```

Scaling

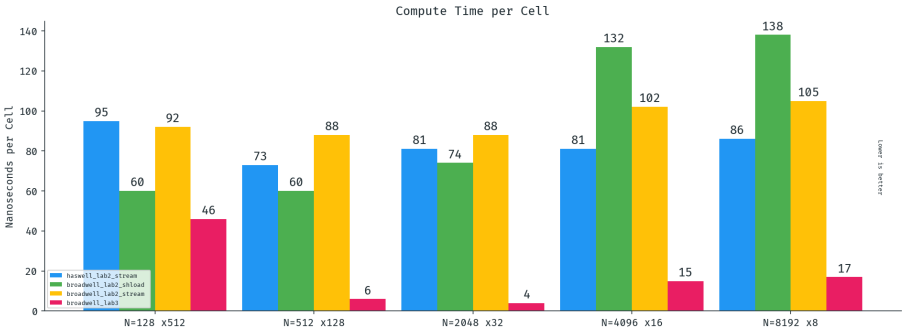
After all the migration, we measured the outcome of the program associated with the number of threads that were used. As long as we increase the number of threads, more performance we get, but less efficient we are. No matter how much threads we add, results tend to be enclosed by a certain threshold.

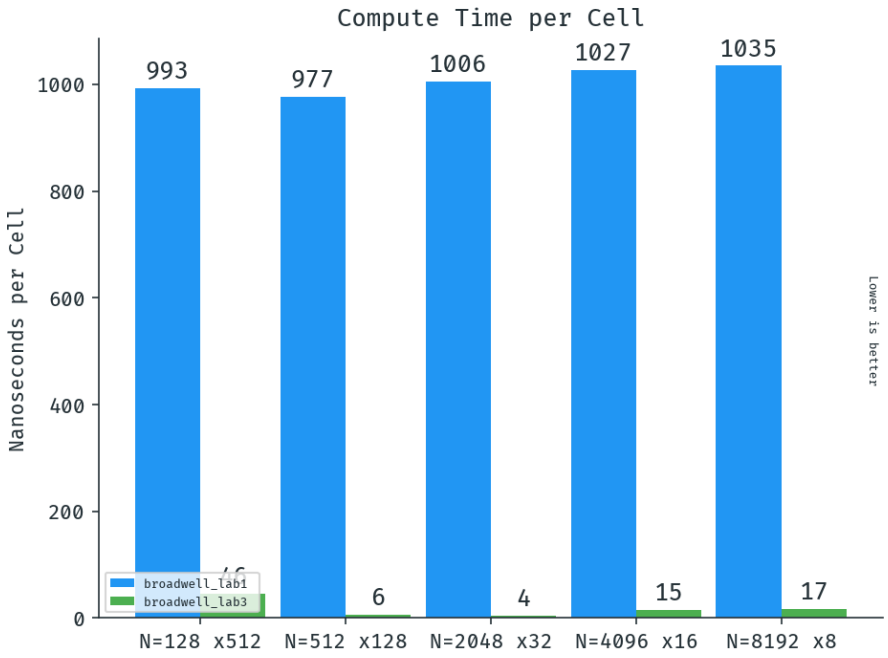

Raw Scaling
Inverted Nanoseconds per Cell
20
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
-100
-110
-120
-130
-140
-150
-160
-170
N=128
N=512
N=1024
N=2048
N=3072
N=4096
N=6144
N=8192
1 threads
2 threads
4 threads
8 threads
14 threads
16 threads
21 threads
24 threads
28 threads


Efficiency Scaling
Efficiency (%)
100
80
60
40
20
0
N=128
N=512
N=1024
N=2048
N=3072
N=4096
N=6144
N=8192
1 threads
2 threads
4 threads
8 threads
14 threads
16 threads
21 threads
24 threads
28 threads

Final Results

Finally, this implementations were compared along the best results obtained in lab2 and lab3 in order to see how much we have improved.





References

- Blaise, B. (2020) "OpenMP Tutorial" in *Lawrence Livermore National Laboratory*. May. 18, 2020. Available in https://computing.llnl.gov/tutorials/openMP/

- Yliluoma, J. (2007) “Guide into OpenMP: Easy multithreading programming for C++”. Available in https://bisqwit.iki.fi/story/howto/openmp/
- Lameter, C. (2013) “NUMA (Non-Uniform Memory Access): An Overview”. in *acmqueue*. August 9, 2013. Available in https://queue.acm.org/detail.cfm?id=2513149