Exercise on Parallelization

Caveats: what I suggest here is my way to proceed, but I'm far to be an expert of parallelization!

So, of course, it is possible to better...

When we want to parallelize

- Reduction of the wall-time: we want to achieve better performance, defined as (results response/execution) times
- Memory problem: large data sample, so we want to split in different sub-samples
- Remember the two strategies:
  - SPMD: Same program, different data
  - MIMD: Different programs, different data

Typical problem suitable for parallelization

- The problem can be broken down into subparts:
  - Each subpart is independent of the others
  - No communication is required, except to split up the problem and combine the final results
  - Ex: Monte-Carlo simulations
- Regular and Synchronous Problems:
  - Same instruction set (regular algorithm) applied to all data
  - Synchronous communication (or close to): each processor finishes its task at the same time
  - Local (neighbor to neighbor) and collective (combine final results) communication
  - Ex: Algebra (matrix-vector products), Fast Fourier transforms

Before parallelization

- Parallelization is not the first solution when your program is slow
  - Look if you can improve the performance improving the code
  - Sometimes compiler optimizations can make the difference (and they do the work for you!)
  - Try to understand if there are better implementations of your problem (see our example)
  - Do not re-invent the wheel
    - Use parallel libraries and look if there are already similar parallel implementations
  - Remember that parallel implementations are more difficult to debug than serial ones

Parallelization Suggestions

- General Law: THINK PARALLLEL!
  - Start to write your program directly thinking parallel implementations
    - Can be challenging for beginners
  - Write a serial version of the code and then move to parallelization
    - Good for beginners (use serial as reference)
    - Anyway it is wasting time! It can be not so straightforward to move from a serial to a parallel implementation of the code
- So, again, THINK PARALLLEL

Real-life case

- Usually we start to think in parallel when our serial implementations are too slow (or in general we want to achieve better results response/execution times)
  - In this case we start with a serial implementation (or in general we "inherit" the code from previous users)
  - Worst situation: sometimes it can be useful to write the code from scratch (when convenient)
    - Many complex serial code implementations are strictly serial, very difficult to parallelize (no thread-safe, complex data structure,…)

More practical suggestions (1)

- Either if you are writing a new program or you have a serial implementation:
  - Understand which part of the code is useful to parallelize
  - Understand your data structure
    - Which data you want to share, which data are private
  - Consider the communications and synchronizations
    - Keep low the communication-time/calculation-time
  - Start with a simpler parallel implementation, for example reducing the data structure
  - Each parallel implementation MUST have the possibility to run in serial (a single process)
    - Make sure that when run in parallel it gives the same results

More practical suggestions (2)

- Remember to balance the load between the processes
  - Final time is given by the slowest process!
- Scalability:
  - Depends on your problem, usually on data decomposition
    - Ex. a parallelization of a simulation of 10 particles, you can have a limit of 10 processors
- Speed-up:
  - Do not expect to run a program of 1 week in 1 second!
  - Remember the Amdahl's Law:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

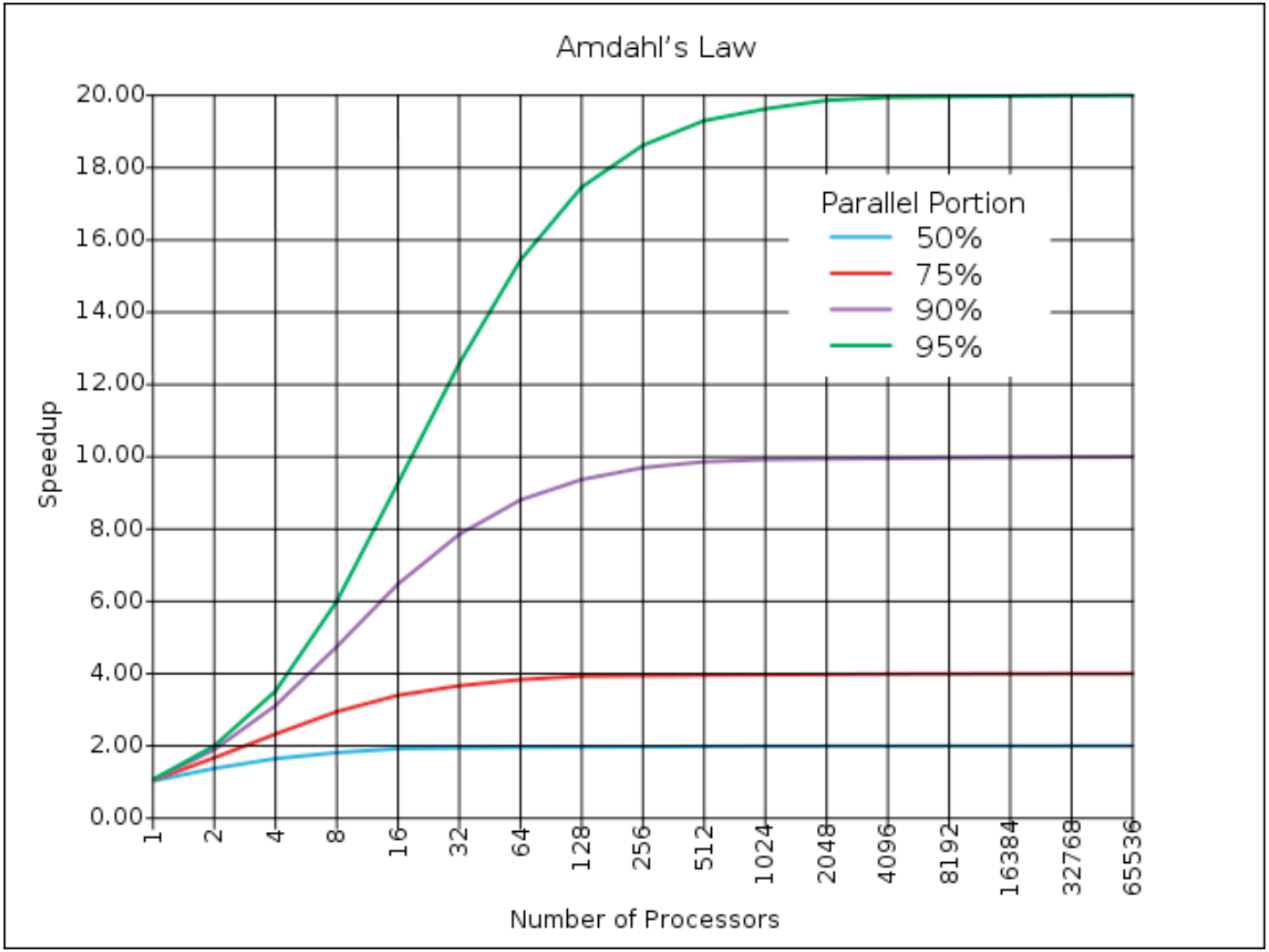
    S → speedup

    P → portion of code which is parallelized

    N → number of simultaneous process
  - Need to find good algorithms to be parallelized!

$$S(N \to \infty) = \frac{1}{(1-P)}$$

Amdahl's Law

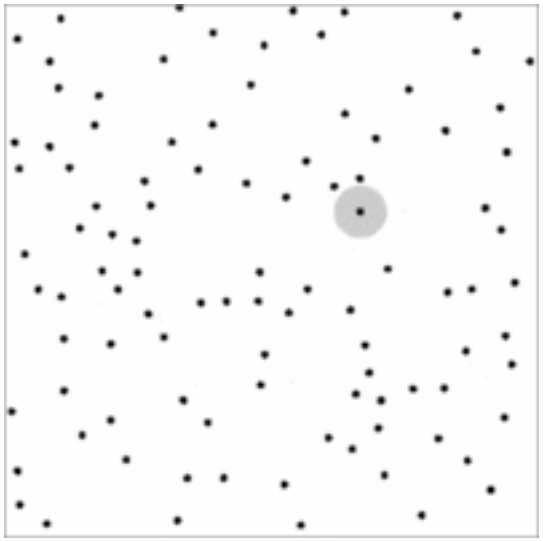

Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

The exercise

- Simulation of N interacting particles in a 1D box
  - Example from Par Lab Boot Camp

http://www.cs.berkeley.edu/~volkov/cs267.sp09/hw2

  - Short-range interaction



- Common simulation problem
  - Same implementation can be applied in several other cases

How to proceed

- Copy the directory

  `/nfsmaster/innocente/parallel`

  in your area
- Inside this directory you find a README.txt file
- 3 proposed serial implementations (see corresponding directories)
- Make copies of the serial.cxx, renaming in openmp.cxx, mpi.cxx, and thread.cxx
  - Look at the comments inside the file to understand where to apply parallelization (essentially require modifications only inside these files)
  - You find all “solutions” in our lectures, but you can look in the web or ask me to find better solutions

case1

- Compile the code with

  `make serial`

- Run `./serial -h`

```
Options:
-h to see this help
-d draw the particles
-n <int> to set the number of particles
-o <filename> to specify the output file name
```

case1

- Basically two loops

```
for (int i=0; i<N; i++)
   for (int j=0; j<N j++)
            // interaction between [i, j]
```

Example (serial execution):
N = 1000 ---> **5.49 seconds**
N = 500 ---> 1.38 seconds ---> x3.98
N = 200 ---> 0.22 seconds ---> x24.95

SCALE as N^2!!!

The parallel implementation in this case is easy...
Example (OpenMP)
P = 2, N = 1000 ---> 2.78 seconds ---> x1.97
P = 3, N = 1000 ---> 1.86 seconds ---> x2.95
P = 4, N = 1000 ---> 1.40 seconds ---> x3.92
P = 8, N = 1000 ---> 0.72 seconds ---> x7.62

SCALE as P processors

case2

- Note that the interaction between B and A is the opposite of between A and B
  - We can calculate an half of the interactions

```
for (int i=0; i<N-1; i++)
  for (int j=i+1; j<N j++)
  // interaction betwen [i, j] and [j, i]
```

Example (serial execution):
N = 1000 ---> 2.66 seconds ---> x2.06
N = 500 ---> 0.70 seconds ---> x7.84
N = 200 ---> 0.11 seconds ---> 49.91x

SCALE as N^2!!!

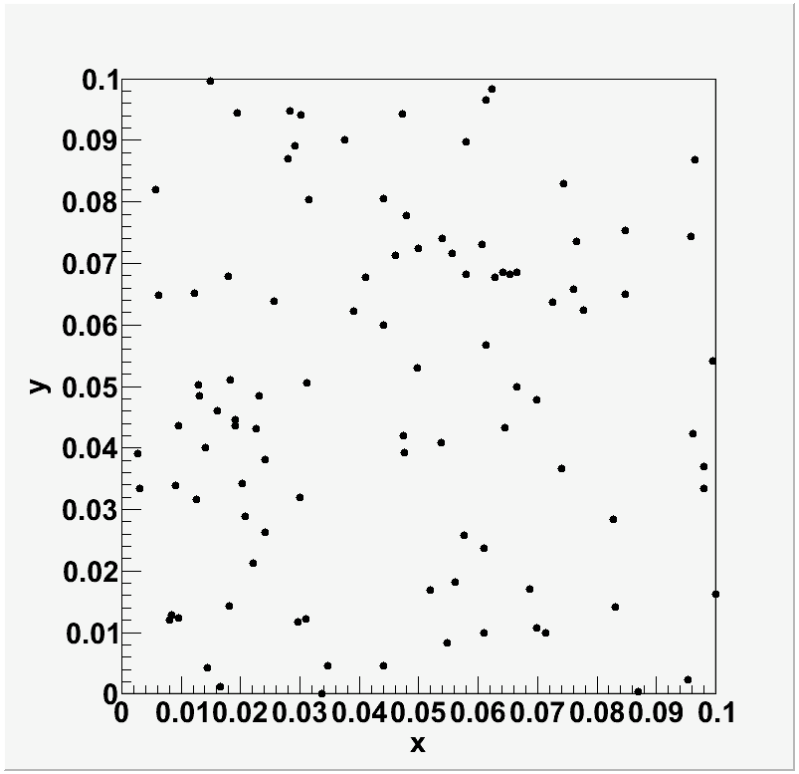
Requires some attention to avoid race conditions

case3

- How can we scale as N (not N^2)?

case3

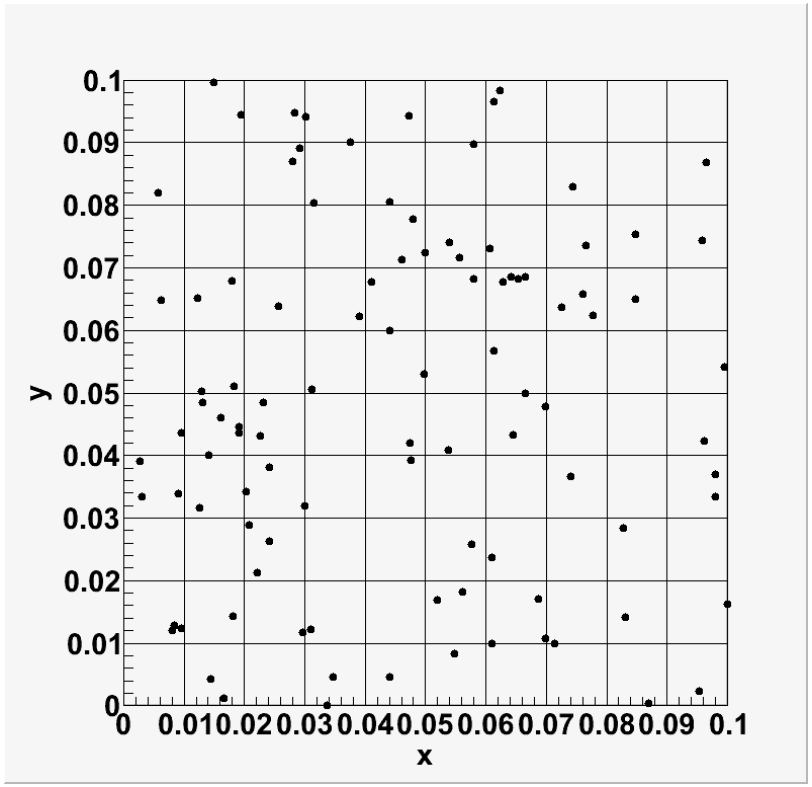
- How can we scale as N (not N^2)?
  - Hint: remember that we have a short-range interaction, i.e. do not need interaction between all particles



case3

- How can we scale as N (not N^2)?
  - Hint: remember that we have a short-range interaction, i.e. do not need interaction between all particles

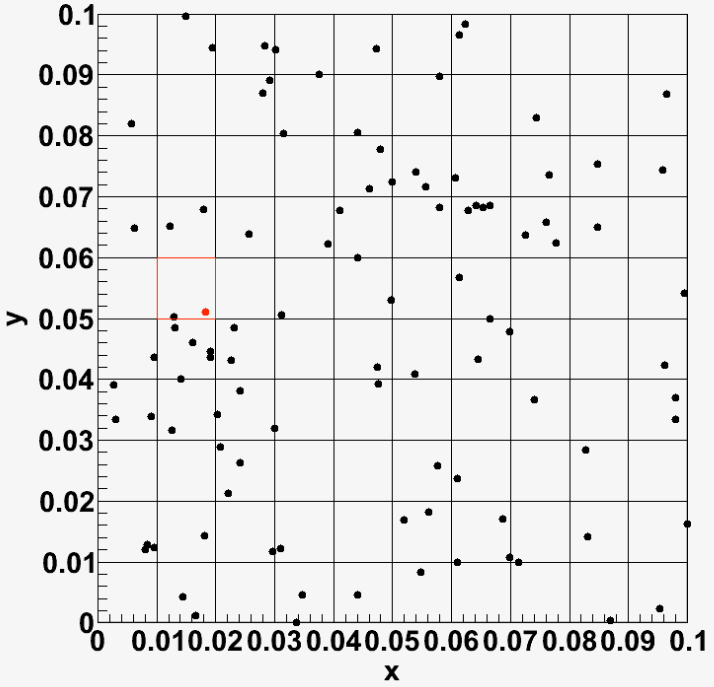
Do a mesh (decomposition of the data sample), where the size of the cells is the range of the interaction



case3

- How can we scale as N (not N^2)?
  - Hint: remember that we have a short-range interaction, i.e. do not need interaction between all particles

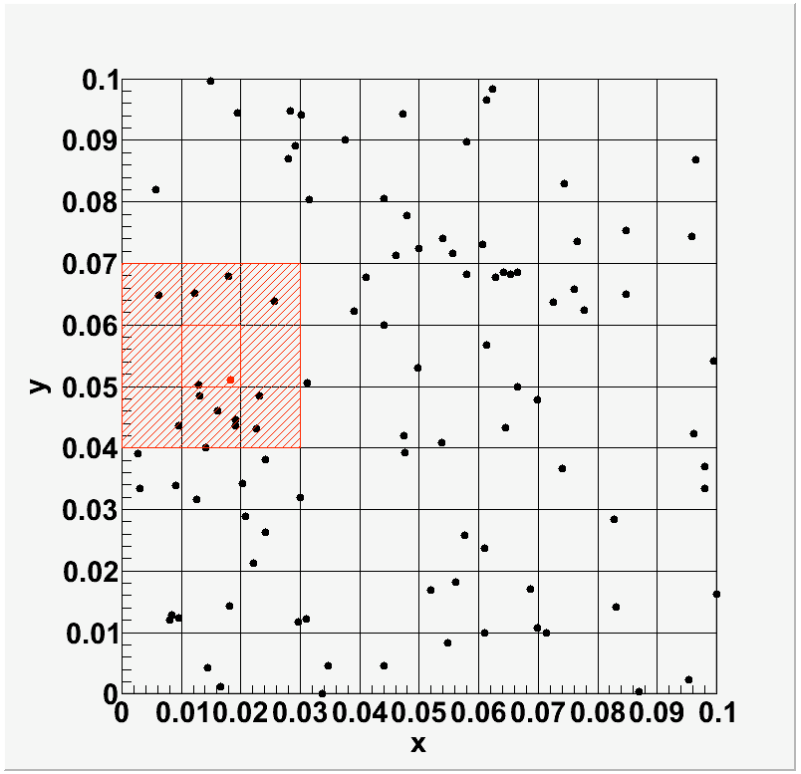
Loop over the cells.
For each cell, loop over his particles and make the interactions with the particles of the neighboring cells



case3

- How can we scale as N (not N^2)?
  - Hint: remember that we have a short-range interaction, i.e. do not need interaction between all particles

What do you expect as speed-up for the serial implementation?



case3

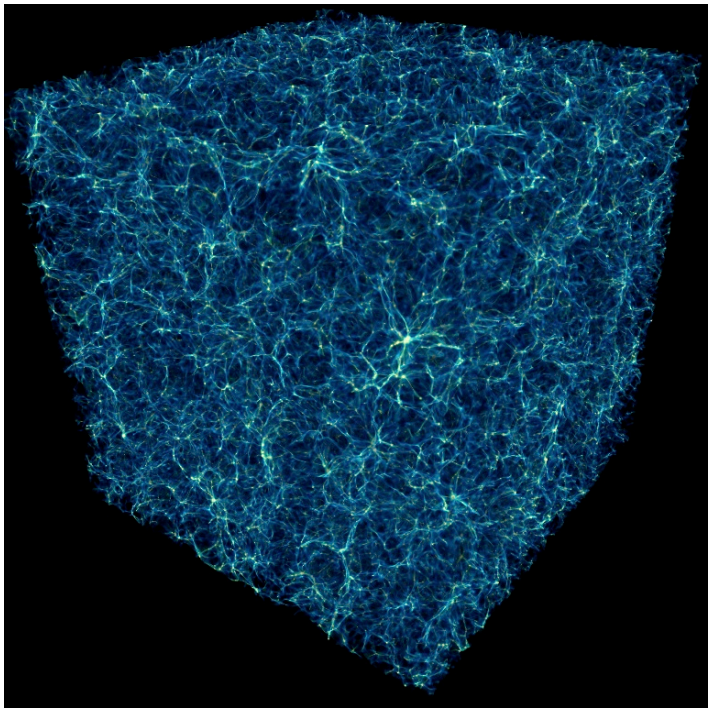
N = 1000 ---> 0.19 seconds ---> x28.89
N = 2000 ---> 0.39 seconds ---> x14.08
N = 4000 ---> 0.79 seconds ---> x6.95

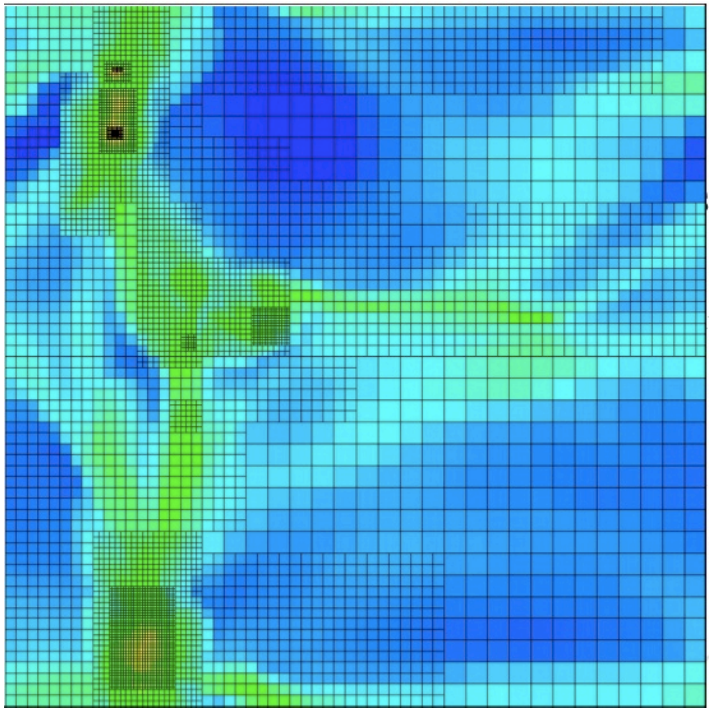
SCALE as N

Better performance are still possible…

- Decomposition problem are common in many problems
  - You can split the data over the processors
  - To have a good balance you can do an adaptive mesh (or more complex adaptive mesh refinement)

- Galaxy formation (example from http://www.isgtw.org/?pid=1001250)
  - a total of about one billion individual grid cells
  - adaptive mesh refinement





The 3D domain (2 billion light years of side).
Colors represent the density of the gas

And now enjoy the exercise!