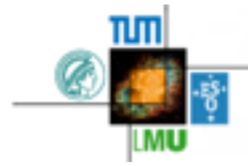
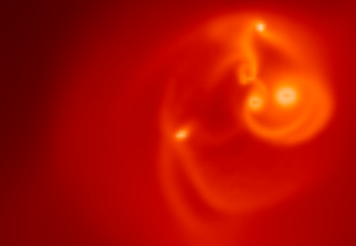


Running GANDALF with openMP/MPI

Giovanni Rosotti

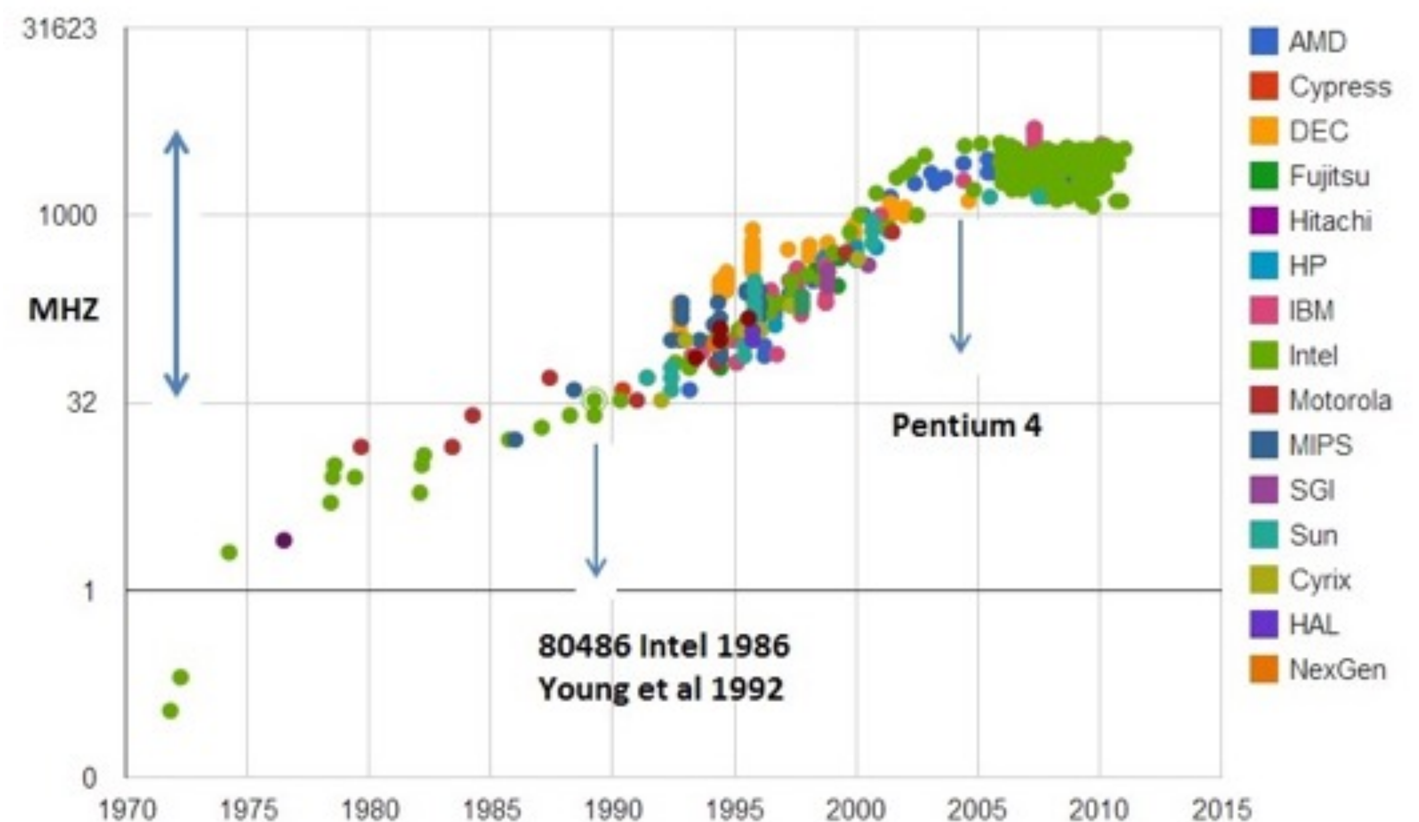
IoA Cambridge

30th October 2015



Parallelisation - some background

- At the end of the '90s CPUs almost reached a bottleneck
- Up to then improvements in performance were mainly coming by increases in the clock frequency
- A hard boundary of a few Ghz was reached
- Since then the way to go to improve performance has been mostly in having more than one core
- Note: single core performance is still increasing! But at a much slower rate than what it used to be



How a CPU works

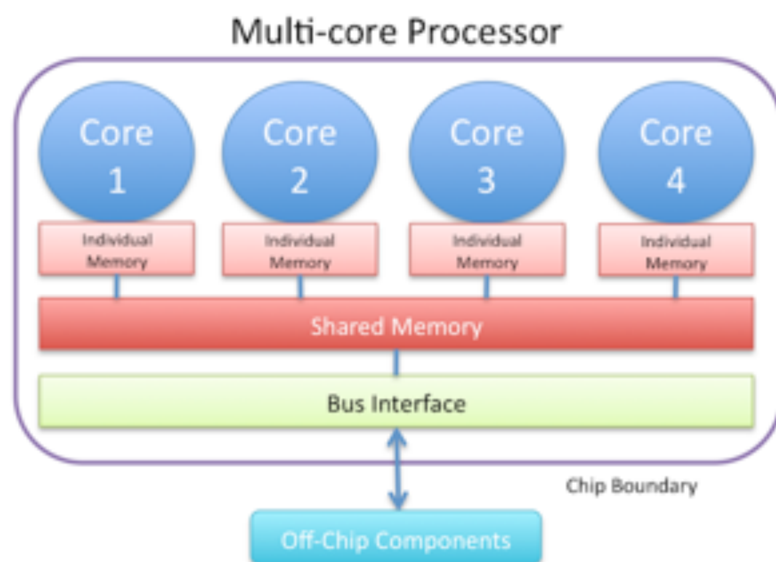
- How can you improve the performance of a CPU without increasing the clock frequency?
- In theory when you compile a code it is converted into a sequence of instructions
- The CPU goes through them one at time. The clock frequency determines how fast it goes
- In practice, not that simple. Instructions do not all take the same time (e.g. floating point division takes much longer than the other operations). Also the memory (=RAM) takes much longer than a CPU clock to be read
- You can guess that it is then possible to exploit many techniques to improve performance while keeping the same clock frequency

More about tricks

- Out of order execution: the CPU doesn’t actually go from top to bottom (but the result is the same as it did!)
- Cache: faster (but smaller…) ram
- Super-scalar processors: your CPU can do more than one operation at time (e.g. multiplying two ints while it multiplies two floats)
- Other ones: pipelining, branch prediction, vectorization, …
- These tricks improve the performance…
- …some of them also exploit parallelism
- But we do not usually talk about this level of parallelization because we cannot code for it; it is handled automatically by the CPU
- You can still try to code in certain ways that (you think) will help the CPU using these tricks

Parallelism

- Can be either inside your machine…
  - modern CPUs have more than one core
  - machines used for supercomputing (or for servers) can have more than one CPU
- …or more machines can be connected together
  - all modern supercomputers are a collection of many (10^3-10^4) nodes
  - Latency and bandwidth of the interconnection determines how good (or bad) your code is going to scale



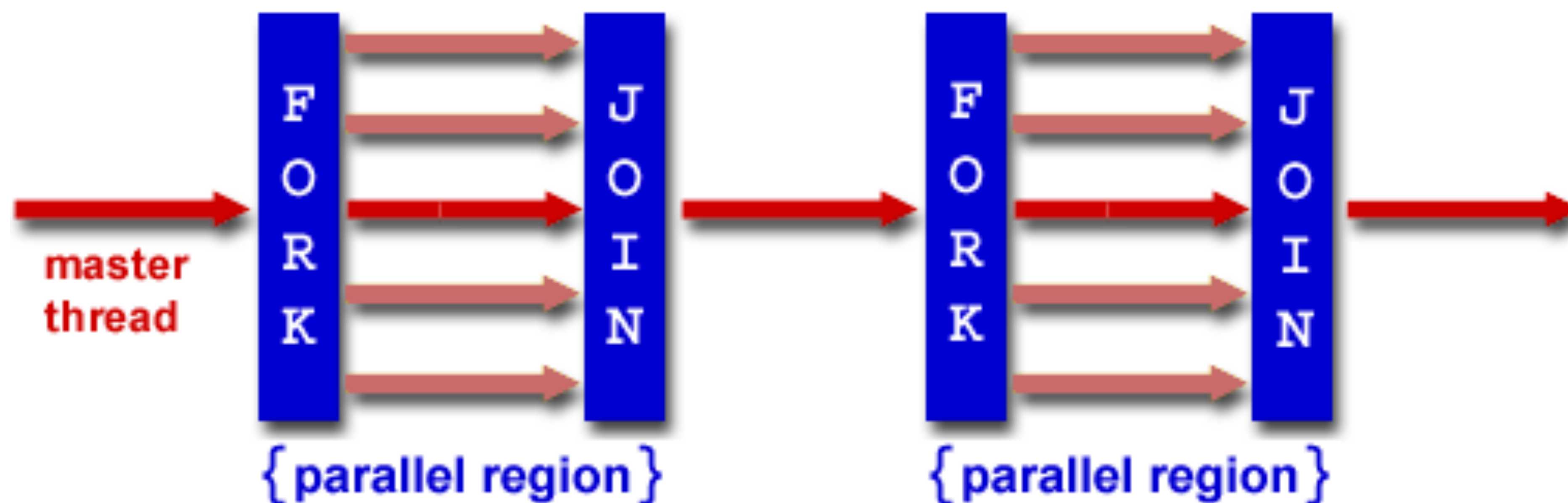
How to code for parallelism

- Short answer: thousands of ways
- But mostly used in supercomputing:
  - OpenMP
  - MPI

- I will NOT talk about GPU computing in this talk
  - It's so different that it should get a talk on his own
  - GANDALF does not exploit it at the moment

OpenMP

- **Shared memory** (i.e. on the same machine)
- Start parallel regions
- The rest of the code remains serial
- Easy to add parallelisation incrementally
- Note that the paradigm of repeated fork/join is typical of openMP and NOT of shared memory in general (e.g. pthreads is different)



An example of OpenMP

```
#pragma omp parallel for
for (int i=0; i<N; i++) {
    c[i]=a[i]+b[i];
}
```

- You can add instructions to every loop. The code will still work even if you haven't parallelized the other loops!
- Incremental parallelisation
- To parallelize a loop, the iterations need to be independent of each other. Or, better, there needs to be no **data dependency**
- You can still parallelize if the dependencies are simple (e.g.: sum of the elements)

MPI

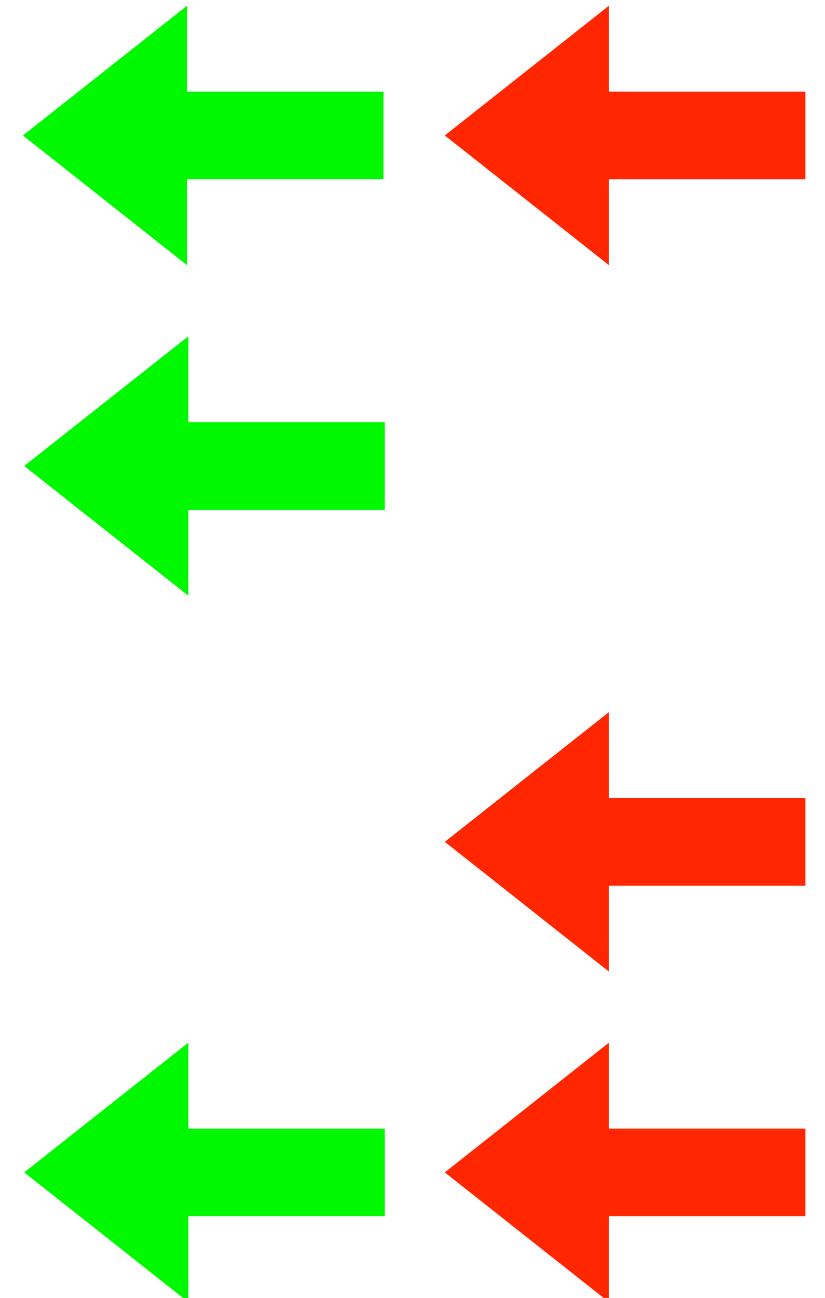
- Works both on the same machine or on different ones
- **Start as many replicas of your program as you want**
- But the replicas cannot see the memory of the other ones (distributed memory)
- The different replicas can communicate only by sending messages. **The library abstracts the actual way the communication takes place** (in-memory, ethernet, infiniband, …)
- A variety of functions for sending messages (one-to-one or global communication)

MPI in 30 seconds

- MPI_Init
- MPI_Comm_size - tells you how many processors you have
- MPI_Comm_rank - tells you your id
- MPI_Send
- MPI_Recv
- MPI_Finalize

MPI in 30 seconds

```
#include <iostream>
#include <mpi.h>
int main(int argc, char** argv) {
    int rank, ncpus;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&ncpus);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if (rank==0) {
        int a=5;
        MPI_Send(&a,1,MPI_INT,
1,0,MPI_COMM_WORLD);
    }
    else if(rank==1) {
        int b;
        MPI_Status status;
        MPI_Recv(&b,1,MPI_INT,
0,0,MPI_COMM_WORLD,&status);
        std::cout << b << std::endl;
    }
    MPI_Finalize();
}
```



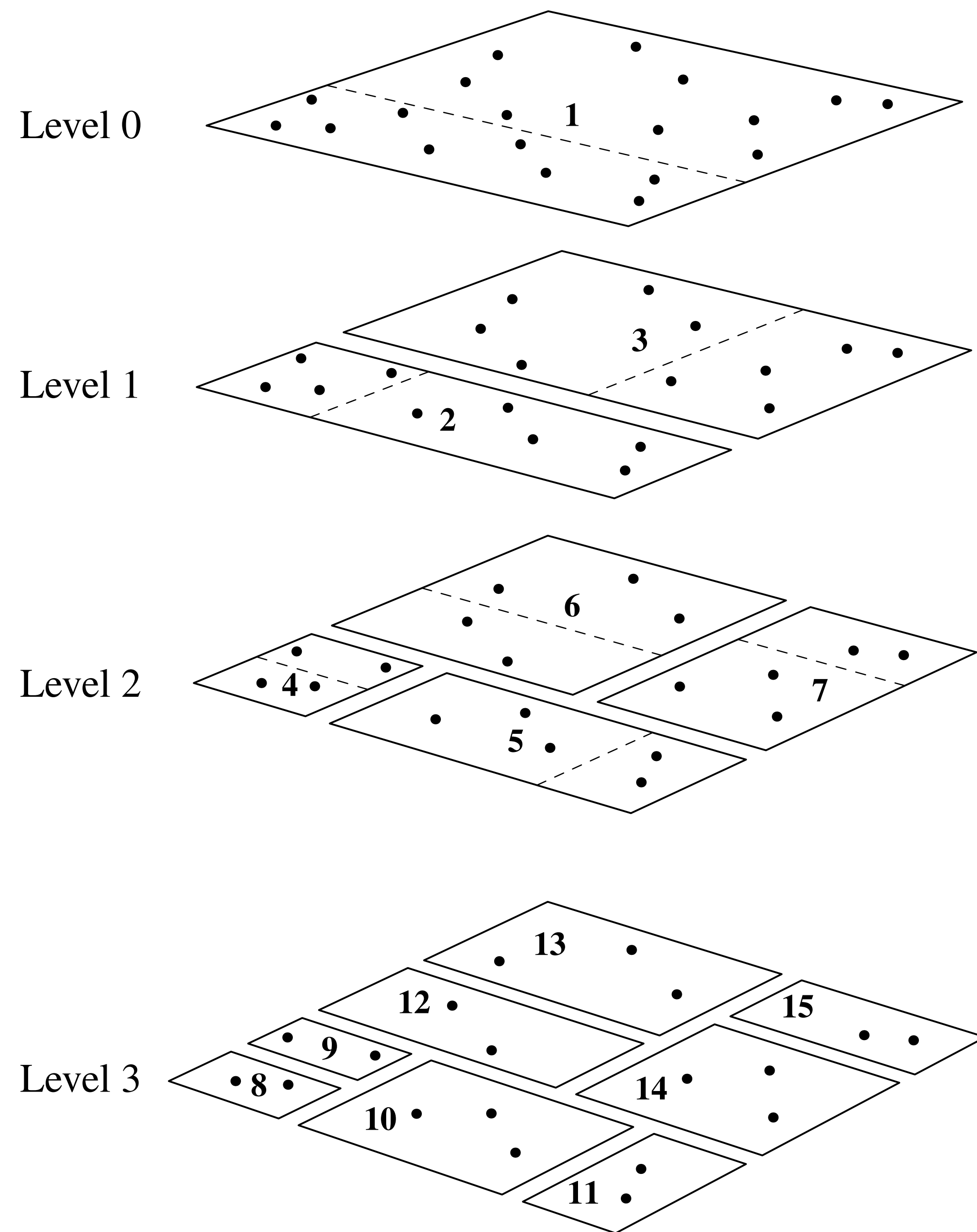
In reality much more difficult!

- MPI_Allreduce
- MPI_Gather
- MPI_Scatter
- MPI_Alltoall
- … (the library contains more than 100 functions)
- most importantly, you need to think HOW you parallelise the code!

Parallelisation in GANDALF

- You don't need to know how GANDALF is parallelised to use it
  - But you might want to know for your culture. Also, you do need to know it if you added some physics and you need to parallelise it!
- OpenMP parallelisation is mostly trivial
- Just add the right directives to your loop
- Only complicated point is tree construction
    - Done level by level. Means that you start taking advantage of parallelism only on the lower levels
    - In theory, this approach sets a hard boundary on the maximum speed-up that can be reached
    - In practice, tree construction is only a minor fraction of the run-time anyway

Tree



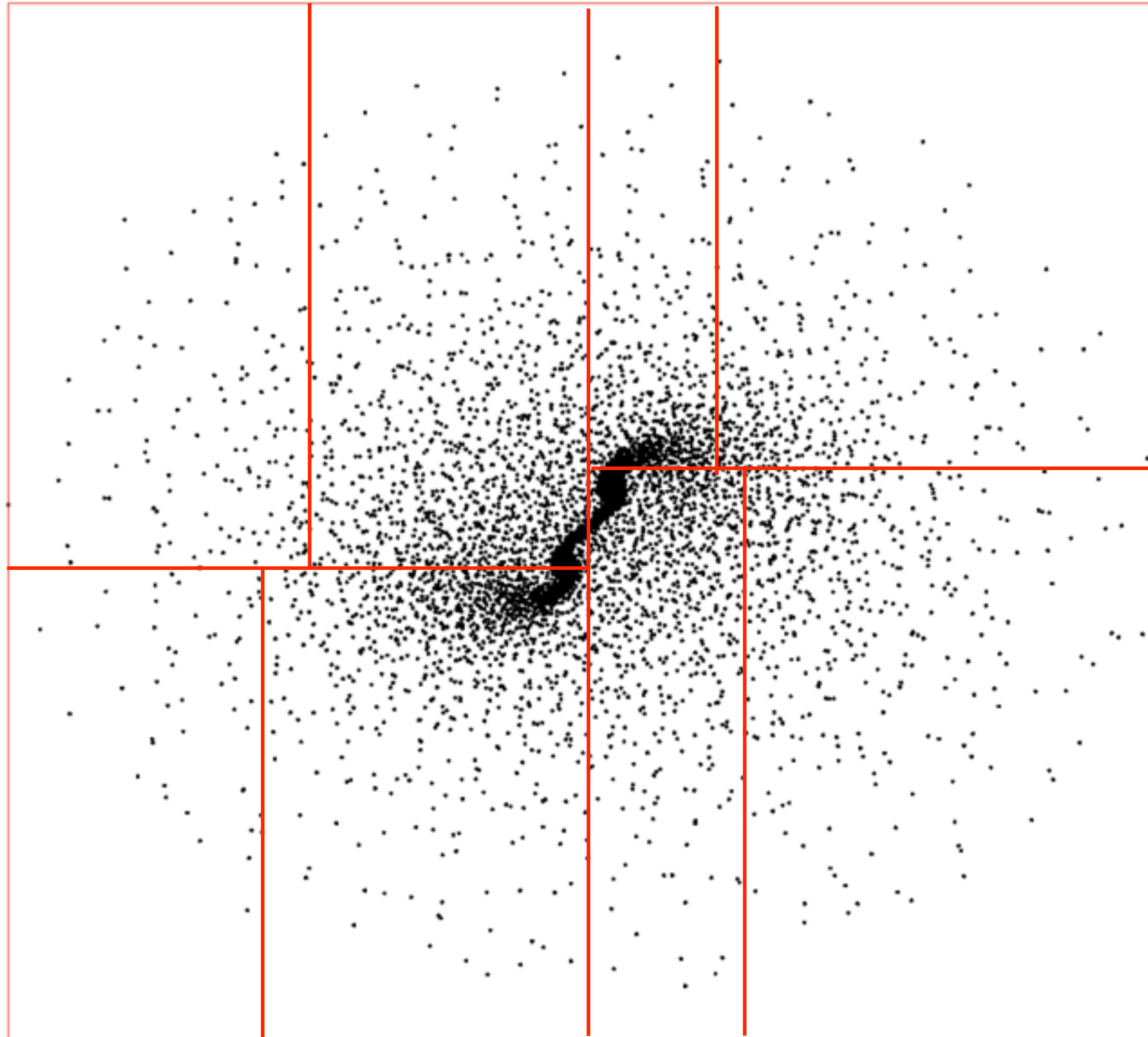
KD Tree

Widely used

- PKDGRAV (Stadley+ 2001)
- GASOLINE (Wadsley+ 2004)
- Gafton & Rosswog 2011

Figure from Gafton & Rosswog 2011

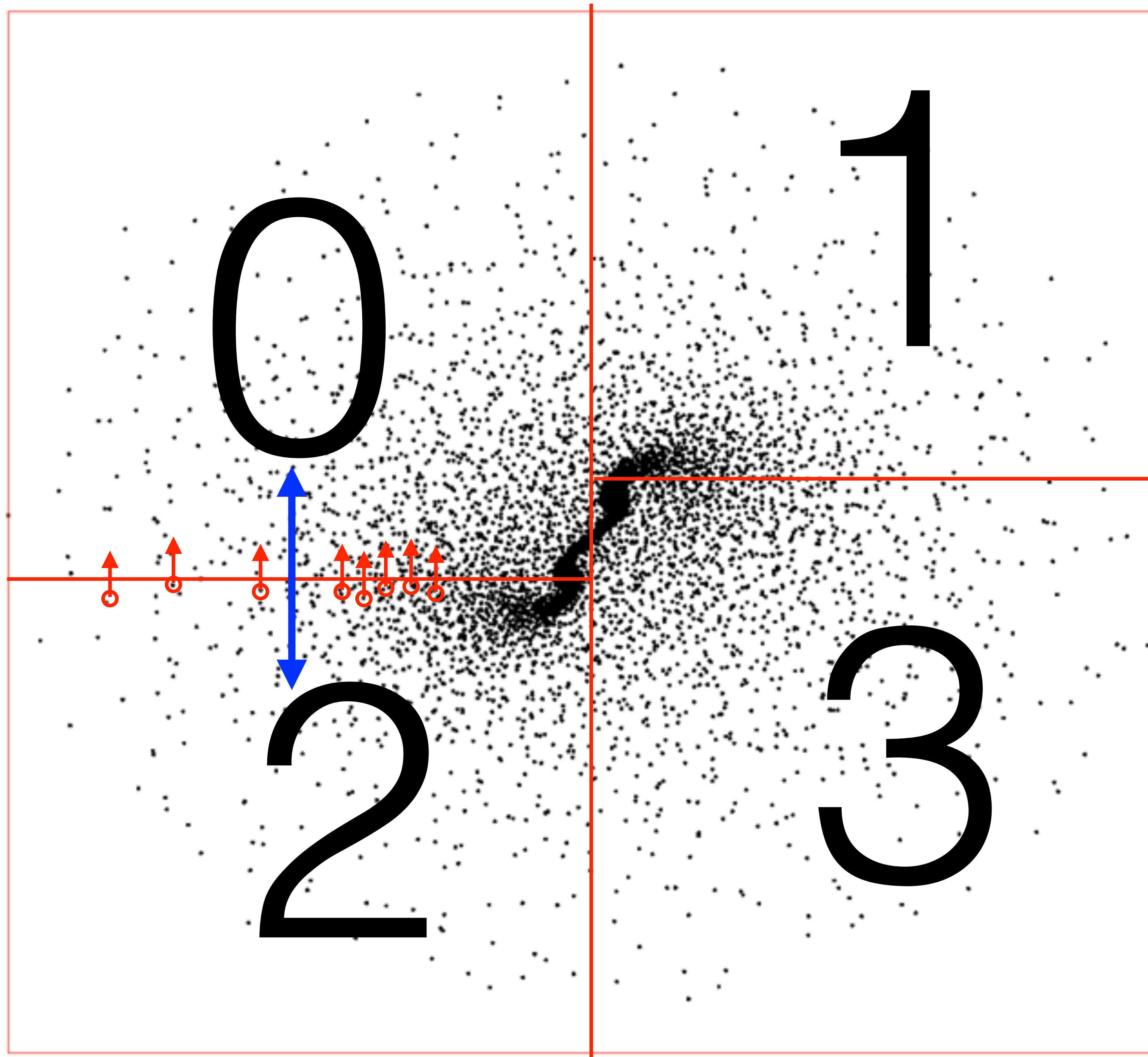
Tree construction



MPI - Domain decomposition

- MPI: much more complicated! Currently works, but not super robust
- Uses **domain decomposition**
- Uses the same tree used for neighbour finding
- Note: this means that **number of MPI processes needs to be power of 2!**
- Simplifies the coding
- Load balancing is achieved by “shifting” the domain boundaries

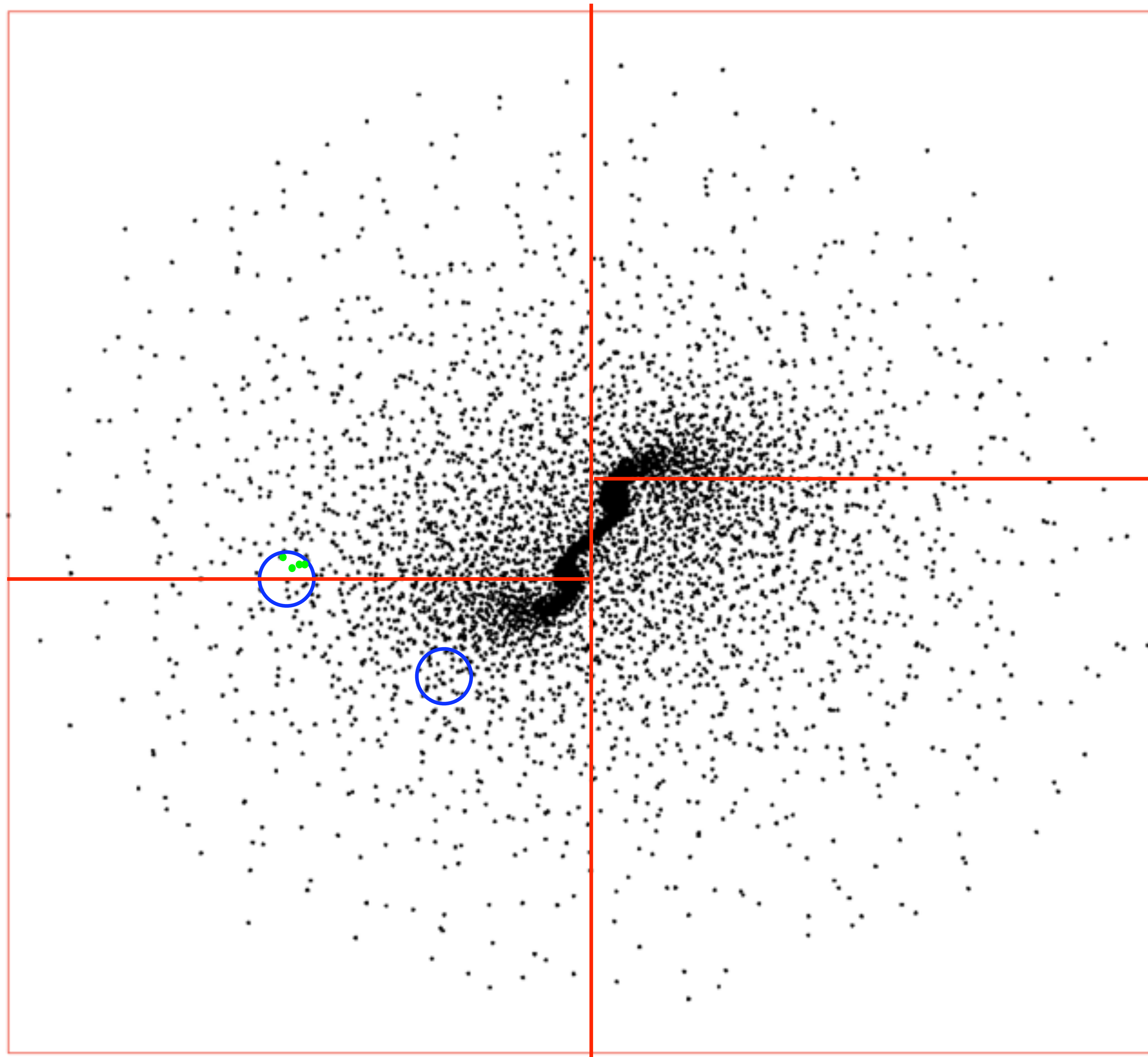
Tree again



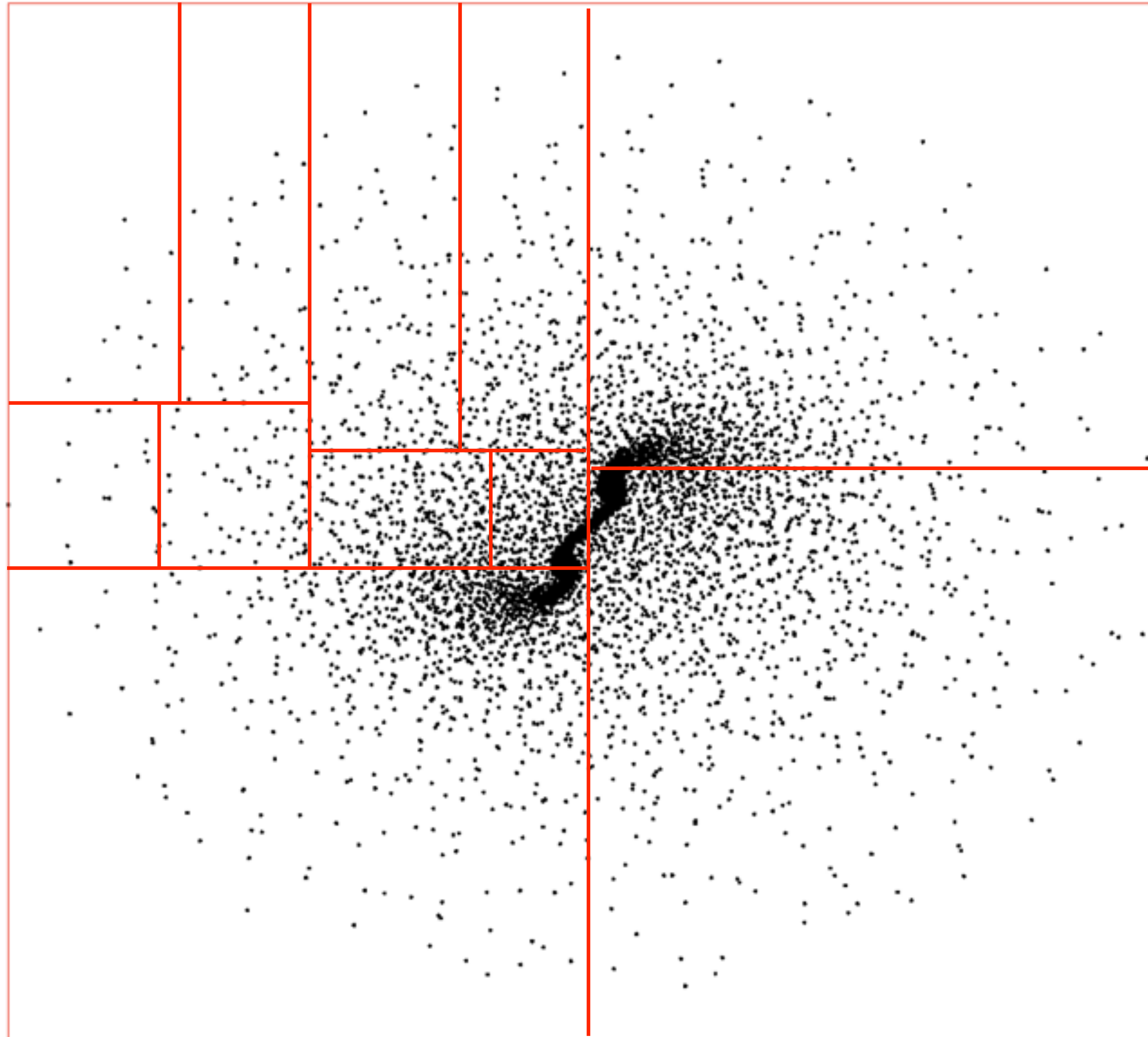
Parallelisation in GANDALF

- For h calculation need to have ghosts
- For force calculation each node has a “pruned tree”, that is a simplified version of the tree of the other processors
- The pruned tree never contains particles, only cells
- Pruned trees are used also for load balancing
- The gravity force can be (sometimes) computed using the pruned trees. Particles for which this is not possible, and particles that interact hydrodynamically with other domains, need to be “exported”. This means a domain asks to another one to compute the force on its behalf

Ghost and exported particles



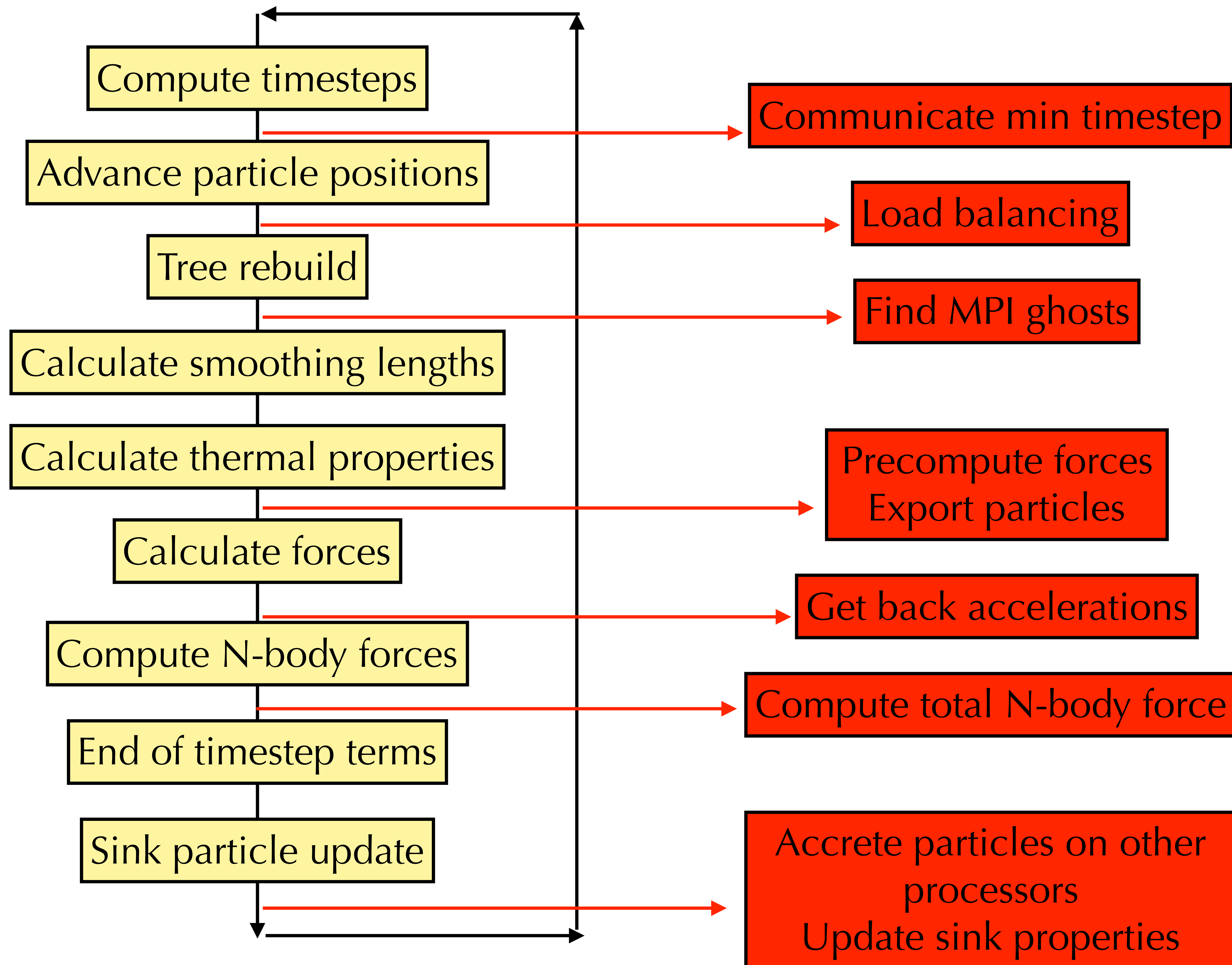
Pruned trees



continued

- Once particles have been exported, the other processor computes locally the force. Then the forces from all processors are summed up
- Stars are simpler: each processor has a copy of them. Only the forces from local gas particles are computed, and then they are summed up. Calculation of forces from other stars instead are repeated on each processor (so **don't use GANDALF as a pure N-body code with MPI**!)
- Forces of the gas on the stars are computed locally by each processor and then summed up

Main loop with MPI calls

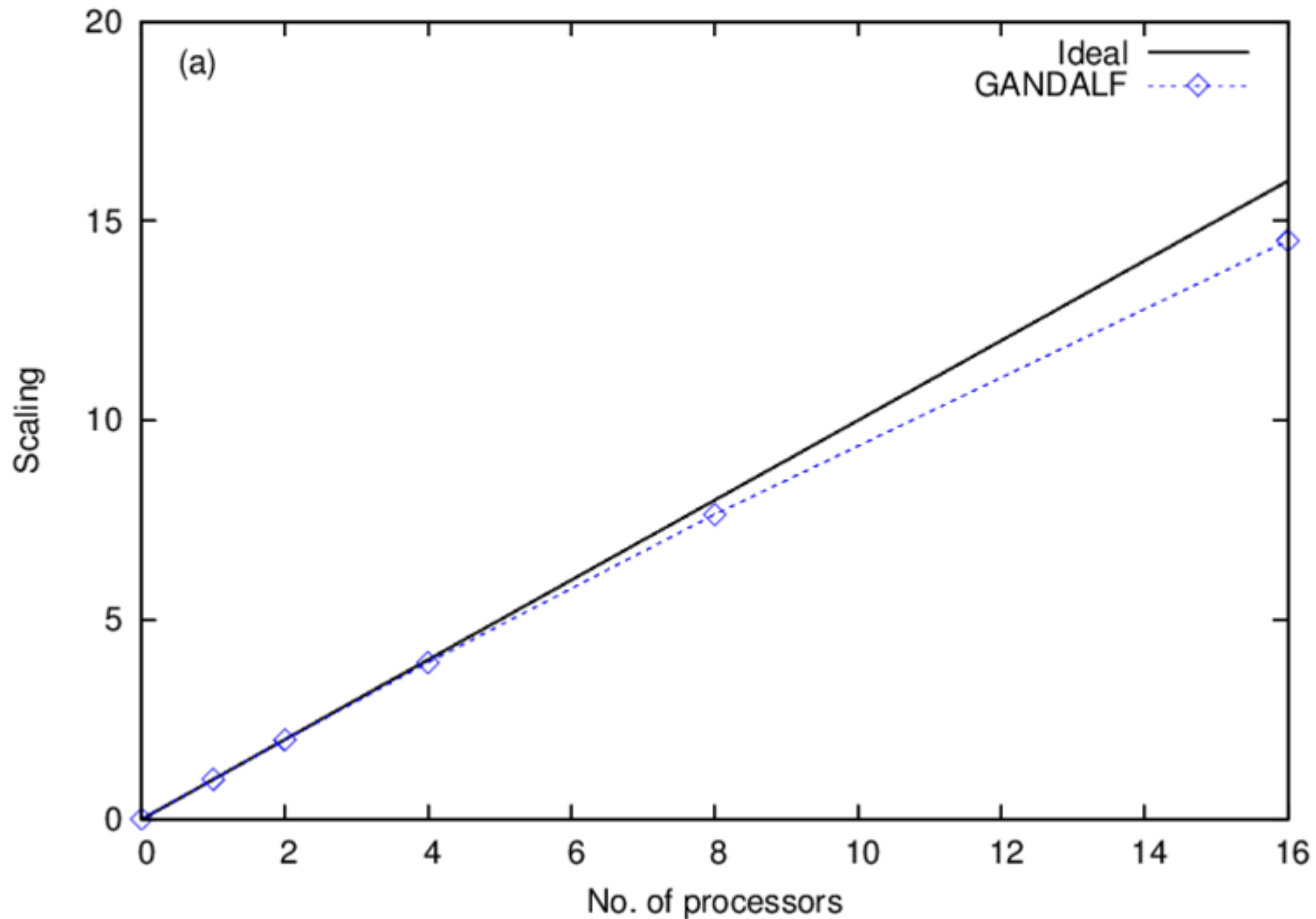


Recommendations

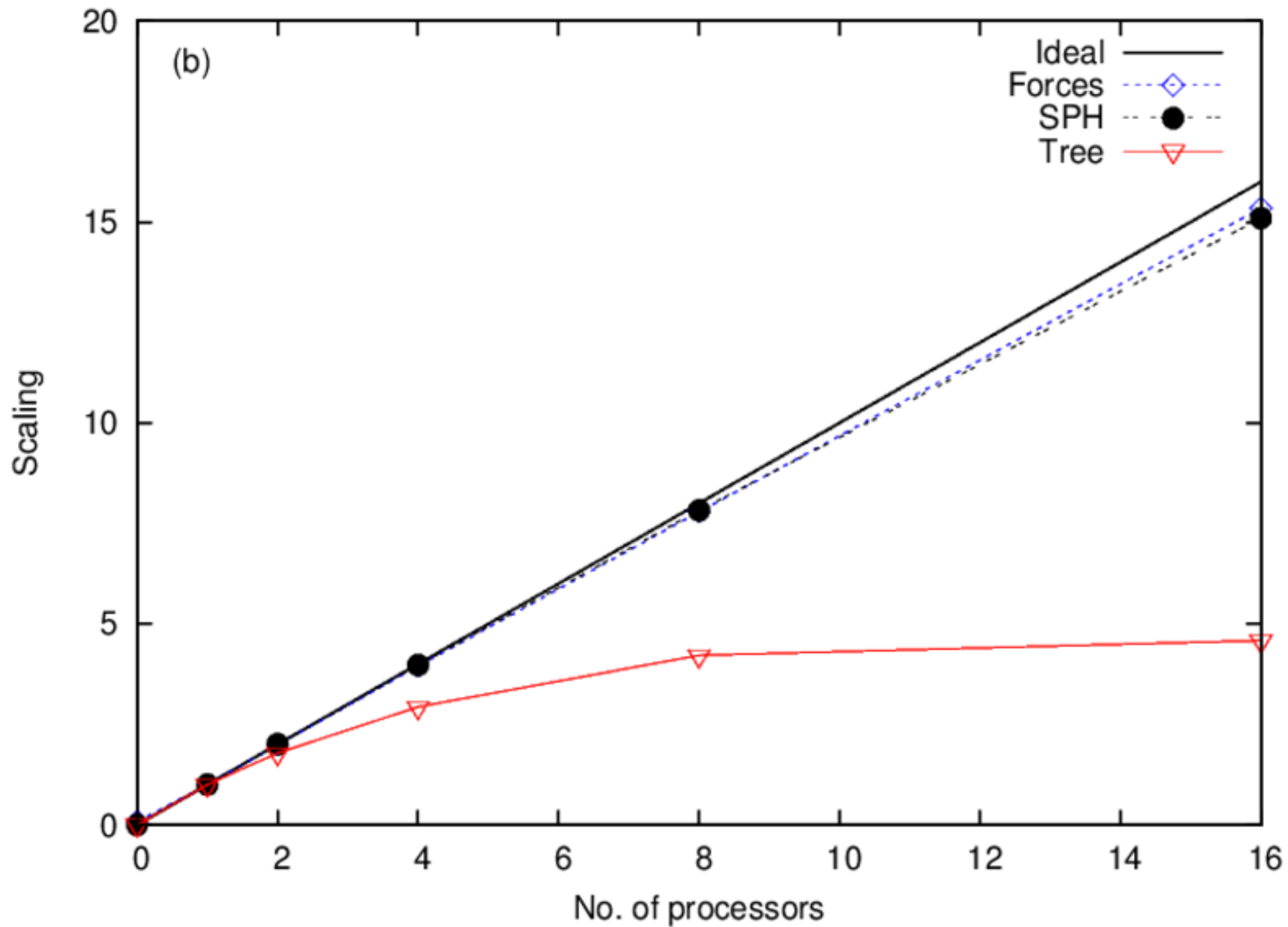
- GANDALF is NOT designed to scale to ~1000 cores. Normally not needed for disc/star formation problems. If people from different fields have different needs, let us know!
- MPI parallelisation is relatively basic - good on a few processors. But combined with openMP, one can efficiently use even 100 processors.
- Do expect MPI parallelisation to improve with time
- Get a sense of how your problem scales before running the "production" simulations; too many processors could actually slow the code down (happened to me several times with other codes)

Scaling tests - OpenMP

- Freefall test with 5e5 particles

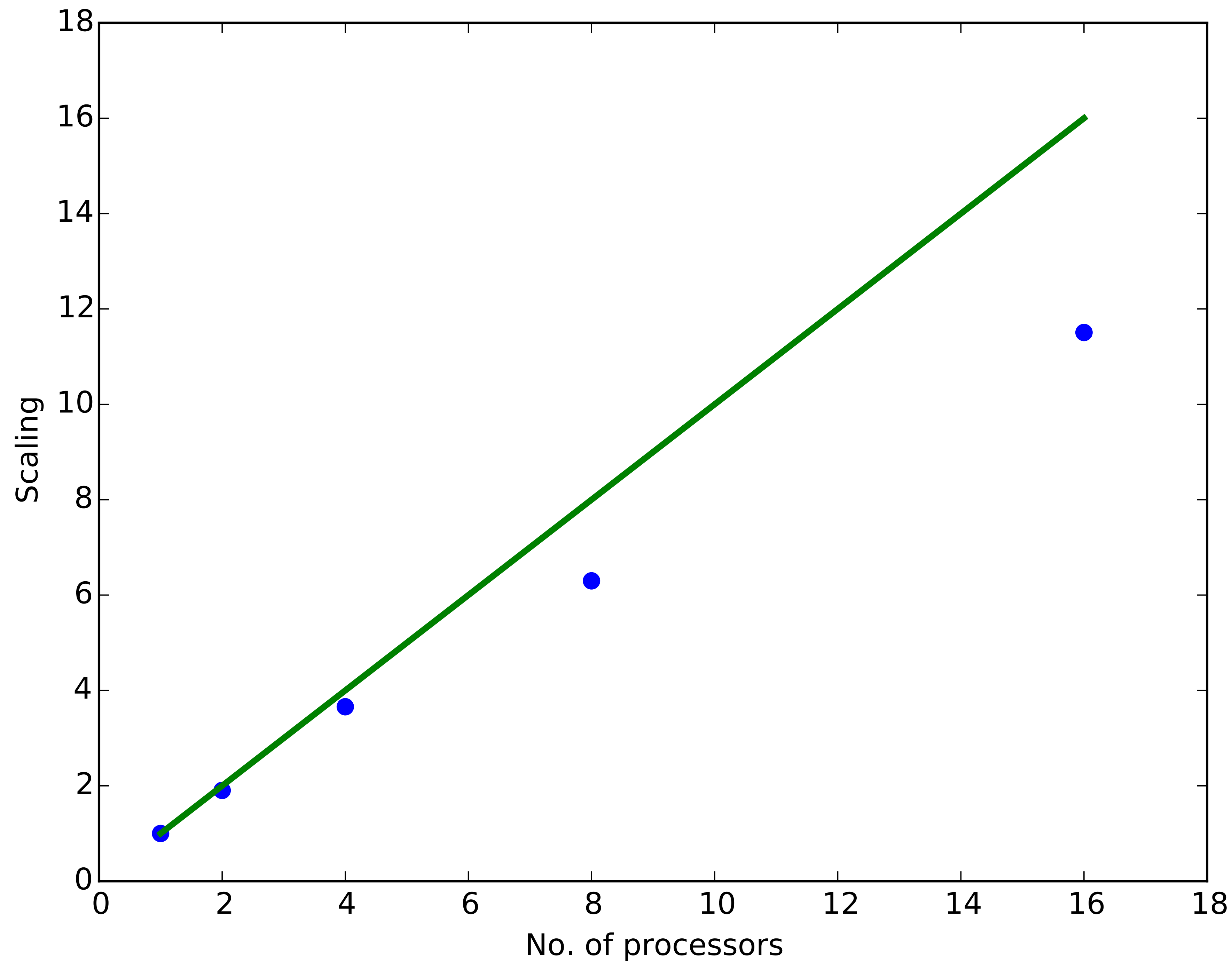


Scaling tests



Scaling tests - MPI

- Freefall test with 1e6 particles



How to compile with openMP/MPI

```
CPP                 = g++
PYTHON              = python
COMPILER_MODE       = FAST
PRECISION           = DOUBLE
OPENMP              = 0
OUTPUT_LEVEL        = 1
DEBUG_LEVEL         = 0
```

Change it to your MPI compiler (normally mpic++)
Make sure that src/makefile is picking up the right options for your compiler

Simple switch
You might have to change the options in src/Makefile if using a non-standard compiler

- With openMP, start your program normally. The number of threads is controlled by the OMP_NUM_THREADS environment variable
- With MPI, you normally do like this:

```
mpirun -np numthreads program
```

- Note that on supercomputers it might be different - check their documentation
- Try doing a scaling test on a simple problem! (e.g.: freefall.dat)