Parallelisation of a Staggered Grid solver

Heisig, Hammer, Ernst

February 3, 2014

Outline

1. Parallelization basics
2. Implementation
3. Results
4. Plans for the future

Why parallelize your code?

Pro

- more compute power
- more memory
- more caches (!)
- parallel computing is the future

Con

- added code complexity
- communication overhead
- Increased power consumption

Don't parallelize without profiling and performance modelling!

MPI in a nutshell

The Message Passing Interface

- call your program with `mpirun -np <N> <NAME> <ARGS>`
- spawns `<N>` identical processes
- only `MPI_MPI_Comm_rank(...)` gives different results

Typical usage:

- split domain between all processes
- perform local updates
- exchange the borders
- repeat

Implementation

The following steps must be parallelized

- SOR::solve()
- SOR::residual()
- SOR::normalize()
- determineNextDT()
- refreshBoundaries()
- computeFG()
- composeRHS()
- updateVelocities()

Most of the time is spent in the SORSolver, so this is the focus.

Domain partitioning

- Usually domain is split in roughly quadratic tiles
- We chose the simpler approach: Split in horizontal stripes

Pro

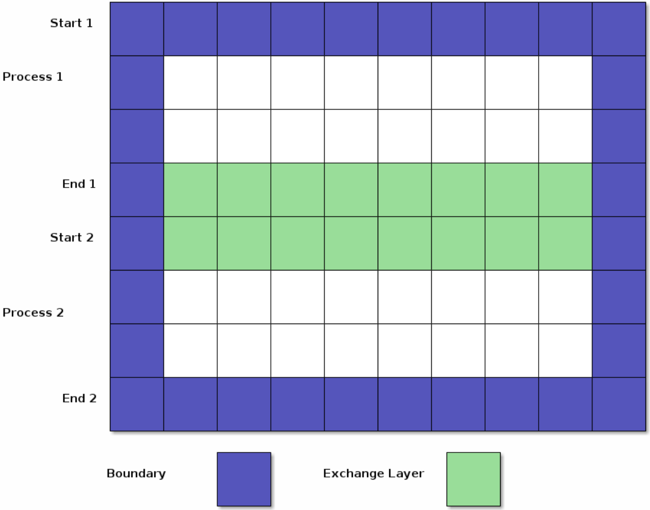
- easier to implement
- fast access patterns along the cachelines

Con

- bad surface / size ratio for large number of processes
- more communication overhead

Domain partitioning (cont.)

Example of a 8 x 9 domain with 2 processes



Continuous migration

How do we migrate our serial codebase to a parallel one without the agonizing pain™ ?

Migration phase:

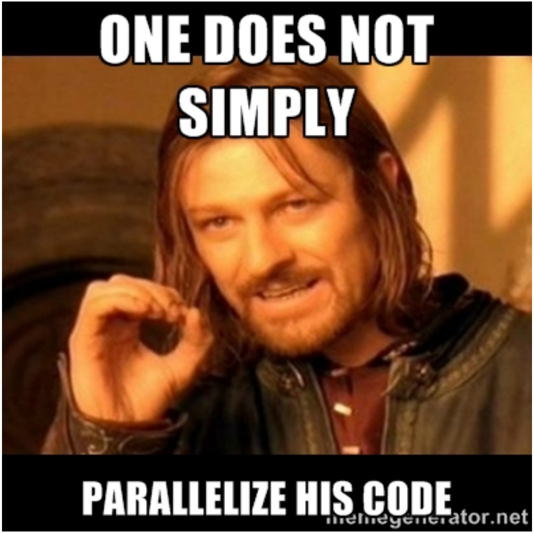
- Every process still has all the data
- Parallelize only one operation at a time
- Methods can be tested individually

When all methods are converted, switch the Array implementation to store only local elements.

Pitfalls

TODO Domi

Regarding parallelization



Results

Was it worth the effort?

Explanation:

- SOR or Jacobi solver does not scale well

Use a better algorithm before writing parallel code!

Plans for the future

Current numerical programs share several problems:

Plain C/C++/FORTRAN

is good for performance but inappropriate for high level tasks, especially runtime features.

Huge codebase

of several hundred thousand lines of code ($\rightarrow$ Huge maintenance effort)

Hardly reusable parts

that are tightly connected to each other

How can we escape this mess?

Hybrid approach

Use (at least) two languages!
One low level language with focus on:

- compiler optimisations
- vectorisation
- Typically C/C++ or FORTRAN

One high level language with focus on:

- abstraction
- beauty
- features
- rapid prototyping
- foreign function interface
- Recommended: Guile/Scheme, Python

The End

Thank you for your attention!