Best practices of parallel programming for HPC

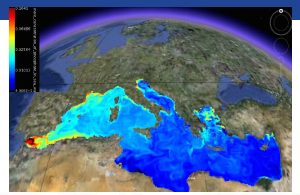
Ivan Girotto – igirotto@ictp.it

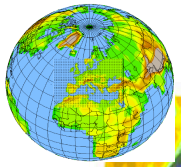
International Centre for Theoretical Physics (ICTP)

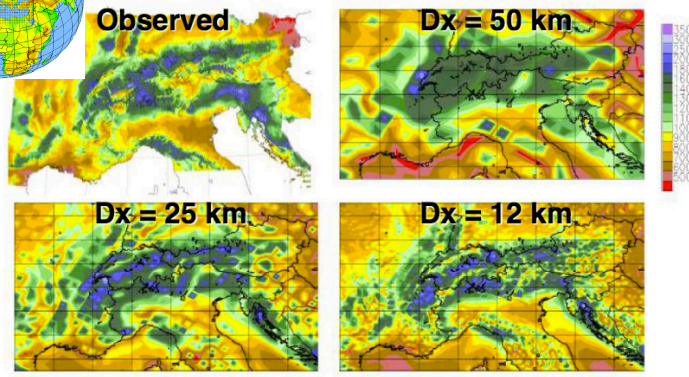
Why use Computers in Science?

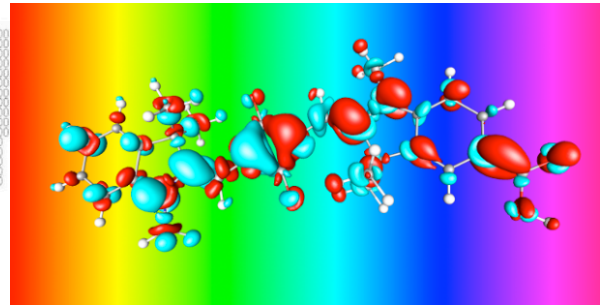
- Use complex theories without a closed solution: solve equations or problems that can only be solved numerically, i.e. by inserting numbers into expressions and analyzing the results
- Do “impossible” experiments: study (virtual) experiments, where the boundary conditions are inaccessible or not controllable
- Benchmark correctness of models and theories: the better a model/theory reproduces known experimental results, the better its predictions

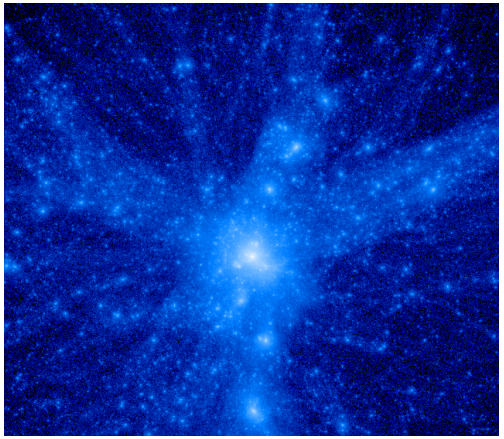
SW in Science



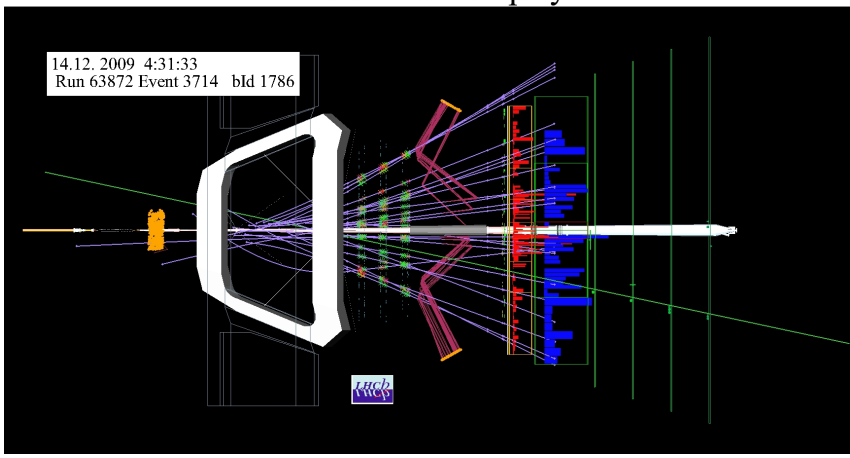


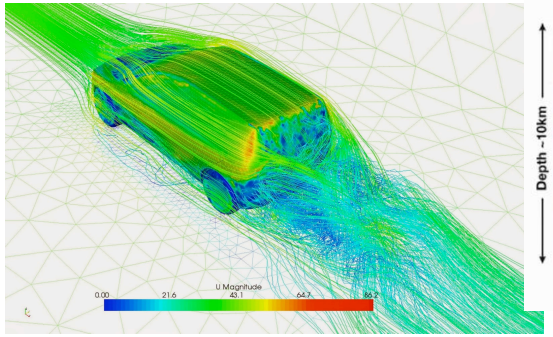


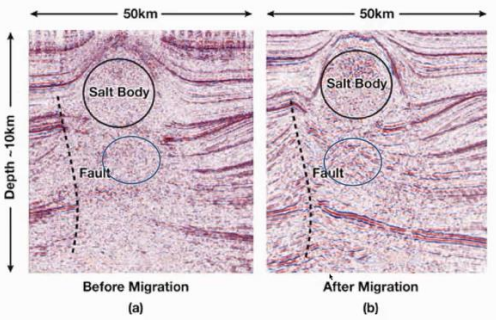




LHCb Event Display







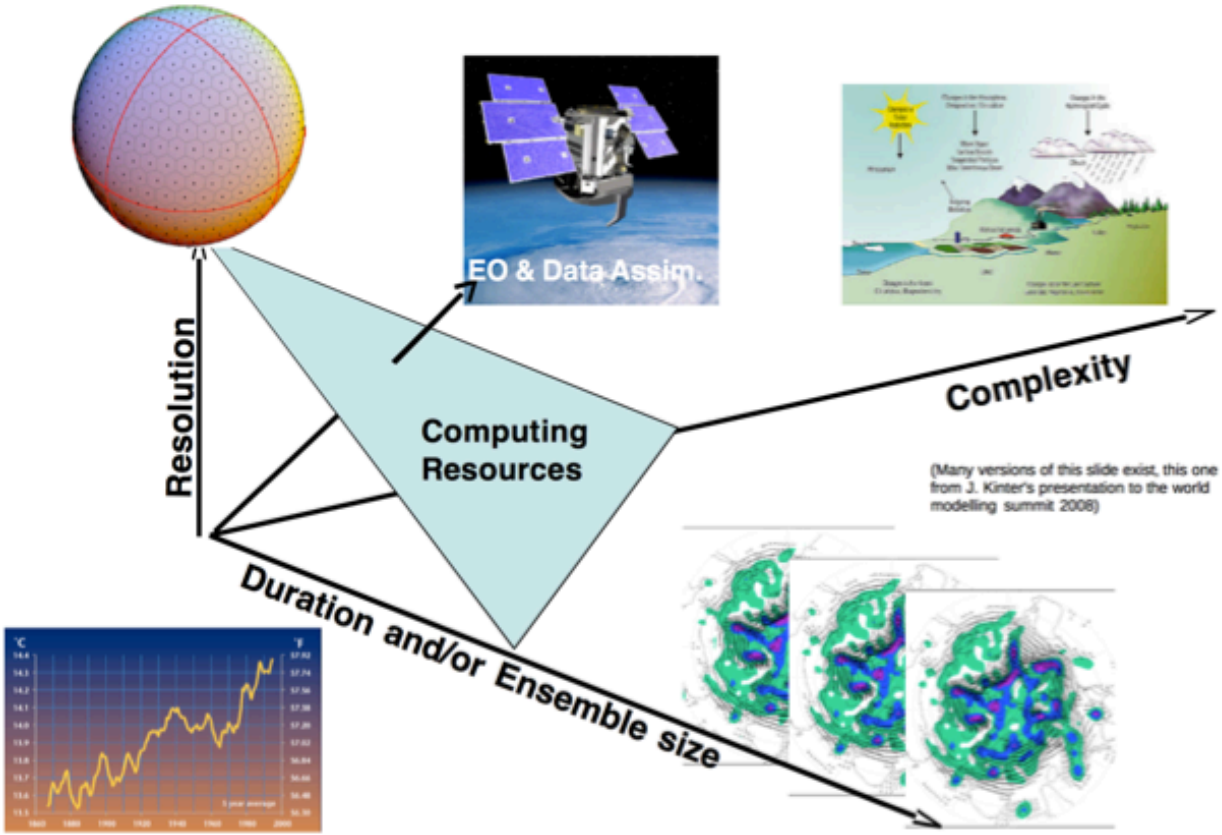
What is High-Performance Computing (HPC)?

- Not a real definition, depends from the prospective:
  - HPC is when I care how fast I get an answer
- Thus HPC can happen on:
  - A workstation, desktop, laptop, smartphone!
  - A supercomputer
  - A Linux Cluster
  - A grid or a cloud
  - Cyberinfrastructure = any combination of the above
- HPC means also High-Productivity Computing

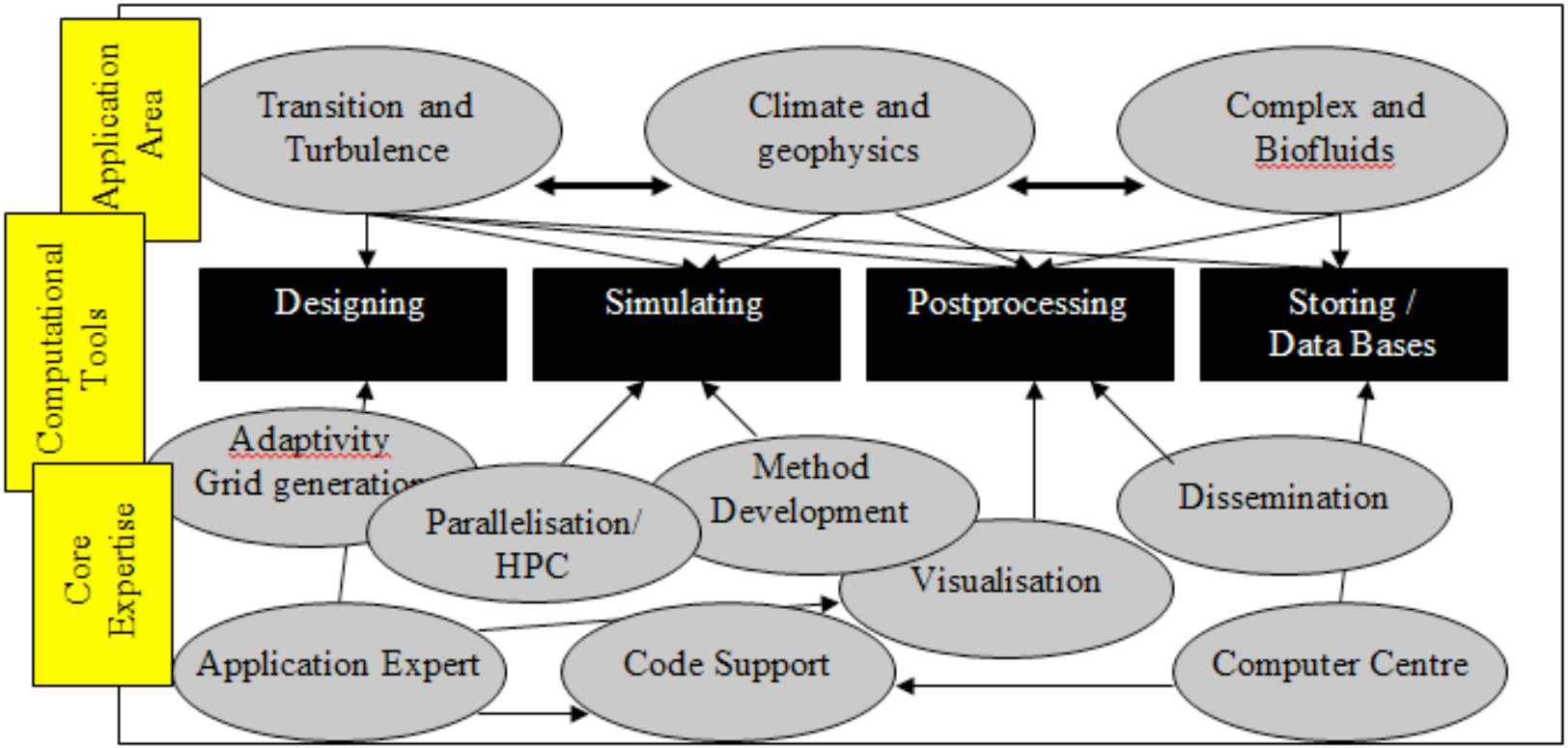
Why would HPC matter to you?

- Scientific computing is becoming more important in many research disciplines
- Problems become more complex, thus need complex software and teams of researchers with diverse expertise working together
- HPC hardware is more complex, application performance depends on many factors
- Technology is also for increasing competitiveness

More & More Computing …



SW flow in science



Conventional Software Development Process

- Start with set of requirements defined by customer (or management):
  - features, properties, boundary conditions
- Typical Strategy:
  - Decide on overall approach on implementation
  - Translate requirements into individual subtasks
  - Use project management methodology to enforce timeline for implementation, validation and delivery
- Close project when requirements are met

What is Different in the Scientific Software Development Process?

- Requirements often are not that well defined
- Floating-point math limitations and the chaotic nature of some solutions complicate validation
- An application may only be needed once
- Few scientists are programmers (or managers)
- Often projects are implemented by students (inexperienced in science and programming)
- Correctness of results is a primary concern, less so the quality of the implementation

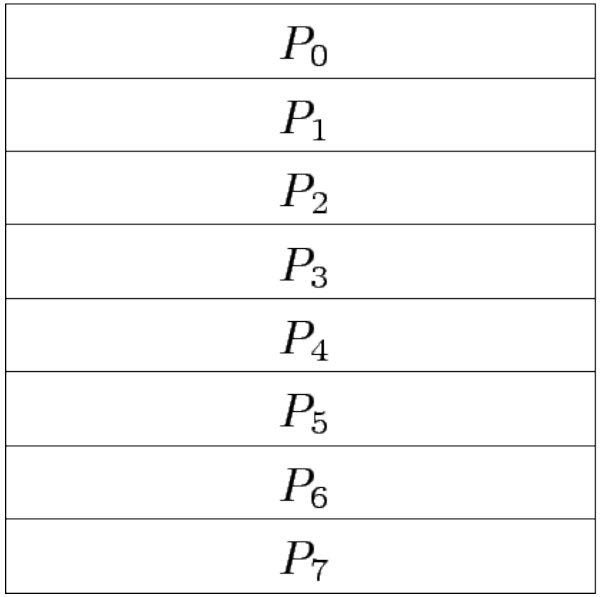
Parallelism - 101

- there are two main reasons to write a parallel program:
  - access to larger amount of memory (aggregated, going bigger)
  - reduce time to solution (going faster)

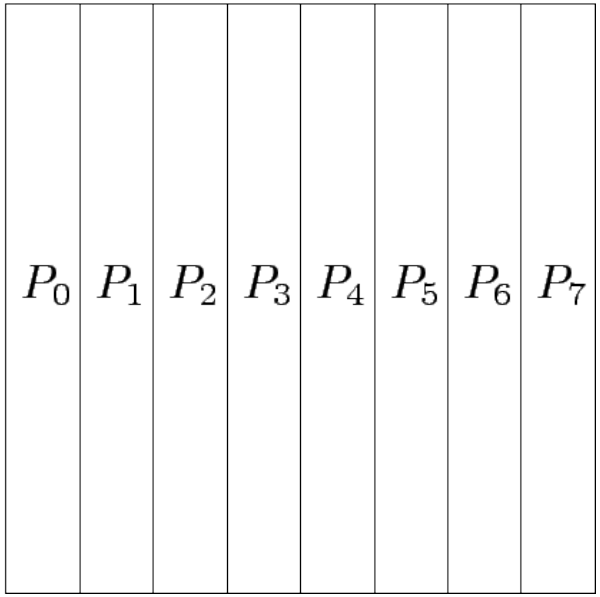
Static Data Partitioning

The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.

row-wise distribution



column-wise distribution

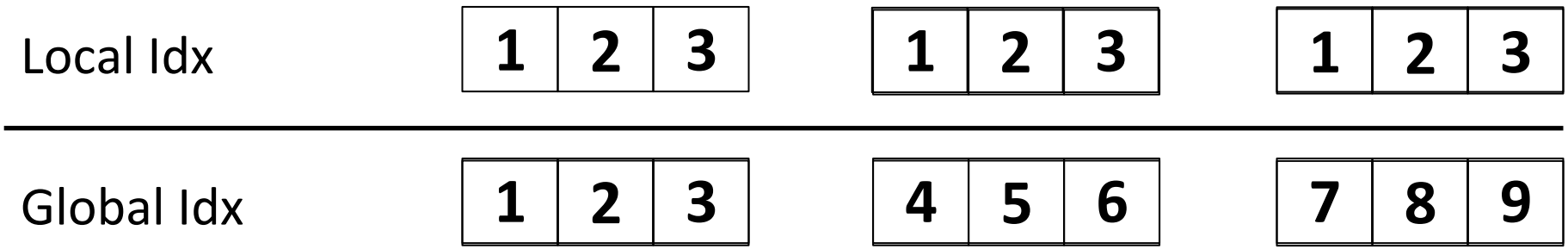


Distributed Data Vs Replicated Data

- Replicated data distribution is useful if it helps to reduce the communication among process at the cost of bounding scalability
- Distributed data is the ideal data distribution but not always applicable for all data-sets
- Usually complex application are a mix of those techniques => distribute large data sets; replicate small data

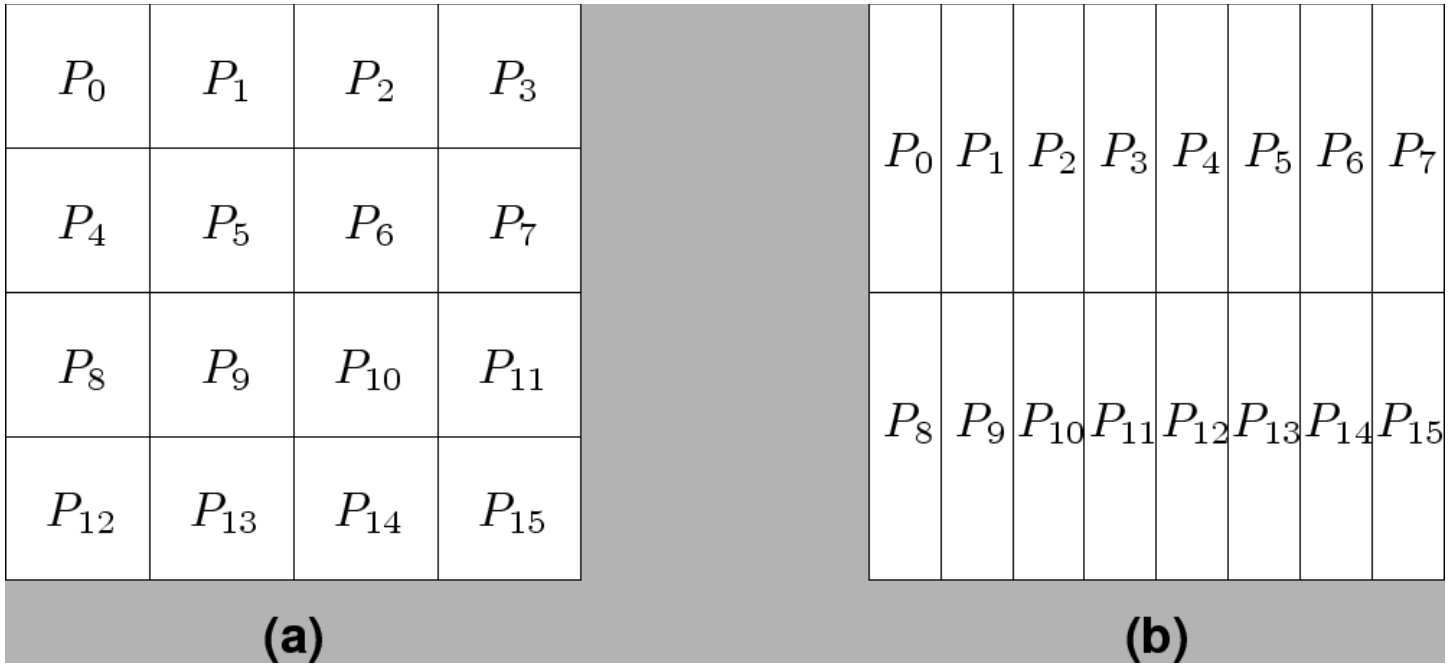
Global Vs Local Indexes

- In sequential code you always refer to global indexes
- With distributed data you must handle the distinction between global and local indexes (and possibly implementing utilities for transparent conversion)



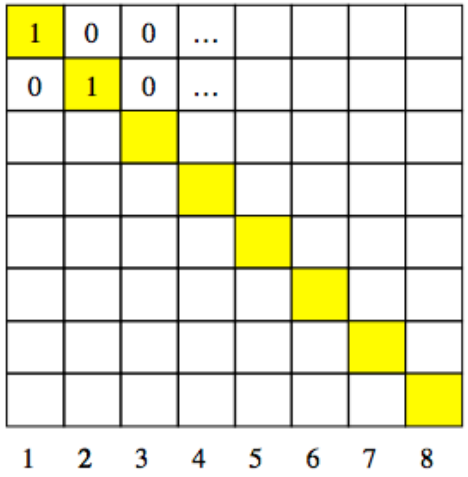
Block Array Distribution Schemes

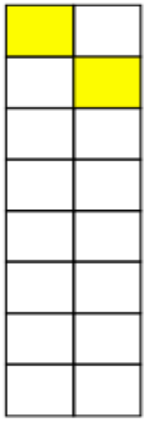
Block distribution schemes can be generalized to higher dimensions as well.

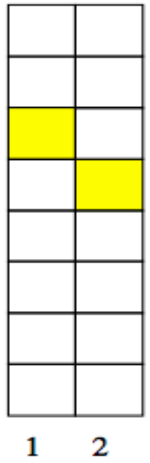


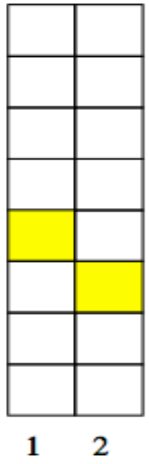
Degree to which tasks/data can be subdivided is limit to concurrency and parallel execution!!

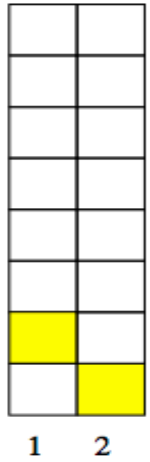
Collaterals to Domain Decomposition /1







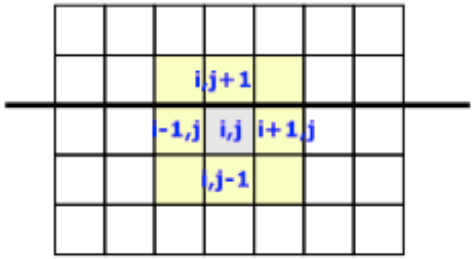




Are all the domain's dimensions always multiple of the number of tasks/processes we are willing to use?

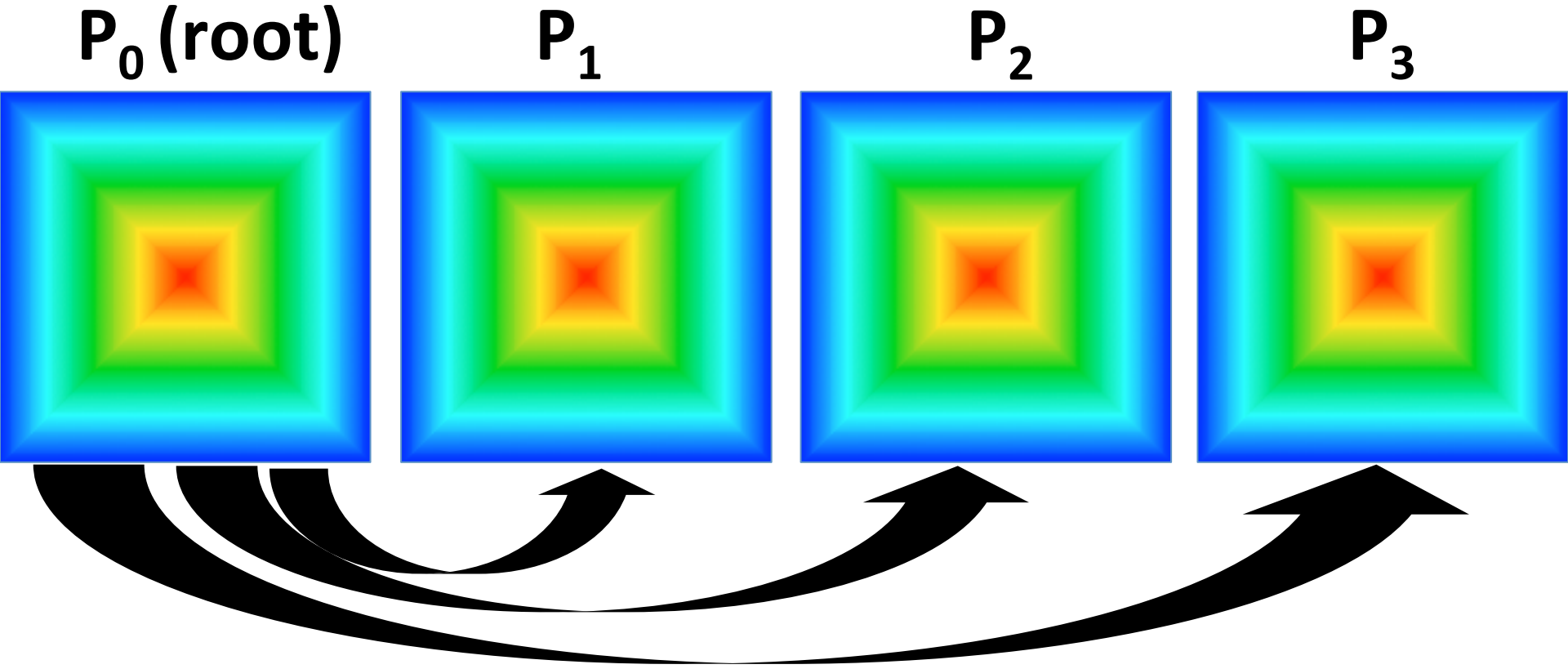
Again on Domain Decomposition

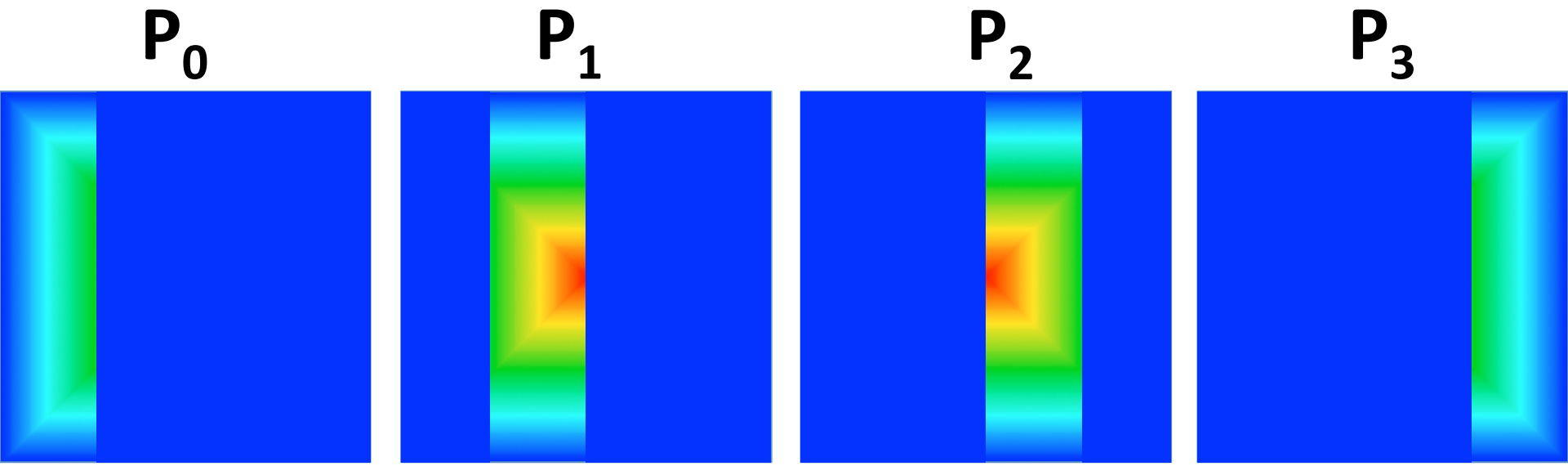
sub-domain boundaries



P_0

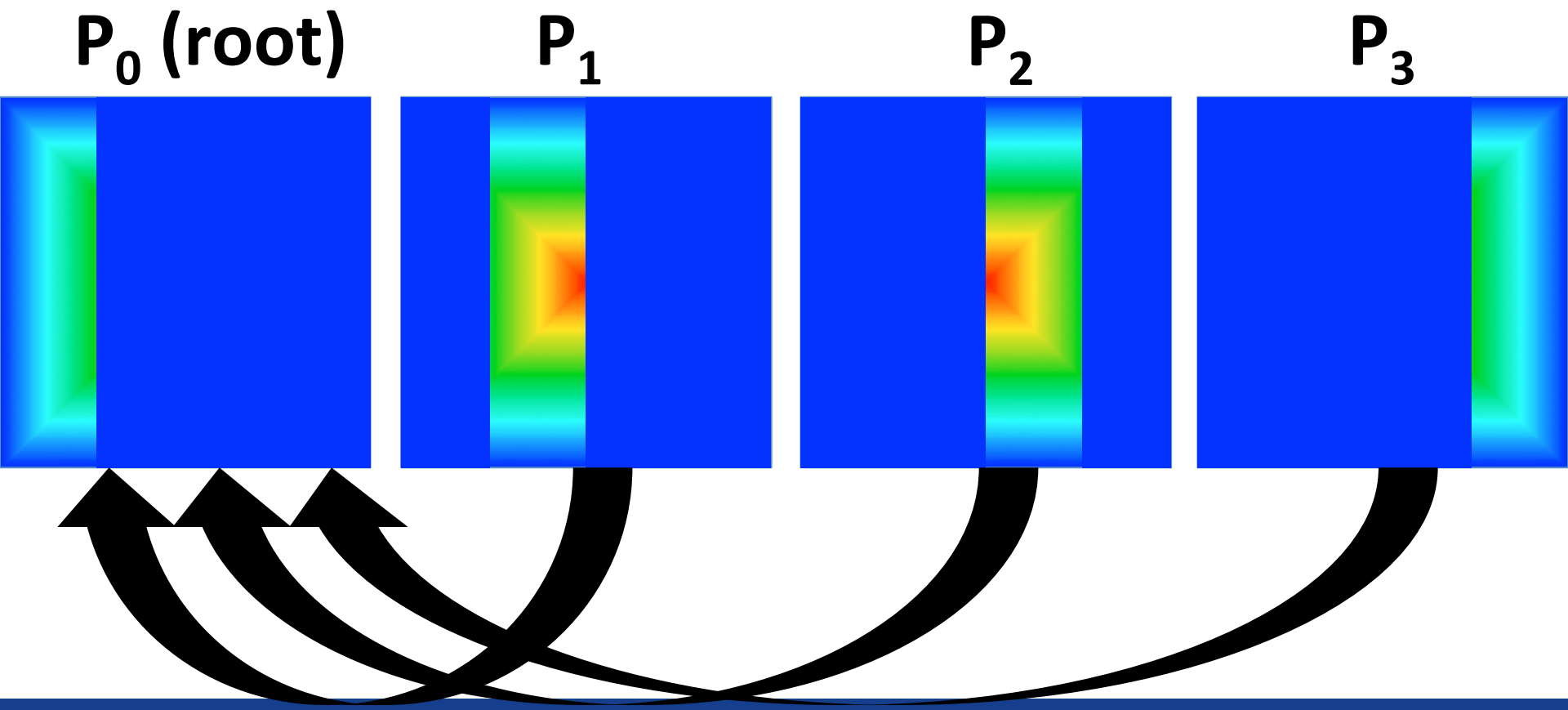
call MPI_BCAST(...)





call evolve(dtfact)

call MPI_Gather(…, …, …)

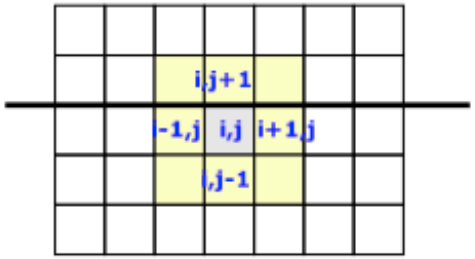


Replicated data

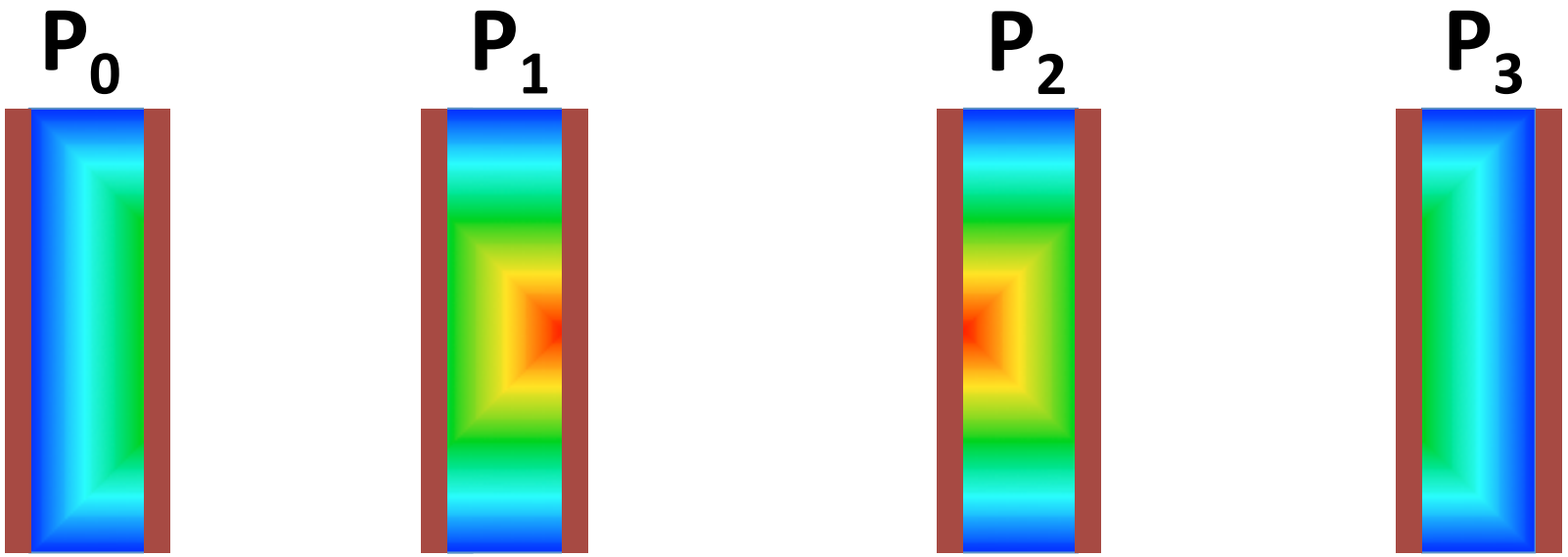
- Compute domain (and workload) distribution among processes
- Master-slaves: P_0 drives all processes
- Large amount of data communication
  - at each step P_0 distribute data to all processes and collect the contribution of each process
- Problem size scaling limited in memory capacity

Collaterals to Domain Decomposition /2

sub-domain boundaries

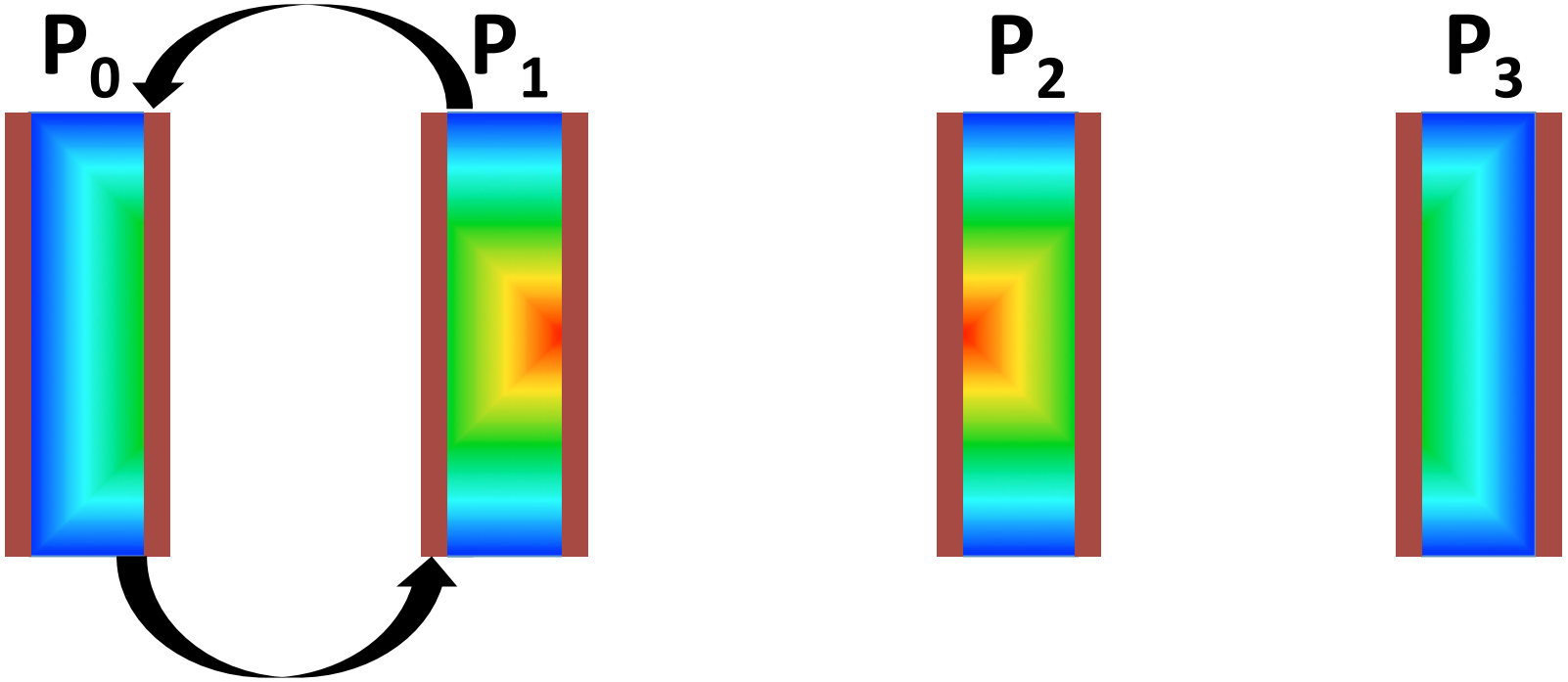


The Transport Code - Parallel Version

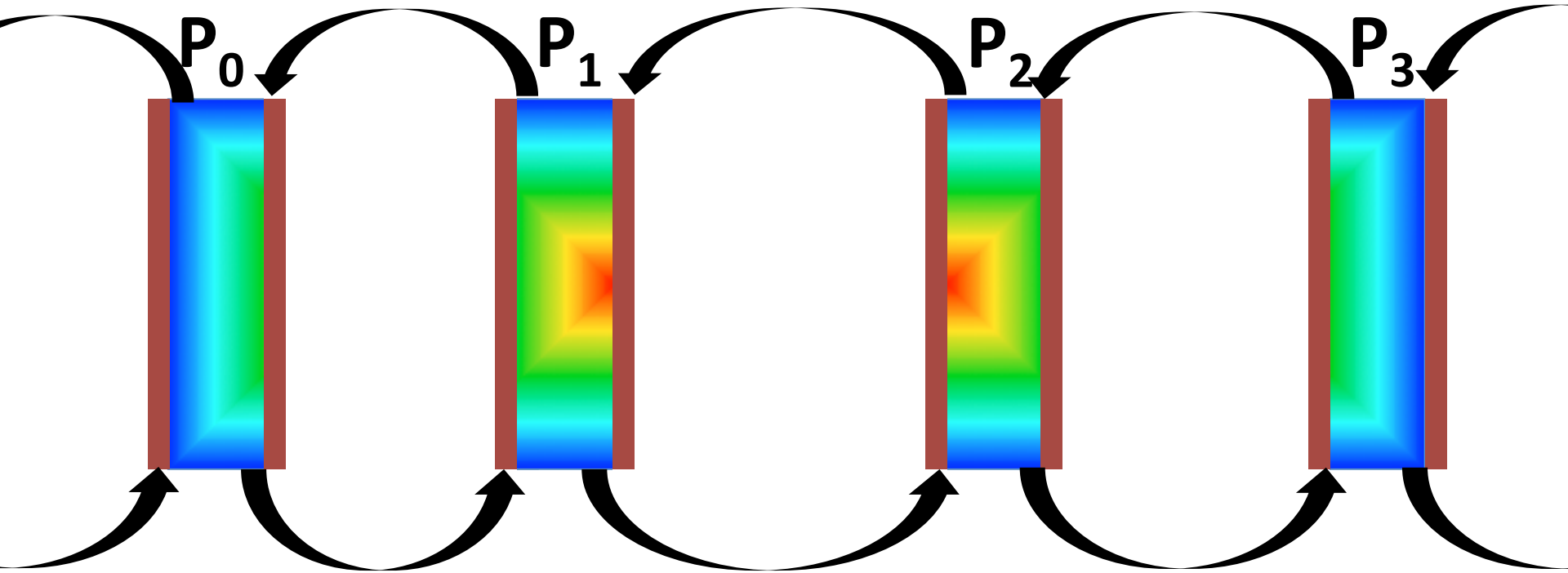


call evolve(dtfact)

Data exchange among processes



proc_down = mod(proc_me - 1 + nprocs , nprocs)

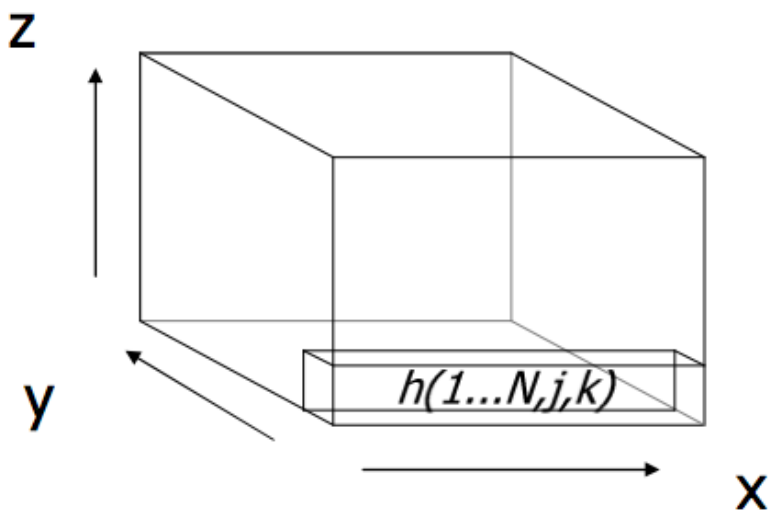


proc_up = mod(proc_me + 1 , nprocs)

Distributed Data

- Global and Local Indexes
- Ghost Cells Exchange Between Processes
  - Compute Neighbor Processes
- Parallel Output

Multidimensional FFT



$$f(x,y,z) = \frac{1}{N_z N_y N_x} \sum_{z=0}^{N_z-1} \Big(\sum_{y=0}^{N_y-1} \Big(\underbrace{\sum_{x=0}^{N_x-1} F(u,v,w)\, e^{-2\pi i \frac{xu}{N_x}}}_{\text{DFT long x-dimension}} \Big) e^{-2\pi i \frac{yv}{N_y}} \Big) e^{-2\pi i \frac{zw}{N_z}}$$

DFT long y-dimension

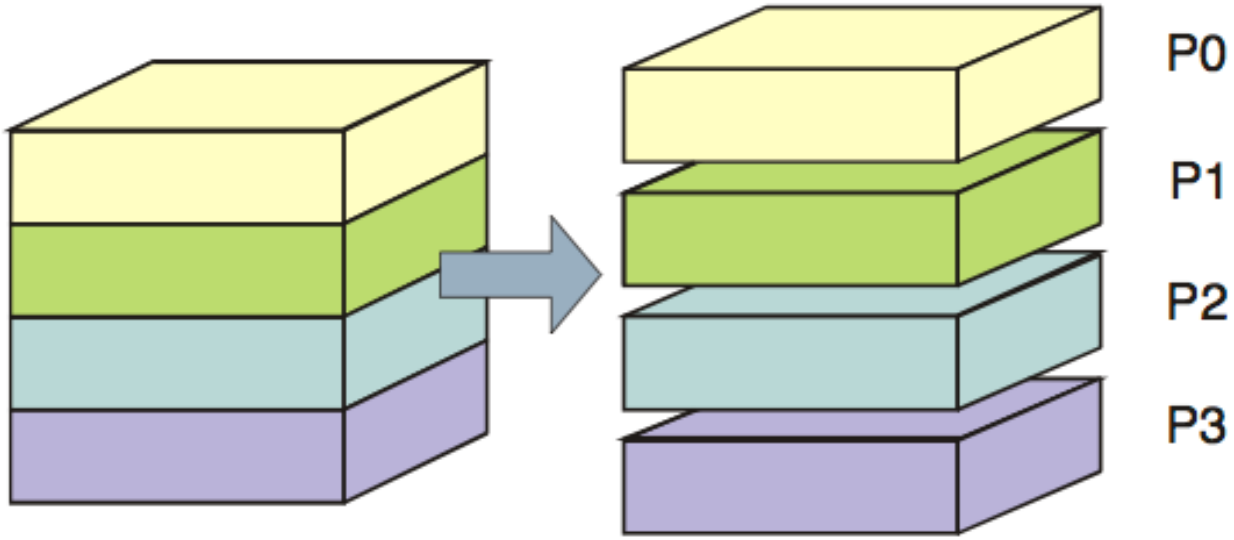
DFT long z-dimension

1) For any value of ***j*** and ***k*** transform the column (1…N, j, k)

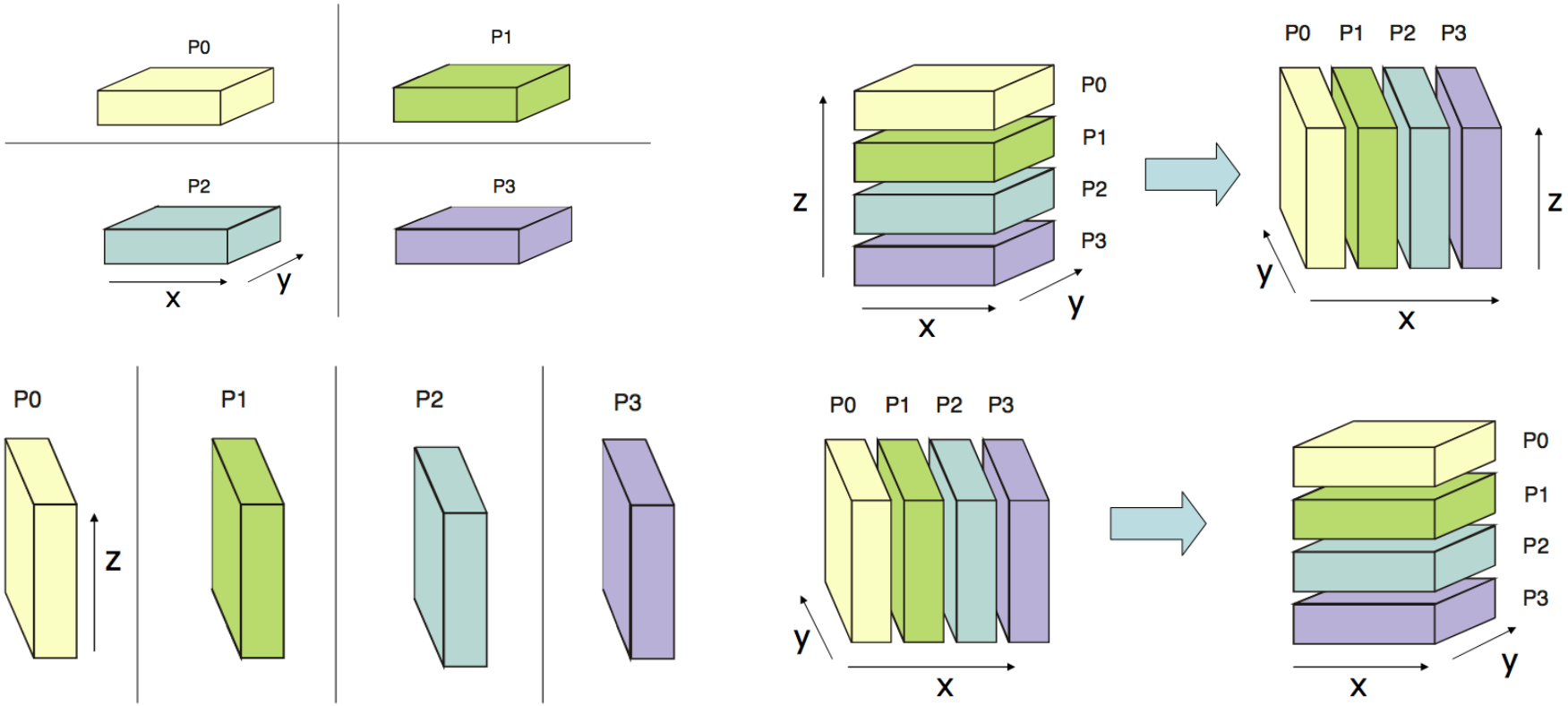
2) For any value of ***i*** and ***k*** transform the column (i, 1…N, k)

3) For any value of ***i*** and ***j*** transform the column (i, j, 1…N)

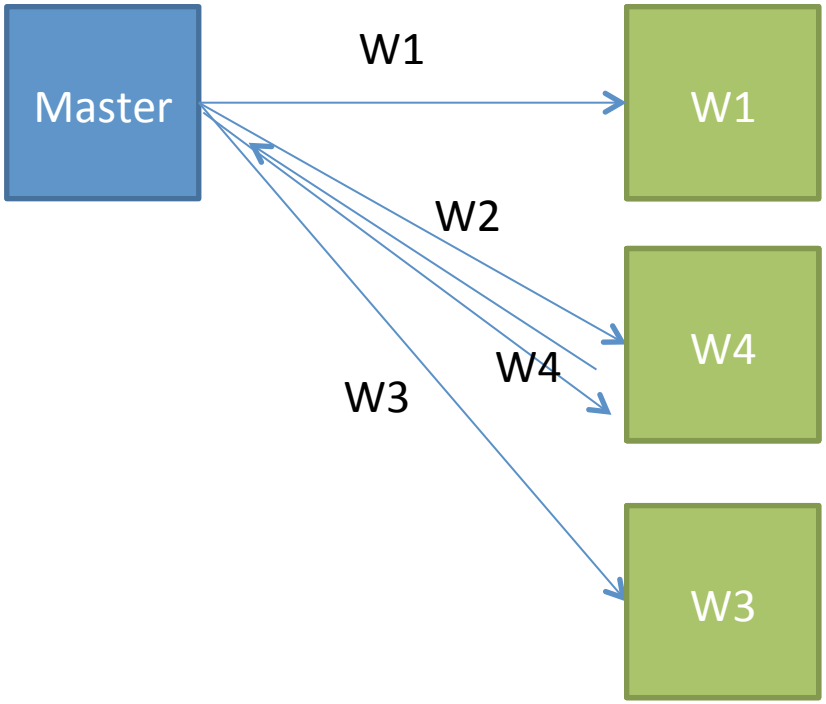
Parallel 3DFFT / 1



Parallel 3DFFT / 2



Master/Slave



Task Farming

- Many independent programs (tasks) running at once
  - each task can be serial or parallel
  - "independent" means they don't communicate directly
  - Processes possibly driven by the mpirun framework

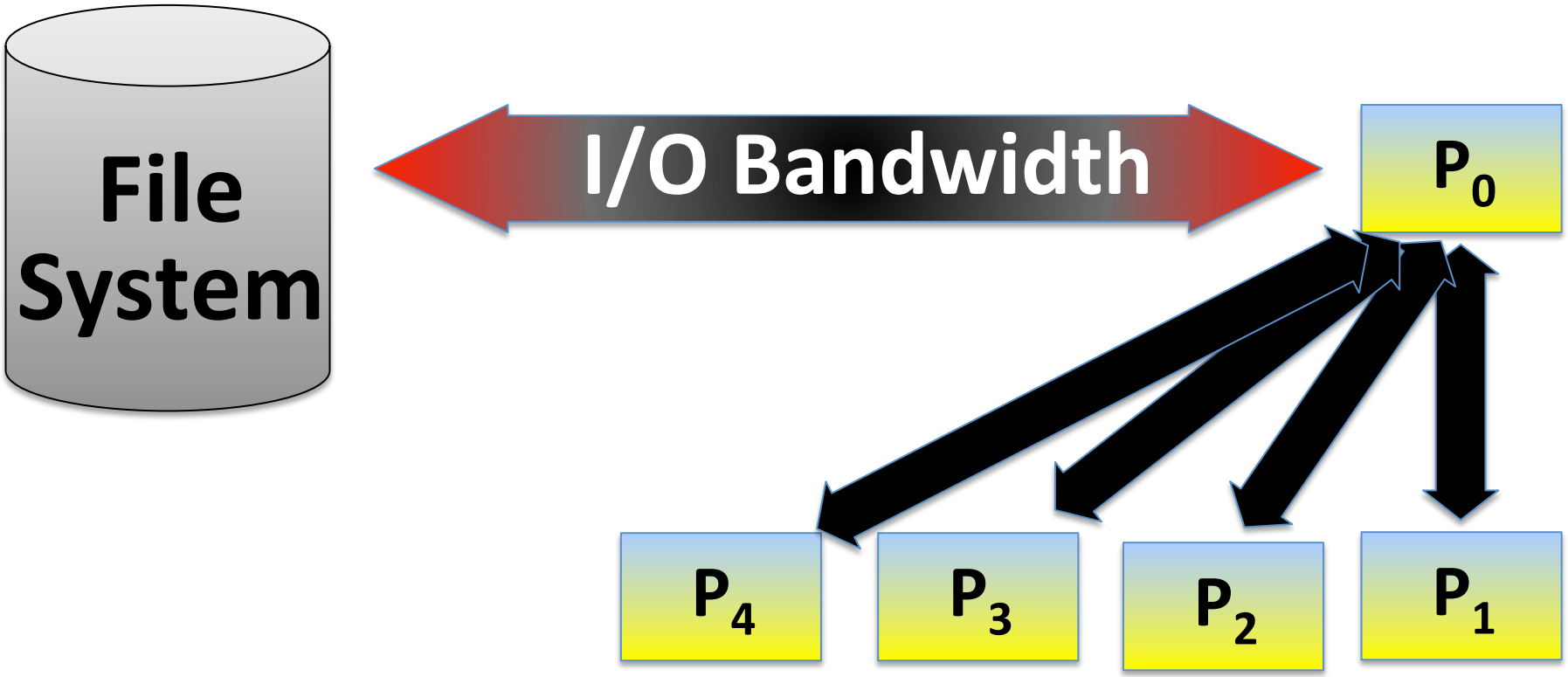
```
[igirotto@localhost]$ more my_shell_wrapper.sh
#!/bin/bash
#example for the OpenMPI implementation
./prog.x --input input_${OMPI_COMM_WORLD_RANK}.dat

[igirotto@localhost]$ mpirun -np 400 ./my_shell_wrapper.sh
```

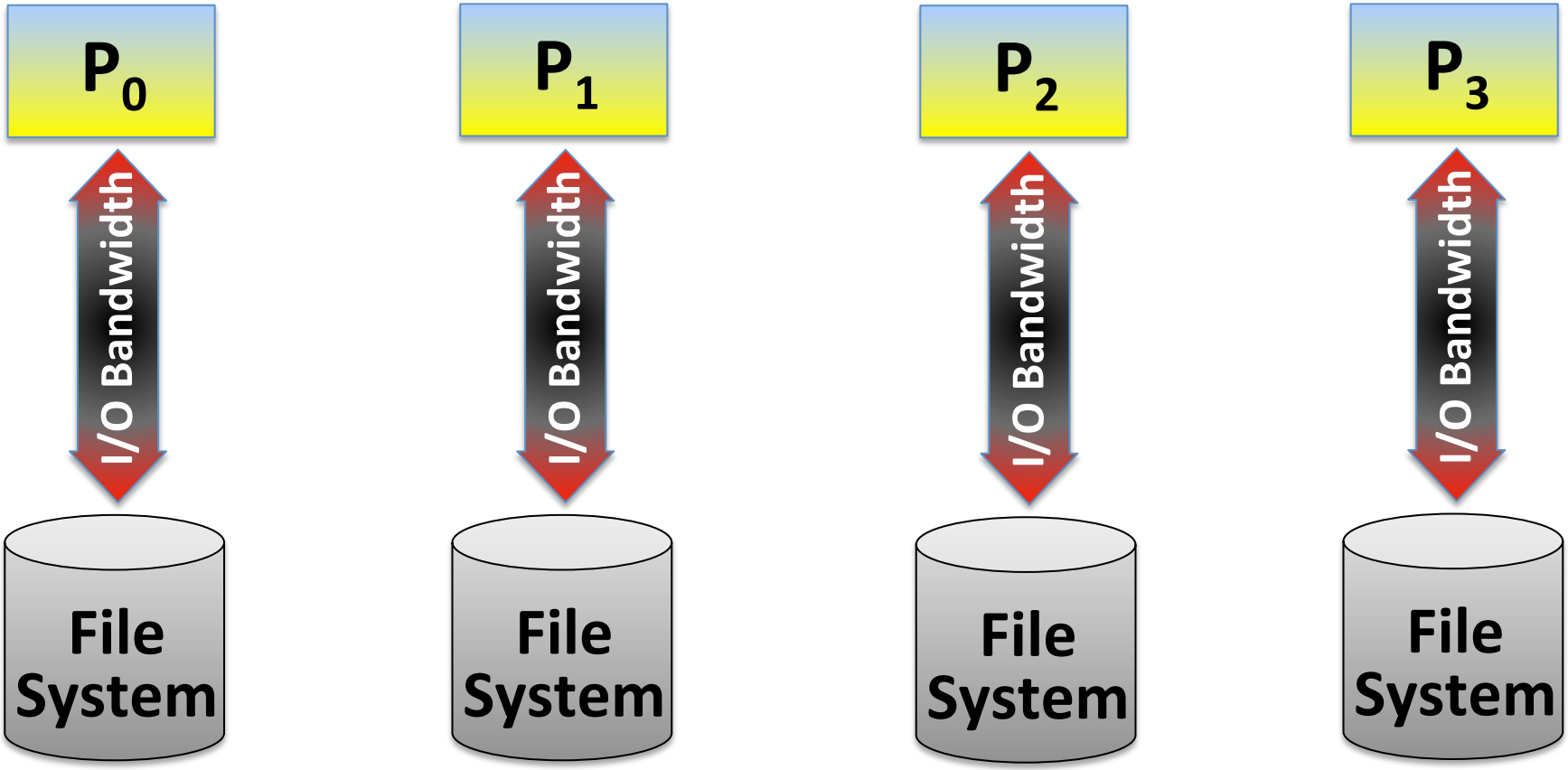
Easy Parallel Computing

- Farming, embarrassingly parallel
  - Executing multiple instances on the same program with different inputs/initial cond.
  - Reading large binary files by splitting the workload among processes
  - Searching elements on large data-sets
  - Other parallel execution of embarrassingly parallel problem (no communication among tasks)
- Ensemble simulations (weather forecast)
- Parameter space (find the best wing shape)

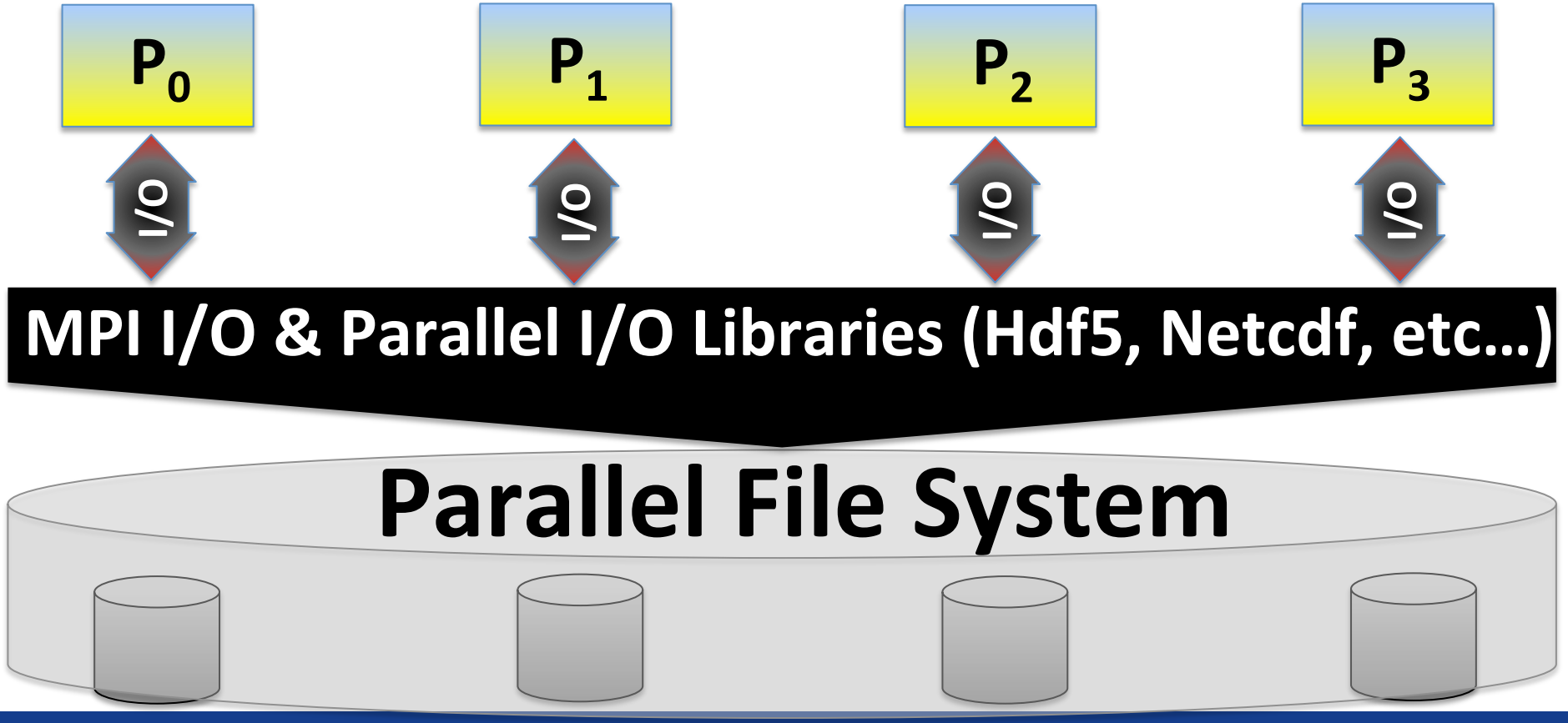
Parallel I/O



Parallel I/O


P0
P1
P2
P3
I/O Bandwidth
I/O Bandwidth
I/O Bandwidth
I/O Bandwidth
File System
File System
File System
File System

Parallel I/O


P0
P1
P2
P3
I/O
I/O
I/O
I/O
MPI I/O & Parallel I/O Libraries (Hdf5, Netcdf, etc…)
Parallel File System

Make Use Freely Available Parallel Libraries

- Scalable Parallel Random Number Generators Library (SPRNG)
- Parallel Linear Algebra (ScaLAPACK)
- Parallel Library for Solution of Finite Elements (dealii)
- Parallel Library for FFT (FFTW)
- Parallel Linear Solver for Sparce Matrices (PETSc)

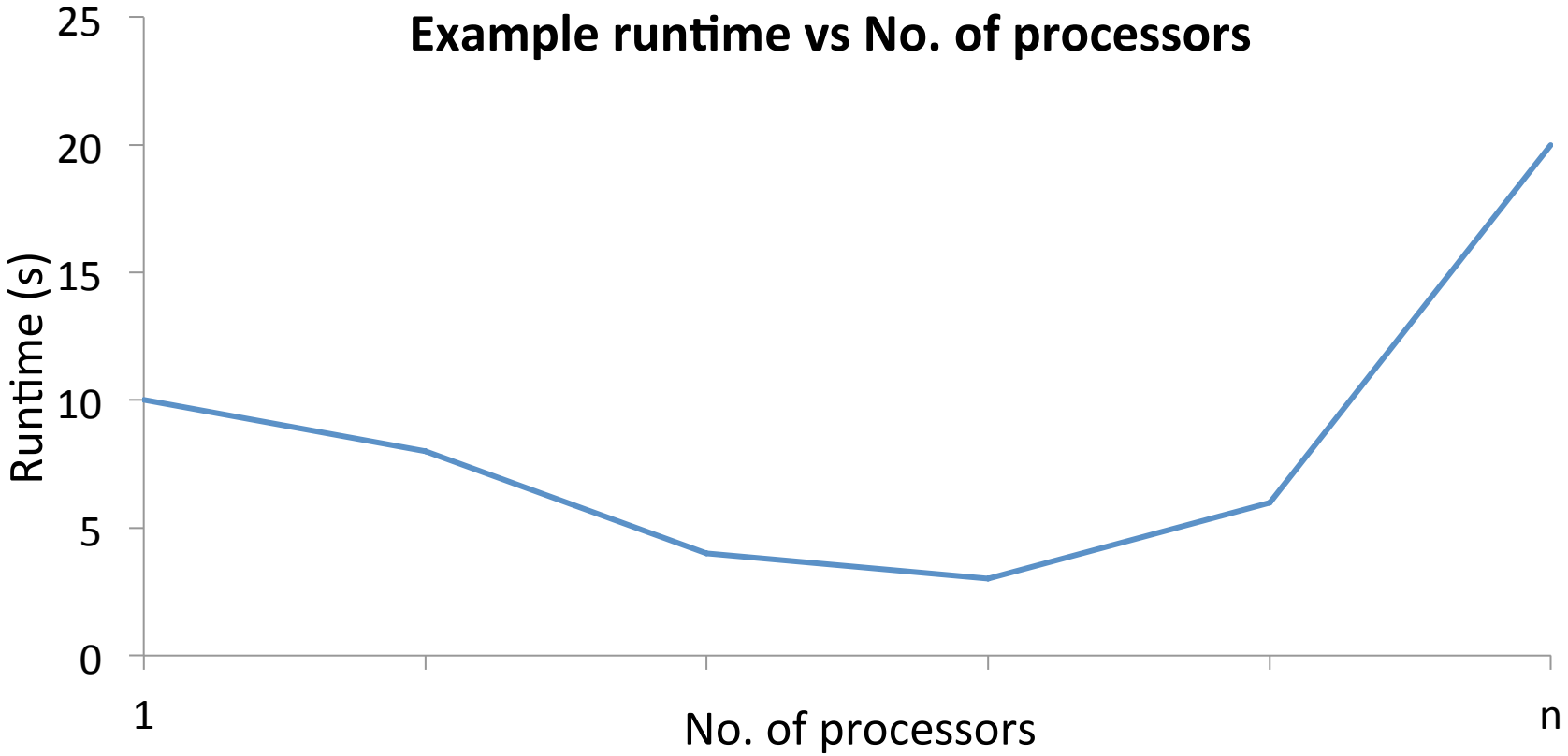
Measure of Parallelism

- evaluation of the performance improvement is not always straightforward, specially considering at the increasing the problem complexity

Scaling

- *Scaling* is how the performance of a parallel application changes as the number of processors is increased
- There are two different types of scaling:
  - *Strong Scaling* – total problem size stays the same as the number of processors increases
  - *Weak Scaling* – the problem size increases at the same rate as the number of processors, keeping the amount of work per processor the same
- Strong scaling is generally more useful and more difficult to achieve than weak scaling

Strong scaling

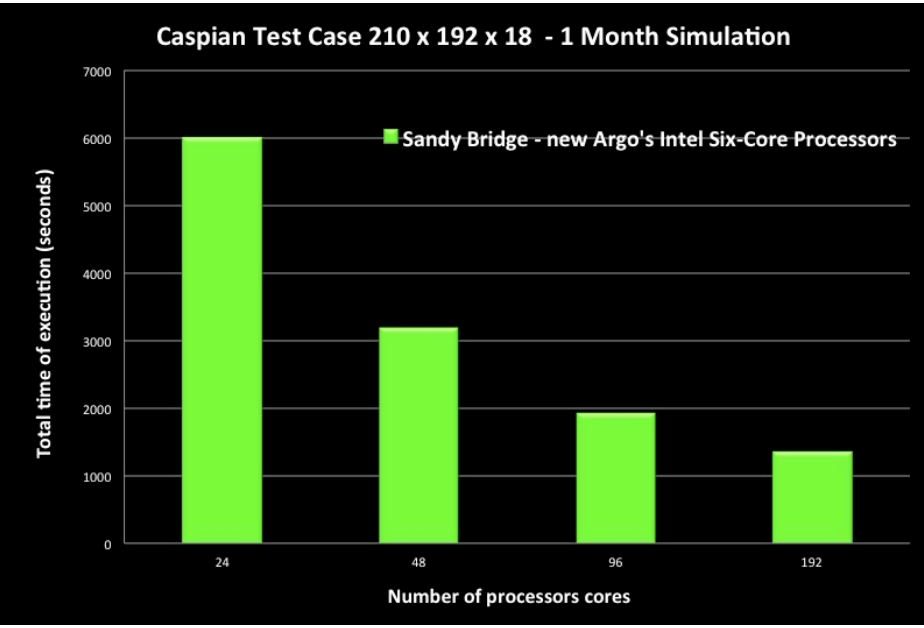


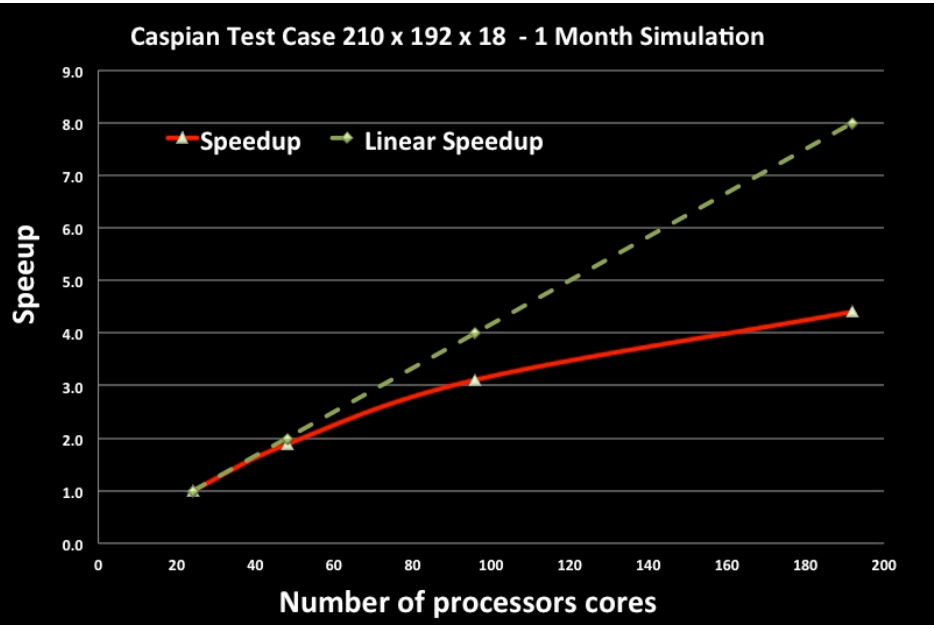
How do we evaluate the improvement?

- We want estimate the amount of the introduced overhead => $T_o = n_{pes} T_P - T_S$
- But to quantify the improvement we use the term **Speedup**:

$$S_P = \frac{T_S}{T_P}$$

Speedup


Caspian Test Case 210 x 192 x 18 - 1 Month Simulation
Total time of execution (seconds)
7000
6000
5000
4000
3000
2000
1000
0
Sandy Bridge - new Argo's Intel Six-Core Processors
24
48
96
192
Number of processors cores

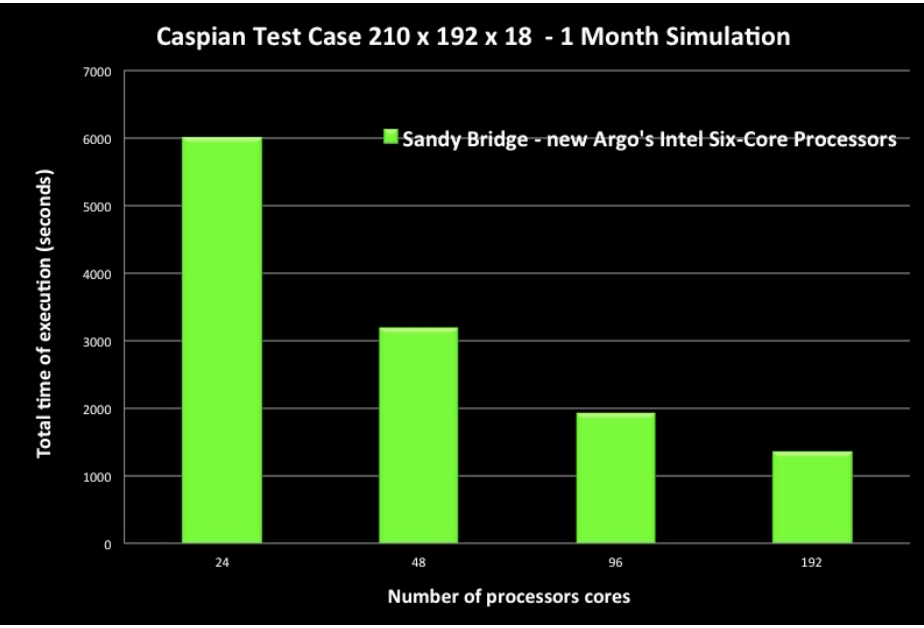

Caspian Test Case 210 x 192 x 18 - 1 Month Simulation
Speeup
9.0
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
Speedup
Linear Speedup
0
20
40
60
80
100
120
140
160
180
200
Number of processors cores

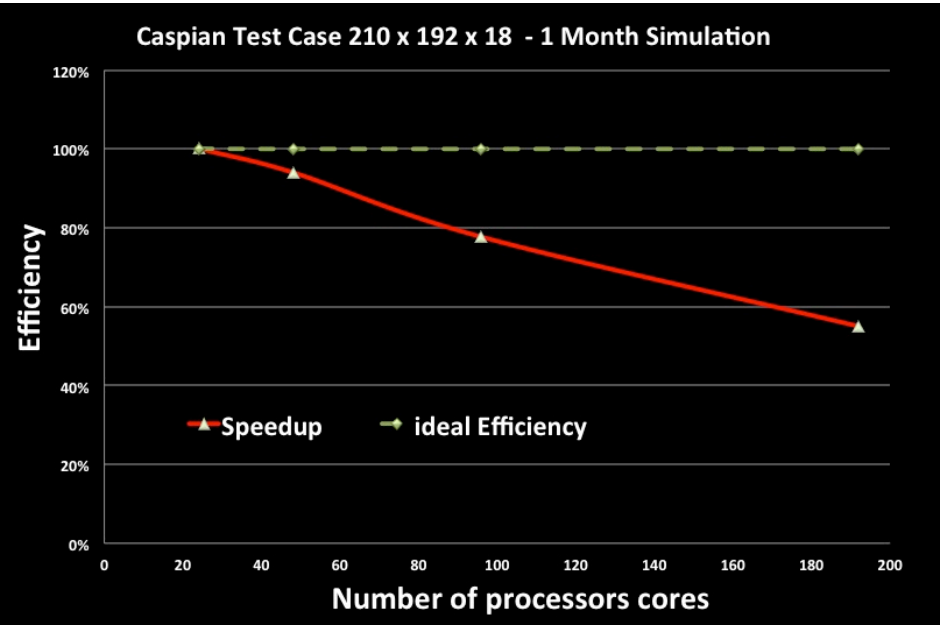
Efficiency

- Only embarrassing parallel algorithm can obtain an ideal Speedup
- The **Efficiency** is a measure of the fraction of time for which a processing element is usefully employed:

$$E_P = \frac{S_P}{p}$$

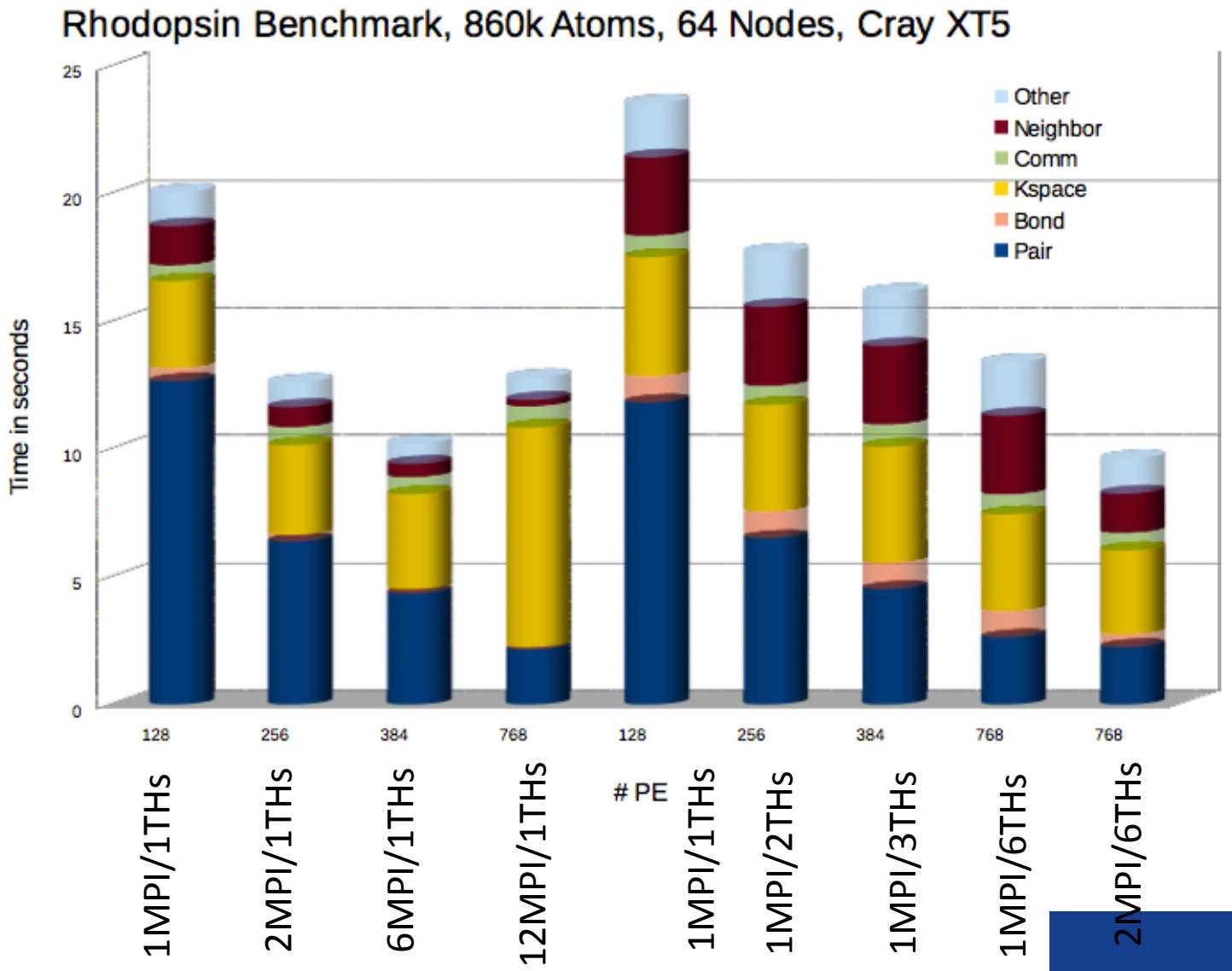
Efficiency


Caspian Test Case 210 x 192 x 18 - 1 Month Simulation
Total time of execution (seconds)
7000
6000
5000
4000
3000
2000
1000
0
Sandy Bridge - new Argo's Intel Six-Core Processors
24
48
96
192
Number of processors cores


Caspian Test Case 210 x 192 x 18 - 1 Month Simulation
Efficiency
120%
100%
80%
60%
40%
20%
0%
Speedup
ideal Efficiency
0
20
40
60
80
100
120
140
160
180
200
Number of processors cores

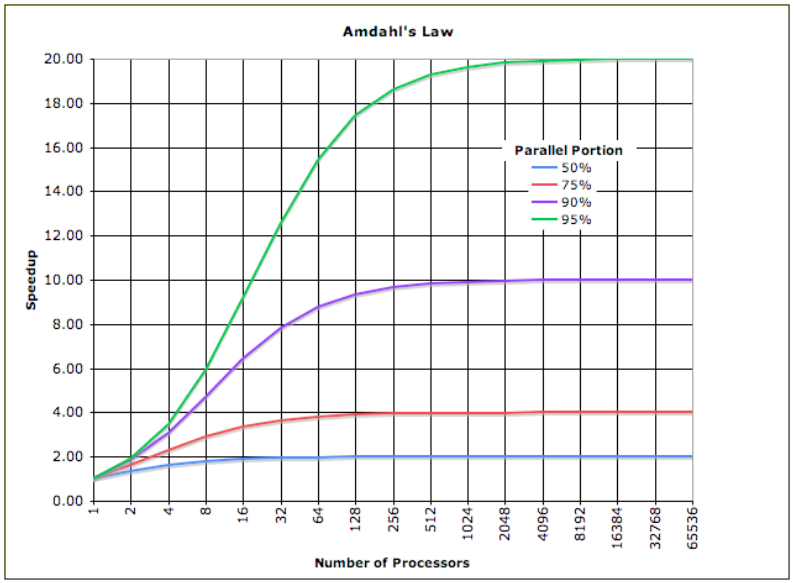
Scalability Limits

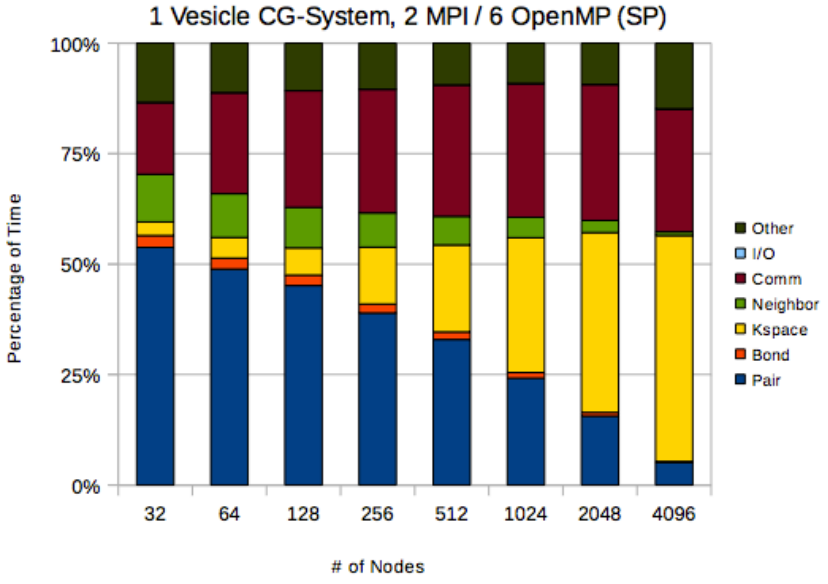
- Amdahl's law (there is a serial part which is not effected)
- Parallelism introduces overhead due to communication, synchronization, additional operations (I/O, memory copies, reordering, etc...)
- Sometimes the limit of scalability can be somehow predicted:
  - is there enough work for any process?
- When considering complex problem not all part of the problem scale eqaully


Rhodopsin Benchmark, 860k Atoms, 64 Nodes, Cray XT5
Time in seconds
0
5
10
15
20
25
Other
Neighbor
Comm
Kspace
Bond
Pair
128
256
384
768
128
256
384
768
768
PE
1MPI/1THs
2MPI/1THs
6MPI/1THs
12MPI/1THs
1MPI/1THs
1MPI/2THs
1MPI/3THs
1MPI/6THs
2MPI/6THs

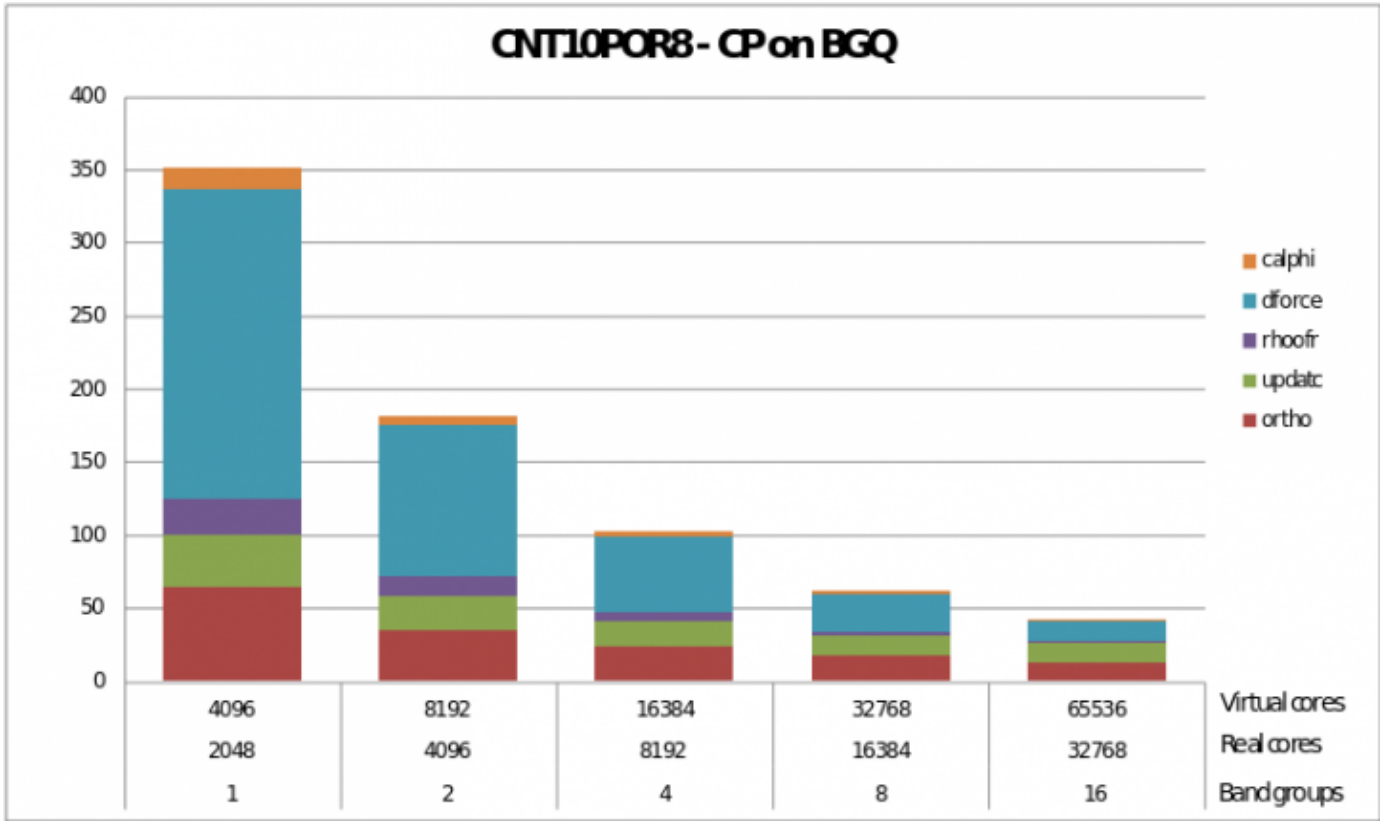
Amdal's Law And Real Life

- The speedup of a parallel program is limited by the sequential fraction of the program
- This assumes perfect scaling and no overhead

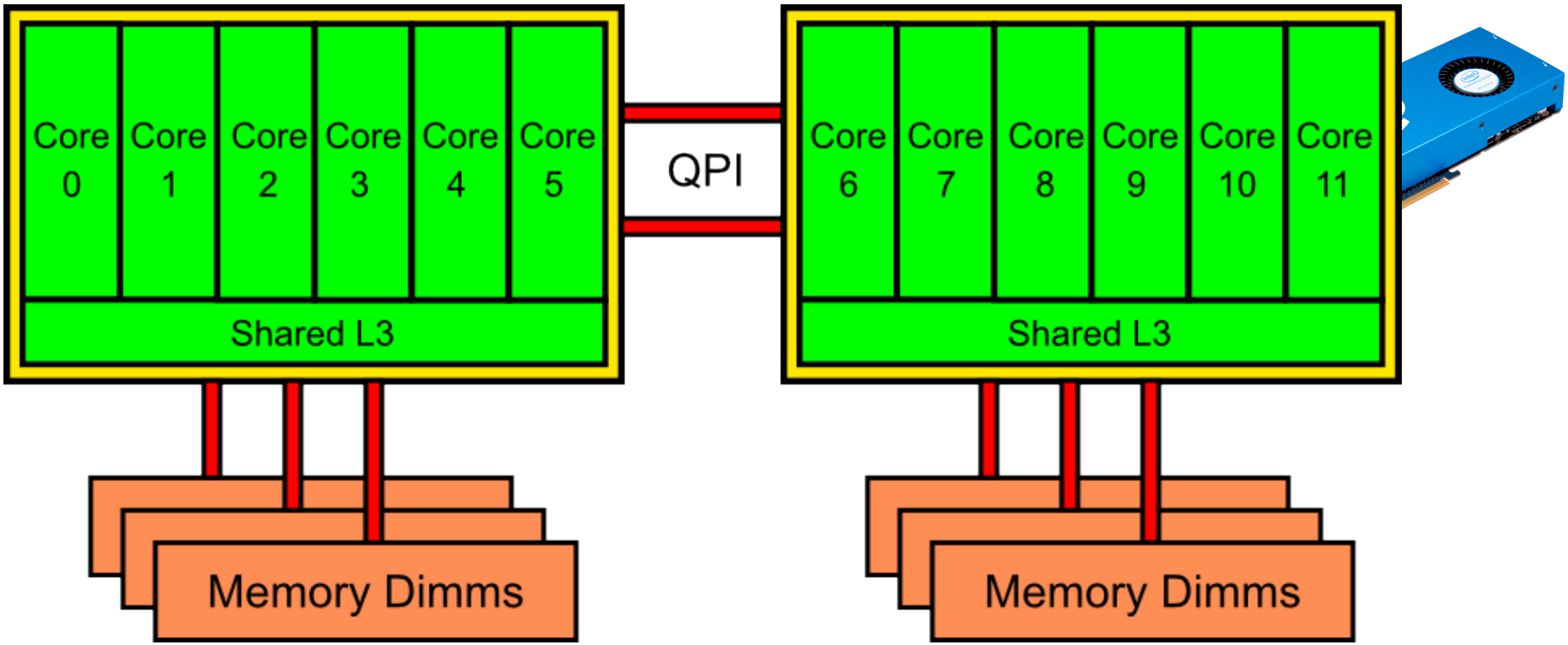




Scaling - QE-CP on Fermi BGQ @ CINECA



Multiple Socket CPUs + Acceleretors



Timing

- Time measurement becomes a relevant task:
- MPI_WTIME()
- Clock
- Seconds()

```
convergence NOT achieved after   5 iterations: stopping

Writing output data file c8_atm213_k111.save

init_run     :     93.79s CPU     93.79s WALL (       1 calls)
electrons    :    961.37s CPU    961.37s WALL (       1 calls)

Called by init_run:
wfcinit      :     69.37s CPU     69.37s WALL (       1 calls)
potinit      :      4.76s CPU      4.76s WALL (       1 calls)

Called by electrons:
c_bands      :    883.32s CPU    883.32s WALL (       5 calls)
sum_band     :     40.30s CPU     40.30s WALL (       5 calls)
v_of_rho     :      1.10s CPU      1.10s WALL (       6 calls)
mix_rho      :      1.51s CPU      1.51s WALL (       5 calls)

Called by c_bands:
init_us_2    :      0.50s CPU      0.50s WALL (      11 calls)
cegterg      :    882.01s CPU    882.01s WALL (       5 calls)

Called by *egterg:
h_psi        :    259.11s CPU    259.11s WALL (      17 calls)
g_psi        :      9.02s CPU      9.02s WALL (      11 calls)
cdiaghg      :    401.37s CPU    401.37s WALL (      16 calls)

Called by h_psi:
add_vuspsi   :     22.44s CPU     22.44s WALL (      17 calls)

General routines
calbec       :     17.25s CPU     17.25s WALL (      17 calls)
fft          :      0.52s CPU      0.52s WALL (      66 calls)
ffts         :      0.63s CPU      0.63s WALL (     117 calls)
fftw         :    231.61s CPU    231.61s WALL (   10260 calls)
davcio       :      4.72s CPU      4.72s WALL (       5 calls)

Parallel routines
fft_scatter  :     63.50s CPU     63.51s WALL (   10443 calls)
ALLTOALL     :     10.66s CPU     10.67s WALL (   10252 calls)
EXX routines

PWSCF        : 17m42.94s CPU    17m42.94s WALL
```

```
convergence NOT achieved after   5 iterations: stopping

Writing output data file c8_atm213_k111.save

init_run     :    119.48s CPU    119.48s WALL (       1 calls)
electrons    :   1369.53s CPU   1369.53s WALL (       1 calls)

Called by init_run:
wfcinit      :     98.55s CPU     98.55s WALL (       1 calls)
potinit      :      2.15s CPU      2.15s WALL (       1 calls)

Called by electrons:
c_bands      :   1289.41s CPU   1289.41s WALL (       5 calls)
sum_band     :     56.06s CPU     56.06s WALL (       5 calls)
v_of_rho     :      1.39s CPU      1.39s WALL (       6 calls)
mix_rho      :      1.23s CPU      1.23s WALL (       5 calls)

Called by c_bands:
init_us_2    :      0.13s CPU      0.13s WALL (      11 calls)
cegterg      :   1288.89s CPU   1288.89s WALL (       5 calls)

Called by *egterg:
h_psi        :    409.59s CPU    409.59s WALL (      17 calls)
g_psi        :      2.35s CPU      2.35s WALL (      11 calls)
cdiaghg      :    528.61s CPU    528.61s WALL (      16 calls)

Called by h_psi:
add_vuspsi   :     32.96s CPU     32.96s WALL (      17 calls)

General routines
calbec       :     31.22s CPU     31.22s WALL (      17 calls)
fft          :      0.62s CPU      0.62s WALL (      66 calls)
ffts         :      0.86s CPU      0.86s WALL (     117 calls)
fftw         :    376.02s CPU    376.04s WALL (   82004 calls)
davcio       :      6.38s CPU      6.38s WALL (       5 calls)

Parallel routines
fft_scatter  :     81.64s CPU     81.65s WALL (   82187 calls)

PWSCF        : 24m57.48s CPU    24m57.48s WALL


This run was terminated on:  12:25:36  12Oct2012
```

How do we handle that?

- Write more modular, more reusable software
  - build frameworks and libraries
- Write software that can be modified on an abstract level or where components can be combined without having to recompile
  - combine scripting with compiled code
- Write software where all components are continuously (re-)tested and (re-)validated
- Write software where consistent documentation is integral part of the development process

What makes such projects successful?

- Success or failure of scientific software projects is not decided on technical merit alone
- The true factors are beyond the code! It is not enough to be a good programmer! In particular, what counts:
  - Utility and quality
  - Documentation
  - Community
- All of the big libraries/packages provide this for their users.

Thanks for your attention!!