EFDC – High Performance Computing and MPI Parallelization

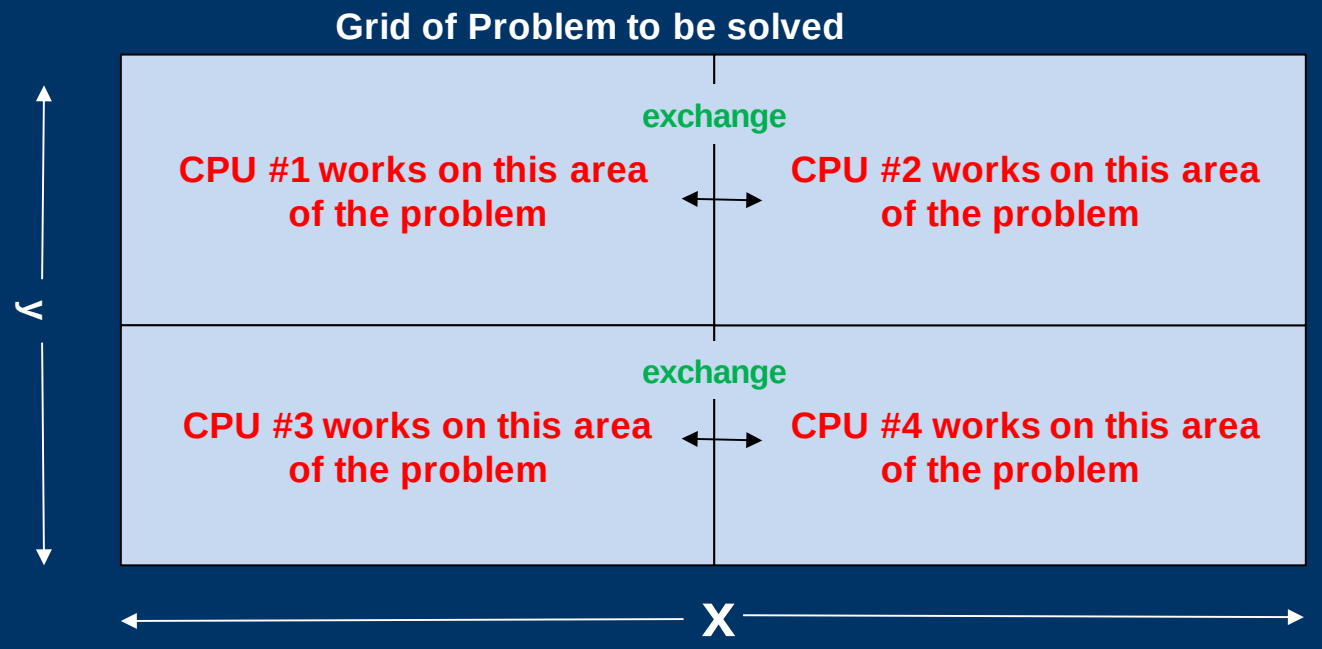
Fearghal O'Donncha

What is High-Performance Computing (HPC)

- First HPC systems were vector based systems
  - Named “supercomputers” because they were an order of magnitude more powerful than commercial systems
  - Very complex architecture; custom built or produced as per order
- Now, “supercomputer” has little meaning
  - Large systems are just scaled up versions of smaller systems
  - Performance measured in terms of FLOPS
  - Alternative definition: fastest computer at any point in time
- HPC practical definition: *any* computational technique that solves a large problem faster than possible using *single, commodity* systems
    - Custom-designed, high-performance processors
    - Parallel computing
    - Grid or distributed computing – collection of compute resources from multiple locations

What is Parallel Computing

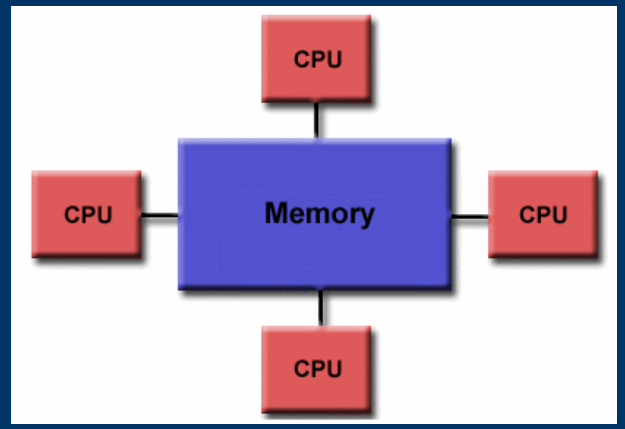
- Parallel computing: the use of multiple computers or processors working together on a common task
  - Each processor works on its section of the problem
  - Processors exchange information with one another



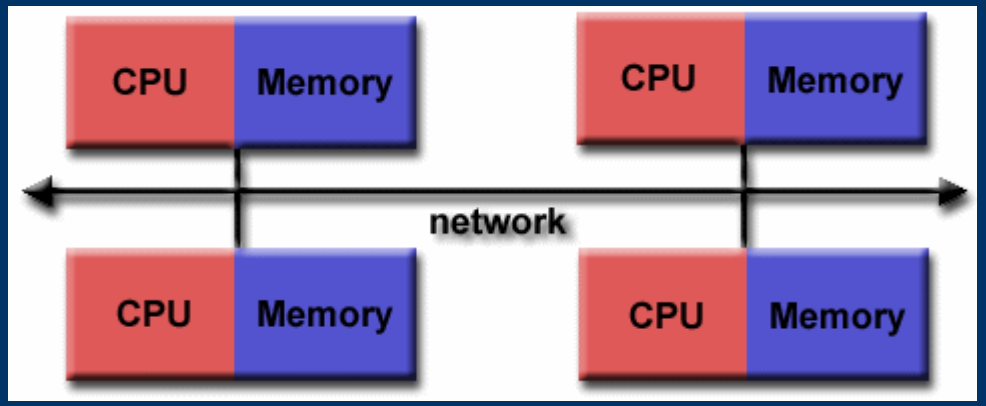
Why do Parallel computing

- Limits of single-CPU computing:
  - Available memory
  - Performance
- Parallel computing allows:
  - Solve problems that don't fit on a single CPU's memory space
  - Solve problems that can't be solved in a reasonable time
- We can run…
  - Larger problems
  - Faster
  - More cases
  - Run simulations at finer resolution
  - Model physical phenomena more realistically

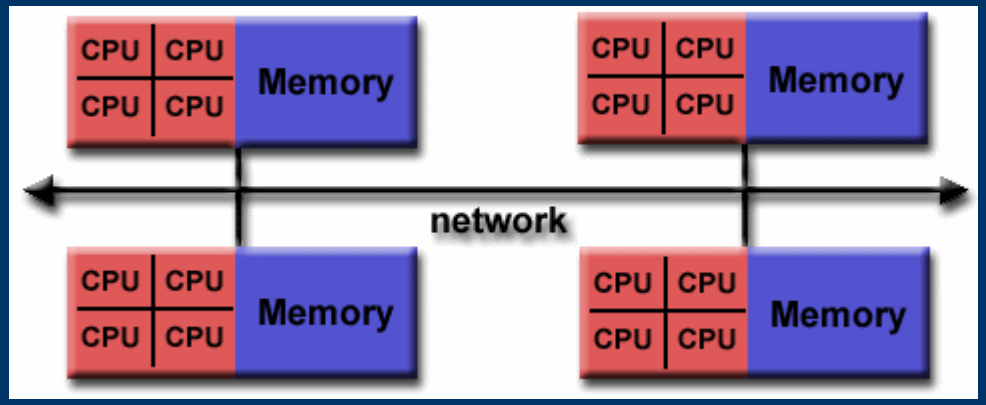
Modern parallel computers are classified by their memory model



Shared memory - single address space. All processors have access to a pool of shared memory.



Distributed memory - each processor has it's own local memory. Must do message passing to exchange data between processors.



Hybrid distributed-shared memory
The largest and fastest computers in the world today employ both shared and distributed memory architectures.

Programming methodologies

- Shared-memory model
  - Standard Fortran or C and let the compiler do it for you
  - Directive can give hints to compiler (OpenMP)
    - Loop-level parallelism
  - Libraries (OpenBlas, ScaLAPACK)
  - Thread-like methods
  - Message passing can also be used but is not optimal

- Distributed memory model
  - Mostly message passing using MPI
  - Data distribution Languages
    - data structure is split up and resides as "chunks" in the local memory of each task.

Parallel computing and ocean modelling

- Legacy code, typically FORTRAN 77 or 90
- Community models
- Incremental model evolution
- Development relies heavily on physical oceanographers and engineers
- Distinction between computer science and natural science components is blurry
- Modularity of code not always respected
- “Developability” of the model is crucial – complexity and readability
- Optimized for specific deployment environments
- Load-balancing across cores often neglected
- Focus on “weak scaling” of model

Case study – Legacy code to deploy on blades cluster

The problem:

- High computational cost of advanced numerical model codes + expensive data assimilation schemes + water-quality monitoring = barrier to real-time predictive models

Why?

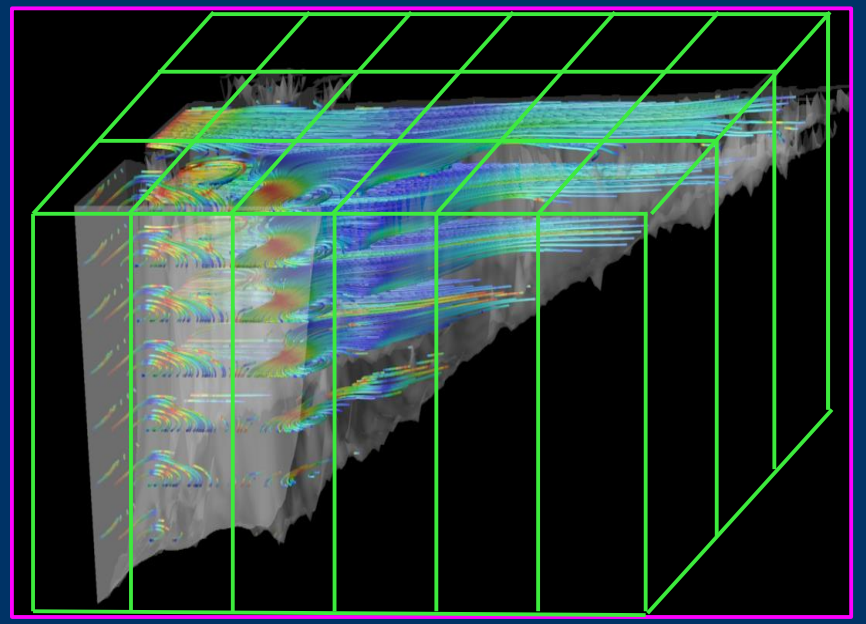
- Practical ability of numerical models to solve real-world problems without sacrificing actual physical phenomena (e.g., small-scale flow processes) constrained; complexity of assimilation schemes restricted.

The Approach:

- Paralellization using message passing (MPI) and domain decomposition
- Surgical changes to existing code
- Deployment on commodity blade cluster
- Targeted load-balancing algorithm
  - Aim: maximise output not “balance load”

Methodology

- Legacy Fortran 77 consisting of approx. 50,000 lines of code and 300 source files
  - Complex structure with multiple application designs
- Surgical changes of existing code a necessity
- Parallelization using MPI and tiling in X,Y, but not Depth
- Most of code partitions cleanly with neighbour-to-neighbour communication
- One routine contains an iterative solver (conjugate gradient)



Conjugate gradient

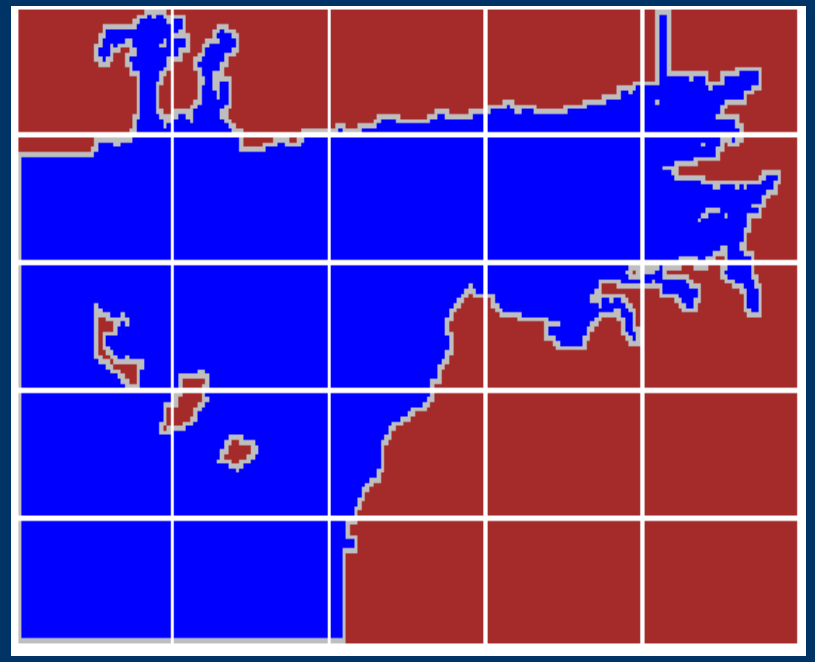
- Numerically efficient method to compute free-surface elevation
  - Iterative procedure until residuals sum below a predefined threshold
- Efficient projection onto parallel computers more difficult due to inherent global communication
- Preliminary concerns to feasibility of parallelization project and expected speedup
- Two approaches considered to address the problem:

1) Parallel ConjGrad
   - Neighbour-to-neighbour communication at each iteration
   - MPI summation checks for convergence
   - Typically 10-40 iterations required for convergence

2) Serial ConjGrad
   - Gather surface data to master node
   - Compute ConjGrad at master
   - Scatter solution to nodes

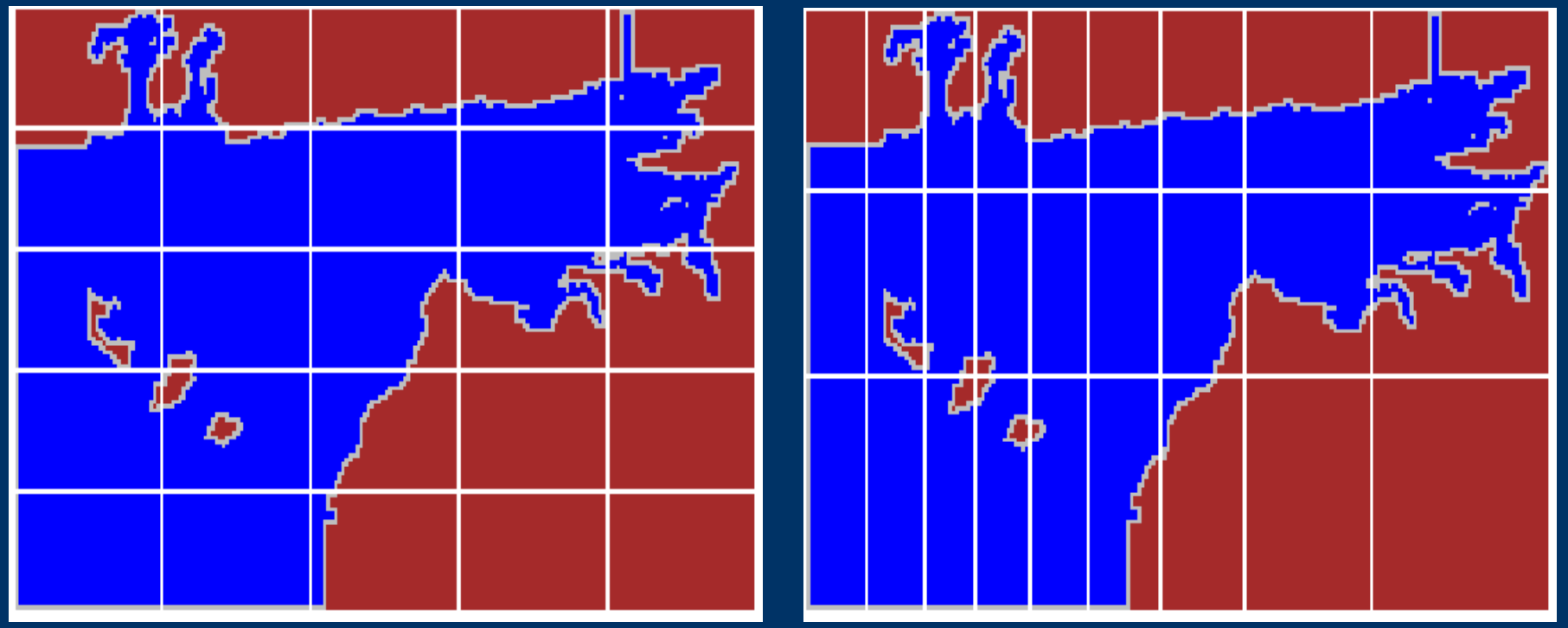
Domain partitioning

- Complicated, irregular mix of water and land with very different computational requirements ranging from completely water to completely land
    - Land cells does not require computation of solution
- Focus on minimizing the maximum load on any processor as opposed to equal distribution, i.e., minimizing time per timestep as opposed to traditional load balancing
- Naïve, equal partitioning will have some processors doing very little or no work (i.e., domains that are entirely land) while others have extremely high computational expense
- In the example on right, the four domains in the bottom right are entirely land (zero computational expense) while domains in the bottom left are entirely water

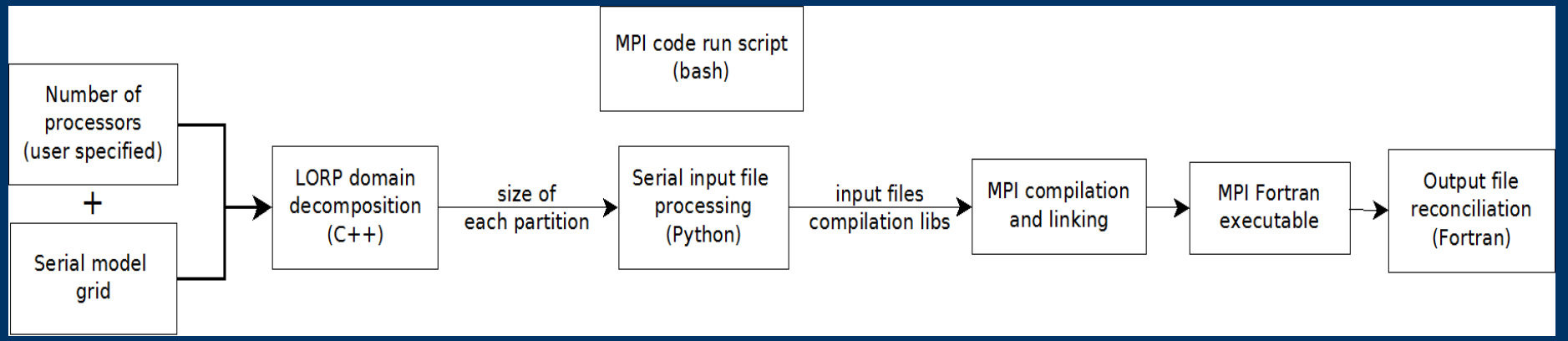


Load balancing: Locally Optimal Rectilinear Partitioning (LORP)

- Starting from a roughly balanced initial partition, adjustments in the partition dividers are made in progressively smaller steps to reduce the cost of the most expensive tile while keeping the number of active tiles equal to the intended processor count
- For a target of 25 processors, the LORP partition is 9×3 with two inactive tiles removed from computation
- Rectilinear domain decomposition demonstrating: (a) a fixed partition assigning equal number of cells to each partitioning and (b) Locally Optimal Rectilinear Partitioning. Note that (a) has four inactive tiles consisting entirely of land while (b) has two but they are larger

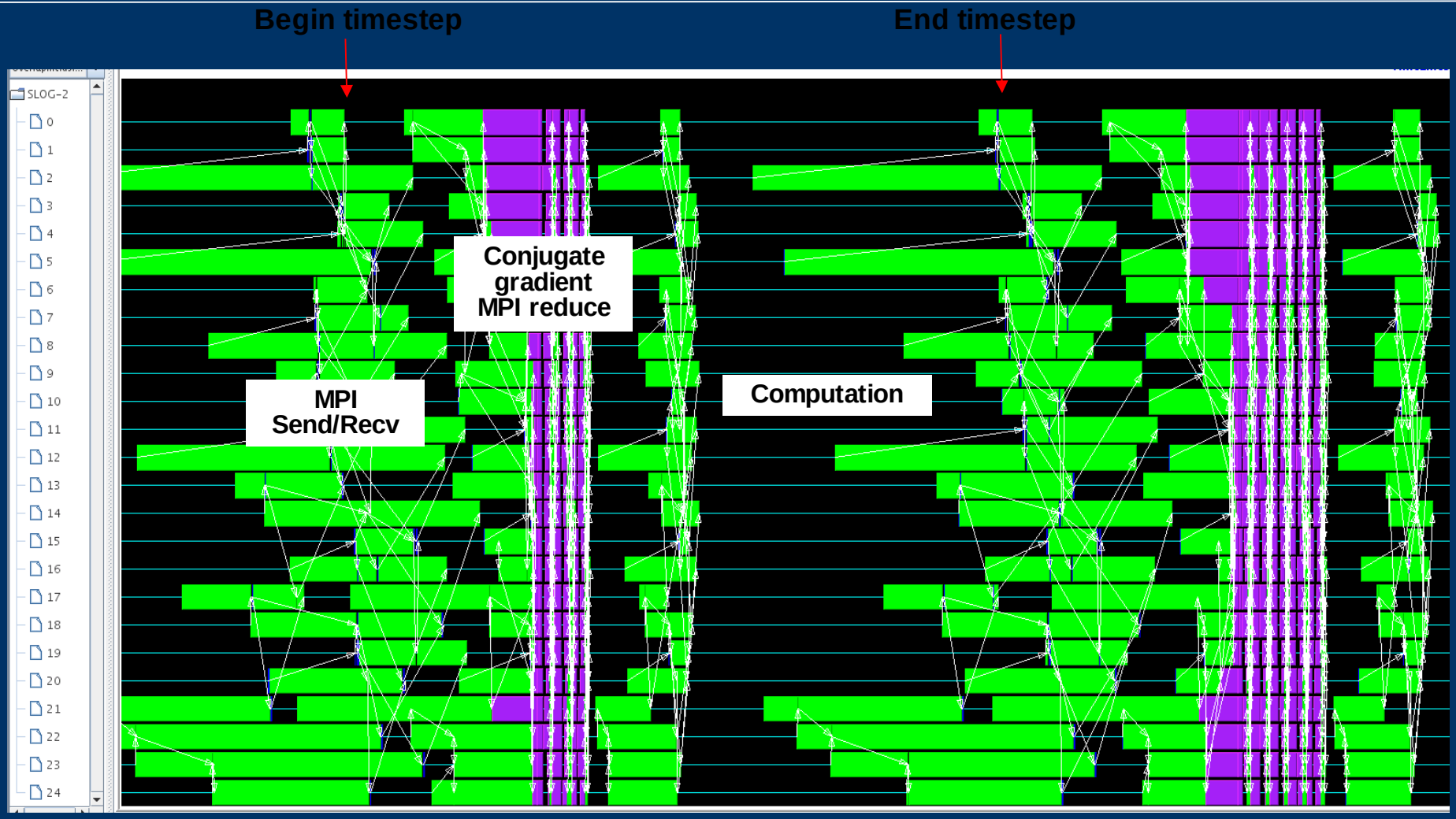


Parallel code execution procedure



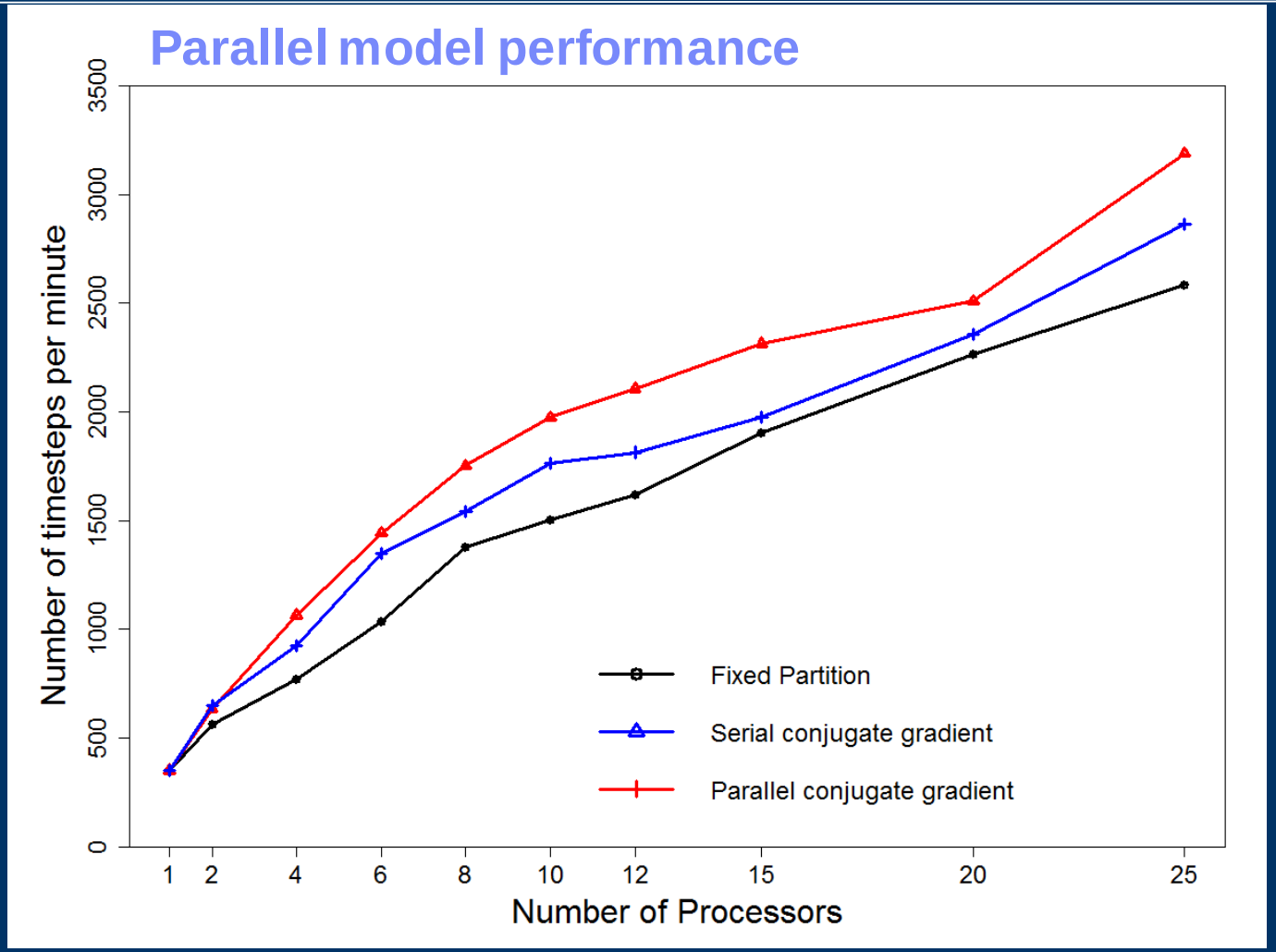
Flowchart of the entire MPI code pre-processing, simulation, and post-processing.

The only required user input is number of available processors; optimal partitioning is computed by LORP, the output of which is used by a Python pre-processing module to amend all input files and memory allocation function as appropriate. After simulation the final stage is reconciliation of output files from each processor into equivalent output from the serial code. The entire routine is conducted by means of a scripting interface to minimize user interaction.



Jumpshot analysis of computation/communication time during a model timestep. Green displays MPI Send/Receive costs, magenta represents global summation conducted by MPI reduce as part of conjugate gradient solver, and black displays computation time.

Parallel model performance



Number of time steps per minute compute by the MPI code varying the number of processors. Three configurations of the MPI code are presented: (1) domain split into equal partitions; (2) domain decomposition performed by LORP and conjugate gradient routine computed in serial; and (3) LORP decomposition with conjugate gradient scheme computed in parallel.

Note on writing and developing successful code

- Continuous unit testing of solution
- Effective source control – github, SVN, etc.
- Collaborative project planning and issue reporting
- Development Environment – Vagrant, IDE, etc.
- Cloud integration if possible/desirable

Cloud Computing and Traditional HPC

Cloud

- Availability – always on
- Scalability – practically unlimited
- Transportability – platform agnostic
- Communication
- Shared nature of the virtualized environment

HPC

- Close to the “metal” – OS-specific-optimised libraries
- User space communication – bypass the OS
- Tuned hardware – based on application needs
- Limited availability
- Capital expense and local administration

Cloud Virtualization

Virtual machine

- Run applications within hosted VM environment
- Most common IaaS offering (e.g., Softlayer, EC2)
- Well-defined isolation with applications contained within its own VM
- Security better defined
- High memory overheads and start-up time
- Hypervisor impacts communication, particularly intra-node

Container-based virtualization

- Extremely lightweight VMs that isolate applications from other containers
- Largely driven by the open-source Docker project
- Aim to accelerate development and ease deployment and distribution
- Much more lightweight
- Reduced memory and start-up times
- Layered approach makes development and source control easier
- Flexible porting of software to other platforms
- Isolation not as well-defined as for VMs
- Performance for communication-intensive applications not well understood