Designing and running turbulence transport simulations using a distributed multiscale computing approach

O. Hoenen[1], L. Fazendeiro[2], B D. Scott[1], J. Borgdorff[3], A G. Hoekstra[3], P. Strand[2], D P. Coster[1]

[1] *Max-Planck-Institut für Plasmaphysik, EURATOM Association, Garching, Germany*

[2] *Chalmers University of Technology, Göteborg, Sweden*

[3] *Institute of Informatics, University of Amsterdam, The Netherlands*

Multiscale simulation involving slow transport and fast turbulent timescales is one amongst the key computational challenges identified by PRACE for Magnetic Confinement Plasmas. Whereas parallelization efficiency is the main challenge global gyrokinetic simulations have to face, difficulty of the multiscale approach is more related to code complexity than to peak performance and high scalability. Instead of implementing a multiscale application as a complex monolithic code, one can consider it as a set of single scale submodels coupled together. Such approach improves ease of development and maintenance for each of the simpler submodel, but it requires some generic coupling methods. These methods should be fast, in order to limit the overhead on simulation time. They also should be portable and easily deployable, to enable distributed execution when a submodel may benefit being run remotely. This situation typically occurs when a submodel has been optimized for a specific hardware (GPU or other accelerators), when it requires a bigger HPC system or when it needs to access a local database.

The MAPPER project [1] aims to deploy a computational science environment for distributed multiscale computing. It provides tools, software and services to help scientists in the design and execution of multiscale applications on European e-Infrastructure. From a formal point of view, MAPPER relies on the modelization of a multiscale phenomena into a set of submodels, each of which covers a single scale within the global phenomenon [1]. Such a single scale can refer to a spatio-temporal domain or to a different physic model. From a practical point of view, MAPPER targets two types of distributed multiscale applications: loosely-coupled (acyclic, data exchange usually done with files) and tightly-coupled (cyclic, data exchange through a coupling library). It provides a software stack built on three layers. At high level, web-based tools provide a user-friendly graphical platform for adding new submodels, building a coupled application and controlling its distributed execution. At intermediate level, middlewares manage reservation and deployement of applications on distributed computing ressources (grid, HPC). At low level, coupling librairies allow communication of data and submodels scheduling.

[1] http://www.mapper-project.eu/

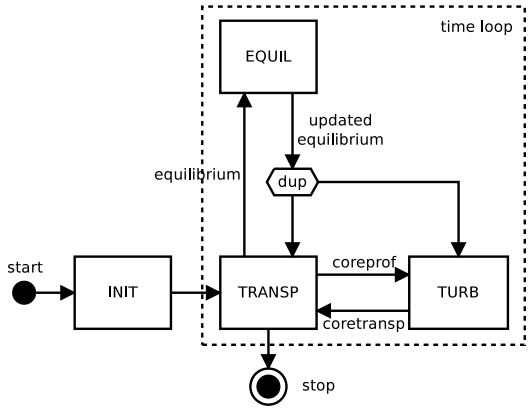


Figure 1: Tightly-coupled application workflow

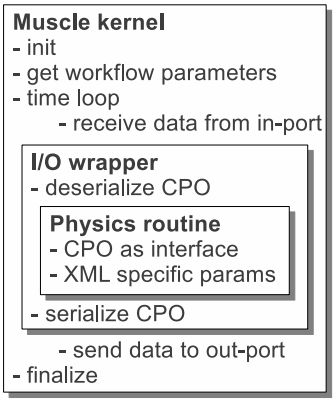


Figure 2: Kernel structure

This work is focused on the usage of MAPPER's coupling library to build and run a distributed tightly-coupled application from a set of legacy codes. The chosen application simulates the time evolution of some plasma profiles. It is composed of three submodels: profiles are evolved by a 1-D transport equations solver, 2-D geometry is given by a fixed-boundary equilibrium code and turbulent transport coefficients are coming from a 3-D flux tube code. These submodels have been implemented within the EDFA Integrated Tokamak Modelling Task Force (ITM [2]), as standalone programs or simple routines, either sequentially or in parallel. Following ITM guidelines, each code is usung a generic datastructure made of a set of Consistent Physics Object (CPO [2]), in particular: an *equilibrium* CPO (2D axi-symmetric tokamak equilibrium), a *coreprof* CPO (1D profiles in the core plasma) and a *coretransp* CPO (generic transport coefficients for the core transport equations). Using a common datastructure gives a direct way of coupling codes and makes different implementations of the same submodel interchangeable. If specific parameters are needed, they are given in a separate XML file. Figure 1 shows how each submodel (rectangular boxes) interact with each others and what CPO (arrows) are exchanged. Data transfer is done by the MUSCLE2 [1] coupling library.

The Multiscale Coupling Library and Environment (MUSCLE2) has been developed within MAPPER as a fully configurable and portable coupling platform, adapted to parallel legacy codes and distributed execution. A coupled simulation is composed of three elements: the MUSCLE2 engine (developed in Java), submodels implementation (called *kernels*) and a configuration file which describes the coupling. On the contrary to previous versions, it is not necessary to develop a kernel in Java, as the API is now also provided in C/C++ and Fortran. This API is simple (only five functions are used for a basic usage) and easy to understand, especially for developers used to MPI. `MUSCLE_init` and `MUSCLE_finalize` are used respectively to connect the kernel to MUSCLE's engine and to disconnect it. `MUSCLE_get_property` is used to

[2] www.efda-itm.eu/

read simulation parameters, which can be local to the kernel or global. Then, `MUSCLE_recv` and `MUSCLE_send` are used respectively to receive (blocking) and send data along different *input* and *output ports*. Communication handles basic interoperable types and *raw* byte streams, used for instance to send complex Java or C++ objects which have been serialized. We are building a submodel on two levels: the I/O wrapper convert native objects into interoperable data, and the kernel itself which implements the time loop and communications using MUSCLE's API. Figure 2 shows such nested structure of the code, allowing to modify both the physics routine and the I/O wrapper implementation without affecting any the kernel and the rest of the simulation.

Once all kernels have been implemented, each one is compiled as a separate executable. Interaction between kernels is then described through a configuration file, written as a small Ruby script. It contains three parts: first the declaration of kernels with their name, type (Core, Native, MPI) and executable's path in the case of Native or MPI kernels, then the declaration of parameters, and finally the description of ports with their coupling as out–in pairs. The coupled simulation is executed by calling `muscle2` (the bootstrap command) with the configuration file as argument. MUSCLE2 starts all executables which are running concurrently: scheduling is simply ensured by the dataflow. As a direct consequence, kernels can run automatically in parallel on a multicore processor if dataflow dependencies afford it. By default, all kernels are running locally on the *master* MUSCLE2 instance, but it is also possible to associate a subset of the kernels with other instances which can be on a remote host. In such case, only IP address and port of the master has to be passed as argument of other instances. Remote data exchange is done transparently through TCP communications. In case muscle's instances are running on computes which can not be accessed from outside world (firewall, local network), the Muscle Transfer Overlay deamon (MTO) can forward (from a frontal node) traffic from muscle.

The simulation presented here is a proof of concept. It couples a 1D transport solver coming from the European Transport Solver [3] (`TRANSP` kernel), a simple circular equilibrium code (`EQUIL` kernel), and the flux tube gyrofluid turbulence code GEM [4] (`TURB` kernel). Initial profiles and external sources (constant) are given by the `INIT` kernel. All codes are written in Fortran 90, and GEM is parallelized with MPI. As Fortran does not provide a built-in serialization procedure to convert Fortran derived types CPO into byte arrays, we are using in this example an out-of-core approach through files. Chosen test case corresponds to an ITER-sized circular tokamak, with flat density profile and steep enough temperature profiles in order to grow instabilities. Temperatures are solved for electrons and one ion specie. Transport solver's time step is $\tau = 0.01$ and GEM is called at each iteration, evolving 8 flux tubes on 16 cores each. Figures 3 shows the time evolution of the electrons temperature profile under the effects of transport coef-

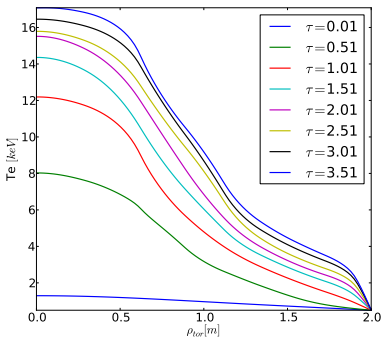


Figure 3: Te profile

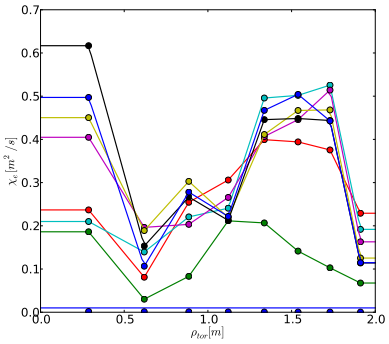
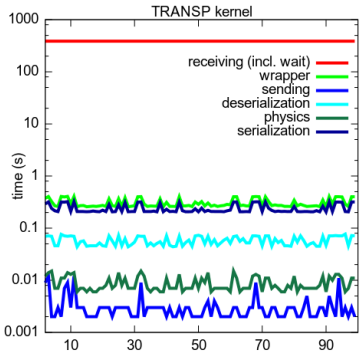


Figure 4: Transport coefficient for Te



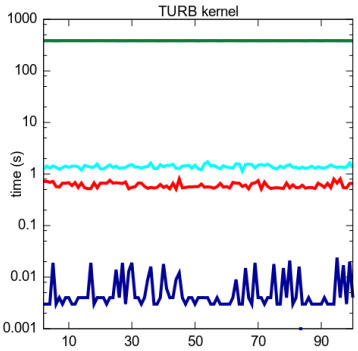


Figure 5: Kernels overhead

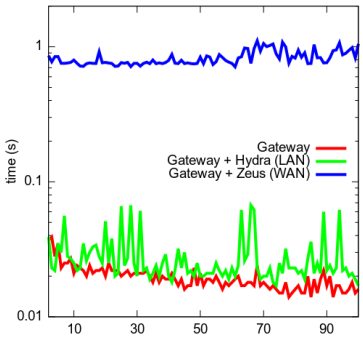


Figure 6: Transfer cost

ficients given in figure 4. On ITM's cluster (Gateway), such simulation requires a total of 10.7 hours on 128 cores for 100 iterations. Figure 5 shows the time spent in different parts of the kernels at each iteration. As expected, the deserialization overhead is not negligeable and should be replaced with an in-memory version for production runs. Overall efficiency (time spent purely on physics routines divided by simulation wallclock time) in that case is 97.9%. Remote test includes a site in Germany (Hydra at IPP, in LAN with the Gateway) and in Poland (ZEUS at Cyfronet, Krakow). For distributed simulation, only TURB kernel is executed remotely. Communication overhead shown in figure 6 is the difference between blocking time in TURB receive and the sum of runtime for distant kernels, thus it includes incoming ($160KB + 2.5MB$) and outgoing ($96KB$) data transfers. On most aspects, overheads remain low compared to the improved adaptability, reusability and maintainability allowed by such approach.

Acknowledgement

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n°261507 (the MAPPER project).

References

[1] J. Borgdorff, et al., Procedia Computer Science **9**, 596 (ICCS 2012)

[2] F. Imbeaux, Computer Physics Communications **181**, 987 (2010)

[3] D. Coster, et al., IEEE Transactions on Plasma Science **38**, 2085 (2010)

[4] B D. Scott, Phys. Plasmas **12**, 102307 (2005)