

Development and evolution of a distributed CFD-code based on the MPI

Essay in *Distributed Information Systems* - DIF8914

Andrea Gruber

Energy and Process Technology Department
NTNU - Trondheim

2002-11-25

Abstract

The distributed computing environment under development at Sintef Energy Research is reviewed and issues related to the distributed execution and development of software are addressed. The general-purpose CFD code SPIDER has been developed at Sintef Energy Research during the last ten years. The present essay reviews several issues related to the particular distributed environment associated to the development and running of the SPIDER code and is therefore not of general character, nevertheless most issues typical of many other distributed information systems are addressed: concurrent program execution, heterogeneous physical (hardware) configuration of the system, software engineering strategies, synchronisation, deadlocks, load balancing and partial failure handling are some of them.

1 Introduction

One of the main market advantages and strength-points of the Thermal Energy Department at SINTEF Energy Research is the professional and technical ability necessary to simulate combustion phenomena using a detailed representation of the chemical kinetics and the integration of this in a general purpose CFD-code (SPIDER: implemented in a mixture of FORTRAN - for the numerical solvers-, C and Motif - for the GUI). This ability, together with the computational power supplied by a variety of heterogeneous hardware architectures (Intel, DEC ALPHA, SGI workstations and clusters) on a high bandwidth communication network, allows very accurate predictions of the formation of the reaction products (i.e. thermal- and

prompt-NO_x , CO, etc.) that is otherwise difficult to obtain with a simplified reaction mechanisms approach. There is though a price to pay when using such a detailed description of the chemical kinetics: high computational time. This price represents a problem especially in research work related to Sintef projects: waiting several weeks to obtain results of a single numerical simulation is simply too much.

Another major problem related to the SPIDER software as a research tool is fragmentation. The development of the SPIDER code started in relation with the PhD work of a graduate student and was carried on by several others PhD students and Sintef researchers. Because of this the CFD code was initially very fragmented and its development ineffective: each developer used to have its own version of the software on its own workstation and the implementation in the code of new features and mathematical models was happening locally. Porting and/or merging of different versions of the code was time consuming and led often to inconsistencies or "bugs".

This report illustrates how these two problems have been, at least partially, solved: the computational time issue was addressed by performing a parallelisation of the computationally expensive part of the code while the software development issue was addressed by complete re-organisation of the code in the frame of a version control system placed at commonly shared locations.

Section 2 reviews some of the issues and challenges related to parallel execution of the SPIDER code in the distributed system: execution concurrency , heterogeneous hardware, synchronisation, deadlocks, load balancing, transparency.

Section 3 reviews the problems associated with distributed software development: general administration rules, access concurrency, merging, tree branching, access transparency.

One definition of *distributed information system* is given in Coulouris *et al.* (2001): **"We define a distributed system as one in which hardware and software components located at networked computers communicate and coordinate their actions only by passing messages"**. Far from resolving or even addressing *all* the issues related to a distributed information system the present work gives a brief description of the problems that have been fixed and those that emerge from the implementation of a particular distributed information environment.

In order to better understand the following sections it is necessary at this point to give a brief description of the physical configuration of the computer system that is in use at the Thermal Energy Department. Figure 1 illustrates the principal parts of the system. The main computational work is usually performed by the DEC ALPHA workstations (RISC EV67 processors) and by the Linux cluster (produced by the Norwegian company SCALI from DELL hardware, features 18 Intel P4 Xeon CPUs). The boxes named "desktop client" in the figure represent the machines

that every single researcher has on his/her desk and span quite a variation both in computational power and hardware architecture (from few to dozens of MFLOP/S, from INTEL cisc to SGI risc processors). The red box in the corner (a DEC ALPHA dedicated file and NIS server) represents in a way the "glue" of the information system providing authorised users with access to all the machines in the network and with several types of transparency (access, location, replication).

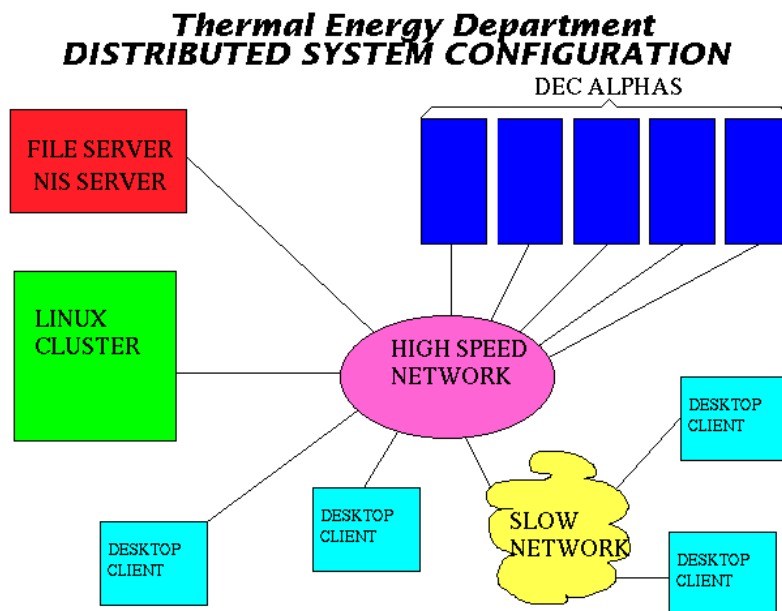


Figure 1: The networked environment

2 Distributed Information System: parallel code execution

A basic assumption which has to be satisfied whenever trying to parallelise and effectively distribute the execution of a computer code on several different CPUs, is that each CPU running the code has *nearly* at all time all the data necessary to perform the calculations and does not depend on the computations performed at the same time on other CPUs. This avoids large latency time (one processor waiting for another one to finish its calculations) and allows fully parallel execution. In the case in which the different CPUs participating in the calculations have to exchange large data sets by passing messages to each other it is very likely that the communication network will represent a bottleneck and the distributed execution will not be very efficient (large latency times).

2.1 The SPIDER code

One instance where the requirements for efficient distributed execution are easily and effectively satisfied is the chemical kinetics calculations that are integrated in the CFD-code SPIDER. In fact the routine performing this kind of calculations simply runs through all control volumes (CVs) in which the computational domain has been divided into and calculates for each one of them the "new" chemical composition using the known thermo-dynamical state (pressure, temperature and composition) of the CV as input. The chemical kinetics calculation algorithm is therefore particularly well suited for parallelisation because the quantity of data to be supplied to each different CPU is limited (the thermo-dynamical state size is of the order of few bytes, i.e. pressure and temperature, and composition of the flow in a particular CV) and the calculation to be performed by each CPU is computationally heavy: in the case of the chemical kinetics mechanism implemented in the SPIDER code (GRIMech version 2.11) 49 species and 277 reversible chemical reaction give a quite formidable system of stiff ODEs to be solved by LIMEX (a stable stiff ODE solver).

Table 1 lists the time percentage (in % of the overall execution time) used by the single subroutines of the SPIDER code when performing a numerical simulation of a turbulent premixed flame with the detailed chemistry approach. As it is evident from the listed data more than 97 percent of the execution time is spent by the numerical simulation code on chemistry calculations, which are performed by the subroutines `lingl`, `ckratt`, `ckwyp`, `ckratx`, `cksmh`, `subst` etc. This formidable computational load carried out by one "module" of the code, module that is quite separate from the rest, was the main reason for which a distributed execution of this part of the code was implemented.

2.2 The Message Passing Interface (MPI)

Given the fact that a computer code or a part of it is logically and efficiently parallelizable (as we have seen is the case for the chemistry calculations in SPIDER) one of the most common ways to obtain a version of the program that can run in parallel on multiple CPUs is its modification using a library of functions called Message Passing Interface or MPI.

MPI is a library of functions used to create message passing programs written in various programming languages (Fortran, C, C++ and Java). MPICH (MPI + Chameleon) is a portable implementation of MPI for different parallel computation systems. It contains visualisation and diagnostic tools, as well. MPICH protocols are used to communicate data between processors either in the same machine or in different machines. Using the most adequate protocol for a parallel system depends on different characteristics such as data size, algorithm taxonomy, and hardware architecture (shared memory, distributed memory). Version 1.2 of MPICH, was compiled by the author both for the Linux and DEC (Compaq) ALPHA platforms, on the Silicon Graphics IRIX platform, SGI's own implementation of MPI was used.

Table 1: Relative computational load of the different subroutines in SPIDER

Profile listing generated Tue Nov 27 14:50:23 2001 with: prof spider mon.out

```
-----
* -p[rocedures] using pc-sampling; *
* sorted in descending order by total time spent in each procedure; *
* unexecuted procedures excluded *
-----
```

Each sample covers 8.00 byte(s) for 0.00078% of 124.4209 seconds

%time	seconds	cum %	cum sec	procedure	file
28.9	36.0098	28.9	36.01	lingl	(linpac.f)
26.0	32.3311	54.9	68.34	ckratt	(cklib.f)
16.2	20.1943	71.2	88.54	ckwyp	(cklib.f)
14.8	18.4033	85.9	106.94	ckratx	(cklib.f)
2.8	3.5107	88.8	110.45	cksmh	(cklib.f)
1.7	2.0938	90.5	112.54	subst	(linpac.f)
1.3	1.6123	91.7	114.16	dnjac	(limex.f)
1.3	1.5986	93.0	115.75	rstate	(rstate.f)
1.3	1.5625	94.3	117.32	djscal	(limex.f)
1.2	1.4775	95.5	118.79	limex1	(limex.f)
1.1	1.3281	96.5	120.12	ckytcp	(cklib.f)
1.0	1.2744	97.6	121.40	ditmat	(limex.f)
0.5	0.6299	98.1	122.03	xlifun	(chemca.f)
0.3	0.4287	98.4	122.46	omeg12	(tranf.f)
0.3	0.3584	98.7	122.81	detcal	(chemca.f)
0.1	0.1348	98.8	122.95	fscsvr2	(lizzie.f)
0.1	0.1191	98.9	123.07	xinter	(tranf.f)
0.1	0.1104	99.0	123.18	mtdma	(tdma.f)
0.1	0.1025	99.1	123.28	pow	(pow.f)
0.1	0.0996	99.2	123.38	jasch	(ckintp.f)
0.1	0.0869	99.2	123.47	edcpsr	(edcpsr.f)
0.1	0.0820	99.3	123.55	d12	(tranf.f)
0.1	0.0635	99.4	123.61	tcalc2	(tcalc.f)
0.0	0.0596	99.4	123.67	inter	(extra.f)
0.0	0.0596	99.4	123.73	xlimex	(limex.f)
0.0	0.0479	99.5	123.78	linsol	(linpac.f)

The chosen setting is the **ch_p4** device (see below) that is configured with shared memory pattern, which allows working with TCP/IP *and* shared memory. This is especially important in a network of heterogeneous hardware consisting of shared

memory Intel SMP workstations and distributed memory DEC ALPHA workstations.

The architecture of MPI consists of 3 layers: Application Programmer Interface (API), Abstract Device Interface (ADI) and Channel Interface (CI). API is the interface between the programmer and ADI. The API uses an ADI to send and receives information. The ADI controls the data flow between API and hardware. It specifies if the message is sent or received, handles the pending message queues, and contains the message passing protocols. The message structure used for ADI is composed of a message body, a field that describes the length of the message body, a field of message tag, context-id, and the destination. CI is implemented by ADI. CI uses protocols to transmit data from a processor to another. There are four protocols in MPICH, namely **eager**, **rendezvous**, **short**, and **get**, for a detailed description of these protocols see Gropp *et al.* (1999). Several CI made for MPICH can be implemented, for instance, **ch_p4**, **ch_shm**. The **ch_p4** is a message-passing library that implements the ADI. Its CI is implemented with Portable Programs for Parallel Processor (P4), that is, for shared-memory and message-passing (distributed memory) models.

Running a program using the MPICH and distributing its execution on several processors in different machines is easy. The computation is performed through a job launcher *mpirun* that spawns a number of processes that all run the parallelised code. The number of processes to be started is given as an argument to the job launcher and the "machine park" that is available for the parallel calculations is specified in a configuration file containing a list of networked computer names (called "nodes"; the number of CPUs available for each node is also specified in the same file). The user is obviously supposed to have authorised access to all the machines where he/she wants to run the parallel code.

2.3 Implementation of MPI in SPIDER

In the SPIDER code the parallel execution of the program was organised as follows (where n_{max} is the number of CPUs minus one), see also Figure 2 for a flowchart scheme:

1. Process 0 starts SPIDER, reads the input files and initialises the chemistry-tables
2. Process 1 to n_{max} start, initialise the chemistry-tables and stop execution at a "mpi barrier" waiting for process 0 to reach the same barrier
3. Process 0 starts the computation of the fluid dynamics by the SIMPLE algorithm (solving the equations for conservation and transport of momentum, energy, turbulence, dissipation and species)
4. Process 0 arrives at the same barrier where the other processes were waiting for it and broadcasts the information needed by all the other processes

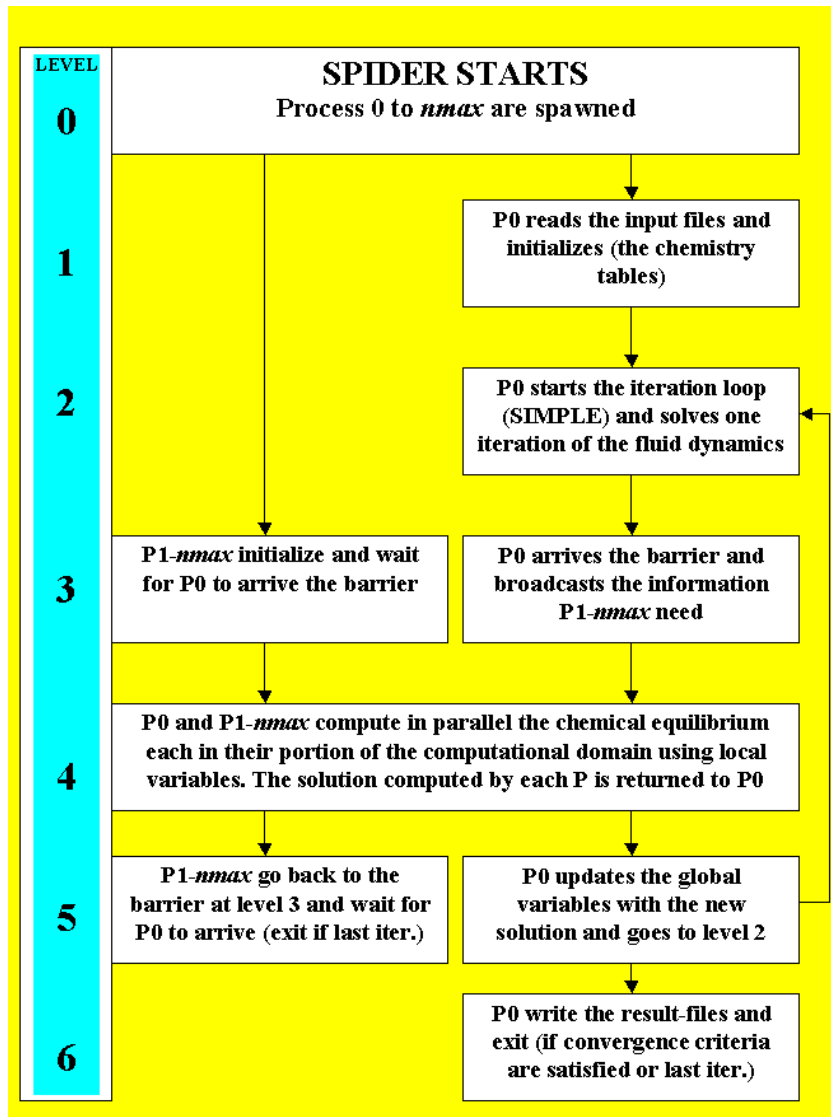


Figure 2: Flowchart of the SPIDER code illustrating the parallel algorithm

5. All the processes solve the chemistry calculations in the CVs that belong to their portion of the computational domain
6. The solution of the chemistry calculation is passed back or "mpi reduced" by all the processes to process 0
7. Process 0 completes the execution of the iteration loop and goes back to point 3), otherwise exits, if the convergence criteria are satisfied or the last iteration is reached

The execution scheme allows for the heavy computations to be split among as many CPUs as available. The scheme has also proven itself to be very stable at

runtime in executing the simulations and very efficient in distributing the computational load on the available CPUs (for details see Gruber and Olsen (2001)) giving an almost linear scaling of execution time with the number of CPUs used. The computational time required by a typical combustion case simulation is now cut down from several weeks to several hours!

Special care when implementing the algorithm illustrated in Figure 2 has to be paid by the developer in order to avoid concurrency, halt or crash situations among one or several of the the spawned processes (for example when concurrently reading the initialisation files or writing output files). In fact the MPI does not guarantee *concurrency and failure transparency*, the implementation of a safe algorithm in this respect is up to the programmer. In the case of the SPIDER code the issue was addressed by careful synchronisation of read operations in all processes and by performing all write operations *only* in process 0. *Partial failure* is tough an issue that has not been addressed yet and in the situation of one node becoming unavailable the parallel calculation would abort. Another issue that requires special attention when using the MPI is the possibility of running into *deadlocks* caused, for example, by a wrongly organised call of the **MPI_SEND** and **MPI_RECEIVE** functions. In a given process both the send and receive operations are in fact blocking the same buffer: if process 0 is trying to send a message to process 1 and the latter process is not ready to receive because it is also trying to send a message to process 0 a *deadlock* situation is reached. In the present implementation of the MPI in SPIDER this problem was avoided by using the **MPI_BROADCAST** and **MPI_REDUCE** functions instead of **MPI_SEND** and **MPI_RECEIVE** whenever spreading and collecting of information was required (as a matter of fact the broadcast and reduce functions improve also *load balancing!*).

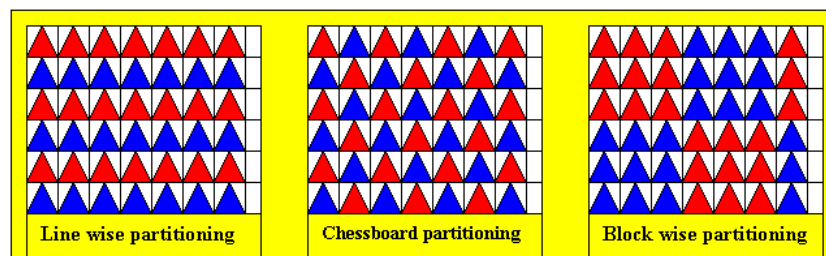


Figure 3: Alternative partitioning of the computational domain

Given the fact that the portions of the computational domain that are "heaviest" computationally are the regions of the physical domain in which the flame is burning and most of the chemical reactions are taking place, it is not difficult to imagine that the CPUs that are associated with these regions experience a computational load that can be much higher than the one experienced elsewhere. If the computational domain is not divided among the available CPUs in a way that evenly distributes the actual load by finely partitioning of the computational domain (i.e. by using a chessboard scheme, see Figure 3), the efficiency of the parallelised de-

tailed chemistry calculations will be very low because of poor *load balancing*. At the same time, a possible future parallelisation of the algorithm (SIMPLE or RK4) that solves the fluid dynamics of the flow in SPIDER may eventually suggest another kind of partitioning (i.e. a block wise scheme, see Figure 3, rightmost alternative) of the computational domain among the parallel processes.

The presence of a NIS (and YP) server gives the users of this particular distributed information system the possibility of running SPIDER not only on the dedicated machines (ALPHAS, SCALI Linux cluster) but also exploiting the idle clock beats of the desktop computers of the other users in the network (the policy here is to run on other people's machines with lowest priority - "nice 19" - so that the other people's work is not affected!), transforming in practice the whole local area network in a single distributed parallel computer.

3 Distributed Information System: parallel code development

A software project of considerable size and complexity simultaneously carried on by many developers often faces problems of inconsistency and incompatibility between the various "branches" of the code programmed by different people working at different parts of the code. The process of developing and maintaining several versions of a software is called *parallel development*. In the case discussed here, the SPIDER code, two major "lines of development" have been the **combustion simulator** (implemented for incompressible flow situations) and the **multiphase flow simulator** (implemented for compressible flow situations). The two branches represents algorithms that are considerably different in many aspects from data structures to solution methods.

The use of the version control software named CVS (Concurrent Version System) was introduced 3 years ago in order to allow efficient *parallel software development* and decrease the costs in man-hours needed by an update or merging of the code to a working version implementing the "latest" models. CVS is free software and does not provide many of the functions available in commercial software configuration management systems (as Atria's *ClearCase* building and multi-site capabilities, see Allen *et al.* (1995)) but it has several important characteristics:

- Its client-server access method lets developers access the latest code from anywhere there's an Internet connection (SSH,NFS,FTP), providing *access transparency*
- Its repository system insulate the different developers from each other. Every developer works in his own directory, and CVS merges the work when each developer is done. This fact resolves any *access concurrency* issue
- Its unreserved check-out model to version control avoids artificial conflicts common with the exclusive check-out model
- Its client tools are available on most platforms

- Its branching and versioning system allows management of multiple variants of evolving software systems and releases. Accounting informations are also stored in the repository allowing for identification of who made a certain change, how and why (providing "log-books" of the software development process)

The characteristics listed above make CVS an invaluable aid to the distributed parallel development of a computer code but CVS alone does not resolve all the issues related to this matter (neither do the commercial version control software packages). Experience teaches two important lessons when using a version control software:

- A set of general administration rules and policies has to be first agreed and then strictly implemented by the developers in order to avoid that one by mistake destroys the work of the others
- No version control software can perform as a valid substitute for the frequent personal communication between developers that is necessary to avoid conflicts, inconsistencies and inefficiencies in the code

The latter consideration is particularly important in the common situation in which, after merging two or more files, a conflict situation arises and the modifications to the code implemented by one developer crash with modifications implemented by others.

4 Conclusions

The issues briefly discussed in this report illustrate some of the improvements achieved and the problems encountered in a particular distributed information system familiar to the author. Many of these issues are common to other distributed information systems. In this case the fact of being able to distribute the execution of a computationally heavy numerical simulation code on a large number of CPUs represents an enormous advantage in a very competitive market: the result is a drastic reduction of the time requirements to obtain valuable knowledge in the field of process engineering (combustion processes). At the same time the costs of the implementation of new mathematical models, that better describe the studied processes, in the computer code has also been greatly reduced by the adoption of a version control software fully exploiting the resource sharing capabilities of the distributed information system.

References

Allen, L., Fernandez, G., Kane, K., Leblang, D., Minard, D., and Posner, J. Clearcase multisite: Supporting geographically-distributed software development. In: *Soft-*

ware Configuration Management: selected papers / ICSE SCM-4 and SCM-5 workshops, pages 194–214. Springer Verlag LNCS 1005, 1995.

Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems: Concepts and Design*. Pearson Education, third edition, 2001.

Gropp, W., Lusk, E., and Skjellum, A. *Using MPI*. MIT Press, second edition, 1999. ISBN 0-262-57132-3.

Gruber, A. and Olsen, R. Parallel version of the cfd code spider - a benchmark study. Technical Report TR A5559, Sintef Energy Research, Sem Sælandsvei 11, 7023 Trondheim, Norway, 2001.