Appendix A

Facilities

A.1 JFF PC cluster at CTTC

The JFF PC cluster is sited at CTTC (Centre Tecnològic de Transferència de Calor) in Terrassa (Spain). This acronym has been adopted in memorial of Joan Francesc Fernandez, computer science professor of the Polytechnic University of Catalonia. The JFF cluster is a network of 35 commodity processors connected with fast ethernet NICs (Network Interface Cards) and two full duplex switches. Each processor runs the operating system Linux with a minimal installation with a source Message Passing library.

Details of the cluster configuration are given in table A.1.

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>model</th>
</tr>
</thead>
<tbody>
<tr>
<td>main board</td>
<td>35xASUS a7V</td>
<td>AMD-751 chipset</td>
</tr>
<tr>
<td>processor</td>
<td>5x600MHz + 28x900MHz</td>
<td>AMD-K7</td>
</tr>
<tr>
<td>cache (L1+L2)</td>
<td>128KB of L1 + 512KB of off-chip L2</td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>2x1GB + 33x512MB</td>
<td>SDRAM 133MHz</td>
</tr>
<tr>
<td>HD</td>
<td>25GB on server + 34x18GB</td>
<td>IDE</td>
</tr>
<tr>
<td>SWAP</td>
<td>double RAM per processor</td>
<td></td>
</tr>
<tr>
<td>NICs (100Mb/s)</td>
<td>2 on server + 34x1</td>
<td>3C905B-TX-NM</td>
</tr>
<tr>
<td>switch (100Mb/s)</td>
<td>24 full duplex ports, scalable</td>
<td>3C16980</td>
</tr>
<tr>
<td>matrix cable</td>
<td>1 matrix cable connecting 2 switches</td>
<td>3C16965</td>
</tr>
<tr>
<td>OS</td>
<td>Linux</td>
<td>Debian v2.2.17</td>
</tr>
<tr>
<td>batch system</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>MP library</td>
<td>MPI</td>
<td>LAM 6.1</td>
</tr>
<tr>
<td>compilers</td>
<td>C,C++,F77</td>
<td>gcc v2.7, g77 v0.5.24</td>
</tr>
</tbody>
</table>

Table A.1: Cluster configuration

Few pictures A.1,A.2 and A.3 of the JFF cluster have been attached to show the distribution and connection of the commodity PCs and switches on shelves.
Figure A.1: Front view of the JFF cluster

Figure A.2: Front of the switches and connections
In order to manage the cluster, there is a server node with x-server and equipped with two NICs (Network Interface Cards), one connected to the 34 nodes of the private net and the other connected to the public network. Users access to the cluster on this second NIC. Tasks of compilation and debugging may be done in this node. The compiler and loader for sequential jobs is the gcc, with the following flags and mathematic library:
> gcc -O3 -w -Wall -pedantic source-files.c -o object-files.o
> gcc object-files.o -lm executable-file

For parallel jobs, the compiler and loader is the hcc, with the MPI library:
> hcc -O3 -w -Wall -pedantic source-files.c -o object-files.o
> hcc object-files.o -lm -lmpi executable-file

After that, directories and executable files needed for execution are copied to nodes with the remote commands rsh and rcp.

Since we are in the beginnings of the JFF cluster, there is not yet a batch system. Therefore the execution of sequential and parallel jobs is done either in interactive mode or in background with the command noshp.

For sequential jobs the following command line is used:
> noshup /work-directory/executable-file &

For parallel jobs with LAM MPI library three steps must be followed:
> lamboot -v nodes
> noshup mpirom -v -c np -O -w /work-directory/executable-file &
> ulps -v nodes
The first step initializes the lam daemon process and 'awakes' the mp node-processors listed in the nodes file. The second step runs with the spiram command mp copies of the executable file in mp processors placed in the work directory. The -o flag tells to the lam daemon that the network of processors is homogeneous (it does not data conversion among nodes), and the -w flag waits for all process to complete before exiting spiram and reports abnormal exit codes. Once the execution has ended, the daemon process is killed with the *wipe command.

Although Linux is a multiprocess operating system it is desirable to minimize the number of processes for performance measures (i.e. timings of subroutines). During these measures, each process whether it is originated by the operating system (i.e. the daemons) or by the executable file (i.e. the output of results) produce perturbations in these measures. For this reason every experiment must be repeated many times to give an averaged measure.

### A.2 SGI/Cray T3E at CIEMAT

The SGI/Cray T3E supercomputer of 32 air cooled nodes is sited at CIEMAT (Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas) in Madrid (Spain). Details of the configuration are given in table A.2.

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>model</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td>32x300MHz, super scalar RISC</td>
<td>DEC 21164</td>
</tr>
<tr>
<td>cache (L1+L2)</td>
<td>8KB of L1 + 96KB 3-way of L2</td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>32x128MB</td>
<td>DRAM</td>
</tr>
<tr>
<td>HD</td>
<td>30GB distributed for every 8np</td>
<td>Giga Ring, SCSI</td>
</tr>
<tr>
<td>SWAP</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>net (1Gb/s)</td>
<td>3-D torus of low latency</td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Unix</td>
<td></td>
</tr>
<tr>
<td>batch system</td>
<td>Network Queueing Environment</td>
<td>N2E v3.3</td>
</tr>
<tr>
<td>MP library</td>
<td>PVM, MPI</td>
<td>optimized for Cray</td>
</tr>
<tr>
<td>compilers</td>
<td>C, C++, Fortran90</td>
<td>optimized for Cray</td>
</tr>
</tbody>
</table>

Table A.2: SGI/Cray T3E configuration

The SGI/Cray T3E has two processors dedicated to the surveillance and maintenance of the global system (i.e. the Global Resource Manager) and file server for the rest of nodes. Four processors for interactive tasks such as user sessions, compiling and debugging. Only they support multiprocessing (time sharing). The rest of processors (i.e. 26 processors) do parallel applications throughout the command spiram. Each processor accepts only one process and the total amount of memory available without swap is 128MB.
Sequential and parallel jobs are submitted to a batch system (for instance, the Network Queuing Environment) using the following script, called `job.sh`:

```bash
# QSUB -l app=g=xp limit
# QSUB -l app=t=hh:mm:ss limit
# QSUB -eo
# QSUB -o job.sh
cd /work/directory
apprun -n np executable-file
```

Further details may be found at the Ciemat's web site [www.ciemat.es](http://www.ciemat.es).
Appendix B

Software layers in CFD code

B.1 Description of layers

The CFD code is structured in four layers: the user layer, the solver layer, the algebra layer and the communication layer. These layers are linked and integrated to build a structure which looks like an iceberg (see Fig. B.1).

![Diagram of iceberg layers](image)

Figure B.1: Iceberg built by layers: user layer, solver layer, algebra layer and communication layer.

The concept of the iceberg comes from the idea that only the top of the iceberg or the last layer is visible while the rest of the iceberg or layers are hidden below the water. This top end is called the user layer, the only layer visible to the end user of the CFD code. The rest of layers are hidden from the user, i.e. the user does not access to the remaining layers. In order to build the CFD code in layers it is very important to specify clearly which are the subroutines of each layer and which layer is supported by another, i.e. to define the dependencies between layers in only one direction. This dependency goes from the top to the base as shown with the arrows in the in figure B.1.
The description of these layers is given below.

- The user layer can be structured into a set of modules, each of them with different functions. For example, the graphic module, the data module and the model module. The graphic module represents the front-end of the CFD code. The data module contains all information needed for running a case. These, a specific information of the case (e.g. geometry, boundary conditions, initial conditions, properties of fluid and flow), and a specific information about how to run the case (e.g. algorithm or procedure, solver, iterations, convergence criteria) must be included. The model module performs the translation from the real case to the set of algebraic systems of equations (e.g. finite volume methods, schemes, set of physic hypotheses). This translation is the discretization of the partial differential equations involved in the case. Since this work has been focused on how to solve such algebraic systems by using solvers and parallel computing, the above modules are compacted in the concept of the user layer.

- The solver layer performs the evaluation of the solution of the algebraic systems of equations. This layer specifies a wide range of solvers based on classic methods (e.g. Jacobi, TDMA, Gauss), decompositions (e.g. complete and incomplete LU decompositions), Krylov space based methods helped by preconditioners (e.g. CG, BCGSTAB, GMRES) and acceleration techniques (e.g. Algebraic MultiGrid and MultiResolution Analysis with wavelets).

- Each of these solvers contains some common basic operations which have been integrated in a more basic layer so called algebra layer. By this, we refer mainly to the algebraic operations between vectors and matrices (e.g. matrix-vector product, inner product between vectors, addition and subtraction between vectors, norm of vectors), and some transformations over vectors, matrix and maps of scalars (e.g. discrete Fourier transform, discrete wavelet transform).

- The above layers are supported by the communication layer. It is also called the communication layer because its main task is the communication inside a layer between processors: mainly the domain decomposition at the user, solver and algebra layers. If the CFD code is thought in sequential, this layer can be eliminated at all, but in parallel, part of the task are done at the communication layer. For example, the parallel algebraic operations are performed in the algebra layer except the exchange of data among the processors that is performed in the communication layer (e.g. the matrix-vector product, the norm of a vector and the maximum or the minimum value of a map distributed among the processors). The subroutines embedded in this layer contain calls to the parallel library MPI. It is worth noting that the from the programming point of view of the CFD code it is based on the SPMD (Single Program and Multiple Data) paradigm.
Most of these subroutines have been summarized for each layer in Fig. B.2.

User Layer
Hypothesis  Boundary Conditions
Discretization  Schemes  Stopping Criteria

Solver Layer
Gauss-Seidel  Band LU  ILU
GC  BICGSTAB  GMRESK  AMG  MRA

Algebra Layer
\[ z = x + y \]
\[ z = x \cdot y \]
\[ a = c_i \cdot y \]
\[ z = x + a \cdot y \quad s = [s] \]
\[ y = A \cdot x \]
\[ r = b - A \cdot x \]
\[ y = \text{DFT}(x) \]
\[ y = \text{DWT}(x) \]

Communication Layer
partition  block vector  block matrix  update  global sum

Operating system  MPI  Network protocol

Figure B.2: Main subroutines of the CFD code grouped by layers.
Bibliography


