Large-Scale CFD Computations

Hrvoje Jasak

`hrvoje.jasak@fsb.hr`

FSB, University of Zagreb, Croatia

Computer Power and Numerics

- CFD simulations are among the largest users of CPU time in the world. Even for a relative novice, it is easy to devise and set up a very large simulation that would yield relevant results
- Other computational fields with similar level of requirements include
  - Numerical weather forecasting. Currently at the level of first-order models and correlations tuned to the mesh size. Large facilities and efforts at the UK Met Office and in Japan
  - Computational chemistry: detailed atom-level study of chemical reactions from first principles
  - Global climate modelling. This includes ocean and atmosphere models, vapour in atmosphere and polar ice caps effects. Example: global climate model facility ("Earth Simulator")
  - Direct numerical simulation of turbulence, mainly as replacement for experimental studies
- In all cases, the point is how to achieve maximum with the available computing resources rather than how to perform the largest simulation. A small simulation with equivalent speed, accuracy *etc.* is preferred

Computer Power and Numerics

Simulation Time

- Typical simulation time depends on available resources, object of simulation and required accuracy. Recently, the issue of *optimal use of computer resources* comes into play: running a trivial simulation on a supercomputer is not fair game
- Example: parametric studies, optimisation and robust design in engineering. Here, the point is to achieve optimal performance of engineering equipment by in-depth analysis. Optimisation algorithms will perform hundreds of related simulations with subtle changes in geometrical and flow setup details in order to achieve multi-objective optimum. Each simulation on its own can be manageable, but we need several hundreds!
- In many cases, the limiting factor is not feasibility, but **time to market**: a Formula 1 car must be ready for the next race (or next season)

Computer Power and Numerics

Reducing Simulation Time

4. **Algorithmic improvements**: faster, more accurate numerics, time-stepping algorithms
3. **Linear solver speed**. Numerical solution of large systems of algebraic equations is still under development. Having in mind that a good solver spends 50-80 % of solution time inverting matrices, this is a very important research area. Interaction with computer hardware (how does the solver fit onto a supercomputer to use it to the best of its abilities) is critical
2. **Physical modelling**. A typical role of a model is to describe complex physics of small scales in a manner which is easier to simulate. Better models provide *sufficient* accuracy for available resource
1. **User expertise**. The best way of reducing simulation time is an experienced user. Physical understanding of the problem, modelling, properties of numerics and required accuracy allows the user to optimally allocate computer resources. Conversely, there is no better way of producing useless results or wasting computer resources than applying numerical tools without understanding.

Computer Power and Numerics

Scope of Lecture

- Our objective is to examine the architecture requirements, performance and limitations of large-scale CFD simulations today
- There is no need to understand the details of high performance programming or parallel communications algorithms: we wish to know what parallelism means, how to use it and how it affects solver infrastructure
- Crucially, we will examine the mode of operation of parallel computers choice of algorithms and their tuning
- The first step is classification of high-performance computer platforms in use today

High Performance Computers

Classification of Computer Platforms

- Basic classification of high performance architecture depends on how instructions and data are handled in the computer (Flynn, 1972). Thus:
  - SISD: single instruction, single data
  - SIMD: single instruction, multiple data
  - MISD: multiple instruction, single data
  - MIMD: multiple instruction, multiple data
- The above covers all possibilities. SISD is no longer considered high performance. In short, SISD is a very basic processing unit (a toaster?)
- We shall concentrate on SIMD, also called a **vector computer** and MIMD, known as a **parallel computer**. MISD is sometimes termed **pipelining** and is considered a “hardware optimisation” rather than a programming technique

Vector Computers

- Computationally intensive part of CFD algorithms involves performing identical operations on large sets of data. Example: calculation of face values from cell centres for grading calculation in cell-centred FVM:

$$\phi_f = f_x \phi_P + (1 - f_x)\phi_N$$

- ϕ_P and ϕ_N belong to the same array over all cells. The result, ϕ_f belongs to an array over all faces. Subscripts P, N and f will be cell and face indices:

```
const labelList& owner = mesh.owner();
const labelList& neighbour = mesh.neighbour();

const scalarField& fx = mesh.weights();

for (label i = 0; i < phiFace.size(); i++)
{
    phiFace[i] =
        fx[i]*phiCell[owner[i]]
      + (1 - fx[i])*phiCell[neighbour[i]];
}
```

Vector Computers

- Performing an operation like this consists of several parts
  - (Splitting up the operation into bits managed by the floating point unit)
  - Setting up the instruction registers, *e.g.* $a = b + c * d$
  - Fetching the data (memory, primary cache, secondary cache, registers)
  - Performing the operation
- In vector computers, the idea is that performing the same operation over a large set can be made faster: create special hardware with lots of identical (floating point) units under unified control
  1. Set up instruction registers. This is done only once for the complete data set
  2. Assume the data is located in a contiguous memory space. Fetching the start of the list grabs the whole list
  3. Perform the operation on a large data set simultaneously

Vector Computers

- A number of joint units is called the **vector length**. It specifies how many operations can be performed together. Typical sizes would be 256 or 1024: potentially very fast!
- Some care is required in programming. Examples:
  - Do-if structure

```
for (label i = 0; i < phiFace.size(); i++)
{
    if (f_x < 0.33)
    {
        phiFace[i] =
            0.5*(phiCell[owner[i]] + phiCell[neighbour[i]]);
    }
    else
    {
        phiFace[i] =
            fx[i]*phiCell[owner[i]]
          + (1 - fx[i])*phiCell[neighbour[i]];
    }
}
```

Vector Computers

- This kills performance: a decision required at each index. Reorganise to execute the complete loop twice and then combine result. Min performance loss: 50%!
- Data dependency

```
for (label i = 0; i < phiFace.size(); i++)
{
    phiCell[i] -= fx[i]*phiCell[owner[i]];
}
```

- Values of `phiCell` depend on each other – if this happens within a single vector length, we have a serious problem!

Vector Computers

Summary on Vector Computers

- Today, vector computers are considered “very 1970-s”. The principle works, but loss of performance due to poor programming or compiler problems is massive
- Compilers and hardware are custom-built: cannot use off-the shelf components, making the computers very expensive indeed
- However, the lesson on vectorisation is critical for understanding high-performance computing. Modern CPU-s will automatically and internally attempt to configure themselves as vector machines (with a vector length of 10-20, for example). If the code is written vector-safe and the compiler is good, there will be substantial jump in performance
- There is a chance that vector machines will make a come-back: the principle of operation is sound but we need to make sure things are done more cleverly and automatically

Parallel Computers

Background on Parallelism

- Recognising that vector computers perform their magic by doing many operations simultaneously, we can attempt something similar: can a room full of individual CPU-s be made to work together as a single large machine
- Idea of massive parallelism is that a large loop (*e.g.* cell-face loop above) could be executed much faster if it is split into bits and each part is given to a separate CPU unit to execute. Since all operations are the same, there is formally no problem in doing the decomposition
- Taking a step back, we may generalise:

  A complete simulation can be split into separate bits, where each bit is given to a separate computer. Solution of separate problems is then algorithmically coupled together to create a solution of the complete problem.

Parallel Computers

Parallel Computer Architecture

- Similar to high-performance architecture, parallel computers differ in how each node (CPU) can see and access data (memory) on other nodes. The basic types are:
  - **Shared memory machines**, where a single node can see the complete memory (also called **addressing space**) with “no cost overhead”
  - **Distributed memory machines**, where each node represents a self-contained unit, with local CPU, memory and disk storage. Communication with other nodes involved **network access** and is associated with considerable overhead compared to local memory access
- In reality, even shared memory machines have variable access speed and special architecture: other approaches do not scale well to 1000s of nodes. Example: CC-NUMA (Cache Coherent Non-Uniform Memory Access)
- For distributed memory machines, a single node can be an off-the-shelf PC or a server node. Individual components are very cheap, the approach scales well and is limited by the speed of (network) communication. This is the cheapest way of creating extreme computing power from standard components at very low price
- Truly massively parallel supercomputers are an architectural mixtures of local quasi-shared memory and fast-networked distributed memory nodes. Writing software for such machines is a completely new challenge

Parallel Computers

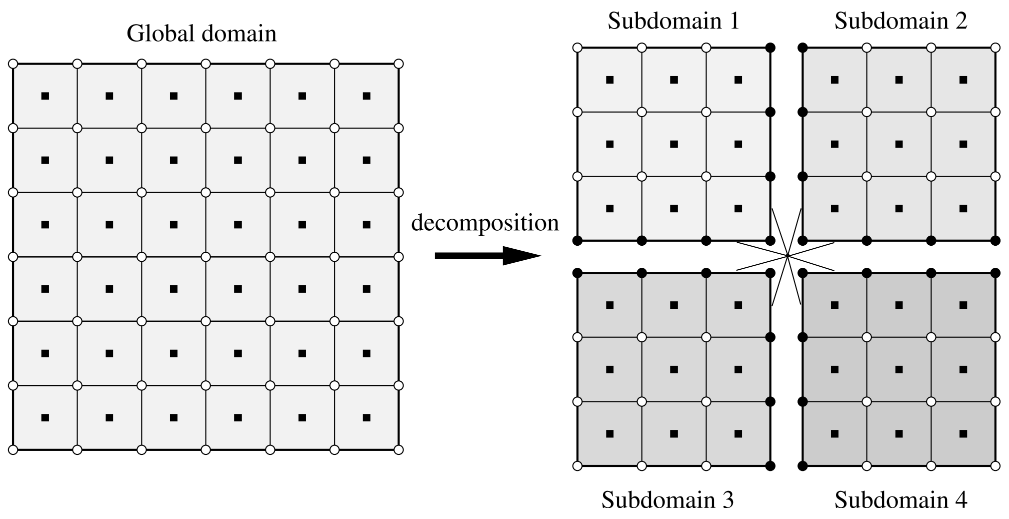
Coarse- and Fine-Grain Parallelisation

- We can approach the problem of parallelism at two levels
  - In **coarse-grain parallelisation**, the simulation is split into a number of parts and their inter-dependence is handled algorithmically. Main property of coarse-grain parallelisation is **algorithmic impact**. The solution algorithm itself needs to account for multiple domains and program parallel support
  - **Fine-grain parallelisation** operates on a loop-by-loop level. Here, each look is analysed in terms of data and dependency and where appropriate it may be split among various processors. Fine-grain action can be performed by the compiler, especially if the communications impact is limited (*e.g.* shared memory computers)

Parallel Computers

Coarse- and Fine-Grain Parallelisation

- In CFD, this usually involved **domain decomposition**: computational domain (mesh) is split into several parts (one for each processor): this corresponds to coarse-grain parallelisation



- While fine-grain parallelisation sounds interesting, current generation of compilers is not sufficiently clever for complex parallelisation jobs. Examples include algorithmic changes in linear equation solvers to balance local work with communications: this action cannot be performed by the compiler

Domain Decomposition Approach

Functionality

- In order to perform a parallel FVM simulation, the following steps are performed:
    - ○ Computational domain is split up into meshes, each associated with a single processor. This consists of 2 parts:
        - ∗ Allocation of cells to processors
        - ∗ Physical decomposition of the mesh in the native solver format

      Optimisation of communications is important: it scales with the surface of inter-processor interfaces and a number of connections. Both should be minimised. This step is termed **domain decomposition**
    - ○ A mechanism for data transfer between processors needs to be devised. Ideally, this should be done in a generic manner, to facilitate porting of the solver between various parallel platforms: a standard interface to a communications package
    - ○ Solution algorithm needs to be analysed to establish the inter-dependence and points of synchronisation
- Additionally, we need a handling system for a distributed data set, simulation start-up and shut-down and data analysis tools
- Keep in mind that during a single run we may wish to change a number of available CPUs and may wish to resume or perform data analysis on a single node

Domain Decomposition Approach

Parallel Communication Protocols

- Today, **Message Passing Interface (MPI)** is a de-facto standard (http://www-unix.mcs.anl.gov/mpi/). A programmer does not write custom communications routines. The standard is open and contains several public domain implementation
- On large or specialist machines, hardware vendor will re-implement or tune the message passing protocol to the machine, but the programming interface is fixed
- Modes of communication
    - Pairwise data exchange, where processors communicate to each other in pairs
    - Global synchronisation points: *e.g.* global sum. Typically executed as a tree-structured gather-scatter operation
- Communication time is influenced by 2 components
    - **Latency**, or a time interval required to establish a communication channel
    - **Bandwidth**, or the amount of data per second that can be transferred by the system

Domain Decomposition Approach

Mesh Partitioning Tools

- The role of mesh a partitioner is to allocate each computational point (cell) to a CPU. In doing so, we need to account for:
  - Load balance: all processing units should have approximately the same amount of work between communication and synchronisation points
  - Minimum communication, relative to local work. Performing local computations is orders of magnitude faster than communicating the data
- Achieving the above is not trivial, especially if the computing load varies during the calculation

Handling Parallel Computations and Data Sets

- The purpose of parallel machines is to massively scale up computational facilities. As a result, the amount of data handled and preparation work is not trivial
- **Parallel post-processing** is a requirement. Regularly, the only machine capable of handling simulation data is the one on which the computation has been performed. For efficient data analysis, all post-processing operations also need to be performed in parallel and presented to the user in a single display or under a single heading: parallelisation is required beyond the solver
- Mesh generation is also an issue: it is impossible to build a complete geometry as a single model. **Parallel mesh generation** is under development

Parallel Algorithms

Mesh Support

- For purposes of algorithmic analysis, we shall recognise that each cell belongs to one and only one processor
- Mesh faces can be grouped as follows
  - Internal faces, within a single processor mesh
  - Boundary faces
  - Inter-processor boundary faces: faces used to be internal but are now separate and represented on 2 CPUs. No face may belong to more than 2 sub-domains
- Algorithmically, there is no change for internal and boundary faces. This is the source of parallel speed-up. Out challenge is to repeat the operations for for faces on inter-processor boundaries

Parallel Algorithms

Gradient Calculation

- Using Gauss' theorem, we need to evaluate face values of the variable. For internal faces, this is done trough **interpolation**:

  $$\phi_f = f_x \, \phi_P + (1 - f_x) \, \phi_N$$

  Once calculated, face value may be re-used until cell-centred ϕ changes
- In parallel, ϕ_P and ϕ_N live on different processors. Assuming ϕ_P is local, ϕ_N can be fetched through communication: this is once-per-solution cost and obtained by pairwise communication
- Note that all processors perform identical duties: thus, for a processor boundary between domain A and B, evaluation of face values can be done in 3 steps:
  1. Collect internal cell values from local domain and send to neighbouring processor
  2. Receive neighbour values from neighbouring processor
  3. Evaluate local face value using interpolation

Parallel Algorithms

Matrix Assembly

- Similar to gradient calculation above, assembly of matrix coefficients on parallel boundaries can be done using simple pairwise communication
- In order to assemble the coefficient, we need geometrical information and some interpolated data: all readily available, maybe with some communication
- Example: off-diagonal coefficient of a Laplace operator

$$a_N = |\mathbf{s}_f| \frac{\gamma_f}{|\mathbf{d}_f|}$$

  where γ_f is the interpolated diffusion coefficient (see above). In actual implementation, geometry is calculated locally and interpolation factors are cached to minimise communication
- Discretisation of a convection term is similarly simple
- Note: it is critical that both sides of a parallel interface calculate the identical coefficient. If consistency is not ensured, simulation will fail
- Sources, sinks and temporal schemes all remain unchanged: each cell belongs to only one processor

Parallel Algorithms

Linear Equation Solvers

- Major impact of parallelism in linear equation solvers is in choice of algorithm. For example, direct solver technology does not parallelise well, and is typically not used in parallel. Only algorithms that can operate on a fixed local matrix slice created by local discretisation will give acceptable performance
- In terms of code organisation, each sub-domain creates its own **numbering space**: locally, equation numbering always starts with zero and one cannot rely on **global numbering**: it breaks parallel efficiency
- With this in mind, coefficients related to parallel interfaces need to be kept separate and multiplied through in a separate matrix update
- Impact of parallel boundaries will be seen in:
  - Every matrix-vector multiplication operation
  - Every Gauss-Seidel or similar smoothing sweep

  . . . but nowhere else!
- Identical serial and parallel operation
  - If serial and parallel execution needs to be identical to the level of machine tolerance, additional care needs to be taken: algorithmically, order of operations needs to be the same
  - This complicates algorithms, but is typically not required

Parallel Algorithms

Synchronisation

- Parallel domain decomposition solvers operate such that all processors follow identical execution path in the code. In order to achieve this, some decisions and control parameters need to be synchronised across all processor
- Example: convergence tolerance. If one of the processors decides convergence is reached and others do not, they will attempt to continue with iterations and simulation will lock up waiting for communication
- **Global reduce** operations synchronise decision-making and appear throughout high-level code.
- Communications in global reduce is of gather-scatter type: all CPUs send their data to CPU 0, which combines the data and broadcasts it back
- Actual implementation is more clever and controlled by the parallel communication protocol