Parallelization issues

Georg Huhs

June 15, 2014

1 Introduction

Not too many years ago overclocking CPUs and estimating the clock frequencies of the next processor generation were the hot topics in the world of gamers and other computer fans. Computing power rose with the frequencies, while transistors became smaller and smaller, so that more of them fit on a single chip, fulfilling the famous law of Moore. But at a certain point it was, amongst other problems, not possible to dissipate the heat from the areas of intense computing, causing a stagnation of the clock frequencies in the early 2000s.

Instead, for increasing the computing power further, manufacturers started to put several computational cores into one processor. Two 3GHz cores are not as fast as one 6GHz processor, but definitely faster than just one, and technically feasible, as well as 4, 6, or even more cores as in modern PCs. While powerful but expensive supercomputers have always been parallel computers, parallelism is a necessity even when using workstations, if we don't want to get stuck at the level of the early 2000s.

The advantage of parallel structures is their scalability. But for utilizing the full computational potential, also the software has to be adopted to the parallel computing approach. And sometimes simple parallelization is not enough. Serial data dependencies can make a parallelization impossible. In such a case the only way out is to exchange the algorithm completely.

Here we don't bother about the programming side, but also on the user's side some understanding of the basic concepts helps a lot for using parallel systems efficiently.

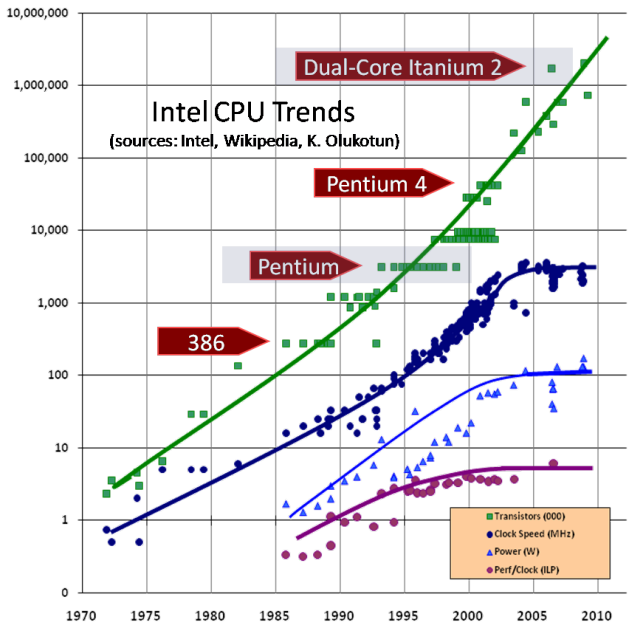


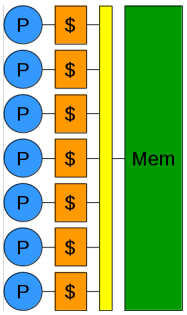
Figure 1: Development of processor performance over time [4]. The clock rates (dark blue) stopped increasing in the early 2000s, but the transistor count (green) continued growing due to the use of multicore architectures, thus Moore's law is still obeyed.

2 Basics

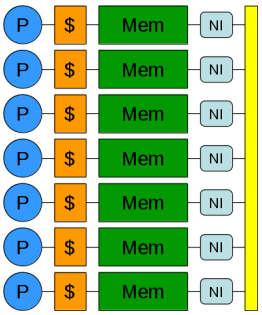
There are different ways of using several computing units in parallel which differ mainly in their access to memory and interconnection, as depicted in figure 2.

`Shared memory` Up to now we talked about several "cores" in a single socket ("multicore"). They communicate with each other basically over the common, thus "shared", memory, and the parallel execution paths are called "threads". This is a very simple approach with short communication ways and little hardware effort, widely used in workstations. It is possible to run more than one thread per core, but since all cores need to fit into a single socket, their number is limited. It is possible to fit several sockets into one "node". But as there are usually not more than 4 sockets per node, a more general concept is needed for a computer of arbitrary size.

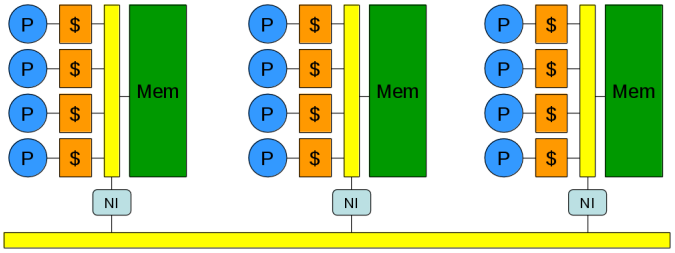
`Distributed memory` The execution paths are called "processes", each one with its own memory, not visible to the others, communicating via messages. In terms of hardware this means, that we have many processing units on many sockets with some interconnection. Often cheap and easily available standard hardware like Intel multicore processors is used. This means there are several cores per socket. Even if they could share the memory, it is possible to distribute it logically, so that each core has access to its



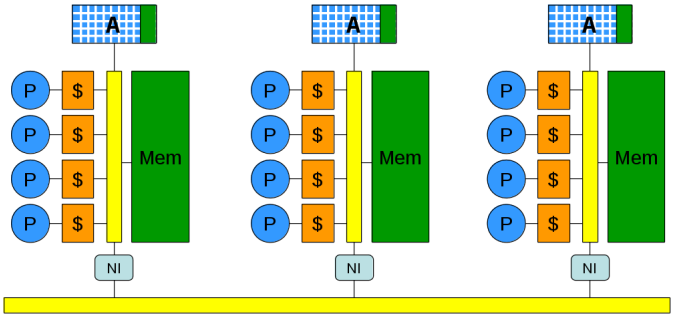
(a) Shared memory



(b) Distributed memory



(c) Combining shared memory nodes with a network connection.



(d) Layout like in (c), with each node hosting a massively parallel coprocessor with its own memory.

Figure 2: Various types of parallel hardware. The meaning of the symbols is: P...Processor, $...Cache, Mem...Memory, NI...Network Interconnect, A...accelerator, and yellow stripes are buses/networks.

share only. On many machines this leaves not more than 2 GB per core, often not enough for a large application. In this case it is possible to reserve several cores for one process, of which only one will be used for computations.

`Hybrid systems` If we would like to have the option to share memory and having a scalable system at the same time, we need a so called hybrid approach. In this case a process runs on a group of cores that share their memory using threads, while the processes work like in a distributed memory environment. The maximum number of cores per process is limited by the hardware (size of a node), but an arbitrary number of nodes can be put together.

`Coprocessors` These are specialized computing devices exploiting massive parallelism. The best known example are Graphics Processing Units (GPUs). They are located on an extra card, connected to a "host", which is a standard multicore processor. Each accelerator has its own memory, so data needs to be transferred from and to the host, forming a considerable bottleneck.
GPUs have an extremely high computational throughput. The maximum performance of the AMD Radeon R9 295X2 is 11.5 TFlop, which equals the performance of top supercomputers only 10 years ago. But due to their special architecture only suitable algorithms with a specialized implementation can use this potential. A more general approach is given by Intel's MIC architecture, which resembles more a multicore processor, but with 60 computational cores, giving it the name "many-core.

`Future systems` In the moment the world of supercomputing is becoming even more diverse. Some of the emerging technologies are:

- The next generation of Intel MICs will not need hosts any more, simplifying the whole system significantly.
- The main challenge for the next big computers is energy efficiency. Several projects tackle this by using energy saving chips of mobile devices combined with GPUs.

For running a scientific application on a supercomputer the computing centers provide the platform and the code owners provide high performance versions of the programs, trying to let the scientists do their work dealing with the infrastructure as little as possible. But a few things remain:

`Machine level` The user has to define how he would like to use the machine. Big computers usually feature a queuing system, which needs at least the information about how many processors shall be used, and an upper limit for the execution time. With this data the execution is scheduled and the user has to wait for it to start, and hopefully end well.
In many cases working out the characteristics of the hard- and software rewards the user with a more efficient use of the system. For example running a multithreaded program on 160 cores of a computer with 16

cores per node, the user has to choose if he wants to run on 10 processes, each with 16 threads, or 160 processes with one thread each. It is also possible, that multithreading allows using more cores, for example in this case using 160 processes with 4 threads might bring down the time to solution considerably, while using 640 single-threaded processes would not.

`Application level` Ideally the application does not need to be told anything about how it should use the available resources. But usually tuning parallelization-related program parameters can improve the performance, and sometimes they even have to be provided.
An example is the PEXSI solver [2], which requires the user to define a number of processor-groups and the amount of processors in the groups, resulting in a set of possible (meaningful) allocations.

Higher specialized or optimized code gives better performance, but also requires more effort to use it efficiently. The same applies to hardware, for example only an algorithm intrinsically featuring massive parallelism, working on a problem with sufficient data to process, can take advantage of a GPU.

3 Siesta in parallel

Siesta features, for now, only distributed memory parallelism in several ways:

- Since the computations on distinct k-points are independent of each other, they can be distributed easily. The resulting parallelization is very efficient, but limited by the number of k-points. It is activated by the fdf option `ParallelOverK`.
- A more general approach is distributing orbitals and spacial grid-points, which is done when using the `diagon` and `orderN` solvers. This allows using an arbitrary number of processors, but practically it does not make sense to go beyond a problem-dependent amount of cores.
- The PEXSI solver features two levels of parallelism: The higher level is a very efficient distribution of the independent "poles". The computations for each pole themselves are also parallelized. The number of poles is related to the desired accuracy, while memory needs and scalability set limits for the number of processors per pole.

For building and running Siesta in parallel a suitable environment has to be provided:

`MPI` is a basic "message passing" communication layer for shared memory architectures and includes basic libraries, compiler wrappers, and an MPI execution binary. Due to its wide distribution it is a quasi-standard for distributed memory architectures.

`BLACS` builds on top of MPI a layer of communication routines specialized for linear algebra operations.

`SCALAPACK` is the parallel version of LAPACK, using MPI and BLACS for parallel tasks, as well as BLAS and LAPACK for (serial) operations within a process.

MPI, BLACS, SCALAPACK, ... are standards only defining interfaces and functionalities. For each of these libraries there are several implementations, often provided by vendors, optimized for their hardware. One measure to get the best performance possible is to link with optimized vendor's libraries like MKL for Intel based computers.

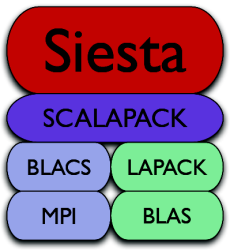


Figure 3: Software stack for a parallel Siesta installation.

3.1 Building parallel Siesta

Building Siesta in parallel is configured in the `arch.make` file, where a parallel compiler and some libraries have to be defined. The configuration for MareNostrum, the supercomputer in Barcelona, is given in Listing 1.

The option `BSC_CELLXC` activates improvements in the construction of the Hamiltionan, in particular when using many processors, but is not compatible with some other features, like the VdW functional.

```
## MPI wrappers to compiler
FC=mpif90

## Flags activating MPI and advanced timing also for MPI calls
FPPFLAGS= -DMPI -DMPI_TIMING       -DFC_HAVE_FLUSH -DFC_HAVE_ABORT
# eventually also -DBSC_CELLXC

## Parallel linar algebra libraries, in this case the MKL implementation
BLACS_LIBS=-Wl,-rpath,/apps/INTEL/mkl/lib/intel64/ ↩
    ↪ -L/apps/INTEL/mkl/lib/intel64/ -lmkl_blacs_openmpi_lp64
SCALAPACK_LIBS=-Wl,-rpath,/apps/INTEL/mkl/lib/intel64/ ↩
    ↪ -L/apps/INTEL/mkl/lib/intel64/ -lmkl_scalapack_lp64 ↩
    ↪ -lmkl_intel_lp64 -lmkl_core -lmkl_sequential
LIBS = $(SCALAPACK_LIBS) $(BLACS_LIBS) -lstdc++

#SIESTA needs an F90 interface to MPI
#This will give you SIESTA's own implementation
#If your compiler vendor offers an alternative, you may change
#to it here.
MPI_INTERFACE=libmpi_f90.a
MPI_INCLUDE=.
```

Listing 1: Relevant lines in `arch.make` for compiling Siesta on MareNostrum.

3.2 Running Siesta in Parallel

For running an application in parallel the system has to be told which resources to use. This is done by calling the application indirectly, for example by `mpirun`:

```
mpirun -n 16 siesta < example.fdf
```

If a queuing system is used, this can't be called directly, but by a launch script, where essential parameters and the call to execute are defined. One example for the LSF system, as used on MareNostrum:

```
#! /bin/bash
#BSUB -J Siesta
#BSUB -cwd ./
#BSUB -n 16
#BSUB -oo std.out
#BSUB -eo std.err
#BSUB -R"span[ptile=16]"
#BSUB -W 00:20

mpirun siesta < Si001+H2.fdf
```

Listing 2: `launch.sh` - Script for running a parallel job on MareNostrum

This job is put into the queue by calling

```
bsub < launch.sh
```

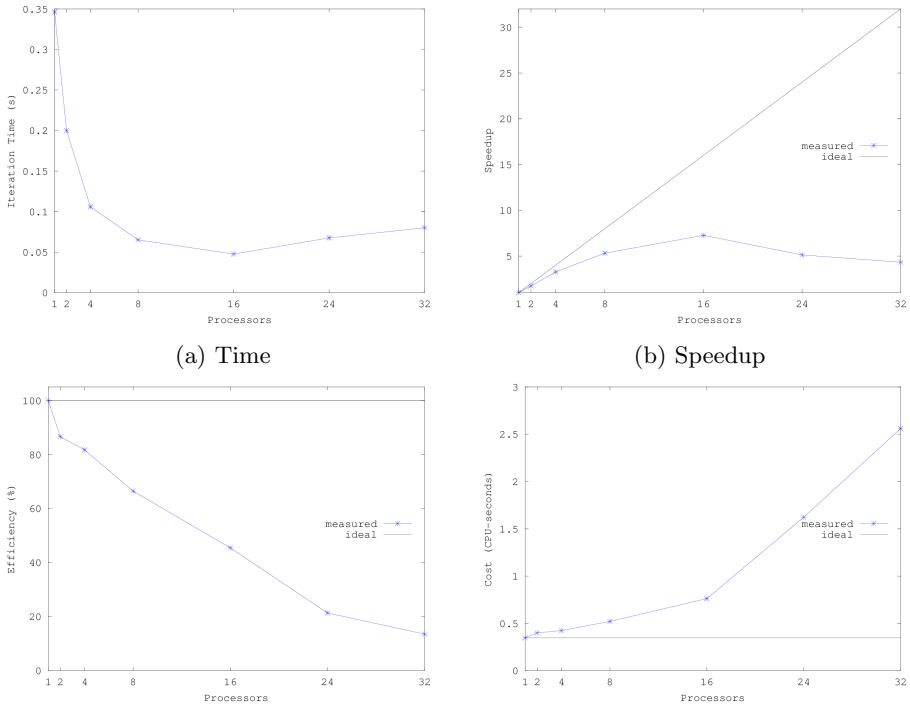
Further there are commands for checking the status of a job, deleting it, ... They can be obtained from the documentation of the system one is using.

4 Scaling

Parallelizing a program involves introducing communication and synchronization, so that time is wasted for waiting for results. The more processors are used for a fixed problem, the more communication is needed while the portions of computation become smaller and smaller. At a certain point, increasing the number of processors does not decrease the **time to solution** any more. Figure 4a demonstrates this for an example featuring 122 orbitals, which can be distributed well to up to 16 processors, while using more processors yields an increase in time.

The time to solution is important for the end user, but does not tell us directly how well a parallelization works. The ratio between the times of the serial and parallel execution is called **speedup**, and can be compared to the ideal value, which equals the number of processors (using twice as many processors should give half the time). This is demonstrated in figure 4b.

Comparing the achieved speedup with the ideal one we get the **efficiency** of the parallelization, depicted in figure 4c. While the speedup graph still shows some scaling up to 16 processors, the efficiency constantly degrades. When applying for computation time at an HPC facility, it is important to show that one can use the resources *efficiently*.



(a) Time (b) Speedup

(c) Efficiency (d) Cost

Figure 4: Analysis of the execution of the Si000-H2 molecular dynamics example with various numbers of processors, based on the average time per SCF iteration.

Allocations on supercomputers are usually granted in CPU-hours, which corresponds to the total **cost** of a computation in terms of time × number of cores. As figure 4d reveals, using 8 or 32 processors gives approximately the same solution time, but the cost increases by a factor of 5.

How well an application scales on a certain hardware depends on the algorithms, their implementation, and the physical problem. If there is more data to process, the work can be distributed more easily, allowing using more processors. With growing problem size, solving the KS eigenproblem takes over most of the time. The effort for this part scales cubically with the number of orbitals, so it is not feasible to solve systems beyond a certain extend. Only profound changes on the algorithmic level can find a remedy. One example is the recently developed PEXSI solver, which reduces the computational complexity and allows the usage of tens of thousands of processors. The price to pay is, that it needs some configuration from the user, and, as figure 6 shows, it is faster than the standard approach only beyond a certain system size. The figure also demonstrates, that for large systems the advantage reaches orders of magnitude, a high gain for just spending a little time on studying the options.

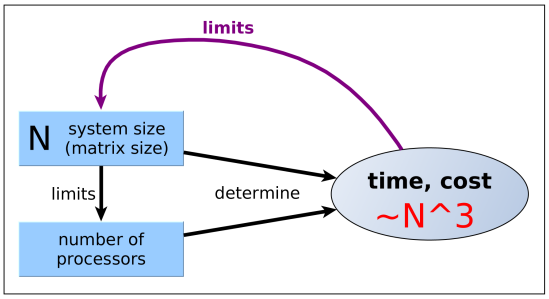


Figure 5: The relations between system size, number of processors, and time to solution. For a certain problem size the number of processors that can be used efficiently is limited. These two parameters define the time needed for the computation. Unfortunately the computational effort scales cubically with the number of orbitals, so increasing the system's dimensions or basis quickly leads to unfeasible effort. Or, seen the other way around, a given allocation or maximum time sets a limit to the system size. A very helpful tool for estimating the effort for a calculation can be found on http://departments.icmab.es/leem/siesta/siestimator/siestimator.php

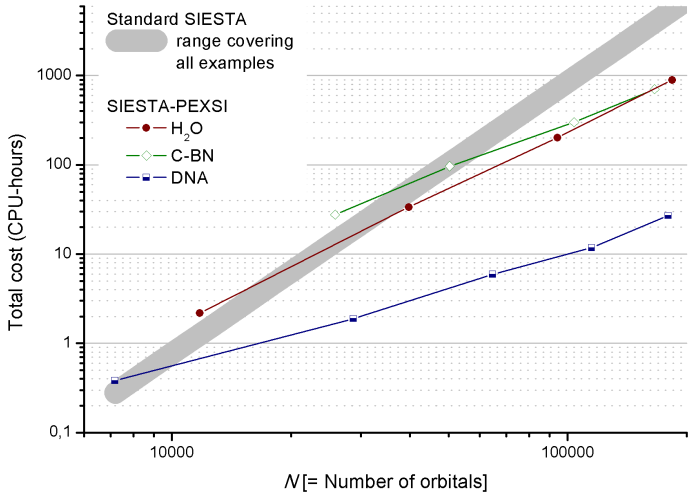


Figure 6: Cost for solving several systems depending on the number of orbitals N ("Weak Scaling"). (Taken from [2].) The solution time with standard Siesta using diagonalization depends almost on N only, while for PEXSI the sparsity of the system and its dimensionality determine the effort. Water is a 3D system, featuring $\mathcal{O}(N^2)$ scaling, C-BN is two-dimensional ($\mathcal{O}(N^{3/2})$ scaling), and DNA shows as 1D system linear scaling. The diagram demonstrates the favorable weak scaling of PEXSI and the crossover-points with diagonalization, which correspond to system sizes of a few thousands of atoms.

5 Access to computing facilities

Relatively cheap standard components and solutions make parallel computers acquirable with a moderate budget, so that many universities or enterprises host such small clusters. Bigger machines are usually located at special centers like the Barcelona Supercomputing Center (BSC), which is also the head of the Spanish supercomputing network RES (http://www.res.es/). On the European level there is a supercomputing initiative called PRACE (http://www.prace-ri.eu). These organizations offer access to the computers they manage in regular calls. For applying one has to explain the scientific content as well as need and the means to use HPC resources. In particular for the PRACE calls one should ask for at least a few thousands of cores, and also the scaling of the code has to be demonstrated. For this purpose PRACE offers two types of projects:

`Preparatory Access` for optimizing codes and doing scaling analysis in order to prepare for a large project.

`Project Access` for compute-intensive applications with mature software.

Successful applications get a certain amount of CPU-time at a specific machine (or several ones) granted. Then using these facilities is free of charge.

References

[1] E. Artacho, E. Anglada, O. Dieguez, J. D. Gale, A. García, J. Junquera, R. M. Martin, P. Ordejón, J. M. Pruneda, D. Sánchez-Portal, and J. M. Soler. The siesta method; developments and applicability. *J. Phys.: Condens. Matter*, 20, 2008.

[2] L. Lin, A. García, G. Huhs, and C Yang. SIESTA-PEXSI: Massively parallel method for efficient and accurate ab initio materials simulation without matrix diagonalization. *Accepted in J. Phys.: Condens. Matter*.

[3] J. M. Soler, E. Artacho, J. D. Gale, A. García, J. Junquera, P. Ordejón, and D. Sánchez-Portal. The SIESTA method for ab initio order-N materials simulation. *J. Phys.: Condens. Matter*, 14:2745–2779, 2002.

[4] H Sutter. The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm.