Master's Thesis

High Performance Computing Applications

Albert Roig Rebato

Tutors:
Clauido Dalla Vecchia,
Andrea Negri

A thesis submitted in partialy fullfilment of
the Master's degree in Astrophysics

in the

Faculty of physics and mathematics
Universidad de La Laguna

June 2018

Contents

Resumen

La ciencia computacional puede considerase ya el tercer pilar del método científico. En los últimos años se han puesto muchos esfuerzos en mejorar el poder de cómputo en las supercomputadoras. Todas las computadoras se actualizan cada 3-4 años ofreciendo grandes novedades en el hardware. Sin embargo, los códigos no se adaptan a estos cambios, de forma que no se le saca todo el rendimiento que se podría.

Una de las formas de aumentar el poder de las supercomputadoras es el *paralelismo masivo*. Un ordenador, o nodo, está formado por dos partes características: el procesador (CPU), que se encarga de ejecutar las instrucciones de los programas; y la memoria, que es donde se encuentran todos los datos. Las supercomputadoras modernas consisten en miles de nodos conectados con múltiples procesadores cada uno.

Generalmente se usan dos librerías para la computación en paralelo: OpenMP y MPI. Ambas son implementables en C/C++ y Fortran. OpenMP es una API que permite paralelizar programas en arquitecturas de memoria compartida (en *multi-core nodes*). Partes del programa se ejecutan en un "hilo principal" (del inglés *master thread*), y otras en "hilos esclavos" (*slave threads*). Todos los "hilos" comparten variables en la misma dirección de memoria, por tanto, pueden acceder y modificarlas sin necesidad de comunicación entre "hilos".

MPI significa *Message Passing Interface*. Se trata de una librería para pasar datos de la dirección de memoria de un proceso a otro proceso. El propósito de MPI es la comunicación entre procesos en arquitecturas de memoria distribuida, aunque también es implementable en memorias compartidas. Hay varías formas de comunicación entre nodos, a grandes rasgos podríamos clasificarlas en bidireccionales o unidireccionales.

La comunicación bidireccional requiere funciones emparejadas para pasar información de un nodo a otro. Es decir, requiere que en el código de ambos procesos haya un mensaje acorde; de envío en una dirección, y de recibo en la otra. Esto se puede de distintas maneras. Mediante comunicación punto a punto, o sea, dos procesos que se comunican. O mediante funciones colectivas, en la que la comunicación involucra a más de dos procesos.

Las últimas versiones de MPI permiten usar funciones de Acceso Remoto de Memoria (*Remote Memory Access* RMA) para obtener o transferir datos de otros nodos. Estas funciones permiten crear una ventana de memoria compartida a la cuál otros procesos pueden acceder. La ventaja de este método es que permite transferencias de datos entre procesos sin necesidad de funciones emparejadas (como un `MPI_Send` necesita un `MPI_Recv`), ahorrando la comunicación entre ambas partes, este método se conoce como comunicación unidireccional (*one-sided communication*).

Una de las ventajas de OpenMP es que su implementación es simple. La mayor desventaja es que no se pude correr en arquitecturas de memoria distribuida. MPI tiene una implementación más complicada pero permite su ejecución en cualquier arquitectura e, incluso, puede llegar a ser más eficiente que OpenMP en memorias compartidas.

Los *profilers* se han convertido en una herramienta imprescindible para cualquier programador. Un *profiler* se encarga de recoger información a lo largo de la ejecución de un código. Por ejemplo, la cantidad de memoria usada por el programa o, el tiempo que pasa en cada función. En este trabajo utilizamos SCALASCA para realizar el análisis de rendimiento de software.

Este trabajo consta de dos partes claramente diferenciadas. En la primera parte, creamos un código que resuelve el problema de N-cuerpos gravitacional mediante un método de integración directa de la segunda ley de Newton. De manera que es un algoritmo $\mathcal{O}(N(N-1))$, donde N es el número de partículas. Calculamos la aceleración, la velocidad y la posición de cada partícula en cada paso de tiempo. Dado que los cálculos para cada partícula son independientes a los cálculos realizados para las otras, el problema es altamente paralelizable. Se distribuir el cálculo de las trayectorias de las partículas entre distintos procesador. Con el objetivo de probar los varios métodos de paralelización comentados, paralelizaremos los cálculos con OpenMP, MPI-punto a punto y MPI-RMA. El código está escrito en Fortran 90.

La paralelización con OpenMP es relativamente simple. Utilizamos la directriz (en inglés *directive*) `parallel do` que permite especificar al compilador la parte de código que se desea paralelizar. También utilizamos la cláusula `private` para determinar qué variables deben o no ser compartidas entre los "hilos".

La paralelización con MPI requiere comunicación entre procesos ya que las variables en cada uno son privadas. Para el cálculo de fuerzas se necesita la posición de todas las partículas, sin embargo, cada proceso calcula las posiciones de una fracción de ellas. Por lo tanto, los procesos deberán comunicarse para tener acceso a las posiciones actualizadas. La comunicación MPI-punto a punto se hace con el método del anillo, es decir, un proceso envía datos al siguiente proceso y recibe del anterior. La comunicación entre procesos se realiza mediante las funciones: `MPI_Isend` y `MPI_Irecv`. La particularidad de éstas es que son funciones sin bloqueo (*non-blocking functions*).

Para la comunicación MPI-RMA se utilizan las funciones `MPI_Win_create` y `MPI_Get`, con las que se crea un espacio de memoria compartida y se accede a los espacios de los otros procesos, respectivamente.

Medimos el tiempo de ejecución en un ordenador de memoria compartida con tal de determinar cuál es el método con mejor rendimiento. El método MPI-RMA es el que da mejores resultados, llegando a hacer el código 6.9 veces más rápido cuando se usan 8 procesadores que cuando se corre el código en serie. En segundo lugar, el método MPI-punto a punto que hace el código 6.7 veces más rápido para el mismo número de procesadores. Finalmente OpenMP, consigue acelerar 6.1 veces el código con 8 procesadores.

La segunda parte consiste en la optimización de un código en serie que sintetiza líneas espectrales. Para ello implementamos métodos de HPC. En este artículo solo nos centramos en la computación, la física no se discute. Lo primero que hacemos es un *profile* del código para encontrar posibles puntos conflictivos (*bottle-necks*). Encontramos que un 56% del tiempo de ejecución se pasa en una función. Así pues, nos centramos en la optimización de está función. Se trata de un código en Fortran 77 que utiliza funciones que están en desuso y deben ser actualizadas. Por ejemplo, eliminamos los COMMON *blocks* ya que son método de organizar las variables muy conflictivo. Finalmente, paralelizamos la función en la que se encuentra el *bottle-neck* mediante OpenMP.

Obtenemos un código hasta 1.28 veces más rápido cuando se ejecuta con 6 procesadores que con el código en serie. Usando más procesadores el aumento de velocidad del código disminuye. Concluimos que el uso óptimo del código es ejecutarlo con 4 procesadores, alcanzando una ejecución 1.27 veces más rpida.

1 Introduction

1.1 Parallel computing

Computer simulations have become an essential tool for understanding processes occurring in the Universe, as large structure formation or galaxy mergers, which due to their long time scales cannot be observed. Hydrodynamical cosmological simulations give detailed information of the period of large-scale formation, including galaxy formation and evolution and the formation of galaxy clusters and superclusters. Massive parallel codes are indispensable to solve a problem of such dimension. For example, the EAGLE simulation (one of the largest hydrodynamical cosmological simulations) uses nearby 10^9 particles to study the Universe evolution. Otherwise, one would not be able to allocate such large amounts of memory and it would take an unreasonable computational time.

High performance computing (HPC) hardware is constantly improving, currently supercomputers consist of a multi-core computing nodes network, and software should be adapted to all changes to reach best performance.

1.1.1 Overview

Solving a problem which involves large amounts of data requires a lot of computational time and memory. Nowadays, one might say the only way to do it is by mean of parallel computing.

Traditionally, codes have been written for serial computation, where a problem is solved executing a series of instructions one after another. This kind of software runs on a single CPU so only one calculation can be done at the same time.

Parallel computing is solving a computational problem carrying out more than one calculation simultaneously. Now the problem is broken into parts which can be solved independently. These parts are distributed in different CPU's so they can be executed simultaneously. Obviously, this procedure can save wall time, but on the other hand, it allows to allocate data in different cores (depending on the computer architecture) which means one can have larger amounts of data. The typical parallel compute resources are either a single computer with multiple processors or more than one of such computers connected by a network.

Given that several events in nature happen concurrently and are related to each other parallel computing has copious applications, the following are some examples of an endless list:

- Cosmological simulations (i.e. GADGET-2)
- Plasma and solar physics, it is used to solve the magneto-hydrodynamic equations (i.e.

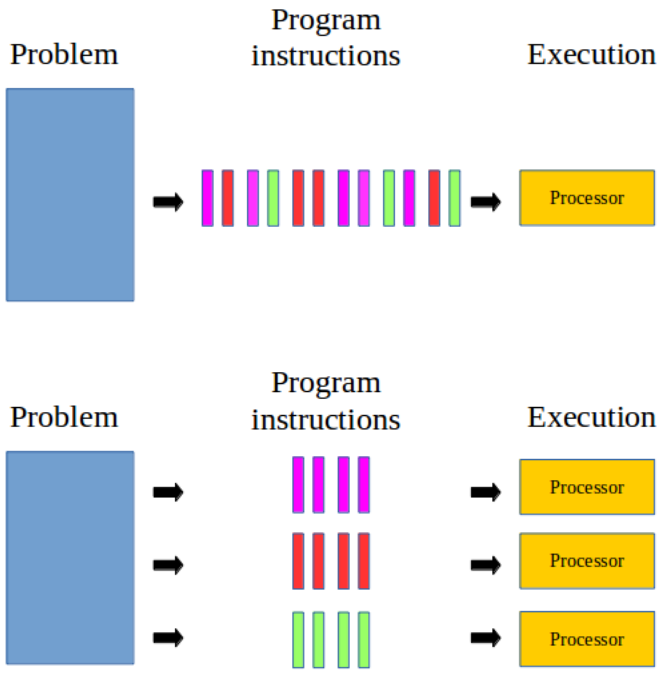

Figure 1: On top, problem solved with a serial program. The problem is borken into a series of instructions which are executed one after another. On bottom, problem is solved with a parallel program. The instructions are distributed among various processors where they are executed. As it is shown each processor has less work in the parallel code than in the serial, so the execution time shall be reduced. This looks like an embarassingly parallel program but communication may be required.

BIFROST).

- Molecule folding and large molecules dynamics (i.e. NAMD).
- DNA and protein sequences analysis.
- Numerical weather prediction and climate change.
- Traffic modelling (i.e. TRANSIMS or AIMSUM).

Of course, parallel programming has some limitations. Amdahl's law relates the program's speedup with the number of processors one may use and it depends on the fraction of code which can be parallelized. The expression for an optimal speedup is,

$$speed\ up = \frac{1}{S + \frac{P}{N_{proc}}} \tag{1}$$

where S is the serial fraction of the code, P is the parallel fraction, so S+P=1, and N_{proc} is the arbitrary number of processors. The parallel fraction must be P<1 and one can notice that if $P \neq 1$

this function tends to a constant value. Hence, speedup will not increase no matter how many processors one uses to execute the program. This limitation is inherent to the problem, one cannot write a code with better performance.

1.1.2 Parallel Computer Architectures

Nowadays, the two approches to increase supercomputers power are: massive parallelism, i.e. large matrices of connected nodes, and accelerators (e.g. GPU's) that are not considered here, but will be briefly described later.

The von Neumann architecture consists in a central processing unit (CPU), a memory and input/output mechanisms. The CPU consists of an arithmetic/logic unit and a control unit. The first carries out the operations, and the latter interprets and executes the instructions stored in the main memory and generates the control signals needed to execute them. All together they constitute a node. Modern computers do not have this specific architecture but the pattern is the same: CPU+memory. All the following architectures follow this pattern.

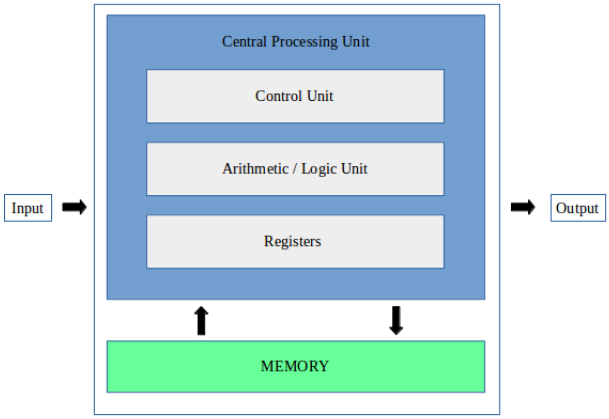

Figure 2: Simple representation of a node/computer, comprised of one CPU and one memory.

- **Shared Memory:** Parallel computers with shared memory architecture have multiple processors that can operate independently and have full access to a common memory. Given the closeness between processors and memory the main advantage of this configuration is that data sharing between tasks is both fast and uniform. The access to memory is of the order of nanoseconds. Nevertheless, adding more CPU's leads to a lack of scalability between CPU's data transfer and the shared memory.

- **Distributed Memory:** Distributed memory consist of multiple nodes, which contain one memory and one CPU, connected by a network. This configuration avoids the scalability

problem, both the number of CPU's and memory increase proportionally. However, data transfer is done through the network (usually optic fibre) so memory access is non-uniform and data traffic between distant nodes takes longer. The delay between sending data from one process and receiving it from another is known as *latency*.

- **Hybrid Shared-Distributed Memory:** Nowadays, modern supercomputers have an hybrid Shared-Distributed memory architecture which consist of multiple shared memory nodes connected by a network. The advantages and disadvantages are those from each configuration, yet the scalability problem is fixed.

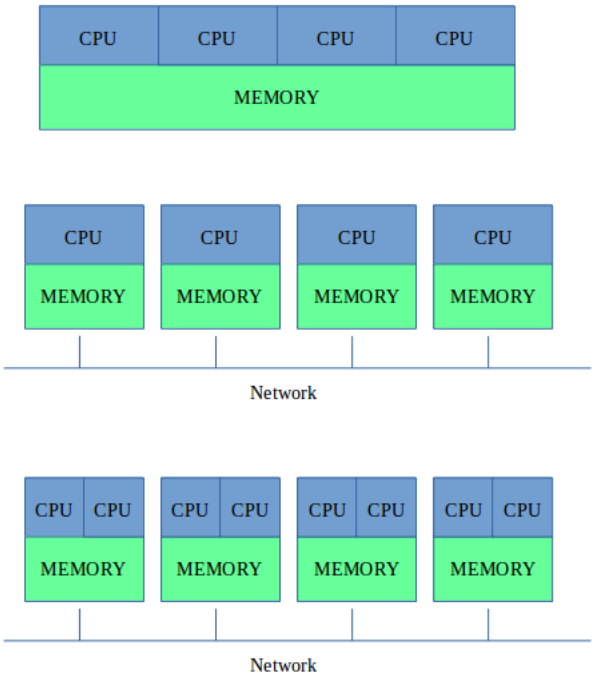

Figure 3: On top, shared memory architecture, one node of multiple CPU's and one memory. In the center, distributed memory, multiple nodes of one CPU and one memory. On bottom, the hybrid Shared-Distributed memory, a network of nodes, where each node has multiple cores and one memory. The hybrid Shared-Distributed architecture is used in all current supercomputers.

- **Graphics processing unit (GPU):** A Graphics processing unit (GPU) is a coprocessor outfitted with a highly parallel microprocessor and a private memory with very high bandwidth. GPU's are specifically disigned to data processing rather than data caching and flow control, making them more efficient for the calculation of graphic information than CPU's. The GPU can lighten the load of information that must be processed by the central unit, and so, the latter can do its job more efficiently.

1.2 Libraries used for parallel communication

There are two main approaches to parallel programming: loop-level or parellel regions. It can also be referred as level of parallelism or granularity. The grain-size stands for a measure of the amount of work performed by each task.

The first refers to individual loop which can be parallelized. This procedure is called *fine-grained parallelism*, where each processor/thread is responsible to execute a low number of tasks. It is useful in architectures with low communication overhead. Thus, fine-grained parallelism should be used in shared memory applications. The grain-size is below $\sim 5 \cdot 10^2$ instructions and short time for communication, $\sim \mu s$.

The latter is focused on parts of codes which can be parallelized. It is known as *coarse-grained parallelism* and it is better suited for distributed memory architectures. The program is broken down to large tasks, the grain-size is above $\sim 10^4$ instructions and long time for communication, $\sim 10s$.

General-purpose parallel computers offer a compromise between fine and coarse-grained parallelism, also known as *medium-grained parallelism*.

1.2.1 OpenMP

OpenMP is an Application Program Interface (API) used for multi-threaded parallel processing on shared memory multi-processors nodes. OpenMP is supported by C/C++ and Fortran programs. Consists in a FORK-JOIN model, i.e. part of the program is executed by one thread (or master thread) and part by multiple threads (slaves threads).

The loop-level parallelism is done as it follows. The execution starts on the master thread, the loop index is divided among the number of threads (defined by the user, named `OMP_NUM_THREADS`), each thread executes a specific range of that index and when the loop finishes, execution continues as serial.

The main advantage of OpenMP is its rather simple implementation. Moreover, OpenMP constructs are written as comments. Therefore, codes can be compiled without modifications whether OpenMP shall be used or not.

A relevant feature of OpenMP are the two different types of variables: Shared and Private. Shared variables have the same address space and private do not. Every thread can access and modify their variables (except the loop index) so if the variable is shared, when one thread modifies it, it changes it for all threads. This can lead into erroneous results. The developer is responsible of declaring which variables are shared and which are private.

The main disadvantage of OpenMP is that code cannot be executed on distributed memory

architectures, thus, it is limited by the number of processors of one node. A purely OpenMP core may not run efficiently in more than ∼100 cores, while other libraries (e.g. MPI) can run efficently so in thousands of cores.

An open issue is a significant overhead in thread creation [6], sometimes MPI programs have better performance.

1.2.2 MPI

MPI stands for Message Passing Interface. It is a standard library that allows message passing for parallel programming in C and Fortran –where message means data–. MPI is used by almost all HPC parallel applications. Its purpose is to move data from the address space of one process to another process. MPI is not a language, but a standard clearly defined set of operations expressed as functions, subroutines, or methods. As a standard, a significant advantage is its wide portability.

MPI is meant for inter-node communication, so to be run on distributed memory multiprocessors and/or networks of workstations, thus, it must be highly scalable. In addition, it is also possible to run it on shared memory architectures.

In computing, latency is known as the delay from the beginning to the end of a communication. Given its implementation on distributed memory, latency is [6] an important issue of MPI . Therefore, the best implementation of MPI is coarse-grained parallelism reducing communication as much as possible.

There are different ways for processors to obtain data form other processors, here we discuss some of them. First versions of MPI involved "two-sided communication", so communication between a process which needs data from another process and the process holding these data. Latest versions allow the so-called *one-sided communication* which does not require synchronisation between nodes.

The "two-sided communication" can be done Point-to-point, where there is communication between two processors only. It requires matching operations by sender and receiver, so a message from the process holding the data (e.g. `MPI_Send()`) and one from the process receiving it (e.g. `MPI_Recv()`). In order to reduce latency, the messages should contain the maximum information as possible so, the number of times nodes communicate between them is reduced the necessary minimum. It can also be done through collective functions. Collective functions consist in message passing involving more than two processors, for instance, one process sharing data with all others (e.g. `MPI_Bcast()`), or one process collecting data from all others (e.g. `MPI_Gather()`). Collective communication –when usable– is much more effective than Point-to-point communication because there are no deadlocks, and the fibers in clusters organized in a way that minimize latency in collec-

tive communication. Obviously, this mode of communication requires high synchronization between processes.

MPI-3.0 and later versions allow one-sided communication, or in other words, Remote Memory Access (RMA) [5]. Here one process can access an address space of another process without communicating if and only if this process shares it. This mode of message passing is meant for programs where the data distribution is constant or slowly changing. A simple way to understand how it works is putting it in terms of public and private windows. One process can make a region of its memory addressable for all processes (as it would be in shared memory implementations), so it shall become a public window. Thus, other processes can obtain or modify the data without any participation of the remote side. The main advantage of RMA communications allows fast and/or asynchronous communication.

Some open issues of MPI are latency, when processes communicate; and high memory overheads, the program needs to be replicated for every process.

1.3 Profilers for parallel applications

Profilers are an essential tool for performance analysis, it helps making the optimisation of any program both more effective and more efficient. They are indispensable to understand the software behaviours in order to improve its performance. Profilers collect information, such as the time or memory used by a program, through a dynamic analysis. As said above, supercomputers are increasing the number of cores/nodes to improve their performance, thus, demanding higher degrees of parallelism to maximise resource utilisation. Profilers help the developer find out where the program spends most of the executing time, known as *hot-function* or *bottle-neck*, so it can be optimised.

The execution of a program could be defined as a sequence of "actions". For instance, the execution of a line of code or the communication between processes (if it is parallel). One can measure visual attributes of these actions by recording them as events [11]. The profiler interrupts the program, collects the statistics or attributes of an action, creating an event, and resumes the program. The typical output of a profiler is:

- **Profile:** A profile is a summary of the statistics of performance metrics of the events, for example the time spent in a function or the number of times a function is called. The size of a profile is linear to the size of the code so requires a small amount of storage.

- **Trace:** A trace is a detailed scan of the events. It shows when and where the events occurred. For example, a trace not only shows how many times a routine is executed, but also the times

at when it is called and the processor/thread which executes it. The size of a trace is linear to the program's instruction path so it requires a large amount of storage.

Performance analysis is done following the following steps: firstly, modifying the program to generate the events (performance data), this is known as *instrumentation*. Secondly, measuring of performance metrics of the events (creation of a profile or trace). Finally, analysing the performance data.

For serial programs a profile is sufficient. On the other hand, for parallel programs, profiles only offer a vague idea of what is happening. Given its strong dependence on time relation of events (communication between/within nodes) it requires a full trace to understand the software behaviour.

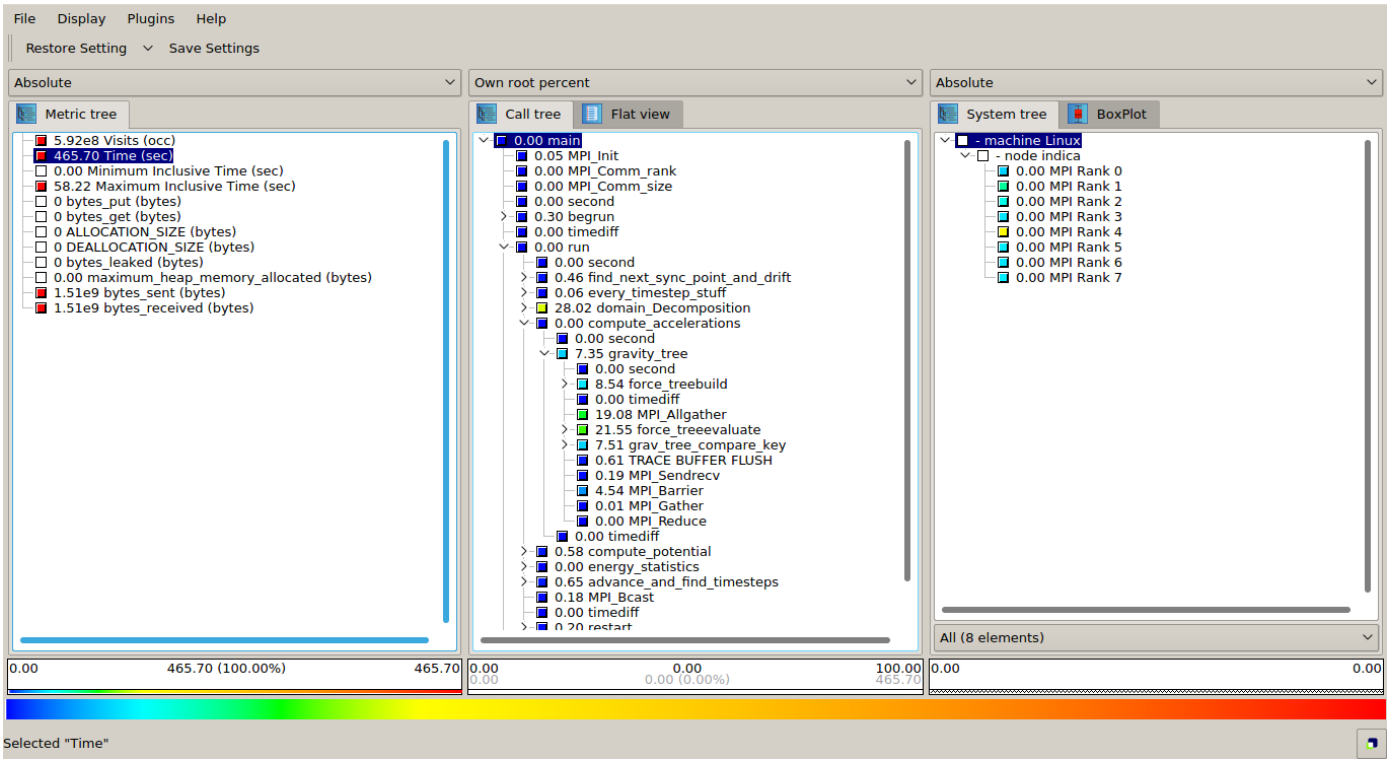

Figure 4: Profile of GADGET-2, a massive parallel code. At left, we can see various performance properties, such as, the total number of calls to any function (Visits) or the execution time (Time). In the middle, the functions and subroutines which are called are shown. In percentage, we see the time spent in each one. Thus, one can notice that, for instance, `force_treeevaluate` is a hot-function so to enhance performance one should focus on that function. At right, the distribution of work among the different processors is shown.

In the following work performance analysis is done with SCALALSCA[10]. SCALALSCA is a free and open-source profiling software, focused on MPI, OpenMP, POSIX threads, and/or hybrids:

MPI+OpenMP/Pthreads. It provides a wide tool-set to measure and analyse the runtime behaviour of parallel programs.

As the Scalalsca User Guide suggests, instrumentation and measurement of user routines, OpenMP constructs and MPI functions are handled by the Score-P instrumenter.

Figure 4 shows the trace obtained through Scalasca of an execution of the hydrodynamical cosmological code Gadget-2. For instace, some performance properties measured in our profiles – among others– are: **Time**, which stands for the total time spent for a program execution. Executions in time-shared environment, such as OpenMP, will also include time slices used by other processors. **Visits**, showing the number of times a call path is visited. And **Wait at Explicit OpenMP Barrier Time** which is the time spent in an explicit OpenMP barrier synchronisation waiting for the last thread.

2 Objectives

As discussed in §1.2, libraries for communication in parallel environments, like the Message Passing Interface (MPI), now include functions for Remote Memory Access (RMA) that should reduce the inter-node data transfer costs. Also, OpenMP multithreading should speed up computation within a node, allowing also less memory consumption. The aim of this work is to test and implement these HPC methods. It has two distinguished parts explained below: the test and the implementation.

In order to test the different methods we will write a simple brute-force N-body problem code. The program will be equivalent using the previously discussed methods of parallelisation (§1.2): MPI point-to-point communication, the MPI-RMA, OpenMP multithreading and the hybrids MPI point-to-point+OpenMP and MPI-RMA+OpenMP. We shall try to find out which method has the best performance and study open issues of MPI and OpenMP libraries. First versions of MPI have a significant problem with latency and implementing the MPI-RMA performance shoud be enhanced. OpenMP should perform better than MPI for shared memory architectures but can have a thread creation overhead issue and ending up having worse performance.

In the second part, we will try to parallelise a serial code which calculates galaxy spectra depending on different initial conditions, such as temperature or metallicity. Physics will not be discussed here, only the programming methods used.

The code is written in Fortran 77 and uses some old fashioned methods. We will update the code to a more recent version of Fortran (F90) enhancing the program performance. Using Scalasca a profile of the code shall be made so we can analyse its weaknesses, such as, possible bottle-necks,

reducing the computing time and increasing scalability using the best suited parallelisation method.

3 Methodology

3.1 Testing HPC methods on a gravitational N-body code

A gravitational N-body simulation approximates the motion of celestial bodies (stars, planets, etc.) which interact gravitationally with one another, by approximating them as massive point-like objects. This is a perfect example of a physical problem which requires computation to be solved. Otherwise, one could not test models of astrodynamics, given that experimentation is not an option. It is also a perfect example of a highly parallelisable problem. Calculations of acceleration, velocity and position for one particle, which are the only time-consuming operations, do not depend on calculations carried out for the others. Here, each process/thread is responsible for calculating these physical paramenters for a fraction of the number of particles. Once one process finishes its calculation, it has to communicate with the other processes to update the positions it did not calculate. Regarding Amdahl's Law (equation 1) one should obtain a linear decrease of computing time: $T_{parallel} = T_{serial}/N_{proc}$, where $T_{parallel}$ is the physical time spent for the execution, T_{serial} is the time it would take to run the code on one process/thread, and N_{proc} is the number of processors.

We solve the N-body problem through direct sumation method, i.e. we directly integrate the Newtonian gravitational force equation.

$$\vec{F}_n = \sum_{i \neq n}^{N} \frac{G m_n m_i}{r_{ni}^2 + \epsilon} \hat{r}_{ni} \tag{2}$$

We also add a smoothing parameter, ϵ, to avoid having particles in the same position. Another approximation we make is the discretized time step, dt, when solving Newton's second law. In other words, we calculate the interaction between one particle and each of the others and do it for every particle. Hence, usually this kind of algorithms are $\mathcal{O}(N(N-1))$. To simplify the code, the gravitational constat and the masses are set to one. Thus, calculations only depend on the distance between particles.

The code is written in Fortran 90 and calculations will be parallelised using OpenMP, MPI point-to-point and MPI Remote Memory Access functions. The code using OpenMP can only be executed on an individual computer, i.e. one single node with multiple cores. In order to compare the three codes we run them in the same computer and measure the physical execution time. As said in §1.2.1, is not sure wether OpenMP or MPI will perform better.

3.1.1 OpenMP N-body code

To parallelise the code we use the OpenMP directive `Parallel Do`. It tells the compiler which region has to parallelise, specifying a `parallel` region that contains a `do` loop.

Note that the distance between particles is going to have different values in each thread. This is only doable if and only if each thread has its own copy of the variable. To accomplish that we use the clause `private`. The rest of variables are shared and that is how OpenMP treats them by default.

```
!$omp parallel do private(i,j,dist)
DO i=1, Npart
   DO j=1, Npart
      IF (j .ne. i) THEN
         dist = (part(i)%pos(1)-part(j)%pos(1))**2.+(part(i)%pos(2)-part(j)%pos(2))**2. &
              +(part(i)%pos(3)-part(j)%pos(3))**2. + 0.000001
         part(i)%acc(1) = part(i)%acc(1) + (part(i)%pos(1)-part(j)%pos(1))/dist**(3./2.)
         part(i)%acc(2) = part(i)%acc(2) + (part(i)%pos(2)-part(j)%pos(2))/dist**(3./2.)
         part(i)%acc(3) = part(i)%acc(3) + (part(i)%pos(3)-part(j)%pos(3))/dist**(3./2.)
      END IF
   END DO
END DO
!$omp end parallel do
```

Figure 5: Parallelisation of the calculation of the accelerations for all particles with OpenMP. Variables `i,j` and `dist` are private, all others are shared. Here the loop is splited among threads, in a way that, each thread is responsible for calculating the forces applied to the particles of a range of the index `i`, i.e. a range between 1 and `Npart`. For instance, if one run it with 2 OpenMP threads, one thread would go from 1 through Npart/2 and the other, from Npart/2+1 through Npart.

3.1.2 MPI N-body code: Point-to-point and RMA

When MPI is initialised the code is replicated for every process, and so do all variables which were already declared. In other words, each process has its own variables in its address space. If one variable is modified by one process, same variable in the other processes will not. In terms of OpenMP variables, one could say all MPI variables are private. Moreover, this replication can lead to a high memory overhead. The developer must know whether is essential to allocate a given variable in a specific process or in all of them.

In our code every particle consists in a structure of three vectors: acceleration, velocity and position. We only allocate the particles which will be calculated for each process, i.e. $N = N_{part}/N_{proc}$. To guarantee synchronization, in case the number of particles is not divisible among the number of processes, the remainder will be distributed to the maximum number of processes as possible,

Figure 6: Representation of a 1-D ring communication. Each process sends its data to the next, and receives it from the previous.

so that none process will carry out more than one calculation than the others. The distribution of particles among processes is done through the collective function `MPI_Scatterv()`.

If each process has a fraction of particles, they will only be able to calculate the forces imparted by the particles within that process. To calculate the contribution of the others, communication of the particle positions between processes will be necessary. Notice that actually there is only need to receive data. This can be done point-to-point or through RMA.

Point-to-point:

In order to get the positions from other processes we implement a ring method to communicate. Firstly, each process sends its data to the next process and receives them from the previous (see Figure 6). Secondly, they send to the second next and receive from the next. Then, send to the third next and receive from the second next and so on.

To accomplish this work one can call `MPI_Send()` and `MPI_Recv()` which are blocking functions, meaning they wait to complete the communication before continuing its execution. In other words, `MPI_Send()` has to find a matching `MPI_Recv()` to resume. To avoid a deadlock, one process must initiate the communication and post its `MPI_Recv()` after the `MPI_Send()`, contrary to the other processes.

The most appropriate functions to do this work are `MPI_Isend()` and `MPI_Irecv()`. These are non-blocking functions, i.e. the function call returns before the transfer is finished. To be sure that data has arrived before starting to calculate an `MPI_Wait()` will be placed after each send/receive.

Remote Memory Access:

With the MPI Remote Memory Access one can create a shared memory window for each process. Processes will now be able to access the allocated data in that window without communicating with the remote process. As said above, we only need to get the positions. Hence, data in the shared window is not modified by other processes. Since we only need to get data from other processes memory, it is not necessary to have synchronization between the tasks.

To create the shared memory window we call `MPI_Win_create()`, which is a collective call. Each

```
send_buff(:,1)=part(:)%pos(1)
send_buff(:,2)=part(:)%pos(2)
send_buff(:,3)=part(:)%pos(3)
CALL MPI_BARRIER(MPI_COMM_WORLD,ierr)

CALL MPI_iSEND(send_buff(:,1), Npart_per_proc(rank+1), MPI_REAL,         &
     proc_send, 1, MPI_COMM_WORLD, request(1), ierr)
CALL MPI_iRECV(part_proc_pos(:,1), Npart_per_proc(proc_recv+1), MPI_REAL,&
     proc_recv, 1, MPI_COMM_WORLD, request(2), ierr)
CALL MPI_WAIT(request(1), status, ierr)
CALL MPI_WAIT(request(2), status, ierr)

CALL MPI_iSEND(send_buff(:,2), Npart_per_proc(rank+1), MPI_REAL,         &
     proc_send, 2, MPI_COMM_WORLD, request(3), ierr)
CALL MPI_iRECV(part_proc_pos(:,2), Npart_per_proc(proc_recv+1), MPI_REAL,&
     proc_recv, 2, MPI_COMM_WORLD, request(4), ierr)
CALL MPI_WAIT(request(3), status, ierr)
CALL MPI_WAIT(request(4), status, ierr)

CALL MPI_iSEND(send_buff(:,3), Npart_per_proc(rank+1), MPI_REAL,         &
proc_send, 3, MPI_COMM_WORLD, request(5), ierr)
CALL MPI_iRECV(part_proc_pos(:,3), Npart_per_proc(proc_recv+1), MPI_REAL,&
     proc_recv, 3, MPI_COMM_WORLD, request(6), ierr)
CALL MPI_WAIT(request(5), status, ierr)
CALL MPI_WAIT(request(6), status, ierr)

DO i=1,Npart
   DO j=1, Npart_per_proc(proc_recv + 1)
      dist = (part(i)%pos(1)-part_proc_pos(j,1))**2.                    &
           + (part(i)%pos(2)-part_proc_pos(j,2))**2.                    &
           + (part(i)%pos(3)-part_proc_pos(j,3))**2.  + 0.000001
      part(i)%acc(1) = part(i)%acc(1) + (part(i)%pos(1)-part_proc_pos(j,1))/dist**(3./2.)
      part(i)%acc(2) = part(i)%acc(2) + (part(i)%pos(2)-part_proc_pos(j,2))/dist**(3./2.)
      part(i)%acc(3) = part(i)%acc(3) + (part(i)%pos(3)-part_proc_pos(j,3))/dist**(3./2.)
   END DO
END DO
```

Figure 7: Parallelisation of the calculation of the accelerations with MPI Point-to-point method. Here we only show the forces imparted by the particles that not belong to the process. We use the non-blocking functions `MPI_Isend()` and `MPI_Irecv()` for communication between processes. This communication is done with every process.

process allocates memory size in bytes, returns a pointer to it, and returns a window object that can be used by all processes to perform RMA operations. To get data from a memory window on a remote process we call `MPI_Get()`. Given that we origanise the data in arrays or derived types, data are not atomic (i.e. not a single number). In order to call a `MPI_Get()` function correctly it is important to know the memory layout in Fortran.

```
buffsize = storage_size(part)*Npart_total/8
! displ = size of memory block when using mpi_get defines the displacement unit
! storage_size(part)/8 means the displacement unit is the size of the structure,
! target_part = j means particle j
CALL MPI_WIN_CREATE(part,buffsize, storage_size(part)/8, MPI_INFO_NULL, MPI_COMM_WORLD, win,ierr)

CALL MPI_WIN_FENCE(0, win, ierr)

DO k=1, nprocs-1
   if (rank-k .ge. 0) THEN
      proc_recv = rank - k
   else if (rank - k .lt. 0) THEN
      proc_recv = rank - k + nprocs
   end if

   ALLOCATE( part_proc_pos(Npart_per_proc(proc_recv+1)*3) )

   DO j=1, Npart_per_proc(k+1)
      target_part = int(j-1,MPI_ADDRESS_KIND)
      CALL MPI_GET(part_proc_pos((j-1)*3+1), 12, MPI_BYTE, &
           proc_recv, target_part, 12, MPI_BYTE, win,ierr)

   END DO

   CALL MPI_WIN_FENCE(0, win, ierr)

   DO i=1,Npart
      DO j=1, Npart_per_proc(k+1)
         dist = (part(i)%pos(1)-part_proc_pos((j-1)*3+1))**2.  &
              + (part(i)%pos(2)-part_proc_pos((j-1)*3+2))**2.  &
              + (part(i)%pos(3)-part_proc_pos((j-1)*3+3))**2. + 0.000001
         part(i)%acc(1) = part(i)%acc(1) + (part(i)%pos(1)-part_proc_pos((j-1)*3+1))/dist**(3./2.)
         part(i)%acc(2) = part(i)%acc(2) + (part(i)%pos(2)-part_proc_pos((j-1)*3+2))/dist**(3./2.)
         part(i)%acc(3) = part(i)%acc(3) + (part(i)%pos(3)-part_proc_pos((j-1)*3+3))/dist**(3./2.)
      END DO
   END DO
```

Figure 8: Parallelisation of the calculation of the accelerations with MPI Remote Memory Acces method. On top, we create the shared window with the function `MPI_Win_create()`. In the first nested loop (`DO j=1, Npart_per_proc(k+1)`) we get the positions of the particles from other processes. In the second loop, we calculate the imparted forces by these particles.

3.2 Optimisation of a serial code (F77)

In this section we profile and parallelise a Fortran 77 serial code that synthesises stellar populations spectra. The code computes the spectral energy distribution of a stellar population, in the spectral range $3540.5 - 7409.6$Å, for a given set of atmospheric parameters. It measures line-strengths for different, such as the metallicity or temperature. To predict the stellar population models it uses either one or all the following stellar libraries: MILES[1], MIUSC[9], NGSL[2], IRTF[8]. In this section we explain the modifications done to the code and thier motivation.

We focus on the functions that use the MIUSC library. First of all, we ran a profiler to find out which functions are the most time consuming. Then, we optimised the serial code updating it to Fortran 90. There are several practices in Fortran 77 which are deprecated in Fortran 90 and ought to be avoided (e.g. COMMON BLOCK's). Finally, we tried to speed up the code parallelising some parts of it.

3.2.1 Serial code profiling: detection of bottle-necks

Running a profiler is essential to analyse the performance of a software. In order to speed up any program, the coder must know where execution spends most of the time. Thus, one can focus on the optimisation of a few functions instead of the whole code. This makes the optimisation much more effictive and efficent.

Figure 9 shows the profile obtained with SCALASCA of the original code. It was run using the MIUSC stellar library, and it measured line-strengths for a total of 20 combinations of μ, α, metallicities, and temperatures. We found out that `sigmam_MIUSC` is the bottle-neck and is responsible for spending a 62.10% of the execution time. Therefore, we focused on this function to do the optimisation and parallelisation.

Notice that a $\sim 96\%$ of runtime is spent in functions comprised in `hrsl_MIUSC`. The operations executed before the call of `hrsl_MIUSC` are dedicated to read the libraries or to write the results, after are all the calculations which are who take most of time. If posible this is what ought to be parallelised to obtain best performance.

We divided the subroutine in smaller functions to detect which is the conflictive operation. As shown in figure 10, we find out that a 56.40% of time is spent in a `STAR_CUBE`, which is part of `sigmam_MIUSC`. This function consists in a big loop with three small loops nested within it. This part of the code is the one we parallelised.

3.2.2 From Fortran 77 to Fortran 90

The code has very old features of Fortran which nowadays are deprecated. Here we discuss two modifications we made to the code in order to update it and to find out possible underlying errors.

The first one was replacing all the `IMPLICIT DOUBLE PRECISION (A-H,O-Z)` statements for `IMPLICIT NONE` statments. The `IMPLICIT NONE` forces the declaration of all variables instead of letting Fortran assume that all variables that start with the letters `i`, `j`, `k`, `l`, `m` and `n` are integers and all other variables are double precision arguments. This avoids possible confusion in the types of variables and facilitates the detection of typographical errors.

Another feature that should never be used is the `COMMON` statement. COMMON blocks are a

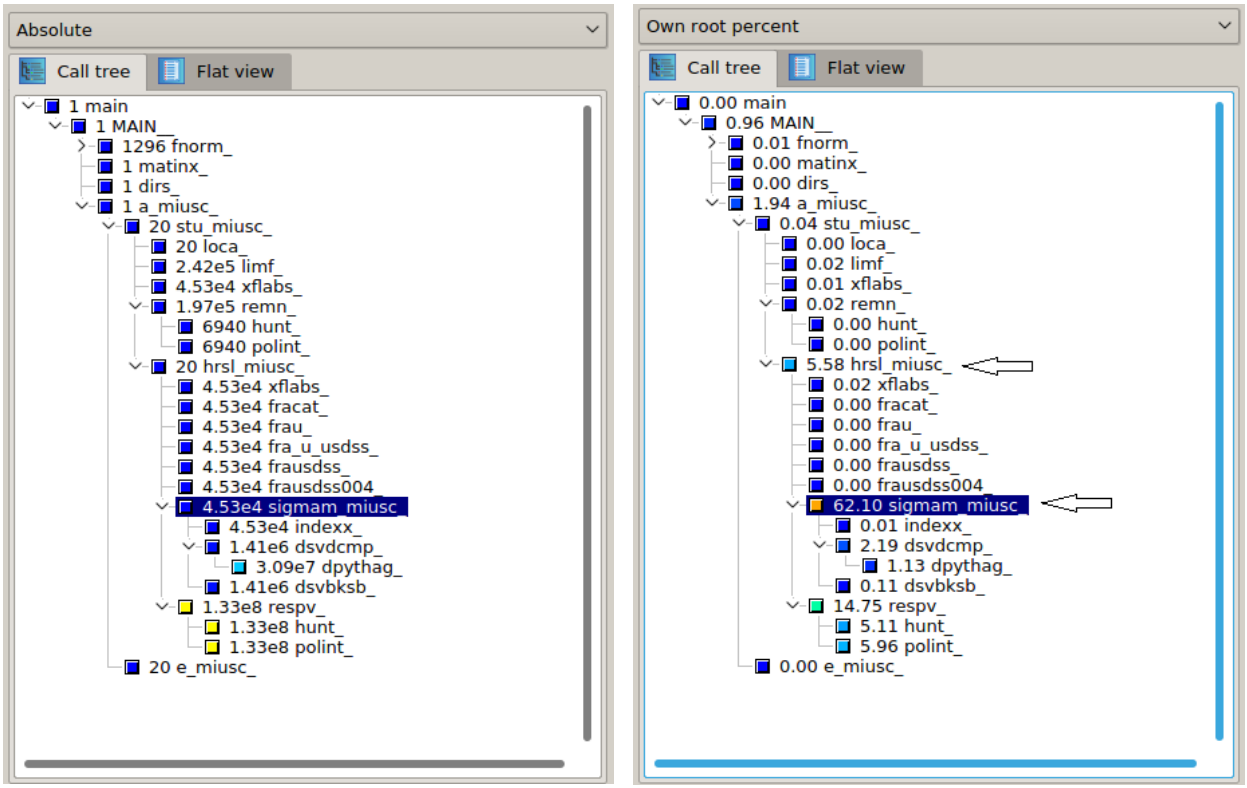

Figure 9: Profile of the original serial code. At left, we see the tree of calls of functions and number how many times each function is called. At right, we see the same but in terms of the percentage of the total execution time spent in each function. Notice that a 62.10% of time is spent in `sigmam_MIUSC`. Thus, efforts to optimise and parallelise the code should focus on speeding up this function.

very useful feature of Fortran 77 to build modular programs. They allow having variables of other routines without having to pass them by procedure arguments. Nevertheless, the global nature of the COMMON blocks variables means memory area of the block is shared, therefore if one routine alters a variable it affects all of the other routines with that `COMMON` statement (a so-called side effect). Placing a variable in a COMMON block inhibits some optimisations that compilers perform, such as code movement or registers optimisations, since the compiler cannot assume that a variable is not modified in a subroutine not having that particular variable in its interface. Furthermore, if one modified the COMMON block statement (e.g. change the name of a variable, or add one) the code would still be compilable. This practice, in addition to inducing to errors, makes the debugging process much harder.

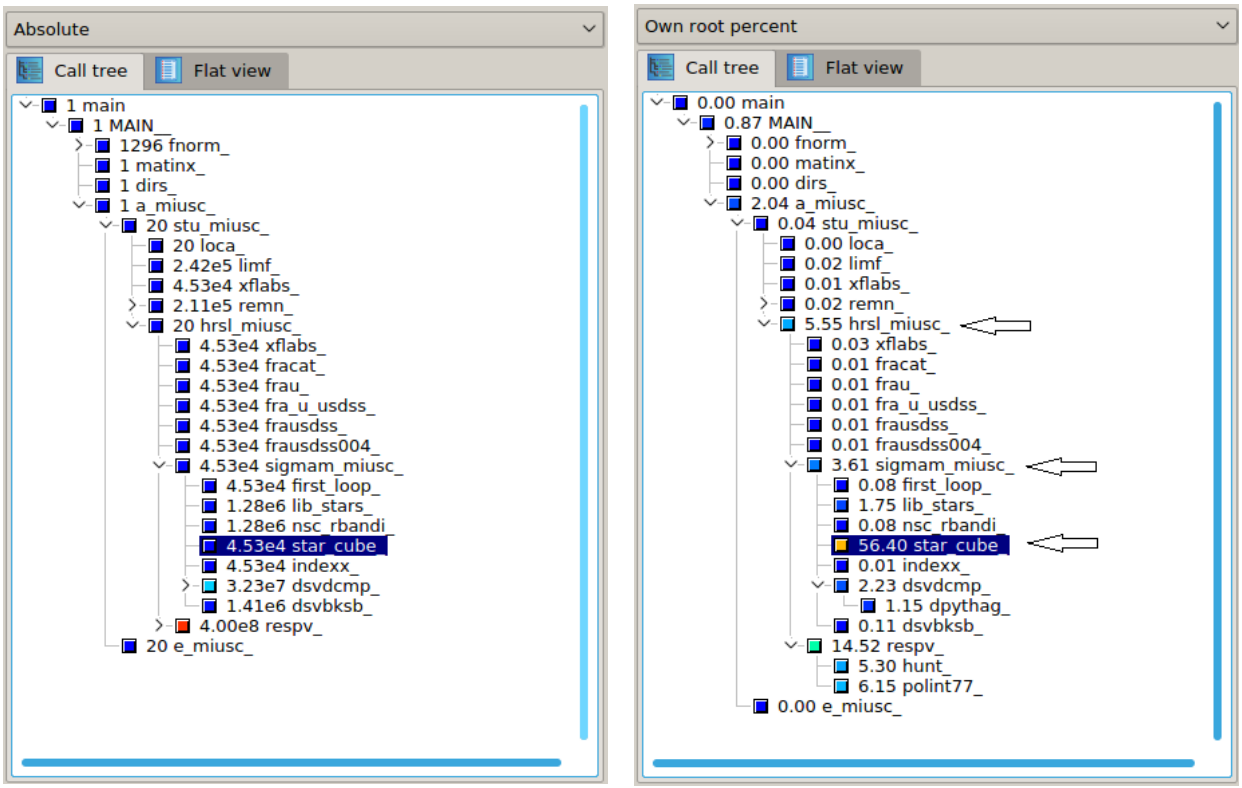

Figure 10: Profile of the serial code. Left panel, we see the tree of calls of functions and number how many times each function is called. Right panel, we see the same in terms of the percentage of the total execution time spent in each function. Here we broke `sigmam_MIUSC` down into smaller parts and found out that the function `STAR_CUBE` is responsible for the 56.40% of runtime.

3.2.3 OpenMP parallelisation

In order to speed up the program we tried to parallelise some parts of it. The idea is to run the program on the shared memory Severo Ochoa Cluster diva@iac, consisting in 192 cores having 4,5TB of shared RAM. Therefore, the parallelisation can be done both with OpenMP and with MPI. From the tests we did with the N-body problem 3.1, we learnt that it is much easier to adapt a code to OpenMP than to MPI. For ease, we implement OpenMP for parallelising the code.

At first we tried to parallelise the outermost loop in the subroutine `hrsl_MIUSC` (see Figure 10), but we had several problems with which variables had to be shared or private. Instead, we parallelised some nested loops located in `STAR_CUBE`. As in 3.1, we use the OpenMP directive `Parallel Do` for the parallelisation, and the clauses `share` and `private` for the variables.

4 Results

4.1 N-body test

In this section we discuss the results obtained when running an equivalent N-body problem parallelised using three different methods. As explained above, they are: OpenMP, MPI point-to-point and MPI-RMA. To test the different methods we measured the runtime of the three codes for the same set of initial conditions. We only measured the time spent in the parallel parts, so that we did not count the serial parts, such as, the ones responsible for reading and writing files.

The tests were done for 2^{10}, 2^{11}, 2^{12} and 2^{13} particles, so the amount of memory used and transferred is different. On the other hand, we ran each set of initial conditions on 1, 2, 4 and 8 threads/processes. All the tests were done in the same computer and we only ran the code in a shared memory architecture, this limits the significance of the results. In table 1 we show the results for 2^{14} particles. We observe that the time decreases almost linearly. This agrees with what we expected from Amdahl's Law (equation 1). However, we notice that the more threads/processes we use, the less linear is decay in the measured time.

The uncertainties in runtime were measured running the code 20 times for each combination of number of particles and number of processes.

Number proc/threads	OMP time (s)	MPI time (s)	RMA time (s)
1	194.1 ± 5.0	197.6 ± 3.6	197.2 ± 1.5
2	108.9 ± 1.8	102.3 ± 0.8	101.5 ± 1.8
4	58.5 ± 1.1	55.8 ± 0.4	53.2 ± 0.9
8	31.8 ± 2.5	29.7 ± 0.3	28.8 ± 0.4

Table 1: Time spent to calculate 10 timesteps of a 2^{13} particles N-body problem. It shows how the runtime is reduced almost linearly adding threads/processes to the execution. Notice that the MPI-RMA is the method with best performance.

In Figures 11,12 and 13 we show the speed up achieved using OpenMP, MPI point-to-point and MPI-RMA, respectively. We observed that when the number of threads/processes increases, the linearity decays. This behaviour is due to the time spent in data transfer. Obviously, if one uses more processes, the code will require more communication between nodes.

We notice that the MPI codes speed up is more linear for larger amounts of particles, conversely, OpenMP parallelisation is less effective. In addition, despite in theory OpenMP should perform better than MPI in shared memory applications, we obtained best results for MPI. We relate this

Number proc/threads	OMP speed up	MPI speed up	RMA speed up
1	-	-	-
2	1.78	1.93	1.94
4	3.32	3.54	3.71
8	6.10	6.65	6.87

Table 2: Speed up achieved with the three different methods dicussed, (OpenMP, MPI Point-to-point and MPI-RMA). It shows how MPI performs better than OpenMP and MPI-RMA reaches the highest speed up.

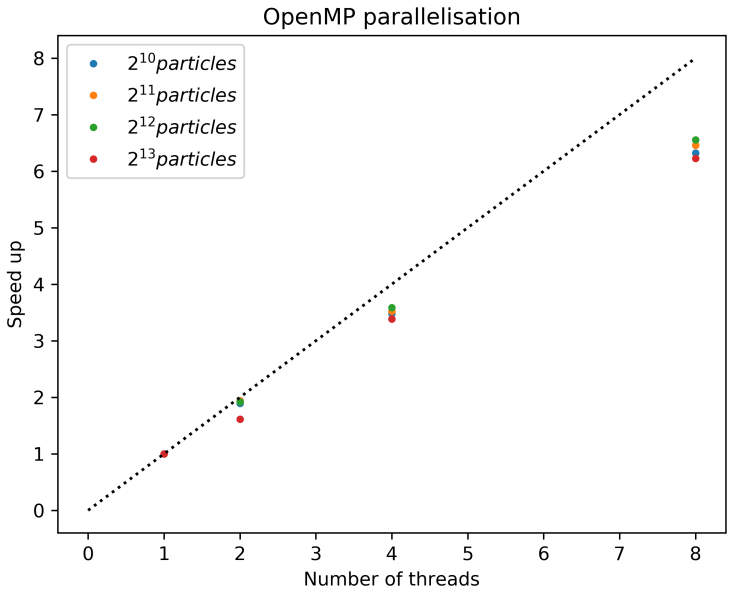

Figure 11: Speed up of an N-body code parallelised with OpenMP. We ran the simulation for a different number of particles. It was executed using 1, 2, 4 and 8 OpenMP theads. Note that the speed up is not entirely linear and that the more particles, the less linear.

to the OpenMP creation of threads overhead issue commented in section 1.2.

Regarding the MPI codes, the Remote Memory Access performs better than the point-to-point no matter how many particles we put in the simulation or how many processes we use. However, the mean runtimes measured are very similar and it might be precipitated to affirm MPI-RMA is the best method. In order to do so, we ought to run the two codes on a distributed memory architecture.

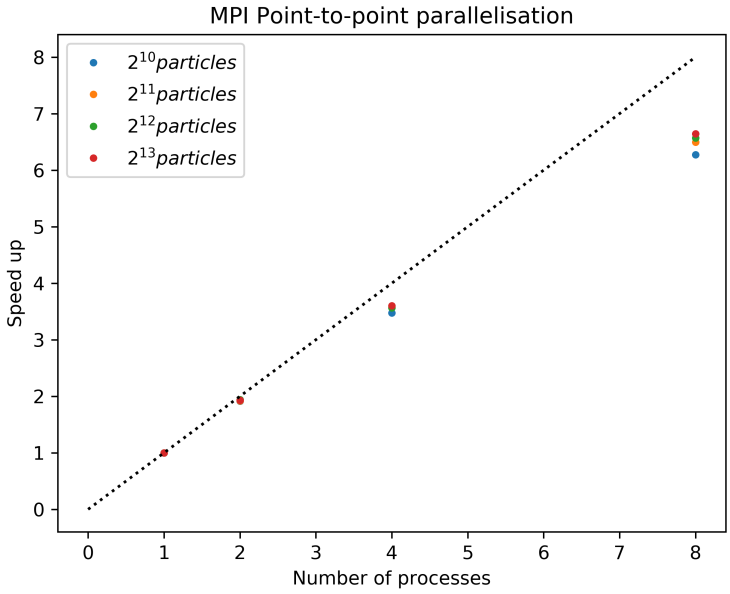

Figure 12: Speed up of an N-body code parallelised with MPI Point-to-point. We ran the simulation for a different number of particles. It was executed using 1, 2, 4 and 8 processes. Note that the speed up is not entirely linear, but now the more particles, the more linear.

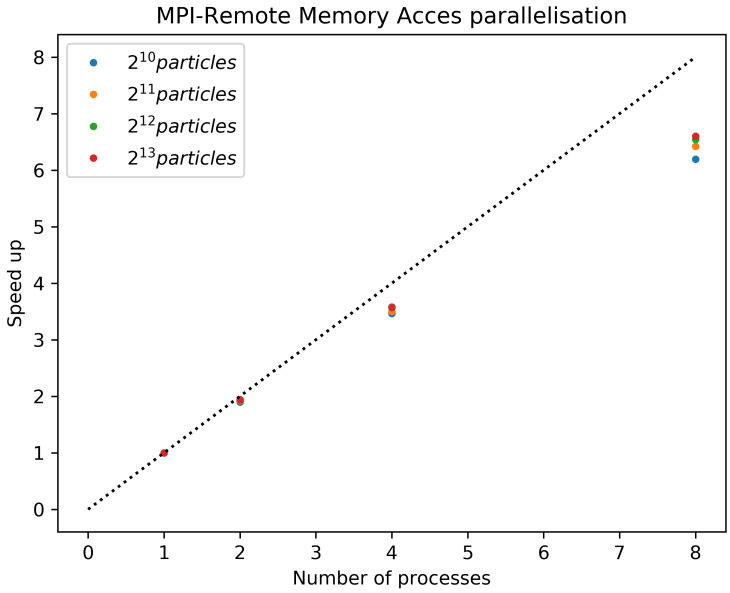

Figure 13: Same as figure 9 but parallelised with MPI-Remote Memory Access.

4.2 Serial code optimisation

In order to speed up the code we parallelised the most time consuming functions. We did the parallelisation with OpenMP. As said in §3.2, we found out that `STAR_CUBE` was the part of the code where it spent a 56.40% of the total time. If we parallelise this part of the code, the serial part will be S∼0.5 and the parallel part P∼0.5. Therefore, regarding Amdahl's Law (equation 1), we will at maximum obtain the following speed up,

$$speed\ up \sim 2\frac{\mathrm{N}}{1+\mathrm{N}} \tag{3}$$

where N is the number of threads. Nevertheless, achieving perfect parallelisation it is not possible. As we saw in the results of N-body test, time consuming functions emerge due to threads creation or communication between them.

In table 3 we show the time spent for the synthesis of one spectrum with the parallelised code. We do not measure the time spent reading the stellar libraries because it is only done once. We

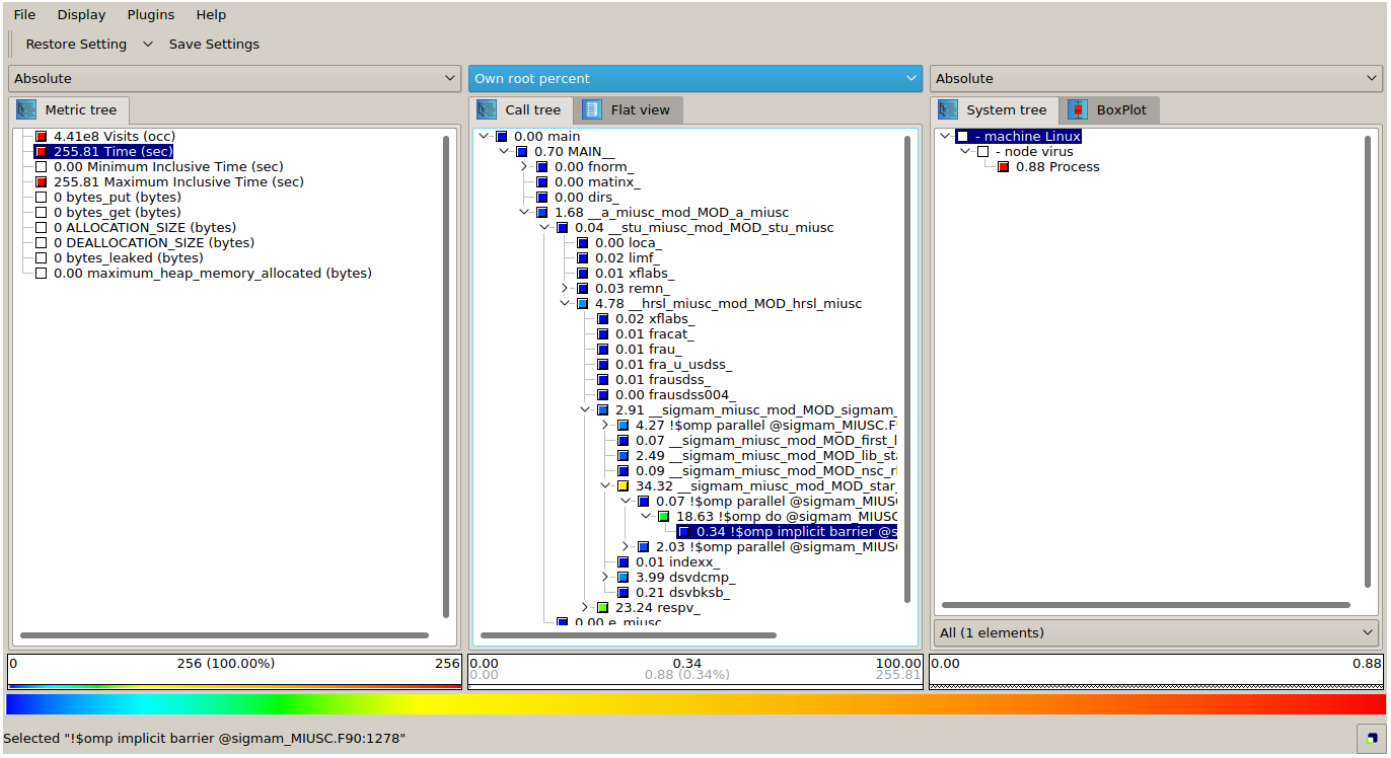

Figure 14: Profile of the code run on 1 thread. Left panel, we can see various performance properties, such as, the total number of calls to any function (Visits), the computational time (Time) or the physical time (Maximum Inclusive Time). In the middle, the functions and subroutines which are called are shown. In percentage, we see the time spent in each one.

obtain best performance when running the code on 6 threads. Conversely to what one might think, if we run it on 8 threads runtime increases. On the other hand, when we run the code on 4 threads the speed up is almost the same than running it on 6. Therefore, we consider 4 threads is the optimal implementation for the code.

Number threads	1	2	4	6	8
Time (s)	7.54 ± 0.65	7.38 ± 0.65	5.91 ± 0.24	5.88 ± 0.38	5.96 ± 0.40
Speed up	-	1.02	1.27	1.28	1.26
Theoretical speed up	-	1.3	1.6	1.7	1.8

Table 3: Time spent for the synthesis of one spectrum, the speed up achieved and the theoretical speed up. We have best performance when running the code on 6 threads but the best ratio $speed\ up/\mathrm{N}_{\mathrm{threads}}$ is for 4 threads.

In figure 15 we show the profiles we obtained running the code on 4 and 8 threads, respectively. Although we observe that the time spent in each loop is fewer with 8 threads, the code runs faster on 4 threads. If one looks at the OpenMP barrier times, will note that the time spent in the barrier running the code on 4 threads it is worse distributed than running it on 8. Nevertheless, the time each thread waits is higher with 8.

In figure 10 we can see `STAR_CUBE` is called several times. For each call the program creates threads and, as explained above, OpenMP has a high time cost issue when creating threads. That is why a higher number of threads does not mean less runtime. Thus, 4 threads perform better than 8.

In relation to the thread creation issue, we tried to parallelise some small loops and the runtime increased instead of decreasing. So it is not evident that if one parallelises a loop the code becomes faster.

The latter problem leads to try other parallelisation approaches. We also tried to parallelise the outermost loop, in other words the function: `hrsl_MIUSC`. If we had achieved this we would have had $\sim 100\%$ of parallelisation, so the run time would have been $t = t_{serial}/\mathrm{N}_{\mathrm{threads}}$. We could not do it because some variables that had be private to parallelise the loop, had to be shared in the way the serial code had been written. In order to do so we needed to change most of the code. It is important that the developer thinks in the possible parallel regions before writing the code, avoiding avoidable relationships between variables (e.g. index counting).

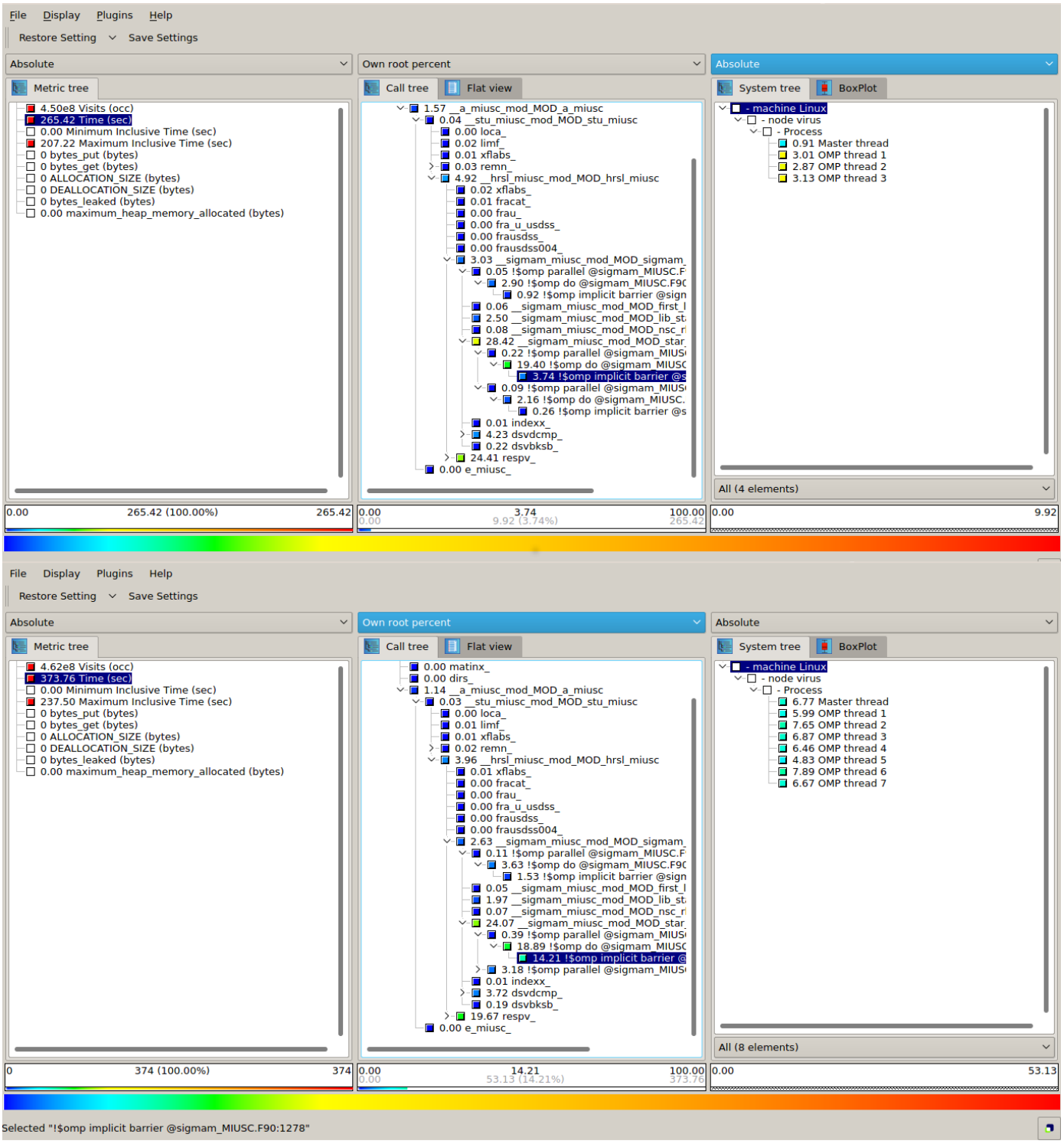

Figure 15: From top to bottom, profiles of the code run on 4 and 8 threads, respectively. Note, at left, that the computational time (Time) has increased but the physical time (Maximum Inclusive Time) has decreased. We also observe that the OpenMP barrier is less compensated when running the code on 4 threads, but the time spent in the barrier is fewer too.

5 Conclusions

- Although OpenMP is designed to be implemented in shared memory architectures, both the classic MPI (point-to-point) and the new Remote Memory Access MPI-3 features have better perfomance. For instance, for 8 threads/processes we obtaind the following speed up's: 6.10 with OpenMP, 6.65 with MPI point-to-point and 6.87 with MPI-RMA.

- The comparison between MPI point-to-point and MPI-RMA is not totally reliable, since they were tested on a shared memory machine. To obtain definitive results, both methods should be tested on a distributed memory architecture, where latency is more noticeable.

- We optimised a Fortran 77 serial code that calculates the SED of a stellar population, updatinng it to Fortran 90, and parallelising it with OpenMP. We obtained a speed up of 1.27 when running it on 4 threads, which is the best ratio performance/$\mathrm{N}_{threads}$.

- Every code should be analysed with a profiler. Otherwise, possible bottle-necks can go unnoticed. Furthermore, it makes the optimisation of a software both more efficient and effective.

- OpenMP does not perform well in fine-grained parallelism. It is due to an overhead in creation of threads. How ever, it has similar performance to MPI when it is implemented in coarse-grained parallelism (of course, only in shared memory architectures).

- There is an actual sofware crisis. On the one hand, the hardware is renewed every 4-5 years. But on the other hand we still write codes in deprecated lenguages (e.g. Fortran 77), and use methods that should be avoided and replaced for new features (e.g. COMMON blocks).

References

[1] Falcón-Barroso, J. et al. 2011, A&A, 532, A95

[2] Heap & Lindler, 2007, ASPC, 374, 409

[3] Hoefler, T. et al., 2013. Remote Memory Access Programming in MPI-3. ACM Trans. Parallel Comput. 1, 1, Article 1 (March 2013), 29 pages.

[4] Message Passing Interface Forum, 2015, MPI: A Message-Passing Interface Standard, Version 3.1

[5] Mikhail B. (Intel), 2017, An Introduction to MPI-3 Shared Memory Programming

[6] Lawrence Livermore National Laboratory, Livermore Computing Center HPC Tutorials

`https://hpc.llnl.gov/training/tutorials`

[7] OpenMP, 2015, OpenMP Application Programming Interface, Version 4.5

[8] Rayner, Cushing, Vacca, 2009, ApJS, 185, 289

[9] Ricciardelli, E. et al. 2012, MNRAS, 424,172189

[10] Scalasca Development Team, 2018, Scalasca 2.4 User Guide

[11] Shende, S. 1999, Profiling and Tracing in Linux