Summary of Parallel programming tools & models

# Measures of performances

## Speedup and efficiency

The speedup us the velocity increment of a parallel program respecto to his sequential variant, it is the ratio between the best sequential time and the parallel time with p processors. The best (and ideal) case is the linear speedup.

<graph with speedup function>

A linear speedup is ideal, the real one is like the red function in the graph, how is it efficient?

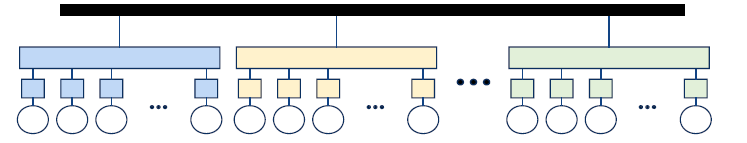
We can calculate the efficiency with this formula:

More the red line is closed to the blue one, more the parallel program is efficient!

E(p) tells that, if we attend a result, it is better attending a little amount of time (like 10 minutes or lower) instead of 1 day or more!

So, the purpose is to find a solution that give a result with the least amount of time possible, how we can do it? We have to try some algorithms and techniques!

## What is a cluster?

A cluster is a supercomputer composed by many “little”machines.

The architecture of a cluster is like in the image on the right, in fact we have:

* Some CPUs that communicate with a shared memory, how? they write in specific location of memory! Every CPU have a cache, it is useful for optimizing the memory access;
* Some CPUs plus a shared memory formed a node, every node runs an operating system and it is connected to the others using a network.

Which parameters could have the cluster and the nodes?

* In the clusters, the number of nodes is important because they allows the parallelism. Moreover, the bandwidth and the latency of the network are important too because they affect directly on the performances of the programs.
* In the nodes, it is important the number of CPUs and cores, the bandwith/latency of the bus and locality. It is important the FLOPS, the number of floating point operations executed in one seconds, it result useful expecially when the node contain a GPU.

## MPI (Message Passing Interface)

What is MPI? It is a parallel approach that parallelizes a program using the data distribution. Each node executes the same program but on different portions of data. At the end of the execution, each node have to communicate the results to the others, how can be the communication? It can be:

* point to point where the communication regards only to node in the cluster. In this case the sender has to invoke a send() and the receiver has to invoke a receive(). We have to be careful because we can fall in deadlocks! To avoid these problems, there are some tools and libraries that implement send() and receive() in the right way;
* the collective communication regards all nodes, in fact there can be some cases where one node have to send a result in broadcast. Also in this case the communication is done with some instructions.

MPI is perfect in memory distribution, because every node run the same programs on different portions of data and at the end share the results, so it is possible to optimize everything.

## OpenMP

Another approach for parallel programming is OpenMP, how does it work? Essentially, the approach is dual to MPI: it is the program to be splitted instead of data!

How does OpenMP split the computation? It uses pragmas that tell to the compiler where are the parallel parts, so it will generate all the code, from the splitting of the computation to the synchronization.

This approach is very fork-joinish because it continue to split the computation and synchronize it in base of the pragmas in the code.

This approach is perfect for shared memory systems, so a program can be parallelized efficiently in a single node of a cluster.

## Is it possible to combine these two approaches?

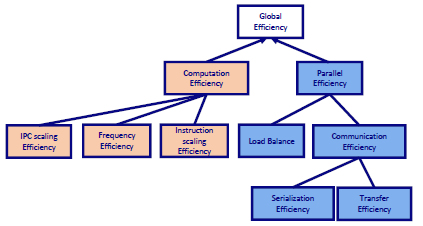
The answer is Yes! And it is called hybrid programming: it is an approach that combine the advantages of MPI and OpenMP, so we can improve load balancing, leverage the capabilities and reduce the communication costs. On the other hand, due to Amdahl law, it could be overheads and locality because of the reduction of calculus for communication.

# Parallel efficiency models

In parallel programming, every model is wrong but some are useful. What does this sentence mean? It mean that

* we can’t get a more corrected model because of more computation, instead we should seek the minimal description of a phenomena.
* We need simple and evocative models, not more elaboration and/or parameters;
* Identify which models are importantly wrong, in fact we don’t have to be concerned about mice when there are tigers!
* we have to ask ourselves if the model is illuminating and useful, not if it is true (because it’s not)!.

## How can we describe the behavior?

We can describe the behavior of a model for a region of space and time, we can do this using top down methods, for example starting with a high view for a sufficient Focus on analysis, then we iterate to go to the low level view.

On the other hand, we have POP model factors quantifying abstract classes of phenomena for a system observable pattern. In these cases, there are scale efficiencies with respect to a reference run.

For this summary, we’ll use a multiplicative model, what does it mean? Considering the photo on the right, we calculate the efficiency of a node using the multiplication of the efficiencies of its sons.

For example:

## Parallel efficiency

The parallel efficiency indicates how a phenomenon is well-parallelized, in this case every thread spends more time in execution instead of communication.

An important factor is load balancing: it determines how the amount of work is equally distributed in every thread. If the amount is more or less the same for each thread, the efficiency of load balancing belongs to 1. Instead if it is not, there is less computation and so a loss of resources belonging to a low efficiency.

The communication is about how time we spend about transferring data from/to threads, we can divide it in:

* Transfer: the time we need to transfer data from/to thread because of relations, casualties and/or order between threads. We can consider it like a sort of delay we do when we split the computation in more parts and we distribute them. We have to consider that these communications are not instantaneous and depends on the model we use.
* Serialization: it is about dependencies and synchronization between threads. Precisely, it means if a task needs some values from another one, it has to attend its finish to continue the execution. And in case of circular dependencies? This is a problematic case because there could be deadlocks, in a better case there is one thread (sometimes the slower one) that unblocks the situation.

## Computation efficiency

The computation efficiency indicates how the computation is well executed on more processors, in fact, respect to the sequential execution, the metrics execution can be different!

This type of efficiency depends on three factors:

* the instruction scaling, in poor words how many instructions are executed for each processor. Ideally the number of instructions per processor should be the same and it is independent from the number of processors we use;
* the frequency of each processor, in fact there is no guarantee that all processors use the same frequency because some of them could be boosted and others reduce this value to avoid the burnout.
* Instructions per cycle (IPC) scaling: in poor words, how fast an instruction is executed! The more this value is high, the less is the time of execution of a group of instructions. For this part we have to consider the caches and the locality properties, in fact these things allow to reduce the latency because we avoid unuseful memory accesses. In multiprocessor architectures, we have a cache for each processor, so we have to synchronize all cache using some methodologies. If a value is not in the local cache, it will be searched in the others and, based on the result, the processor will do or not a memory access.

## What’s about computation?

Considering the photo on the right, we can split the computation in two parts:

* the useful computation (the blue one), that is the time of effective computing;
* the unuseful computation (the yellow one) , all the times not regarding the computation like the synchronization and the communication.

Our purpose is to increase the useful computation and decrease the unuseful one! We can say that a computation is efficient when the blue part is bigger than the yellow one, so we can resume it with this formula:

And in a situation without dependencies? This is an ideal situation, the most efficient case, in fact every processor doesn’t have to attend to some values from the others. From this particular case, we can calculate the load balancing, using this formula:

At the same way, we can calculate the serialization:

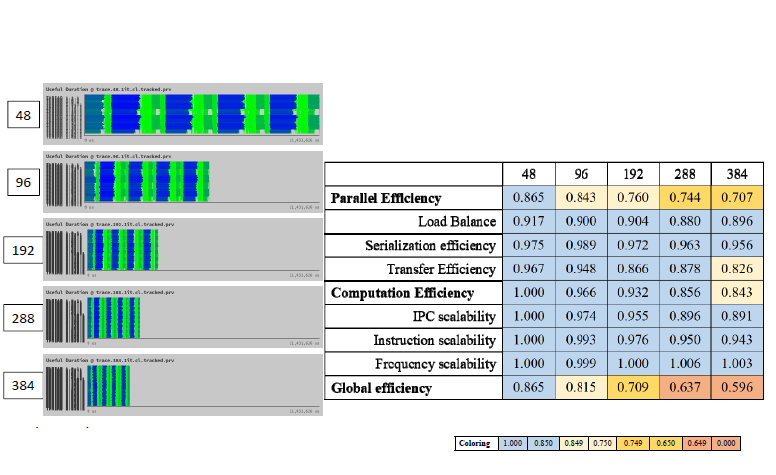
Now we consider another ideal machine with dependencies, zero-latency and infinite bandwidth, what can we say about this? With these assumptions, we get a more realistic model allowing us to save a little amount of time. We’ll use it to calculate transfer with this formula:

## Sequential efficiency scaling

The sequential efficiency regards the computation of a single processor. for this, we can consider three factors:

* ;
* ;
* .

How can we count the instructions? We use the library PAPI, which allows us to count all the instructions in a defined region.

Now we consider the traces on the right: what do we get if we use the same number of processors on every trace? We get the table always on right! In this table we can see that the best results is the computation of 48 traces, why? Because it is a case of strong scalability, a type of scalability where the number of processors remain the same in front of a larger amount of work (or vice versa), so we lose time for communication (i’m not sure about this).

# Methodologies

What about methodologies? Typically we measure the speedup and the efficiency of a parallel program to say if the model/methodology we use is nice or not. In generale, we have to consider the global effect, for example a lack of structural insight, and the coarse aggregation, the risk of speculating about causes with little capability of hypothesis.

For these reasons, we have to analyze the code to find possible issues before doing things.

## Which is the approach?

The approach we use is practically the scientific method: we generate and hypothesis and then we try to validate it?

How can we validate the hypothesis? We can use some tools that acquire data, prepare and verify it!

An important thing is not to mask the symptoms, why? Because if we don’t understand the problem, we cannot go forward! In fact we have to analyze the original version of a code (maybe a not optimize version) without implementing solution, then we can identify the possible causes of performance issues and so a solution!

In base of what we said, we can define the following steps:

* data acquisition and preparation: in this step we define the amount of data, the level of detail, the structure and the FOA;
* Time based scaling: we try to scale the things we define on the time;
* we define the model, an abstract description of the fundamental aspect of behavior, also we quantify the terms of efficiency;
* For each inefficiency we found, we describe the route causes;
* At the end, we do some recommendations.

## How can we identify the structure?

We find the structure in base of the granularity of the problem and in base of differentiated and/or repetitive behavior in the time and space. So, what we need to do it? we have to find relevant data and then reduce, clean and prepare it. Typically we aren’t interested about an initialization/termination, it is sufficient that they have as little perturbation as possible.

After doing this, we focus on the analysis defining precise intervals for computing statistics, then we apply the general analysis to individual phases.

The purpose of the analysis might not be known sometimes, however it is important for productivity. Moreover, the mental models is not always equals to the reality, so we have to acknowledge.

## Selecting FOA and metrics

After we identify the structure, we can select the FOA, how?

firstly, we select a region to analyze, the choice depende from the context and in general the situation. Then we study the chosen region in order to extract speedup and efficiency.

The metrics are general mechanism for describing the parallelism in order to quantify an efficiency loss and doing a fair comparison between two parallel things, also they tell where we have to look for causes.

## Factors and causes

### Load balancing

Which are the causes of a bad load balancing? In general it is a bad distribution of data among processes caused by:

* an imbalanced number of instruction per processor, in this case histograms results useful to identify the problem;
* Imbalanced IPC, in fact some threads have worse locality and/or the mapping on shared resources is unbalanced;
* Different frequencies due to operating system noise, etc.;

### Transfer

A lack of efficiency in transfer indicates that there is an an high overload of the runtime, a poor latency and/or a low bandwidth. Which could be the causes?

It could be the noise of the operating system, but also the propagation time for communication (it depends on the message size and from the hardware) and the protocol we use. For the last one, there could be overheads and interferences, limits in the buffer, etc.

To avoid this problems, we can use metrics like bandwidth to see how is the situation and then do something to solve it.

### Serialization

The principal problems in serialization regards the circular dependencies, in fact they causes irregularities during the execution or noise.

For these problems, it results useful to analyze the microscopic load imbalances, the noise and the chain of circular dependencies.

### Communication

The loss of efficiency in this part is normally caused by the synchronization between task or the overhead. In this case, metrics like MPI calls and granularity could be useful to identify the problem.

### Instruction, IPC and frequency scalability

A low value of this means that the number of instructions grows with core count due to code replication, etc. At the same time, a lack of IPC is about that the specific core count is worse than the reference case. It could be caused by locality, contention of resources, etc. On the other hand, an high IPC could be caused by a cache effect. At the end, respect to the reference case, a lack of frequency scalability is caused preemption, power management measures, etc.

# Personal vision of parallel computing

The purpose is to have a specific thinking about these models. Once upon a time, people wrote new parallel instructions on ISA and APIs to make parallel apps.

This methodology works in the single cores, but in multicore? There could be problems with caches because they’re not synchronized, so we have to find a way to do it.

So, the solution is to modify the algorithm, in case of GPUs:

* we need to load the data in the device’s memory;
* we have to redo the algorithm considering the GPU architecture, so we have to test different algorithms in order to improve the performances.

Doing this is correct? No! We need to provide a maintainable solution that does not affect the future consistently.

So, it’s better improve the portability, maintanability, etc.

### Families of parallel models

StarSs is a family of parallel model telling the best eay to make a parallel app, the idea is to execute in parallel a sequential tasked based program on a single address/namespace with directionality annotations. How can we do this? In order to do this, we have to consider that some tasks are independent, so we can execute them in parallel, instead in case of dependencies we have to respect a certain order.

StarSs is not a single idea but a set of these in base of the available architecture.

In case of OpenMP, we use pragmas to indicate which part of code we want to parallelize, so the runtime generate all the parallel code.

These generated part exist in different implementations, so we can easily convert the program to another architecture.

These pragmas allow us to optimize the code independently from the architecture, moreover they’re useful to detect the dependencies of data and the lookhead.

Basing on these informations we can build the task graph, a graph representing all the dependencies in the parallel application.

## OmpSs

OmpSs is an experiment platform to the forerun of OpenMP, the idea is to minimize the set of concepts relaxing StarSs, looking for elegance and fundamentals and giving power to the runtime.

This platform return useful in order to redo the same program for each architecture, in fact we can use a general structure that we can adapt to the specific one. Also, in case of a less number of threads, we can easily aggregate some tasks in one of these.

A variable in OmpSs can be:

* shared: the task use the original variable;
* private: declared as private and writings in a task affect the global one;
* firstprivate: like private but writiings affects the local variable;
* threadprivate: private variable in the thread, in generale it is not used.

### Array sections

An array section is a subregion of a larger structure we can specify with pragmas in order to parallelize it. These regions could be:

* inputs;
* Outputs;
* each other;

### Incomplete directionality

Because of the directionality is not required in all the variables, forcing it is a risk we represent a group of variables with one, so the code results not portable. When we can, it’s better avoiding it.

### Concurrency

The pragma of concurrency relax inout dependencies executing the tasks in parallel, then the dependent part will be executed.

We use this pragma with the pragmas atomic and/or critical , they specity a set of operation to execute sequentially.





### Commutative

It is a pragma simular to concurrent but here the task are not executed in parallel, instead they’ll be executed sequentially with a non-specific order.



### Nesting

It is the possibility to nest pragmas in order to get a hierarchical task graph, what is the idea?

The idea is to instanciate the inner task before and then the outer ones, this happens because of the dependencies and improve the parallelism. Moreover, in this way every level contribute to the parallelism.

There are some types of nesting:

* top down: every level contributes to the parallelism;
* flattening: increase the concurrency and take out runtime overhead in critical paths;
* granularity control: use pragmas to avoid a too small granularity;

### Multi Dependencies

Task may have more dependencies, so what can we do in these cases? We use the same pragmas, managing them as a single dependency.

### Potential

Given an application, where is the problem in the execution?

The data transfer could be not perfect, is there a way to improve it?

* We have to identity the code structure;
* We have to detect the tasks and their dependencies

Moreover we can aggregate some dependent task in a unique one and split anothers to improve the parallelism.

### Example: file processing

Considering a folder with some files we need to read, how can we parallelize? The idea is the following:

* We read sequentially each file;
* We do what we have to do;
* We save the results sequentially;

Another way is to perform these three phase in a single thread, one for each file, then we add a task merge that will merge all the results.

Last but not least, we can consider one thread for each phase and perform the example as a pipeline.

## Hybrid programming

To explain the idea, we do the following example: we want to accelerate the product between matrixes, how can we do this? Because of each cell is the product with specific lines and columns, we can some task doing this.

We can get an improvement of performances separating task creations from the results and/or using the nesting.

But why do we hybrid?

The hybrid programming allow us to parallelize when we can and to run sequentially otherwise, in fact we can’t parallelize some tasks because of the dependencies.

### MPI + OmpSs

Because of MPI is a too serial approach, each tasks spend cores in case of delay and/or long time transfers, using the CPU for nothing.

For this reason we combine it with OmpSs, it allow to optimize at the best the execution thanks to a top-down approach to do serial things and overlapping the communication.

### Malleability

It is the property of a parallel program to adapt its execution to the parallelism, how is it possible? We can use some libraries like DLB that intercept the runtime, there are APIs hinting the resource demands and there are policy of core reallocation.

THis is an opportunity to fight the ahmdalh’s law that allow to be more productive

### Heterogeneity

It is a way to define how the model works in order to improve the performances, it means:

* the execution of different things on equally functional devices;
* how we can parallelize;

The ISA heterogeneity regards compilers and how they can optimize.

THe separate heterogeneity means that CPU and GPU use different memories, so we need to transfer data and some variables in the device referencing the main one.

A way to homogenize the heterogeneity is OmpSs.

How is possible to access data in the device memory? we use some sentinels to build dependencies, also to access the data we need a valid copy from the device.

### Kernel based programming

It is a way where we can define a function called kernel that will be executed in parallel with a subset of data as input.

This way allows the separation of of the iterations ado loops in order to appear simple, more over the granularity is extremely fine, thanks to the fork-join model.

### Task based programming

In this case, we can support larger granularities defining the tasks and their dependencies.

In this way, we can decrease the overhead, but a too large granularity cause too overhead too, so we choose the finest possible.

### CUDA on OmpSs

Using this methodology, the transfering of data is done in runtime, however we can’t access to the copied data, in fact the created variable are pointers.

A CUDA task doesn’t show the GPU where it is executed. In case of dependencies, there is a way to passa values from a task to another without passing through the CPU.