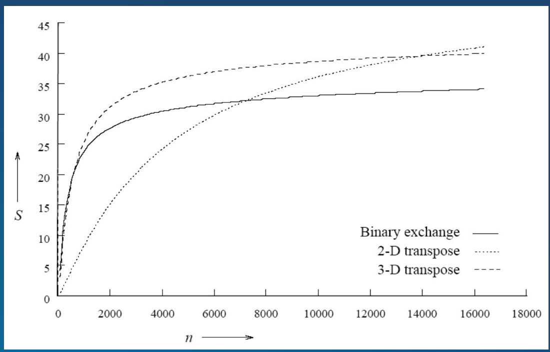
Performance Evaluation

Frédéric Desprez
INRIA



Some references

- **Parallel Programming – For Multicore and Cluster System,** T. Rauber, G. Rünger

- **Introduction to parallel Computing, 2nd Edition**, A. Grama, A. Gupta, G. Karypis, V. Kumar, Addison Wesley

Measuring time

Before parallelizing a program, one must be able to know which part of a program takes the most time in computation

- **Three types of time to consider**
  - **Wall time**
    - The time spent executing a program: the time spent between the beginning of the execution and the end
  - **User time**
    - The time really used by the program
    - It can be much lower than the wall time if the program has to wait a lot, for example for system calls or data exchanges
    - This lost time can give indications for optimizations
  - **System time**
    - Time not used by the program itself but by the operating system (memory allocation, process management, disk access, ...)
    - We try to keep it minimal

Measuring time, contd.

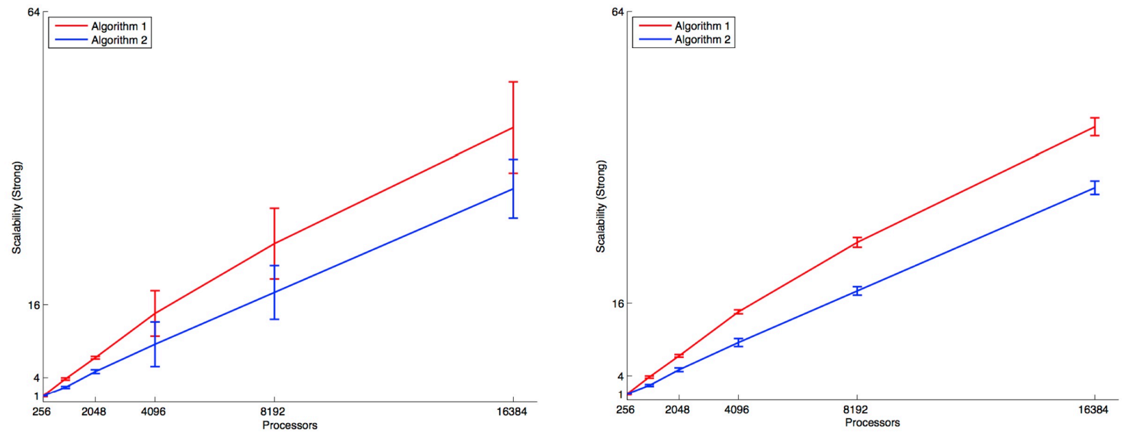
- Unix time command: `time ./executable`
  - Output example

```
real 3m13.535s
user 3m11.298s
sys 0m1.915s
```

  - Measures the total time of the program
- For performance analysis, it is necessary to know the execution time of certain parts of the program
  - Methods dependent on programming languages or operating systems
  - MPI: MPI_Wtime(), OpenMP: omp_get_wtime()
    - Give the wall time between two function calls
- Application profiling
  - If proper compilation, use gprof (`gprof executable > prof.txt`)
  - List of all functions with their execution time, their total time percentage, number of calls
  - Call tree
- Software timers
  - PAPI

Good Measurement Practices

- Choice of number of processors
  - Depending on available resources
  - Beware of physical topology
- Pay attention to the resolution of the clock
- Repeat experiments to understand variability
  - Shared resources (processors, network)
  - Placing jobs / threads on potentially different processors / cores
- Confidence Interval



Need for analytical models of parallel programs

- A sequential program can be evaluated according to its given execution time according to the size of its input data
- A parallel program has its time that depends on other elements
  - Number of processors used
  - Their relative speed
  - The speed of communication between them
  - $\Rightarrow$ A parallel program can not be evaluated independently of these elements
- **Some intuitive measures**
  - The wall time obtained to solve a given problem on a given parallel platform
  - What is the gain obtained in speed with respect to the sequential time: the acceleration (or speedup)

Execution time

- **Sequential execution time (T_s)**
  - It is the time spent between the beginning and the end of an execution on a sequential node

- **The parallel time (T_p)**
  - This is the time between the start of parallel execution and the time the last processor finishes

- **Warning!**
  - To compare, use the same processors!
  - Take the data transfers into account if necessary

Factors Affecting Performance

- The algorithm should be able to be parallelized!
- The volume of data to which it applies must be sufficiently large in relation to the number of processors used
- Additional overhead due to synchronization and memory access conflicts can reduce performance
- Load balancing between processors
- The use of parallel algorithms can increase the complexity of parallel algorithms compared to sequential algorithms
- The distribution of data between multiple memory units can reduce memory contention and improve the locality of the data, which can lead to performance gains

Overhead sources

- **Interactions between processes**
  - A non-trivial parallel algorithm will require interactions between processes during execution (synchronization, intermediate data exchange)
  - Communications are generally the most important sources of performance loss

- **Waiting time**
  - Because of many reasons like
    - A load imbalance,
    - synchronizations,
    - the presence of sequential parts.

Overhead sources

The fastest sequential algorithms for a given problem may prove to be difficult / impossible to parallelize

- Using a parallel algorithm based on a sequential algorithm that is simpler to parallelize (with a high degree of concurrency)
- Example: matrix product using Strassen or Winograd algorithms vs 3 loops

Difference between the number of operations between the best sequential algorithm and the parallel algorithm

- Overhead in number of operations
- But a parallel algorithm based on the best sequential algorithm can still perform more calculations than the sequential algorithm
- Example: Fast Fourier Transform (FFT)
  - In the sequential version, the results of some computations can be reused
  - In the parallel version, generated by different processors (thus performed several times by different processors)

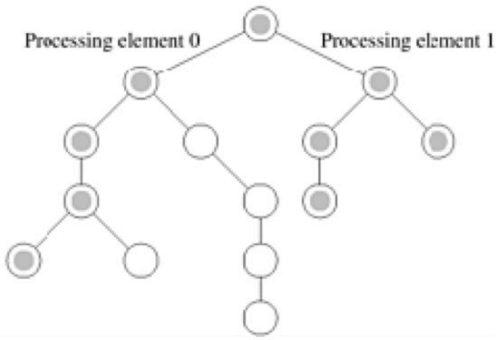
Acceleration (*speedup*)

- What **performance gain can be achieved** by parallelizing an application compared to its sequential implementation?
- The speedup is a measure that captures the relative benefit of solving a problem in parallel
- The speedup *S* is the **ratio of time to solve a problem on a single processor over time to solve a problem on a parallel *p* processors machine**
- It generally ranges between 0 and *p*, where *p* is the number of processors
  - Same type of processors between parallel and sequential execution
  - One should (normally) take the best sequential algorithm to solve the same problem
    - Sometimes it is not known or its implementation makes it ineffective
    - Then take the best implementable algorithm

Superlinear speedup

- There are sometimes accelerations greater than p

- **This happens when**
  - The work done by a sequential algorithm is superior to that of its parallel version
  - Exemple: search, algorithms in trees



  - If the data enters the caches for the parallel version
    - The performance of larger memory sizes is less important

Efficiency

- Efficiency measures the fraction of time for which a processor is used in a useful way

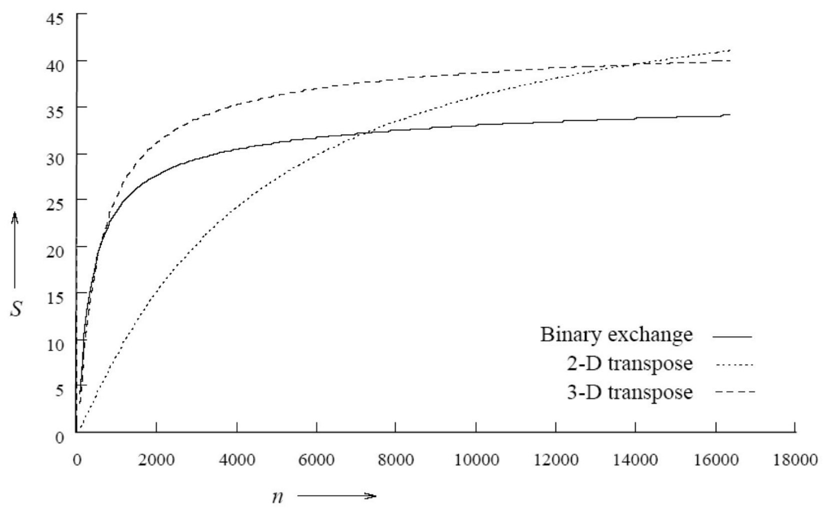
$$E = S/p$$

  - An efficient system has an efficiency equal to 1
  - In practice $0 \leq E \leq 1$

Scalability of parallel systems

- **Extrapolate performance**
  - How to move from a small problem on a small system
  - to a big problem on a larger configuration
- **Examples:** 3 algorithms to compute a n-point FFT on 64 processors

- Choosing this algorithm depending of configurations



Scalable parallel systems

- Total overhead function $T_o(T_s, p)$
  - Best sequential time T_s
  - Number of processors p

$$T_o = pT_p - T_s$$

- Efficiency

$$E = T_s / pT_p = T_s / (T_o + T_s) = 1 / (1 + T_o / T_s)$$

- Often, we have $T_o(T_s, p) / T_s < 1$
  - T_o grows in a sub-linear manner with respect to T_s
  - In this case, the efficiency increases if the size of the problem increases and if the number of processors is constant
- For such systems, it is possible to keep a constant efficiency by
  - Increasing the size of the problem
  - Increasing the number of processors proportionally
- Such systems are **scalable**

Scalability of parallel programs

- In scientific papers we read observations such as

  "We implemented an algorithm on the parallel machine X which obtained an acceleration of 10.8 out of 12 processors with a problem size equal to 100."

- A dot on a curve!
  - What happens if we have 100, 1000 processors?
  - What happens if we have data of size 10, 1000?

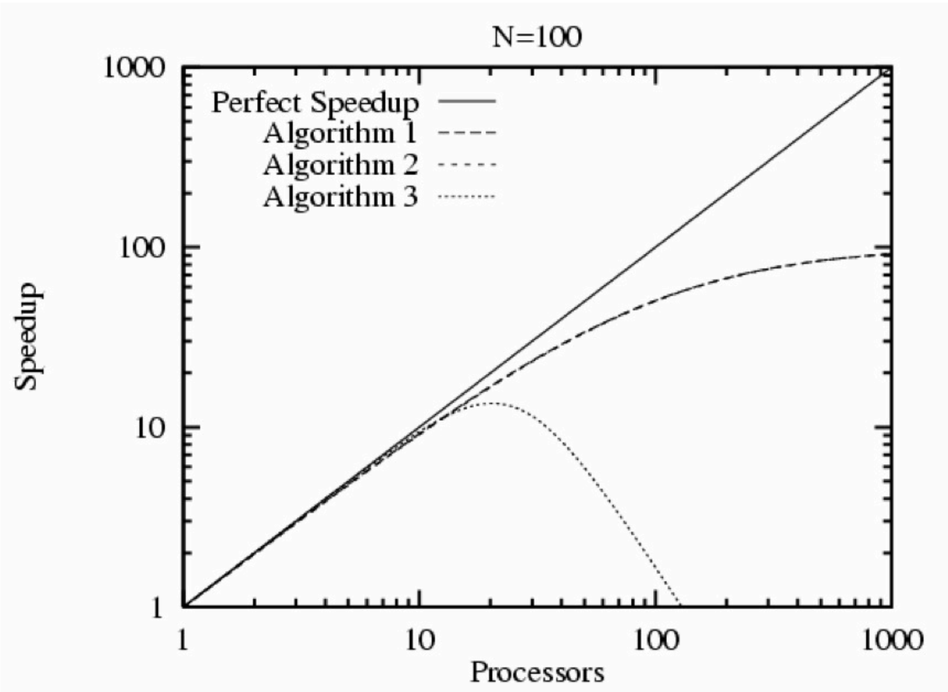
Scalability of parallel programs, contd.

- Three theoretical performance models
  - *$T = N + N^2 / P$*
    - This algorithm splits N^2 computations but also replicates N other computations
    - No other sources of additional cost
  - *$T = (N + N^{2)} / P + 100$*
    - This algorithm splits all the computations and adds an additional cost of 100
  - *$T = (N + N^{2)} / P + 0.6 P^2$*
    - This algorithm splits all the computations and adds an additional cost of $0.6 P^2$

- All these algorithms have an acceleration of 10.8 on 12 processors for *N = 100* !

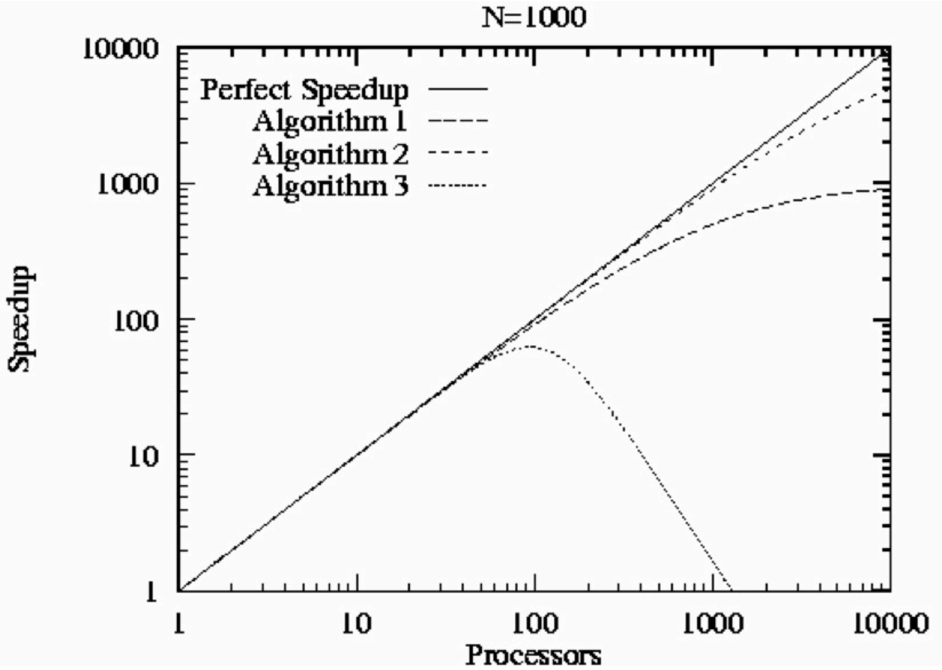
Scalability of parallel programs, contd.

If we increase the number of processors for N = 100



Scalability of parallel programs, contd.

If we increase the number of processors for N = 1000



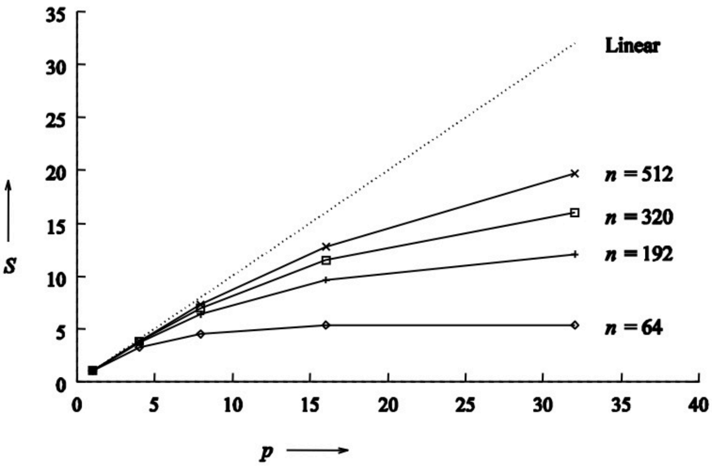
Scalability of parallel programs, contd.

- Adding *n* numbers on *p* processors
- Supposition: addition = communication = 1 time unit

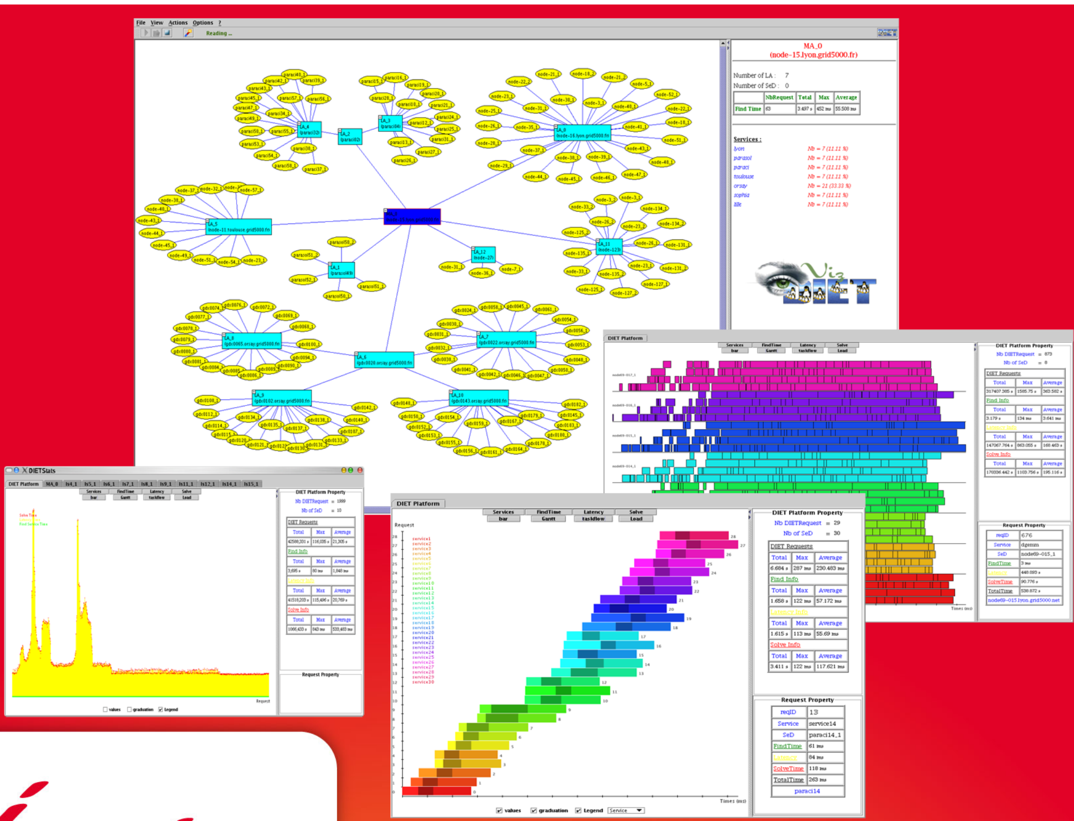
$$T_P = \frac{n}{p} + 2\log p$$

$$S = \frac{n}{\frac{n}{p} + 2\log p}$$

$$E = \frac{1}{1 + \frac{2p\log p}{n}}$$



Acceleration tends to saturate and efficiency decreases



INVENTEURS DU MONDE NUMÉRIQUE