Latinamerican School for Computational Materials Science.

Linux cluster approach to parallel computing

Stefano Cozzini

CNR-INFM DEMOCRITOS, Trieste

Santiago, Chile - Enero, 2009

Agenda

- Parallel computing: a few ideas
- Linux clusters for parallel computing?
- Hardware bricks for Linux Clusters
- Software stack
- How/where/when to choose a Linux Cluster ?

Parallel Programming Paradigms

The two architectures determine two basic schemes for parallel programming

Data Parallel (shared memory)

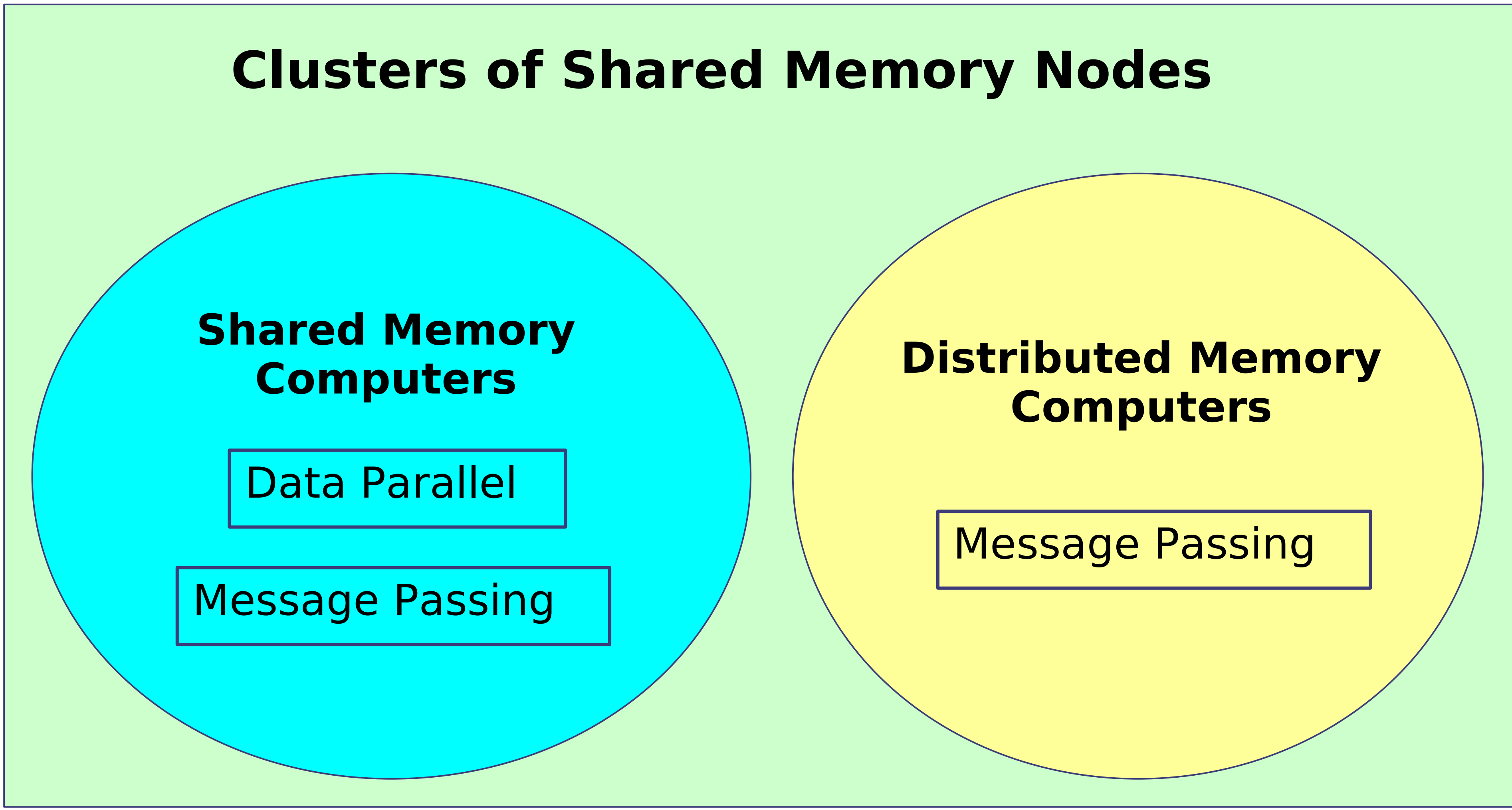
Single memory view, all processes (usually threads) could **directly** access the whole memory

Message Passing (distributed memory)

all processes could **directly** access only their local memory

Its easy to adopt a Message Passing scheme in a Shared Memory computers (*unix process have their private memory*).
Its less easy to follow a Data Parallel scheme in a Distributed Memory computer (*emulation of shared memory*)

Architectures vs. Paradigms



Parallel programming: a short summary..

Architectures	
Distributed Memory	Shared Memory
Programming Paradigms/Environment	
Message Passing	Data Parallel
Parallel Programming Models	
Domain Decomposition	Functional Decomposition

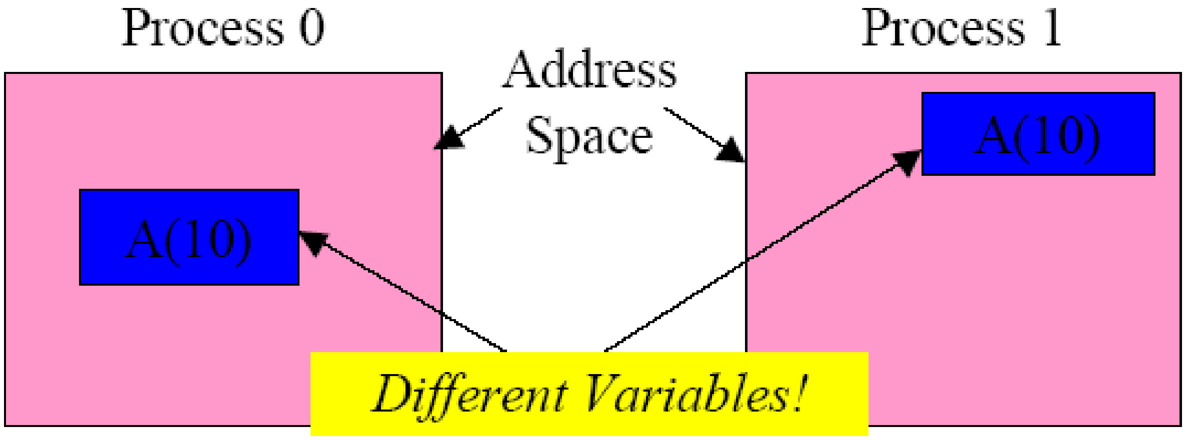
Parallel Programming Paradigms, cont.

Programming Environments	
Message Passing	**Shared Memory**
Standard compilers	Ad hoc compilers
Communication Libraries	Source code Directive
Ad hoc commands to run the program	Standard Unix shell to run the program
Standards: **MPI**	Standards: **OpenMP**

Message passing paradigm

- Parallel programs consist of separate processes, each with its own address space
  - Programmer manages memory by placing data in a particular process
- Data sent explicitly between processes
  - Programmer manages memory motion
- Collective operations
  - On arbitrary set of processes
- Data distribution
  - Also managed by programmer

Distributed memory (shared nothing approach)



Principles of Parallel Computing

- Speedup, efficiency, and Amdahl’s Law
- Finding and exploiting parallelism
- Finding and exploiting data locality
- Load balancing
- Coordination and synchronization
- Performance modeling

All of these things make parallel programming more difficult than sequential programming.

Speedup

- The *speedup* of a parallel application is

  Speedup(p) = Time(1)/Time(p)

- Where
  - Time(1) = execution time for a single processor
  - Time(p) = execution time using p parallel processors
- If Speedup(p) = p we have *perfect speedup* (also called *linear scaling*)
- speedup compares an application with itself on one and on p processors
- more useful to compare
  - The execution time of the best serial application on 1 processor

versus

  - The execution time of best parallel algorithm on p processors

Efficiency

- The *parallel efficiency* of an application is defined as

$$\text{Efficiency}(p) = \text{Speedup}(p)/p$$

  - Efficiency(p) <= 1
  - For perfect speedup Efficiency (p) = 1
- We will rarely have perfect speedup.
  - Lack of perfect parallelism in the application or algorithm
  - Imperfect load balancing (some processors have more work)
  - Cost of communication
  - Cost of contention for resources, e.g., memory bus, I/O
  - Synchronization time
- Understanding why an application is not scaling linearly will help finding ways improving the applications performance on parallel computers.

Superlinear Speedup

Question: can we find "*superlinear*" speedup, that is

$$\text{Speedup}(p) > p \quad ?$$

- Choosing a bad "baseline" for T(1)
  - Old serial code has not been updated with optimizations
  - Avoid this, and always specify what your baseline is
- Shrinking the problem size per processor
  - May allow it to fit in small fast memory (cache)
- Application is not deterministic
  - Amount of work varies depending on execution order
  - Search algorithms have this characteristic

Amdahl's Law

- Suppose only part of an application runs in parallel
- Amdahl's law
  - Let s be the fraction of work done serially,
  - So (1-s) is fraction done in parallel
  - What is the maximum speedup for P processors?

$$Speedup(p) = T(1)/T(p)$$

$$T(p) = (1-s)*T(1)/p + s*T(1)$$

$$= T(1)*((1-s) + p*s)/p$$

assumes perfect speedup for parallel part

$$Speedup(p) = p/(1 + (p-1)*s)$$

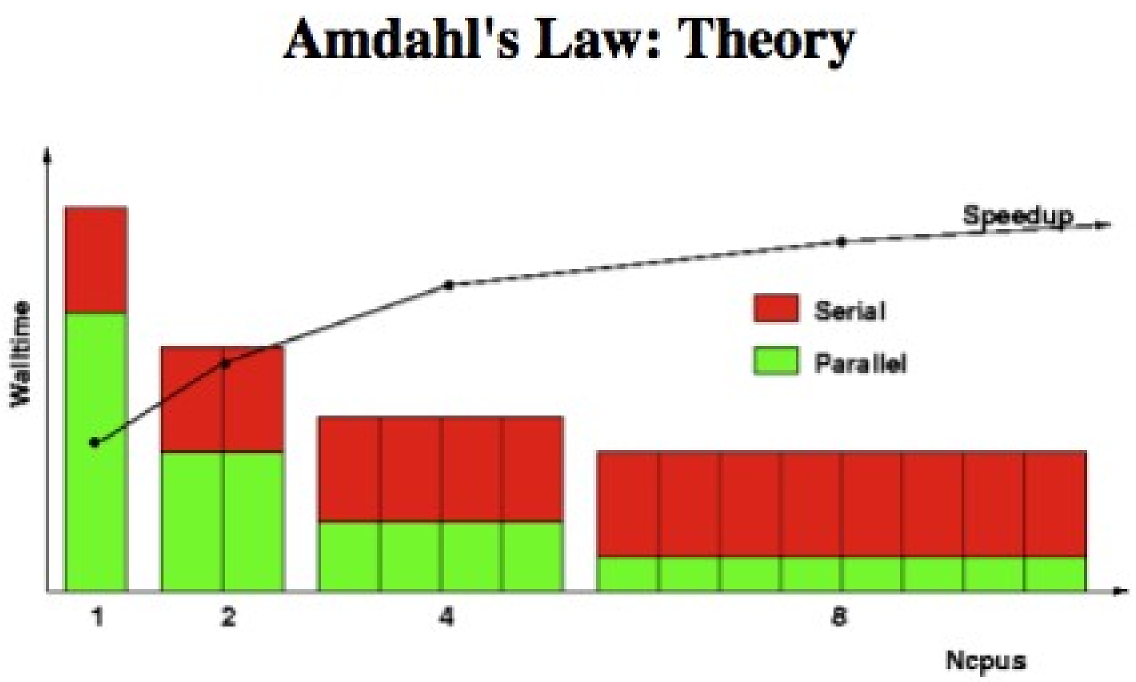
Even if the parallel part speeds up perfectly, we may be limited by the sequential portion of code.

Serial components

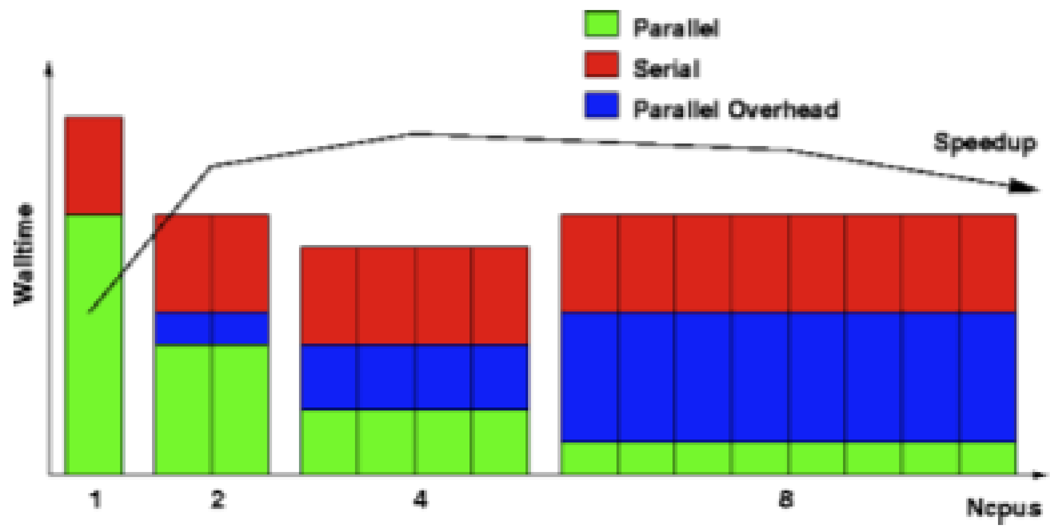
- Code executed by a single process:

  if (rank == 0) { ... }

- IO through a single process
- Operations done redundantly on local copies of redundant data serial fraction:



Ahmdal law in practice..



- parallel overhea
  - Any operations not in the serial code
  - Number of such operations often increases with N cpus
  - Any message passing or synchronization
  - Extra redundant computation
  - Algorithm changes
  - Extra system overhead, e.g. forking threads for threaded libraries

Amdahl's law(2)

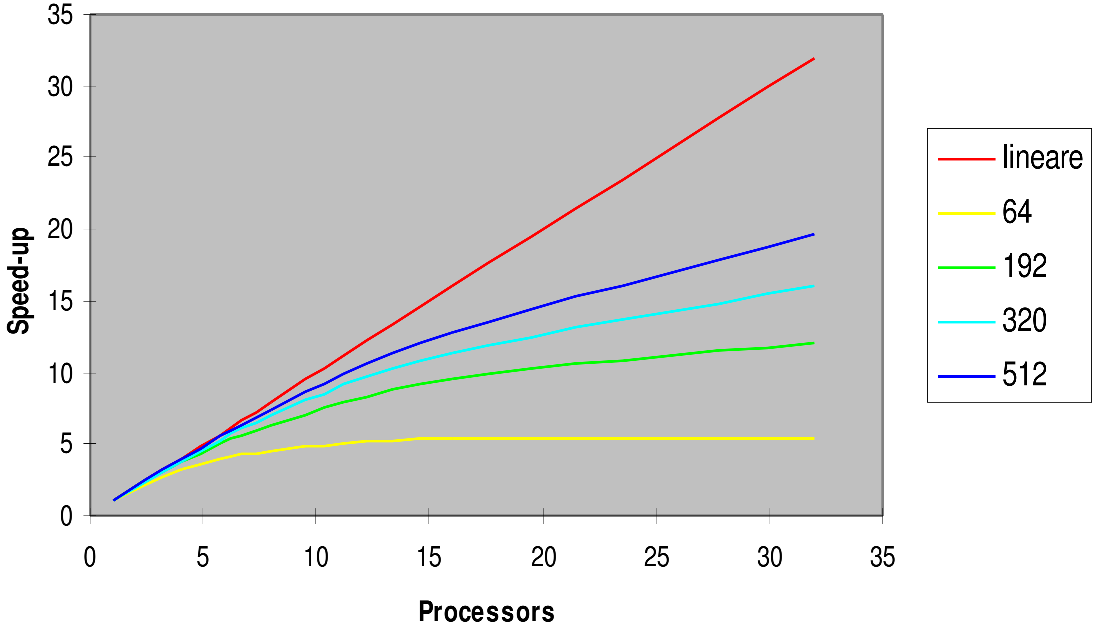
- Which fraction of serial code(parallel overhead) is it allowed ?

>	2	4	8	32	64	256	512	1024
5%	1.91	3.48	5.93	12.55	15.42	18.62	19.28	19.63
2%	1.94	3.67	6.61	16.58	22.15	29.60	31.35	32.31
1%	1.99	3.88	7.48	24.43	39.29	72.11	83.80	91.18

What about Scalability ???

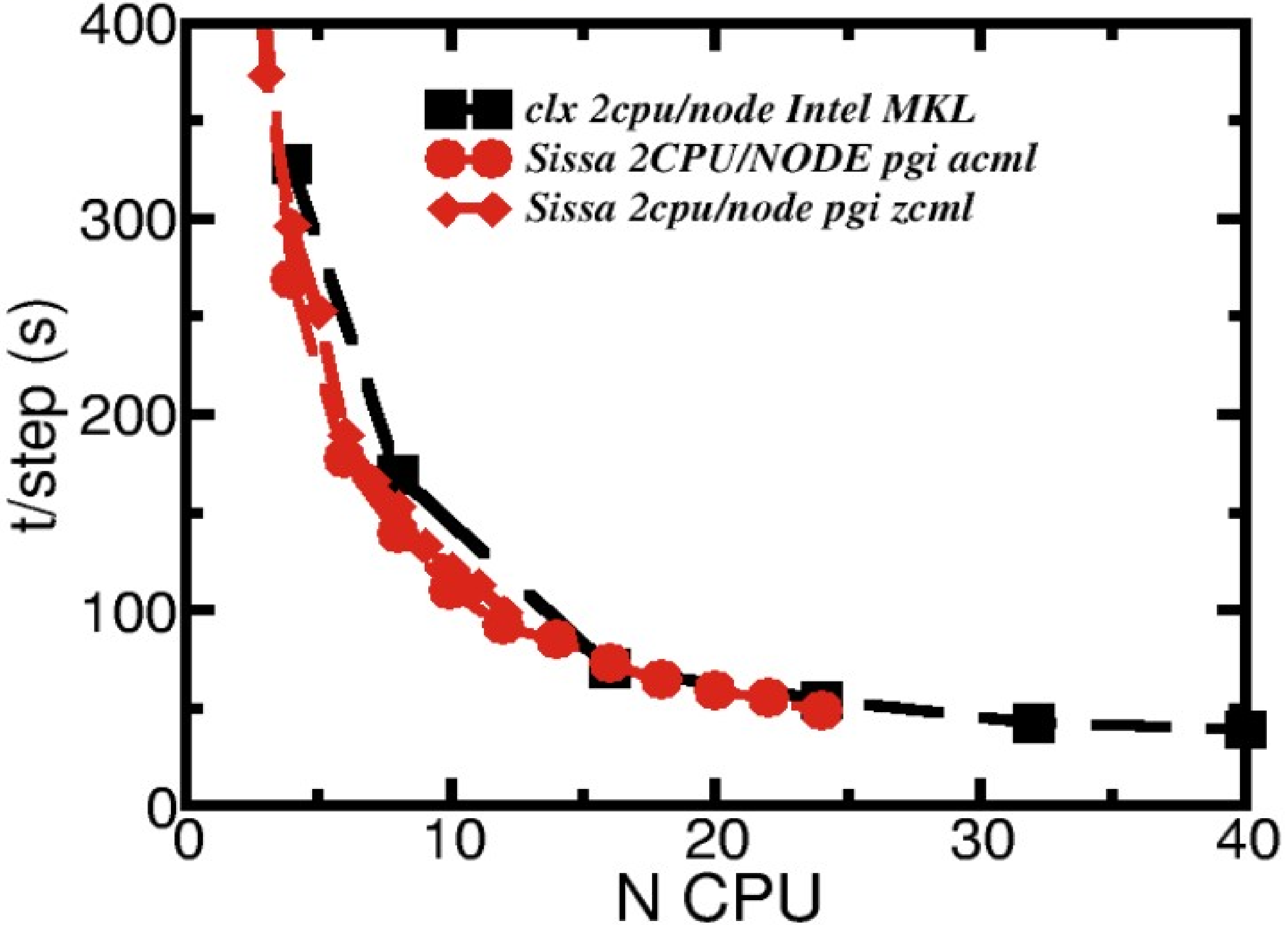
Problem scaling..

- Amdahl’s Law is relevant only if serial fraction is indipendent of problem size, which is rarely true
- Fortunately “The proportion of the computations that are sequential (non parallel) normally decreases as the problem size increases ” (a.k.a. Gustafon’s Law)



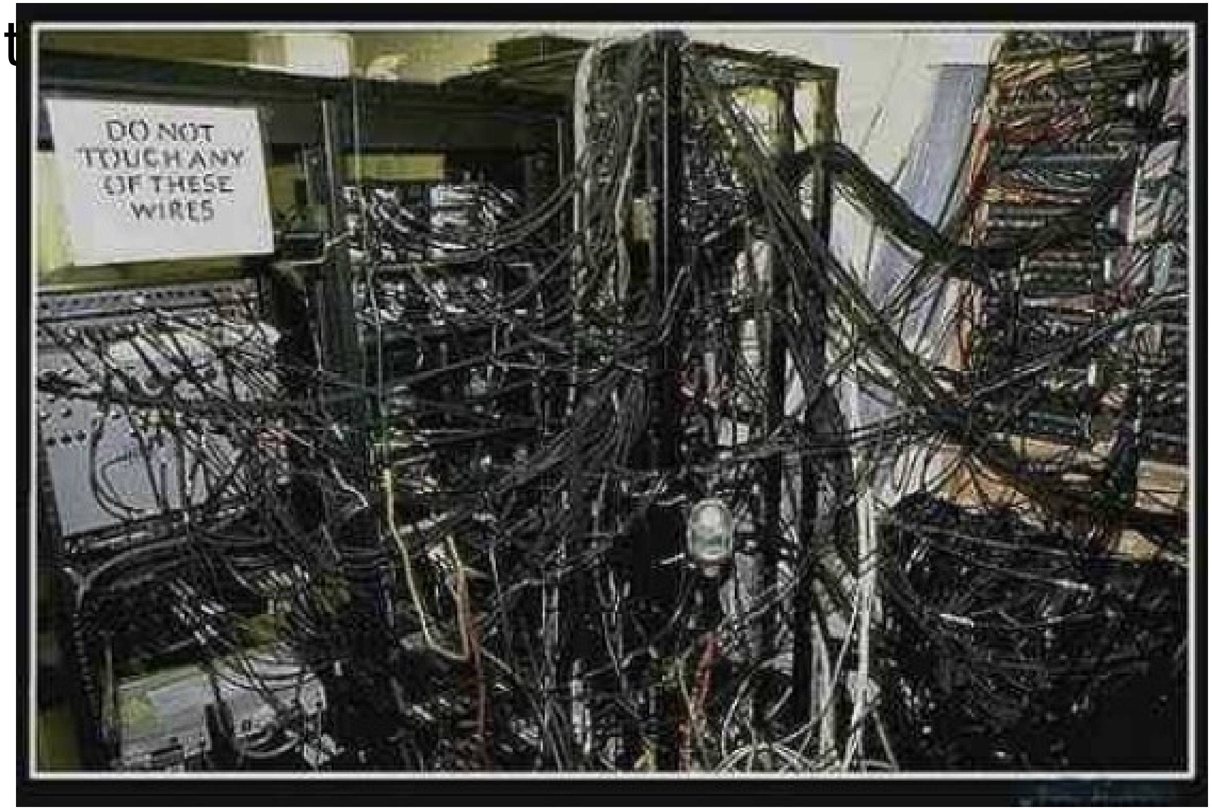
Real parallel programs

Typical comparison of performance of an MPI code on a given Platform is to plot time vs N_{CPU}



About network for clusters

- The characteristics of the network cannot be ignored
  - Topology
    - Diameter
    - Bisection bandwidth
  - Performance
    - Latency
    - Link bandwidth



Interconnect Topologies

- ***Bus***
  - Nodes share a "party line".
  - Not very common any more, except between processors and memory inside a host.
- ***Hypercube***–SGI Origin and Altix
  - Nodes are vertices on an n-dimensional hypercube.
- ***Mesh***–Cray T3D/E and XT-3/4/5, IBM BlueGene
  - A 1D mesh with wrap-around at the edges is called a ***ring***.
  - A 2D (or more) mesh with wrap-around at the edges is called a ***torus***.
- ***Switched***–Ethernet, Infiniband, Myrinet,
  - Nodes are connected to a concentrator called a switch.
  - Multiple switches may be connected hierarchically (i.e. as a tree) or in any of the above topologies.

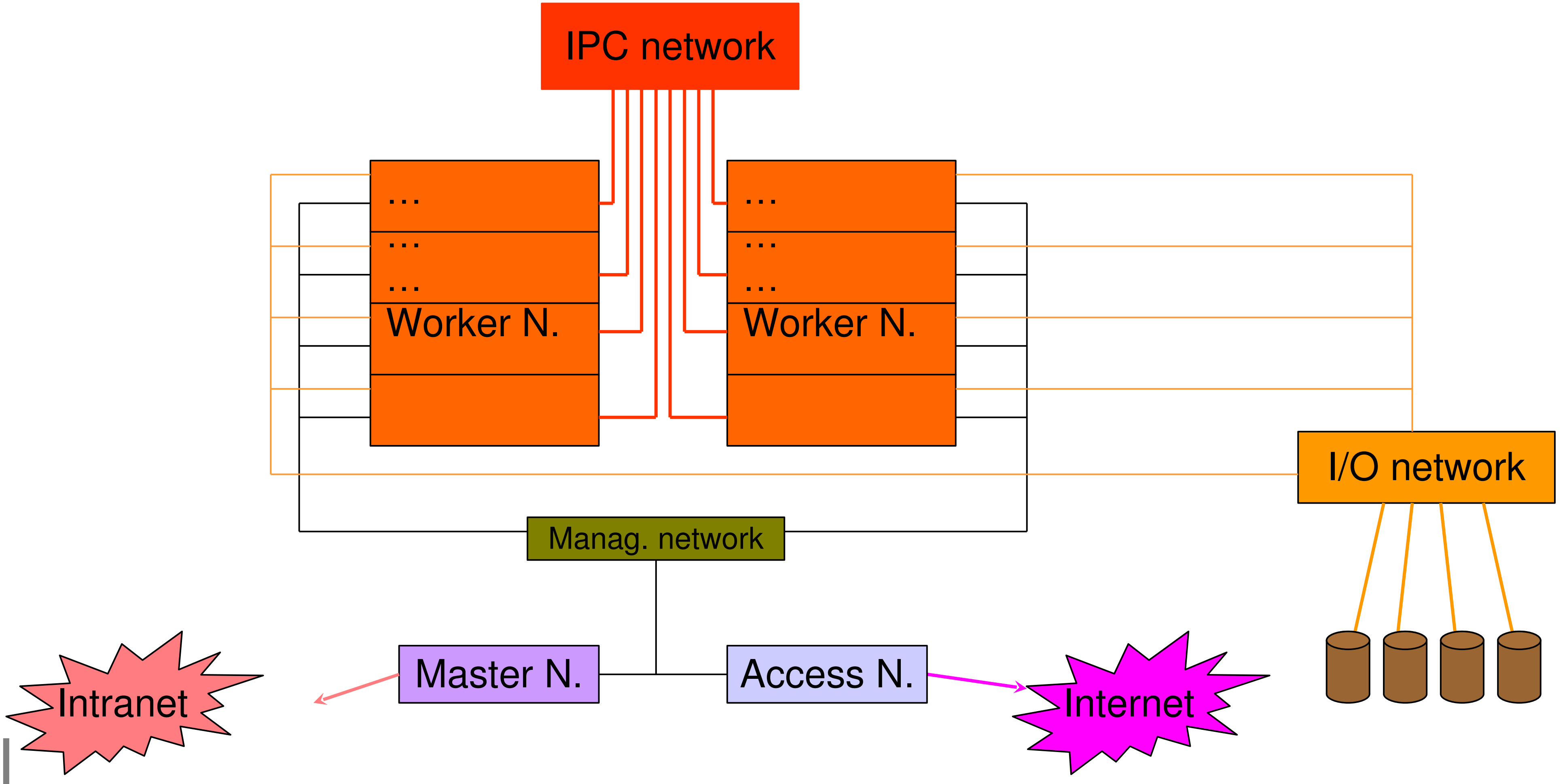
Interconnect Characteristic

- ***Latency:*** Initialization time before data can be sent
- ***Per-link Peak Bandwidth:*** Maximum data transmission rate (varies with packet size)
- ***Diameter***: Maximum number of hops to get between most distantly connected nodes.
  - Hypercube networks have best diameter, at most log 2(N) for N nodes.
- ***Bisection Bandwidth***: Bandwidth available if one half of nodes try communicating with the other half simultaneously.
  - Torus networks typically have the best bisection bandwidth

Which networks for Linux Cluster ?

- Commodity
  - Gigabit Ethernet
- High Speed Network
  - Myrinet
  - Infiniband
- Difficult choice:
  - Which kind of cluster (HTC or HPC) ?
  - Which kind of application ?
    - Serial/Parallel
    - Parallel loosely coupled / tightly coupled ?
    - Latency or bandwidth dominated ?
  - Budget considerations
  - I/O considerations

HPC cluster logical structure



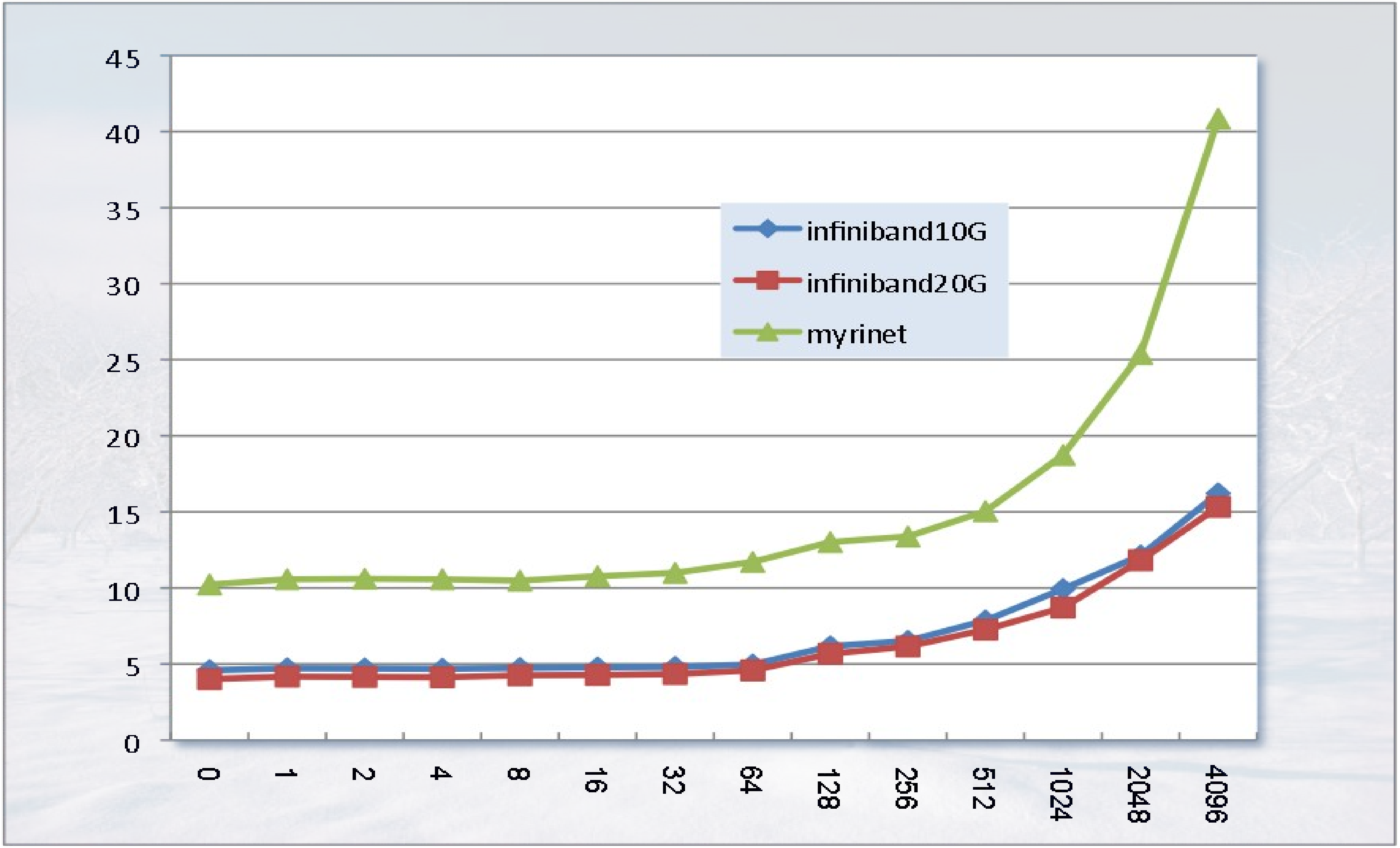
Luxury clusters: 3 networks

- HIGH SPEED NETWORK
  - parallel computation
    - low latency /high bandwidth
    - Usual choices: Myrinet/SCI/Infiniband...
- I/O NETWORK
  - I/O requests (NFS and/or parallel FS)
    - latency not fundamental/ good bandwidth
    - GIGABIT is ok
- Management network
  - management traffic
    - any standard network (fast ethernet OK)

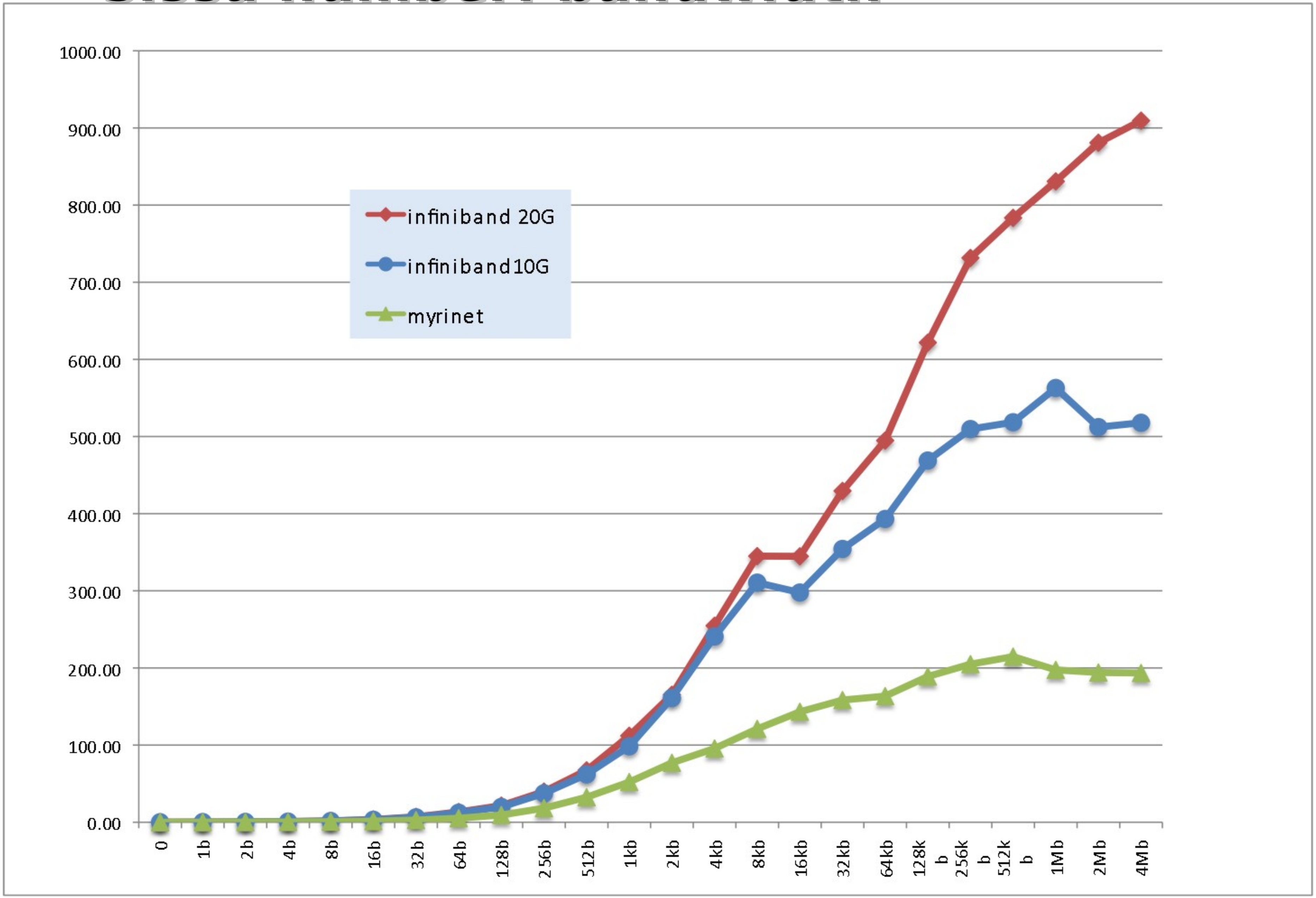
Interconnect Characteristics:

- Latency: Initialization time before data can be sent
- Per-link Peak Bandwidth: Maximum data transmission rate (varies with packet size)
- To measure it:
  - IMB benchmark : it will be use later in the lab..

Sissa cluster: latency



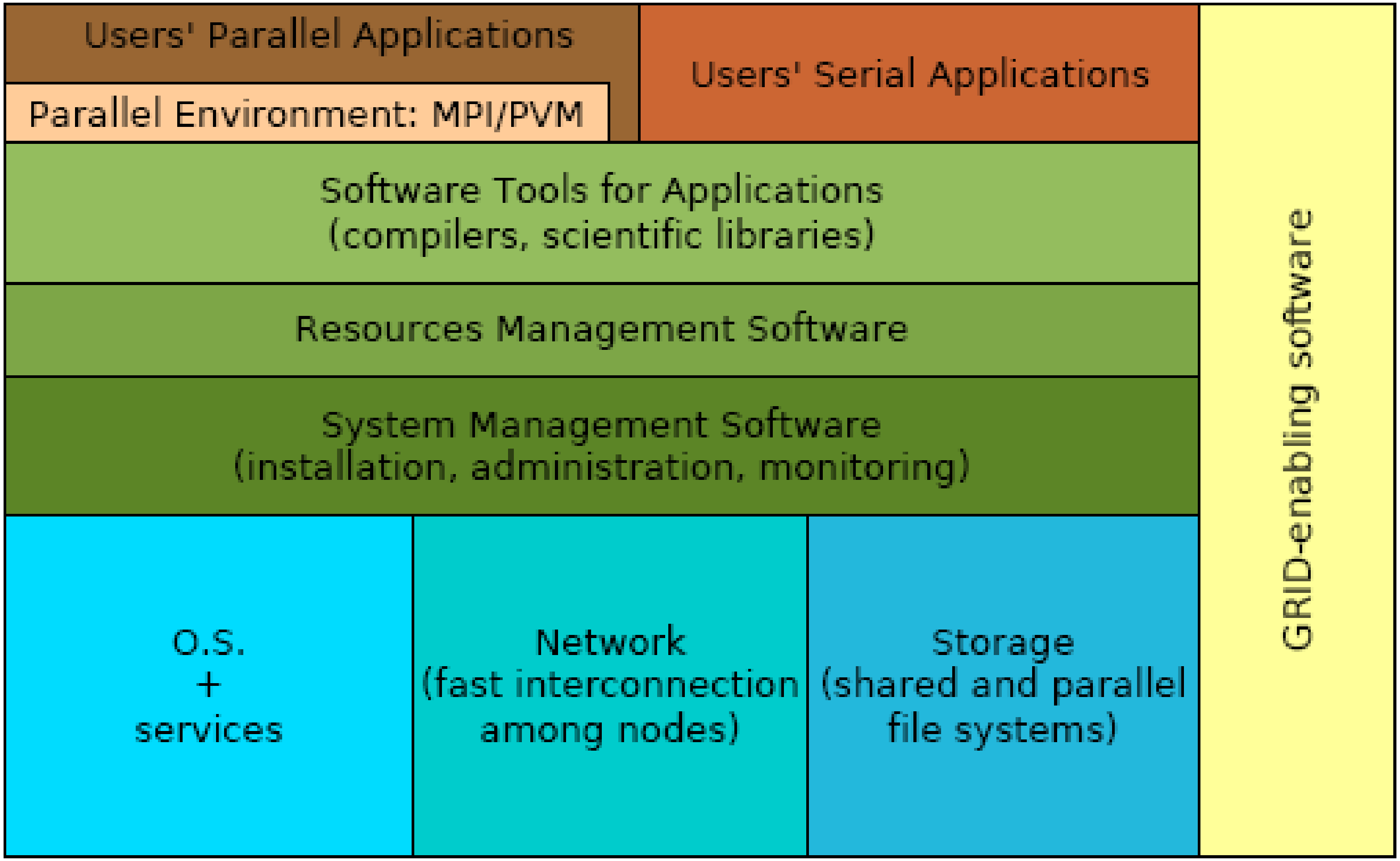
Sissa number: bandwidth



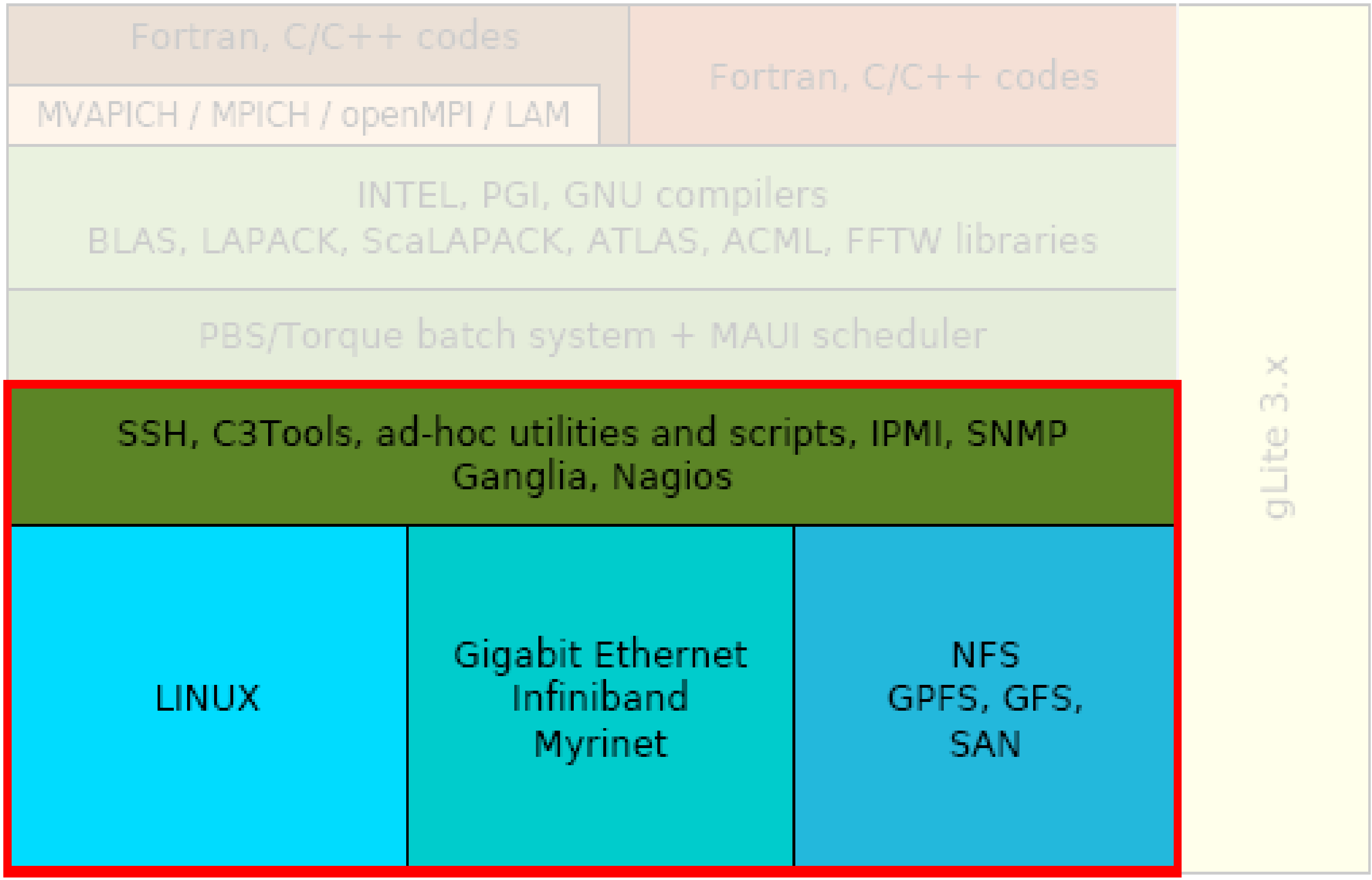
high speed network considerations

- In general the compute/communication ratio in a parallel program remains fairly constant.
- So as the computational power increases the network speed must also be increased.
- As multi-core processors proliferate, it is increasingly common to have 4, 8, or even 16 MPI processes sharing the same network device.
- Contention for the interconnect device can have a significant impact on performance.

Linuux Cluster: the software stacks



Linux Cluster: the sys. Adm. stacks

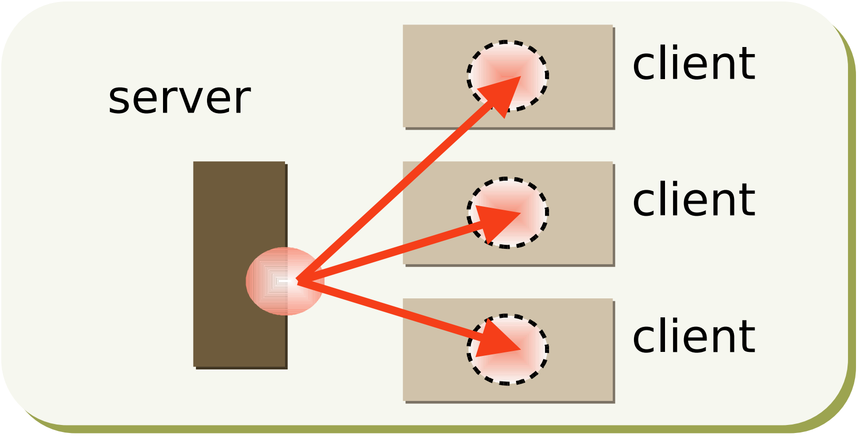


Middleware Design Goals

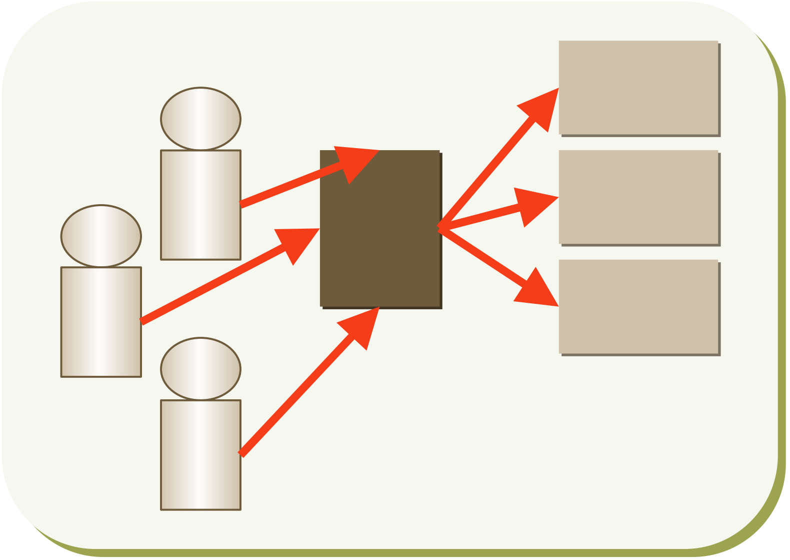
- Complete Transparency (Manageability):
  - Lets the see a single cluster system..
    - Single entry point, ftp, ssh, software loading...
- Scalable Performance:
  - Easy growth of cluster
    - no change of API & automatic load distribution.
- Enhanced Availability:
  - Automatic Recovery from failures
    - Employ checkpointing & fault tolerant technologies
  - Handle consistency of data when replicated..

Cluster middleware: beowulf approach

- Administration software:
  - NFS
  - user accounts
  - NTP



- Resource management and scheduling software (LRMS)
  - Process distribution
  - Load balance
  - Job scheduling of multiple tasks



Cluster Management Toolkits

- Are generally made of an ensemble of already available software packages thought for specific tasks, but configured to operate together, plus some add-ons.
- Sometimes limited by rigid and not customizable configurations, often bound to some specific LINUX distribution and version.

  May depend on vendors' hardware.
- Free and Open
  - OSCAR (Open Source Cluster Application Resources)
  - NPACI Rocks
  - xCAT (eXtreme Cluster Administration Toolkit)
  - Warewulf ....
- Commercial
  - Scyld Beowulf
  - IBM, HP, SUN and other vendors' Management Software...

Cluster Pro&Cons

- Pro:
  - Price/performance when compared with a dedicated parallel supercomputer
  - Great opportunity for low budget institution
  - Flexibility: many ad hoc solution for different problems..
  - Open Technology
    - What you learn in this business can be used everywhere..
- Cons:
  - It is hard to build and operate medium and large cluster
    - Large collection of software that are not “talk to each other”
  - Lot of expertise needed (no plug and play yet)
  - How to use cluster power efficiently

Which cluster do I need ?

- Which applications ?
  - Parallel
    - Tightly coupled
    - Loosely coupled
  - Serial
    - Memory / I/O requirements
- Which user's community ?
  - Large /Small
  - Homogeneous /heterogeneous

- Understand your computational problem before buying/building a cluster !
- Run your own benchmarks before buying/building a cluster !