# Computational Science of Computer Systems

## Méthodologies d'expérimentation pour l'informatique distribuée à large échelle


Martin Quinson

(with the SimGrid Team and others)

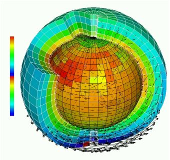

September 10th 2014

ENS Cachan

# What is Science anyway?

## Doing Science = Acquiring Knowledge

$$\frac{\partial}{\partial x_j}\left(\frac{\partial \Phi}{\partial x_i}\right) = \frac{\partial}{\partial x_i}\left(\frac{\partial \Phi}{\partial x_j}\right)$$



**Experimental Science**

- Thousand years ago
- Observations-based
- Can describe
- Prediction tedious

**Theoretical Science**

- Last few centuries
- Equations-based
- Can understand
- Prediction long

**Computational Science**

- Nowadays
- Compute-intensive
- Can simulate
- Prediction easier

*Prediction is very difficult, especially about the future. – Niels Bohr*

# Observation still bases Science

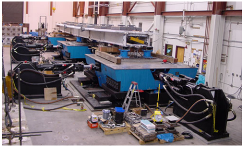
Space telescope

Large Hadron Collider

Mars Explorer

Tsunamis

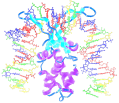
Earthquake vs. Bridge

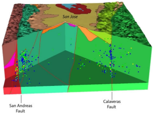


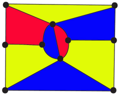
Climate vs. Ecosystems

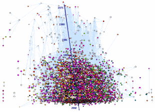
*(who said that science is not fun??)*

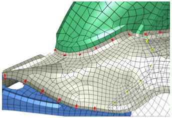
# Computational Science

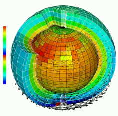




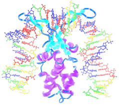
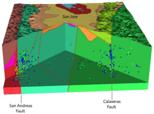



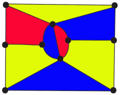
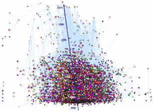
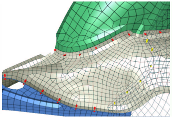
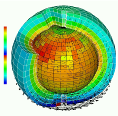






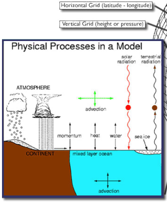
# Computational Science

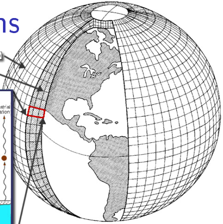



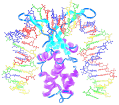
## Understanding the Climate Change with Predictions

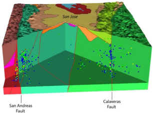


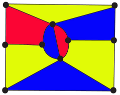


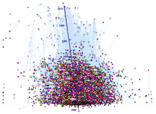


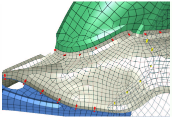
# Computational Science

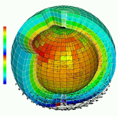




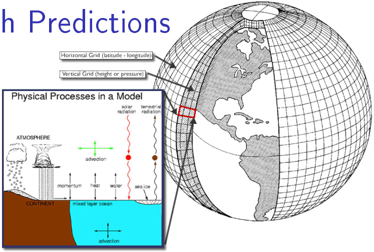




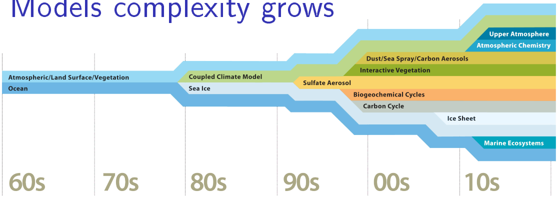




## Understanding the Climate Change with Predictions



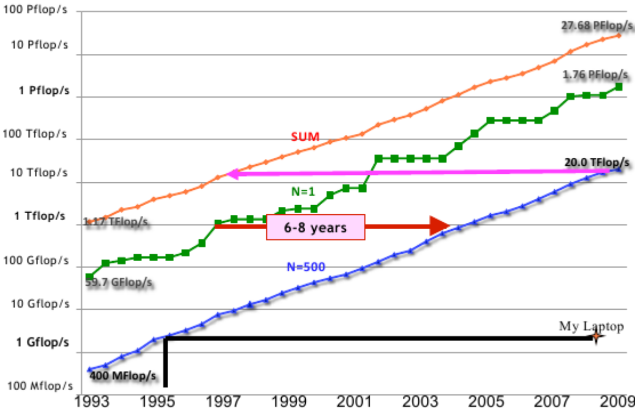

## Models complexity grows




this requires
**large computers**

# How Big are SuperCompting Facilities?

- Size unit: Floating point Operation per Second (FLOP/S)
- Ranking of biggest computers: TOP500. (Bench: Linpack)

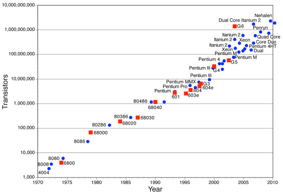



Exponential increase of power

- Laptop ≈ 10-year old SuperComputer (smartphone ≈ 10-year old laptop)

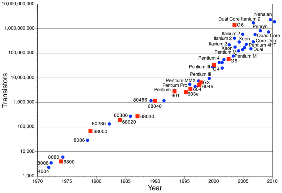
# If there is a Ranking, there is a Race

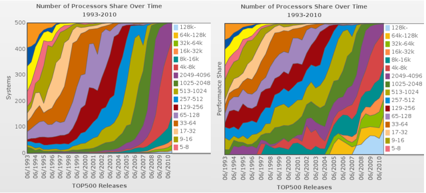
- Leading IT ($\approx$ and CS) since 20 years
- Moore law: Power doubles every 18 months
- US fight hard to keep pole position (Japan 2002–2004, China since 2013)
- Used to be a tough game for founders, increasing transistors per CPU



# If there is a Ranking, there is a Race

- Leading IT ($\approx$ and CS) since 20 years
- Moore law: Power doubles every 18 months
- US fight hard to keep pole position (Japan 2002–2004, China since 2013)
- Used to be a tough game for founders, increasing transistors per CPU




## But nowadays, hardware makers lazily stack up components

- Category "+128k processor": 9 machines out of 500, 19% of computing power
- Programming efficiently 128k processors is a tremendous challenge
- ExaScale systems with billions processors expected before 2020!
- And as software programmers, **we** have to deal with the resulting mess

Why did founders gave up on us? What did they try before?

# At first, they reduced the transistor size

## Intel 4004 (1971)

With todays technologies, we could put 15 complete processors on each transistor of the original

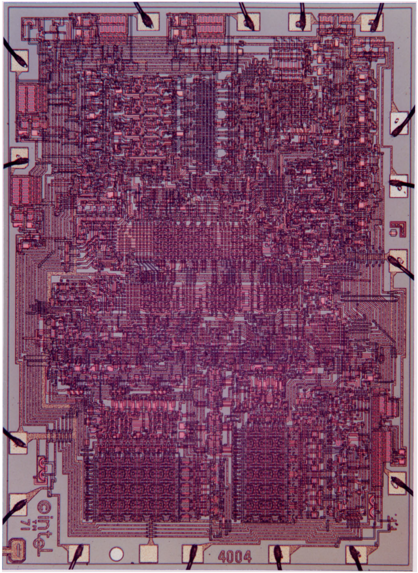
## But there is limits

- Current wire size: dozens of atoms

## Show Must Go On

- Intel and co business plan:
- Computers last only 3 years
- (not only for the TOP500 race)

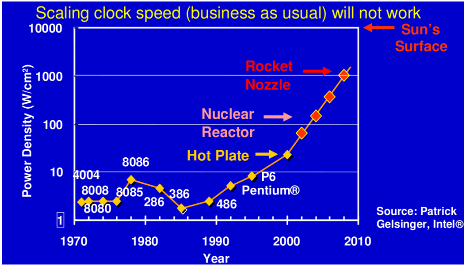
Then they increased the electric frequency



# But it failed! Because of Energy

## Limit #1: Power density

- Increasing frequency consumes and dissipates too much energy
- When frequency ↗, Computation ↗ linearly; Energy ↑ quadratically
- More energy means higher temperature

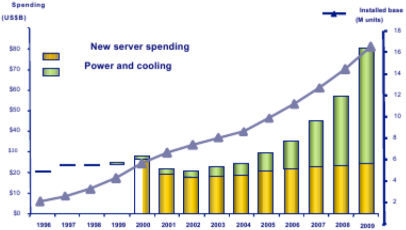




# Electric Power becomes THE problem

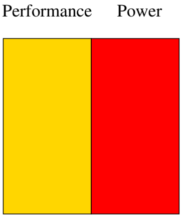
## Limit #2: Energy costs

- IT industry dissipate 1% of world wide electric production
- 1Mw/h is 1M$ per year, and data centers dissipate hundreds of
- Microsoft's DataCenter in Chicago: 198Mw (Nuclear Power Plant: 1000-1500Mw)
- Power becomes more expensive than servers!



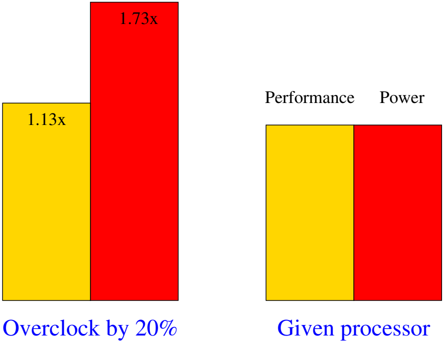

- Can soon put more transistors on chip than can afford to turn on. – Patterson'07

# Stacking up components to save energy

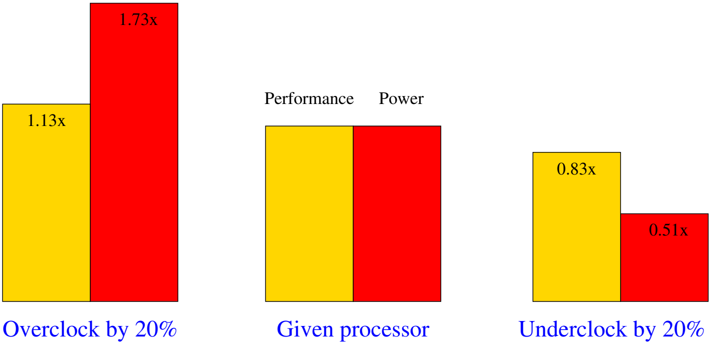



Given processor

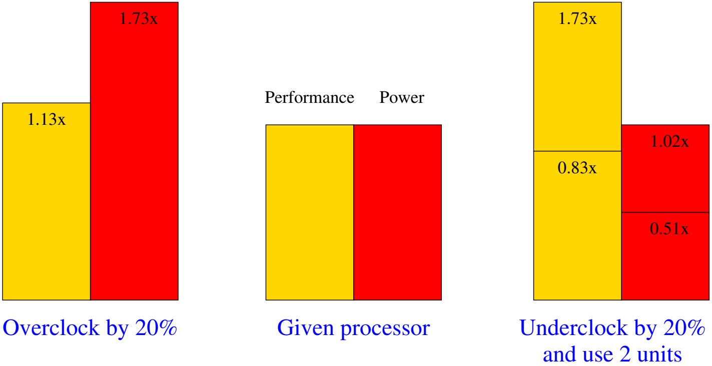
# Stacking up components to save energy



# Stacking up components to save energy



# Stacking up components to save energy




## This explains why our program must go parallel

- That's a real pain to program though
- Good news: There is no physical limit to fear anymore
  Bad news: Nobody knows how to leverage millions of cores efficiently
- Good news: this actually saves energy nowadays (greener is cheaper)

# Energy-Efficient Performance

## Super Computer in 1996

- Performance: 1 TeraFlop
- Efficiency: 1,000 Flop per watt

# Energy-Efficient Performance

Super Computer in 1996

- Performance: 1 TeraFlop
- Efficiency: 1,000 Flop per watt

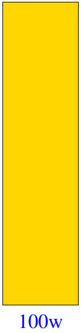
Mainstream component in 2009


- Performance: 2.4 TeraFlop
- Efficiency: 1,600,000 Flop per watt

And there is a lot of further power savings to do

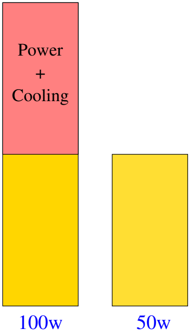
# Data Center Energy Losses

Where go all that energy that data centers consume?



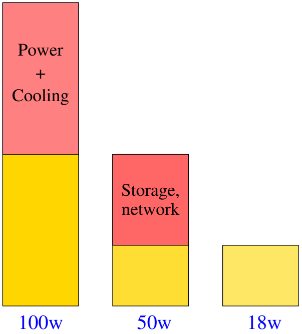
# Data Center Energy Losses

Where go all that energy that data centers consume?



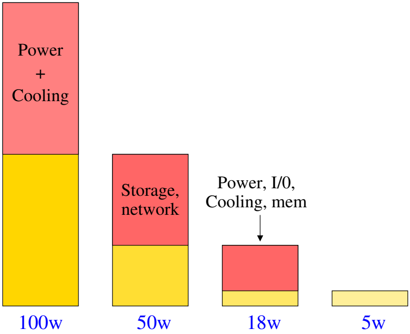
# Data Center Energy Losses

Where go all that energy that data centers consume?



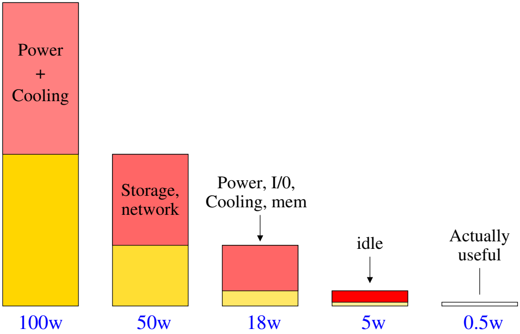
# Data Center Energy Losses

Where go all that energy that data centers consume?



# Data Center Energy Losses

## Where go all that energy that data centers consume?

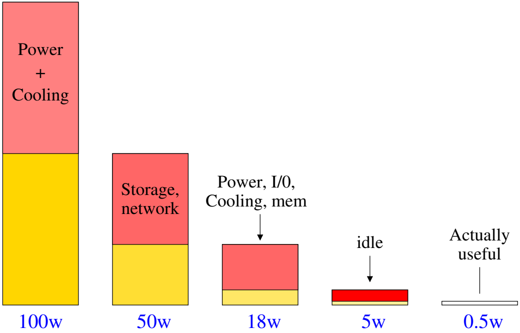



## This is a HUGE waste of resource

- Data centers are 1 to 5% efficient only

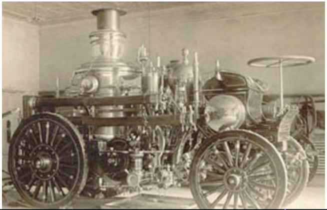
# Data Center Energy Losses

Where go all that energy that data centers consume?



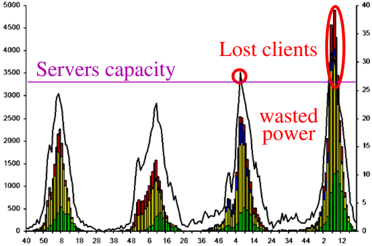

This is a HUGE waste of resource

- Data centers are 1 to 5% efficient only
- Steam engines are 10-15% efficient



# From Idling Computers to Cloud Computing

- Big vendors on the Internet are subject to flash crowd effects
  People tend to buy at day, and more December 20. than on August 15.
- To not loose clients, vendors over-dimension their servers




### Amazon idea

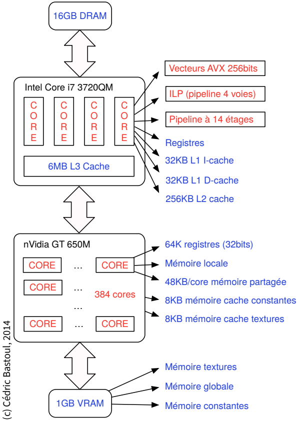
- Rent unused power to others!
- Computers better amortized
  Buy bigger ones, loose no client
- Infrastructure as a Service (IaaS)
- Highly Cost-Efficient Computing

- Elastic Computing: Pay only what you need/use
- Cloud Computing: Rent Infra (IaaS), OS+apps (PaaS) or even Software (SaaS)
- Virtualization: ease things, allows optimization (=hardware over booking)
- IT services gets externalized to specialists, that cut costs through scale

# What about Domestic Systems?

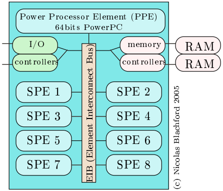
## Laptops

- 2.7 billions of transistors
- 5 kind of parallelism
- 15 sorts of memory
- 4 programming models




## Gaming Systems

- Cell processors (Playstation3)
- Heterogeneous cores




## A hardware issue turned into a software one

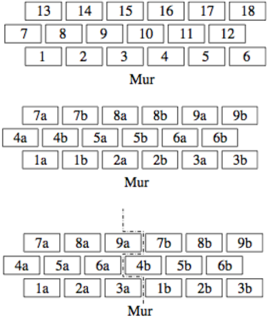
- Intel would sell 1024+ core CPUs, if someone could have any use of these

# What is Parallelism in practice??

- It's about splitting the work to do in sub-elements
- and letting several entities do these tasks
- Hard points: Spliting work, Coordinating entities

# What is Parallelism in practice??

- It's about splitting the work to do in sub-elements
- and letting several entities do these tasks
- Hard points: Spliting work, Coordinating entities




## To build a wall, you could

- Do it alone, but it's slow!
- (a) and (b) take one stone after the other
  They hinder each other
- Split the space correctly for better interactions

- Numerical analysis: split your data while preserving data dependencies
- Data Parallelism can be near to impossible on some problem
- You can often combine it with Task Parallellism

# Problème de l'exclusion mutuelle

## Exemple : deux banques modifient un compte en même temps

**Agence Nancy**

```
1. courant = get_account(1867A)
2. nouveau = courant + 10
3. update_account (1867A, nouveau)
```

**Agence Karlsruhe**

```
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 10
3. update_account(1867A, neue)
```

- variables partagées + exécutions parallèles entremêlées ⇒ différents résultats :
- ( 0 ; ? ; ?) N1( 0 ; 0 ; ? ) N2( 0 ;10 ; ? ) N3(10 ;10 ; ?)
  K1(10 ;10 ;10) K2(10 ;10 ;0) K3( 0 ;10 ; 0) → compte inchangé
- (0 ; ? ; ?) N1 (0 ;0 ; ?) K1 (0 ;0 ;0) N2 (0 ;10 ;0)
  K2 (0 ;10 ;-10) N3 (10 ;10 ;-10) K3 (-10 ;10 ;-10) → compte −= 10
- (0 ; ? ; ?) K1 (0 ; ? ;0) N1 (0 ;0 ;0) K2(0 ;0 ;-10)
  N2(0 ;10 ;-10) K3(-10 ;10 ;-10) N3(10 ;10 ;-10) → compte += 10

C'est une **condition de compétition** (*race condition*)

- Solution : opérations **atomiques** ; pas d'exécutions entremêlées
- Cette opération est une **section critique** à exécuter en **exclusion mutuelle**

# Réalisation d'une section critique

## Schéma général

Processus 1

```
...
entrée en section critique
    section critique
sortie de section critique
...
```

Processus 2

```
...
entrée en section critique
    section critique
sortie de section critique
...
```

- Exclusion mutuelle garantie par les opérations `(entrée en section critique)` et `(sortie de section critique)`

## Réalisation

- Attente active : processus à l'entrée section critique boucle un test d'entrée
  - **Inefficace** (sur mono-processeur)
  - Parfois utilisé dans conditions praticulière dans le noyau
- Primitives spéciales : fournies par le système
  - Primitives générales : sémaphores, mutex (on y revient)
  - Mécanismes spécifiques : comme verrouillage de fichiers (idem)
  - Les primitives doivent être atomiques...

# Notion d'interblocage

Utilisation simultanée de plusieurs verrous ⇒ problème potentiel

## Situation

- Deux processus verrouillent deux fichiers

**Processus 1**

```
...
verrouille (f1) /* 1A */
accès à f1
...
verrouille (f2) /* 1B */
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

**Processus 2**

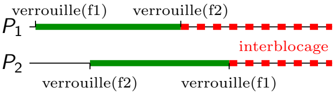
```
...
verrouille (f2) /* 2A */
accès à f2
...
verrouille (f1) /* 2B */
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

## Déroulement

Exécution (pseudo-)parallèle

- Première possibilité : 1a ; 1b ; 2a ; 2b
- Seconde possibilité : 2a ; 2b ; 1a ; 1b
- Troisième possibilité : 1a ; 2a ; 1b ; 2b

## Exécution de 1a ;2a ;1b ;2b




- P1 et P2 sont bloqués *ad vitam eternam* :
  - P1 attend le `deverrouille(f2)` de P2
  - P2 attend le `deverrouille(f1)` de P1
- C'est un **interblocage** (*deadlock*)

# Situation d'interblocage

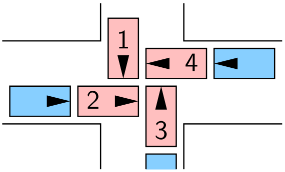
## Définition

- Plusieurs processus bloqués dans l'attente d'une action de l'un des autres
- Impossible de sortir d'un interblocage sans intervention extérieure

## Conditions d'apparitions

- Plusieurs processus en compétition pour les mêmes ressources
- Cycle dans la chaîne des attentes

## Exemple : carrefour lyonnais un vendredi à 18h




**Exercice : quelles sont les ressources ?**

*chaque quart du carrefour*

**Exercice : comment sortir de l'interblocage ?**

*impossible (sans bate de baseball)*

# Situation réelle d'interblocage

# Problèmes de synchronisation (résumé)

- Condition de **compétition** *(race condition)*
  - Définition : le résultat change avec l'ordre des instructions
  - Difficile à corriger car difficile à reproduire (ordre ≪aléatoire≫)
  - Également type de problème de sécurité :
    - Un programme crée un fichier temporaire, le remplit puis utilise le contenu
    - L'attaquant crée le fichier avant le programme pour contrôler le contenu
- **Interblocage** *(deadlock)*
  - Définition : un groupe de processus bloqués en attente mutuelle
  - Évitement parfois difficile (correction de l'algorithme)
  - Détection assez simple, mais pas de guérison sans perte
- **Famine** *(starvation)*
  - Définition : un processus attend indéfiniment une ressource pourtant libre
  - Servir équitablement les processus demandeurs

# Schémas de synchronisation

## Situations usuelles se retrouvant lors de coopérations inter-processus

- Exclusion mutuelle : ressource accessible par une seule entitée à la fois
  - Compte bancaire ; Carte son
- Problème de cohorte : ressource partagée par au plus N utilisateurs
  - Un parking souterrain peut accueillir 500 voitures (pas une de plus)
  - Un serveur doom peut accueillir 2000 joueurs
- Rendez-vous : des processus collaborant doivent s'attendre mutuellement
  - Roméo et Juliette ne peuvent se prendre la main que s'ils se rencontrent
  - Le GIGN doit entrer en même temps par le toit, la porte et la fenêtre
  - Processus devant échanger des informations entre les étapes de l'algorithme
- Producteurs/Consommateurs : un processus doit attendre la fin d'un autre
  - Une Formule 1 ne repart que quand tous les mécaniciens ont le bras levé
  - Réception de données sur le réseau *puis* traitement
- Lecteurs/Rédacteurs : notion d'accès exclusif entre *catégories* d'utilisateurs
  - Sur une section de voie unique, tous les trains doivent rouler dans le même sens
  - Un fichier pouvant être lu par plusieurs, si personne ne le modifie
  - Tâches de maintenance (défragmentation) quand pas de tâches interactives

Comment résoudre ces problèmes avec les sémaphores ?

# Modern Computers are Large and Complex

## Massive Parallelism

- Cannot miniaturize further (atom limit)
- Cannot increase frequency (energy limit)
- Solution: Multiply compute cores!
- Sequoia, third fastest computer: 1,572,864 cores

## ExaScale Systems, used in Computational Science

- Systems doing one Exaflop per second by the end of the decade
- 1 Exaflop = $10^{18}$ operations. One million million million operations...
  At humanly doable speed, that requires 10 times the age of the universe
- Each node: 20 millions lines of code ($10\times$ Encyclopedia Britannica)

## Other very large computer systems in the wide

- Google computers dissipate 300MW on average (150,000 households, $\frac{1}{3}$ reactor)
- Botnets: BredoLab estimated to control 30 millions of zombie computers
- In addition, these systems are heterogeneous and dynamic

So, how do we *study* these beasts?

# Computational Science of Computer Systems

## My Research Field: Methodologies of Experimentation

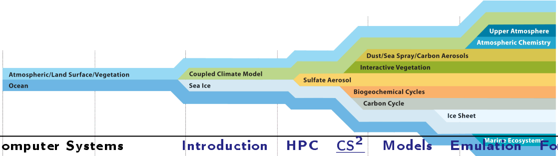
- Assessing the performance and correctness of large-scale computer systems
- Meta-research on producing scientifically sound results
- Main contribution: SimGrid, a large-scale computer systems simulator

## First title (rejected)

Simulating Applications for Research in
Simulation Applications for Research

## Epistemological Stance

- Empirically consider large-scale computer systems as natural objects
- Eminently artificial artifacts, but complexity reaches "natural" levels
- Other sciences routinely use computers to understand complex systems



# Computational Science of Computer Systems

## My Research Field: **Methodologies of Experimentation**

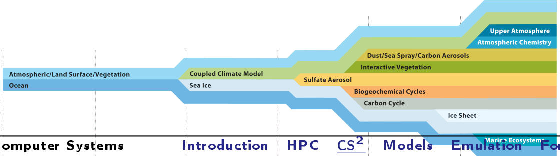
- Assessing the performance and correctness of **large-scale computer systems**
- Meta-research on **producing scientifically sound results**
- Main contribution: **SimGrid, a large-scale computer systems simulator**

## First title (rejected)

**Simulating Applications** for **Research** in
**Simulation Applications for Research**

## Epistemological Stance

- Empirically consider large-scale computer systems as natural objects
- Eminently artificial artifacts, but complexity reaches "natural" levels
- Other sciences routinely use computers to understand complex systems



# Assessing Distributed Applications

## Correctness Study $\rightsquigarrow$ Formal Methods

- Tests: Unable to provide definitive answers

## Performance Study $\rightsquigarrow$ Experimentation

- Maths: Often not sufficient to fully understand these systems

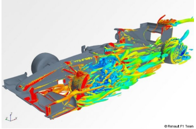
# Assessing Distributed Applications

## Correctness Study ⇝ Formal Methods

- Tests: Unable to provide definitive answers
- Model-Checking: Exhaustive and automated exploration of state space

## Performance Study ⇝ Experimentation

- Maths: Often not sufficient to fully understand these systems



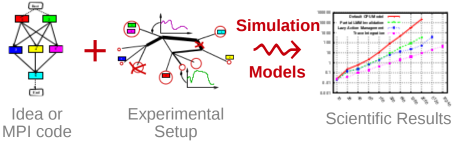
Courtesy of Lucas Nussbaum

- Experimental Facilities: Real applications on Real platform *(in vivo)*
- Emulation: Real applications on Synthetic platforms *(in vitro)*
- Simulation: Prototypes of applications on system's Models *(in silico)*

# Simulating Distributed Systems

## Simulation: fastest path from idea to data

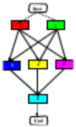
- Get preliminary results from partial implementations
- Experimental campaign with thousands of runs within the week
- Test your scientific idea, don't fiddle with technical subtleties (yet)




## Simulation: easiest way to study distributed applications

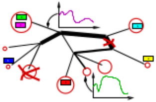
- Everything is actually centralized: Partially mock parts of your protocol
- No heisenbug: (Simulated) time does not change when you capture more data
- Clairevoyance: Observe every bits of your application and platform
- High Reproducibility: No or very few variability
- Capacity planning: Can we save on component? *What if* network were faster
- Don't waste resources to debug and test (up to 50% on some production infra)

# Simulation Challenges



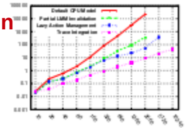
Idea or
MPI code

+

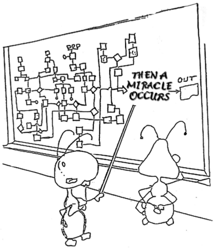


Experimental
Setup

**Simulation**
⇝
**Models**



Scientific Results




## Challenges for the Tool Makers

- Validity: Get realistic results (controlled experimental bias). That's hard.
- Scalability: *Fast enough* and *Big enough*
- Tooling: runner, post-processing, integrated lab notes

## Major Components of any Simulation-based Experiment

- An observation of your application: either a trace or the live application
- Models of your platform: CPU, network, any other relevant resource
- A configuration describing the experimental settings

# SimGrid: Versatile Simulator of Distributed Apps

## Scientific Instrument

- Versatile: Grid, P2P, HPC, Volunteer Computing and others
- Sound: Validated, Scalable, Usable; Modular; Portable
- Community-driven: 30 contributors (5 not affiliated), 5 contributed tools, GPL

## Scientific Object

- Allows comparison of network models on non-trivial applications
- High-Performance Simulation on realistic workload
- Full model checker of distributed applications; Emulator under way

## Large Established Project

- Impact: 120 publications (110 distinct authors, 5 continents), 4 PhD
- Started in 1998 at UCSD; Now collab accross many individuals and institutions
- 7 partners, 20+ researchers (CNRS, Universities, Inria)
- Public funding (≈3M€ ANR/Inria); Community based (User days, hackfests)

# Simulation Validity

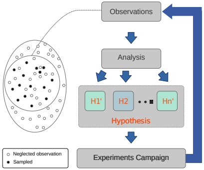
SotA: Models in most simulators are either simplistic, wrong or not assessed

- PeerSim: discrete time, application as automaton;
- GridSim/CloudSim: naive packet level or buggy flow sharing
- OptorSim, GroudSim: documented as wrong on heterogeneous platforms
- Dimemas: aim at performance trends and bottleneck identification

# Simulation Validity

SotA: Models in most simulators are either simplistic, wrong or not assessed

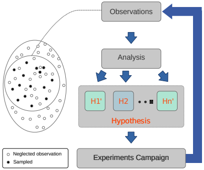
- PeerSim: discrete time, application as automaton;
- GridSim/CloudSim: naive packet level or buggy flow sharing
- OptorSim, GroudSim: documented as wrong on heterogeneous platforms
- Dimemas: aim at performance trends and bottleneck identification
- SimGrid
  - 10-years effort on validity
  - Same methodology than physicist: we try to (in)validat our models
  - Observe, analyze, hypothesis

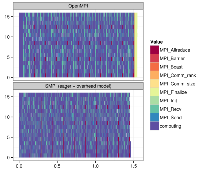


# Simulation Validity

SotA: Models in most simulators are either simplistic, wrong or not assessed

- PeerSim: discrete time, application as automaton;
- GridSim/CloudSim: naive packet level or buggy flow sharing
- OptorSim, GroudSim: documented as wrong on heterogeneous platforms
- Dimemas: aim at performance trends and bottleneck identification
- SimGrid
  - 10-years effort on validity
  - Same methodology than physicist: we try to (in)validat our models
  - Observe, analyze, hypothesis




## We need to combine

- Usage model: Predict ending time of each task in isolation
  - On Network, both one hop models, and multi-hops paths
- Contention model: predicts how tasks interfere with each others
  - On Network, needs to take topology (and routing) into account
- Applicative model: Complex operations (eg, MPI global communications)
- For both CPU and Network, and disks if possible

# Fine-grain Simulation of CPU

## Many Cycle-accurate Models and Simulators exist

- We could simulate entirely each core, each node, each site, etc.
- Most resources are modeled separately: cores, buses, networks, disks
- Popular belief: more details means more accurate simulation

**we could combine these tools together!**

## Microscopic Modeling (only) is not an option

- Immensely slow: x1000 slowdown when host machine ≈ studied system
  - So folding a larger system into a smaller host machine is impossible
  - This approach is sensible, for other scientific workflows
- More details actually bring more chaos and less insight
  - Complex models are hard to instantiate and fragile (Flash project)
  - Phase effects: clean simulations lead to resonance effects [Floyd 91]
  - *A wealth of information creates a poverty of attention* [Simon 71]
- Mixing *macro* and *micro* models sounds appealing but difficult
  - As done in SST project and also by the BSC group

# Simplistic CPU Model

## How it works

- Computation load measured in Flops; CPU's power measured in Flops/s
- Timing is obtained by simply dividing one by the other
- Basically, this is just like reinjecting timing.

## What is this Model Good for?

- Allows to see what you would get with a CPU twice faster
- Almost every projects does this (SimGrid, Dimemas, ...)

## Known Limits

- Hardware extrapolation to other kind of CPUs, w/ cache contention
  - Dimemas can adjust per SEB; PSINS extrapolates from hardware counters
  - SST mixes Micro (cycle accurate) and Macro models to that extend
- Multicore memory contention (could hack something but haphazard)
- Scalability extrapolation: what would happen with more nodes
  - BigSim can model the SEB perf as a polynomial of #processes
  - PSINS tries to fit a model from the SEB's parameters

# Elaborate Analytic CPU Model

## The Promise

- Get a bunch of hardware-level counters while benchmarking the SEBs
- Automatically build portable performance models out of it

## The (many) Challenges

- Relating the hardware counters you see to the actual timing you get
- You need a performance model taking the counters as an input
  - PSINS has the convolver for that, but hard to get and understand it
  - Our preliminary results: encouraging for some kernels, deceiving for others
- How to obtain the hardware counters?
  - Measurements? SimGrid/Dimemas use PAPI on real runs (hard to extrapolate)
  - Cache simulation? PSINS goes this way
  - Code analysis? Maqao does it
- How generic and portable will the models be?
  - Things are very different e.g. on ARM

# More on CPU modeling

## Upcoming complexity is somehow depressing

- Multicores, Implicit Mem Accesses, OpenMP, Memory/ PCI contention
- Modern processors overclock themselves when only one core is used
- GPU, SOC systems, dedicated accelerators

## Don't seek for a complete model

- KISS is better, and the advantage of more complex CPU models is unclear
- At least in the use-cases that we target (at our scale)
- We are not competing with cycle-accurate simulators
- So simply refine your simple models, only when the need is blatant

*Essentially, all models are wrong, but some are useful.* – G. Box

## Much more insight can be injected into the Network Models

- Things are very complex too, but maybe less integrated by vendors
- We can work at the level of standard protocols (TCP, InfiniBand)

# Agenda

- Introduction
- Modern Large Computing Facilities
- Computational Science of Computer Systems ($CS^2$)
- Simulation Models
  - CPU models
  - Modeling Communications
  - MPI Operations
- Emulation with SMPI
- Dynamic Verification of Distributed Applications
- Conclusion

## Components of a good model

- Point to point communications: latency, protocol switch
- Topology: shared memory $\neq$ remote, latency penalty for remote cabinets
- Contention

# Fine Grain Network Simulation

## Packet-level simulators run the full protocol stack

- Hopefully perfect, since "everything's taken into account"
- But complex models $\rightsquigarrow$ hard to instantiate and unstable

Flores Lucio, Paredes-Farrera, Jammeh, Fleury, Reed. *Opnet modeler and ns-2: Comparing the accuracy of network simulators for packet-level analysis using a network testbed.* WSEAS Transactions on Computers 2, no. 3 (2003)

- Inherently slow, and parallelism won't save you here!
  BigSim proved that distribution is for size (memory) issues, but sequential is faster
- Sometimes wrongly implemented
- Not really helping to understand the macroscopic behavior

## Same bias and drawbacks than cycle-accurate CPU simulation

- Perfectly fitted to study TCP variants or wireless algorithms
- Very bad choice to study MPI algorithms (IMHO)

# Modeling Point to Point Networks

## Basic Model: $Time = L + \frac{size}{B}$

- Resource work at given *rate* (B, in Mb/s); Uses have a given latency (L, in s)
- Very similar to the basic CPU model (simply adds latency)
- This somehow works for Multi-Hops Networks

## Better Model of TCP Multi-Hops Networks

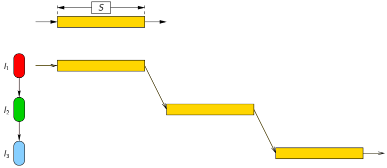
- Several models proposed in Networking Literature, such as [Krusoe 2000]

$$B = min\left(\frac{W_{max}}{RTT}, \frac{1}{RTT\sqrt{2bp/3} + T_0 \times min(1, 3\sqrt{3bp/8}) \times p(1+32p^2)}\right)$$

  - $T_0$: retransmission timeout; RTT: round-trip t; $W_{max}$ max window size
  - p: loss rate; b: #packages acknowledged per ACK (hard to instanciate)
- Keep It Instanciable, Silly: use $\beta' = min(\beta, \frac{W_{max}}{RTT})$ *(TCP windowing)*

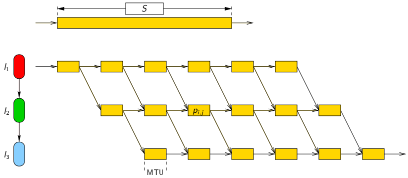
# Taking the Network Topology into Account

## Store & Forward




- Sounds Natural:
  cf. time to go from city to city
- But Plainly Wrong:
  Data not stored on routers

## Wormhole



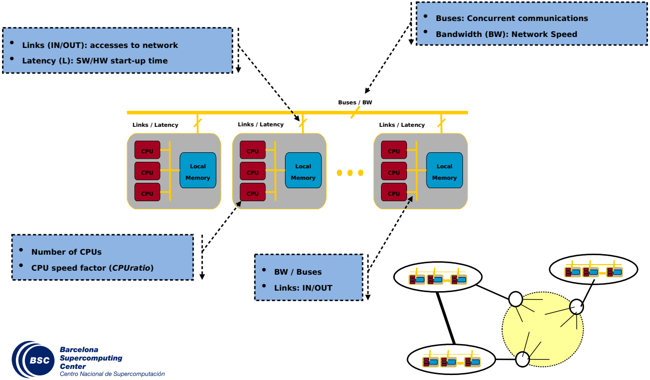

- Appealing: (& widely used ☹)
  Remember networking class?
- Really inaccurate:
  TCP congestion, etc

What's in between these two approaches?

## Packet-level Simulators

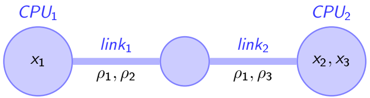
- ☺: Realism commonly accepted; ☹: Sloooooow
- No usable models of HPC networks in generic tools (NS2/3)

# Exclusive Resource Usage




- In Dimemas, resources are allocated exclusively with more than one token
- Nicely models buses' backplane: up to N flows get through, others do wait
- Then a delay-model computes the time of each communication
- Applied at each models (memory, networks), with no overlap between both
- Similar mechanism in BigFastSim (?)

# Analytic Network Models




$$x_1 \leqslant Power\_CPU_1 \quad (1a)$$
$$x_2 + x_3 \leqslant Power\_CPU_2 \quad (1b)$$
$$\rho_1 + \rho_2 \leqslant Power\_link_1 \quad (1c)$$
$$\rho_1 + \rho_3 \leqslant Power\_link_2 \quad (1d)$$

## Computing the sharing between flows

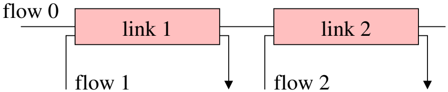
- Objective function: maximize $\min_{f \in \mathcal{F}}(\rho_f)$ [Massoulié & Roberts 2003]
- Equilibrium: increasing any $\rho_f$ decreases a $\rho'_f$ (with $\rho_f > \rho'_f$)
- (actually, that's a simplification of SimGrid's real objective function)

## Efficient Algorithm

1. Search for the bottleneck link $l$ so that: $\frac{C_l}{n_l} = min\left\{\frac{C_k}{n_k}, k \in \mathcal{L}\right\}$
2. This determines any flow $f$ on this link: $\rho_f = \frac{C_l}{n_l}$
3. Update all $n_l$ and $C_l$ to remove these flows; Loop until all $\rho_f$ are fixed

# Max-Min Fairness

## Homogeneous Linear Network



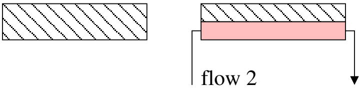

$C_1 = C \qquad n_1 = 2$
$C_2 = C \qquad n_2 = 2$

$\rho_0 =$
$\rho_1 =$
$\rho_2 =$

- All links have the same capacity $C$
- Each of them is limiting. Let's choose link 1.

# Max-Min Fairness

## Homogeneous Linear Network



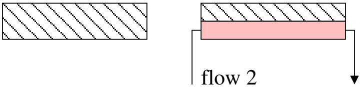

$C_1 = 0 \qquad n_1 = 0$
$C_2 = C/2 \qquad n_2 = 1$

$\rho_0 = C/2$
$\rho_1 = C/2$
$\rho_2 =$

- All links have the same capacity $C$
- Each of them is limiting. Let's choose link 1.
- This sets $\rho_0$ and $\rho_1$. Remove flows 0 and 1; Update links' capacity and uses

# Max-Min Fairness

## Homogeneous Linear Network




$C_1 = 0 \qquad n_1 = 0$
$C_2 = 0 \qquad n_2 = 0$

$$\rho_0 = C/2$$
$$\rho_1 = C/2$$
$$\rho_2 = C/2$$

- All links have the same capacity $C$
- Each of them is limiting. Let's choose link 1.
- This sets $\rho_0$ and $\rho_1$. Remove flows 0 and 1; Update links' capacity and uses
- Link 2 sets $\rho_1 = C/2$.
- We are done computing the bandwidths $\rho_i$

## SimGrid Implementation is **efficient**

- Dedicated LMM solver with Lazy updates, Trace integration, and Cache locality

# Flow-level Models Facts

Several sharing methods are possible, many have been evaluated in SimGrid

## Pros

- rather flexible (add linear limiters whenever you need one)
- account for network topology
- account for many non-trivial phenomena
  e.g., RTT-unfairness of TCP and even reverse-traffic interference to some extent

## Cons

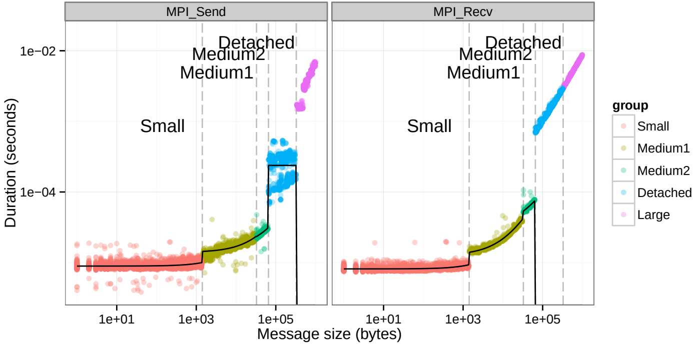
- ignores protocol oscillations, TCP slow start
- ignores all transient phases
- does not model well very unstable situations
- does not model computation/communication overlap

## Conclusion

- Common belief: this cannot scale, so often ruled out
- Yet, when correctly implemented and optimized, it's a strong alternative
- Captures contention if TCP is in steady state (when $size > 1Mb$)

# MPI Point-to-Point Communication on Ethernet

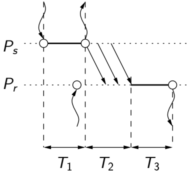
Randomized measurements (OpenMPI/TCP/Eth1GB) since we are not interested in peak performance but in performance characterization



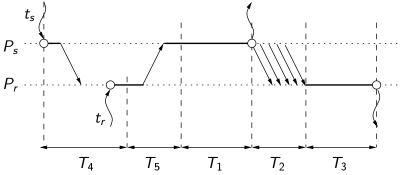

- There is a quite important variability
- There are at least 4 different modes, each is piece-wise linear and discontinuous

# LogGPS in a Nutshell

- LogP model initially designed for complexity analysis and algorithm design
- Many variations account for protocol switch through continuous linear functions




Asynchronous mode ($k \leqslant S$)




Rendez-vous mode ($k > S$)

$$T_1 = o + kO_s \qquad T_4 = \max(L+o, t_r - t_s) + o$$

$$T_2 = \begin{cases} L + kg & \text{if } k < \boxed{s} \\ L + sg + (k-s)G & \text{otherwise} \end{cases}$$

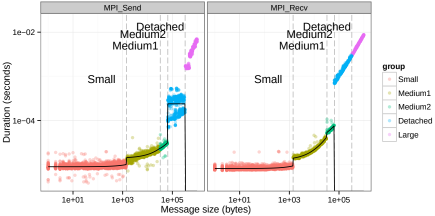
$$T_3 = o + kO_r \qquad T_5 = 2o + L$$

| Routine | Condition | Cost |
|---|---|---|
| MPI_Send | $k \leqslant S$ | $T_1$ |
| | $k > S$ | $T_4 + T_5 + T_1$ |
| MPI_Recv | $k \leqslant S$ | $\max(T_1 + T_2 - (t_r - t_s), 0) + T_3$ |
| | $k > S$ | $\max(o + L - (t_r - t_s), 0) + o + T_5 + T_1 + T_2 + T_3$ |
| MPI_Isend | | $o$ |
| MPI_Irecv | | $o$ |

- May reflect the operation of specialized HPC networks from the early 1990s. . .
- Ignores many factors: contention, topology, complex protocol stack, . . .
- So? What's the best? Fluid or LogP? None! They are complementary!

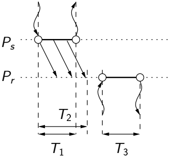
# SimGrid Network Model

## Measurements

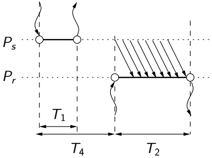



## Hybrid Model

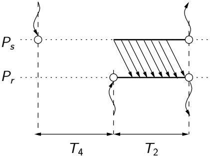
Asynchronous ($k \leqslant S_a$)



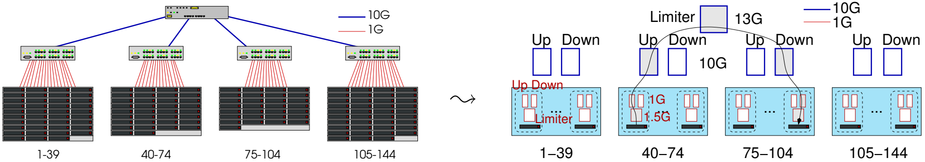

Detached ($S_a < k \leqslant S_d$)




Synchronous ($k > S_d$)



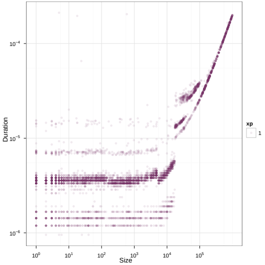

## Fluid model: account for contention and network topology

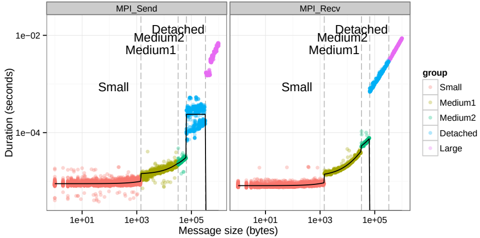


# MPI Point-to-Point Communication on IB

## IB have supposedly simpler and more predictable performance

- It should be clean and stable, with less intelligence in the protocol
- Indeed, it's faster and cleaner than TCP, but *IB is not that different*




## Surprisingly, Modeling InfiniBand is complex wrt Bandwidth Sharing!

- Strictly fair share of IB buffers (in and out)
- Preliminary feelings: bandwidth is not fairly shared, but handling time is
- Counter-intuitive results, but results got confirmed (+ we have a candidate model)

# Conclusion on Network Modeling

## Analytic Models are possible

- TCP: Algorithmic model for synchronization + Equation-based for sharing
- IB: Still ongoing but encouraging (even with strange sharing)

## Models are Getting Complex (but that's ok)

> *For today's complex simulations [from Computational Sciences], the computer program* **is** *the model. Questions such as* Does program X correctly implement model A?, *a question that made perfect sense in the 1960s, have become meaningless.* — Konrad Hinsen

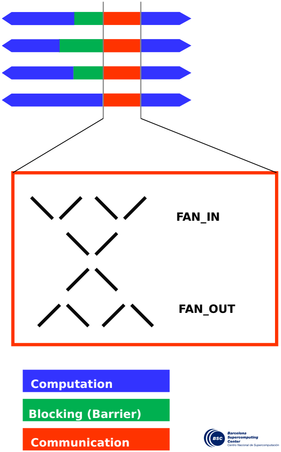
## The runtime also induce protocol switches

- e.g. Eager mode vs. Rendez-vous mode
- Presented (SimGrid) Results are somehow specific to MPI
- MPI collective operations absolutely have to be modeled too

# Analytic Collective Models (1/2)

## Dimemas' Simple Models

- Regular and similar Algorithms:
  - Some Fan In, a middle operation, and some Fan Out
- To model a given collective algorithm, you specify
  - Amount of Fan In/Out and cost of each tree level
  - Cost of the middle operation
- Example of Scatter/Gather:
  $\left\lceil \frac{\log N}{\log \text{fan}_{\text{in}}} \right\rceil \times (\text{latency} + \frac{\text{size}}{\text{bw}}) + \left\lceil \frac{\log N}{\log \text{fan}_{\text{out}}} \right\rceil \times (\text{latency} + \frac{\text{size}}{\text{bw}})$
- Cost of All2All: (no FAN in/out but similar)
  $N(N-1) \times (\text{latency} + \frac{\text{size}}{\text{bw}})$
- Add a barrier before to nicely fit to the picture



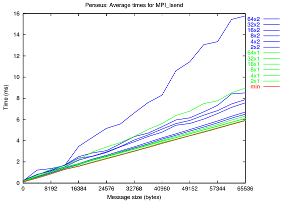
# Analytic Collective Models (2/2)

## Cons of Dimemas' Collective Models

- Models are simplistic compared to algorithms' sophistication, barrier is artificial
- Topology not taken into account, Contention through bus' tokens

## Approach of [Grove, Coddington 2003]

- Don't model performance, benchmark and replay it
- On given cluster, benchmark every communicator size
- Also benchmark communicator geometries
- This gives the self-interference of collectives
- Could be extended to interference between collectives




## Pros of Dimemas' Collective Models

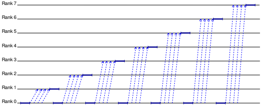
- You can easily extrapolate to other network characteristics and topology
- Easy to instanciate on a given platform

# Collective Communications Through Trace Replay

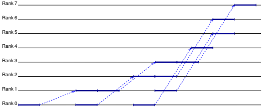
## Improving the realism while enabling extrapolation

- Decompose any collective into a set of point-to-point comms
- Tracing is not trivial, as staying at PMPI level is not enough
- LogGOPSim: collectives are rewritten in a DSL called GOAL
- BigSim: traces are collected in Charm++, underneath

```
rank 0 {
   l1: calc 100 cpu 0
   l2: send 10b to 1 tag 0 cpu 0 nic 0
   l3: recv 10b from 1 tag 0 cpu 0 nic 0
   l2 requires l1
}
rank 1 {...
```




**Linear Broadcast/Scatter Pattern.**

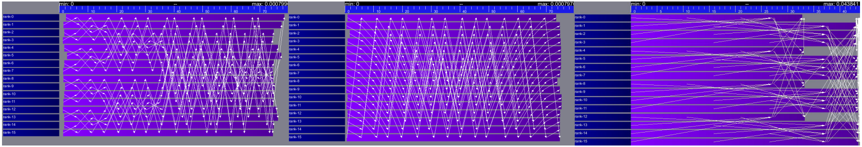



**Binomial Tree Pattern.**

# Collectives' Code Scavenging

## SimGrid's Approach

- SimGrid implements more than 120 algorithms for the 10 main MPI collectives



## This code was ... *integrated* (OpenMPI, MPICH, and StarMPI)

- Selection logic from OpenMPI, MPICH can be reproduced

## Future Work

- Expand this selection logic and autotuning possibilities to allow better selection
- See how all of this behaves on Multicore systems, with SMP-aware algorithms
- Implement MVAPICH2 Selector
- (In)validation on real platforms, with Infiniband, torus networks

# Agenda

- Introduction
- Modern Large Computing Facilities
- Computational Science of Computer Systems ($CS^2$)
- Simulation Models
  - CPU models
  - Modeling Communications
  - MPI Operations
- Emulation with SMPI
- Dynamic Verification of Distributed Applications
- Conclusion

# What is SMPI?

- Reimplementation of MPI on top of SimGrid
- Imagine a VM running real MPI applications on platforms that does not exist
  - Horrible over-simplification, but you get the idea
- Computations run for real on your laptop, Communications are faked

## What is it good for?

- Performance Prediction ("what-if?" scenarios)
  - Platform dimensioning; Apps' parameter tuning
- Teaching parallel programming and HPC
  - Reduced technical burden
  - No need for real hardware, or hack your hardware

## Studies that you should NOT attempt with SMPI

- Predict the impact of L2 caches' size on your code
- Interactions of TCP Reno vs. TCP Vegas vs. UDP
- Claiming a simulation of 1000 billions nodes

# Features and Limitations

## Features

- Complex C/C++/F77/F90 applications can run unmodified out of the box
  - MPI ranks folded as threads in an unique UNIX process
  - Global variables automatically privatized
- Traces from various projects can be used offline
- Basic but sound coarse-grain CPU models (with multicores)
- Extensively tested on Linux, Mac and Windows

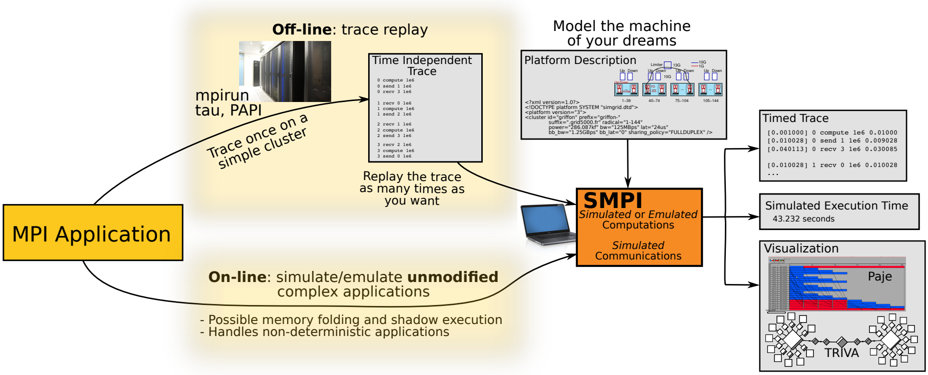
## Limitations

- Partial MPI API coverage: $\approx$ 100 primitives supported (more to come on need)
  - No MPI-IO, no one-sided, MPI3 collectives, spawning ranks, ...
  - Still passes many of MPICH3 standard complience tests
- Non-multithreaded applications, neither pthread nor OpenMP

## Success Story

- Accurate Ethernet (soon IB) models, accurate collectives, mid-range apps
- Misprediction of BigDFT on Tibidabo turned out to be a hardware issue

# Observing the Application




## Offline Simulation

- Obtain a trace of your application
- Replay quickly and easily that trace
- Hard to extrapolate, adaptative apps?

## Online Simulation

- Directly run your application
- Technically very challenging
- No limit (but the resources)

Most existing tools go for offline simulation

# Challenges in Observing Applications

Offline: Many contributions in the literature

- Reduce intrusiveness while capturing the traces to avoid heisenbugs
- Compact the traces (that can grow very quickly)
- Extrapolate the trace to new conditions

Online: HPC codes are resource hungry

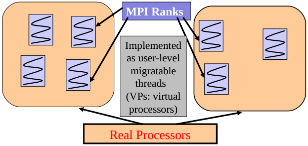
- It does not fit easily on single laptop or node
- Sometimes, host machine must be larger than studied machine
- Some tricks allow to cheat here
  - Memory folding to allocate once, and share between processes
  - Kernel sampling to reduce execution time

# Challenges in Observing Applications Online (2/2)

## Folding the application is difficult

- Global variables of distributed processes hard to fold into thread locals
  - Manual modification: works but burdensome
  - Source-to-Source: turn globals into arrays of locals
  - Compiler's pass: move globals into TLS area
    changes toolchain (no `icc`) $\rightsquigarrow$ alters SEBs (as any previous solution)
  - GOT injection: rewrite the ELF symbol table when switching contextes
    static variables are not part of the GOT unfortunately
  - `mmap` of `.data` and `.bss`: preserves SEBs but forces sequential exec
  - Run real processes, MPI interactions turned into external `mmap`. Perf?

## Architecture (in AMPI)




## Approaches implemented

- AMPI: Source-to-source with Photran
  GOT injection; Compiler's pass for TLS
- SMPI: source-to-source (coccinelle, f2c)
  Recently implemented mmaping
- Full processes not implemented yet (?)

# Agenda

- Introduction
- Modern Large Computing Facilities
- Computational Science of Computer Systems ($CS^2$)
- Simulation Models
  - CPU models
  - Modeling Communications
  - MPI Operations
- Emulation with SMPI
- Dynamic Verification of Distributed Applications
- Conclusion

# Assessing the Correctness of HPC codes?

Writing Distributed Apps is notoriously difficult, but:

## The Good Old Days

- MPI codes circumvented the difficulty with rigid communication patterns
- Correctness established through testing
- Only performance matters anyway:
  - Most prefer a fast code that rarely fail-stop to a slow code that always work
  - (at least, that's my feeling for most of the numerical applications)

## These Days are Now Over

- But rigid patterns do not scale! We now have to release the grip
- But this is dangerous! We now have to explicitly seek for correctness

## Slowly, old ignored problems resurface...

# Model Checking and Dynamic Verification

## These are Automated Formal Methods

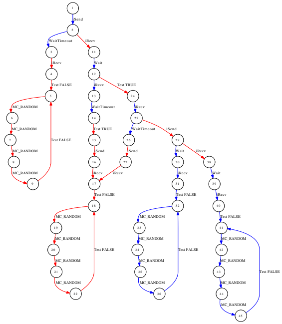
- Try to assess the correctness of a system by actively searching for faults
- If you find a fault, then you have something to work on
- If don't find any after an exhaustive search, correctness experimentally proved
- Dynamic Verification: Model Checking applied to real applications

# Model Checking and Dynamic Verification

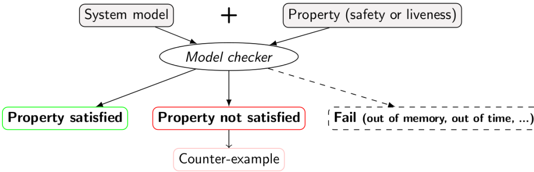
## These are Automated Formal Methods

- Try to assess the correctness of a system by actively searching for faults
- If you find a fault, then you have something to work on
- If don't find any after an exhaustive search, correctness experimentally proved
- Dynamic Verification: Model Checking applied to real applications

## Exhaustive Exploration



## Model Checking: the Big Idea




- My preferred outcome: a counter-example
  If not, I fear my property to be wrongly expressed
- We tend to bug finding, not certification

# Formal Properties

## Safety Properties

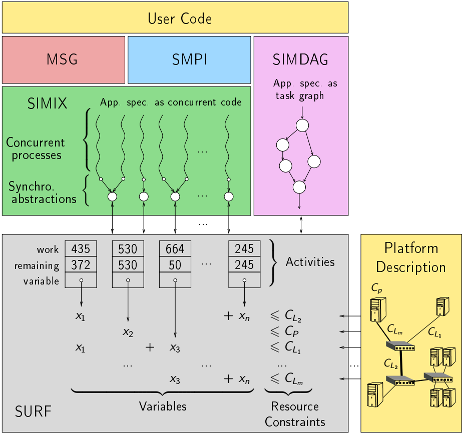
- "A given bad behavior never occurs"
- Can be expressed as boolean (assertion): no deadlock, $x \neq 0$, ...
- Work on all states separately
- Counter example: a faulty state

## Liveness Properties

- "An expected behavior will happen in all cases"
- Example: Any process that asks a resource will obtain it eventually
- Must be expressed in a temporal logic such as CTL (safety ones *could* too)
- Work on execution path
- Counter example: an infinite path (ie, a cycle) that violates the property

Liveness properties are much more challenging to verify in practice

# SimGrid and SMPI

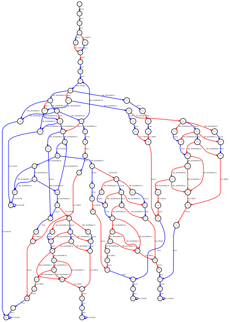


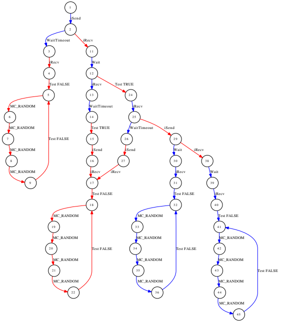

- SMPI can run complex C/C++/Fortran applications on top of SimGrid
- Let's leverage this unconventional virtualization layer for verification!
- + collective code scavenging $\rightsquigarrow$ verify even runtime's collectives

# SimGridMC: Formal Methods in SimGrid

## Verify any application that would run in SimGrid

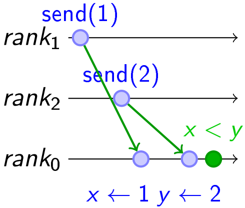
- Replace the simulation kernel underneath with a model checker
- Tests all causally possible orders of events to dynamically verify the app
- Reuse the mediation mechanism that base the simulator
- System-level checkpoints the app to then rewind and explore another path
- Works with SMPI, and MSG (our simple API for the study of CSP algorithms)



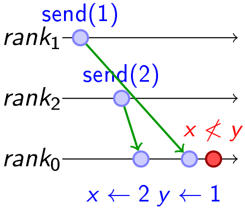



# Example: Out of order receive

- Two processes send a message to a third one
- The receiver expects the message to be in order
- This may happen. . . or not




```
if (MPI_rank() == 0) {
  MPI_Recv(&x , MPI_ANY_SOURCE);
  MPI_Recv(&y , MPI_ANY_SOURCE);
  MC_assert(x < y);
} else {
  MPI_Send (&rank , 0);
}
```




```
*************************
*** PROPERTY NOT VALID ***
*************************
Counter-example execution trace:
[(1)recver] iRecv (dst=recver, buff=(verbose only), size=(verbose only))
[(3)sender] iSend (src=sender, buff=(verbose only), size=(verbose only))
[(1)recver] Wait (comm=(verbose only) [(3)sender -> (1)recver])
[(1)recver] iRecv (dst=recver, buff=(verbose only), size=(verbose only))
[(2)sender] iSend (src=sender, buff=(verbose only), size=(verbose only))
[(1)recver] Wait (comm=(verbose only) [(2)sender -> (1)recver])
```

# Mitigating the State Space Explosion

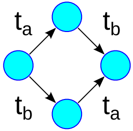
Many execution paths are redundant ⇝ cut exploration when possible

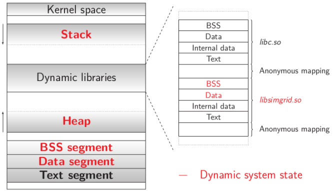
## Dynamic Partial Ordering Reduction (DPOR)

- Works on histories: test only one transitions' interleaving if independent
- *Independence theorems:* Local events are independent; iSend+iRecv also; ...
- Must be conservative (exploration soundness at risk!)
- It works well (for safety properties)

## System-Level State Equality

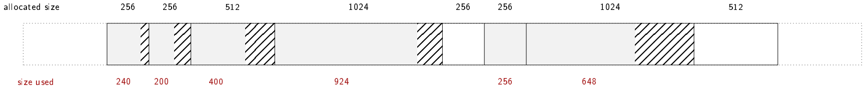
- Works on states: detect when a given space was previously explored
- Complementary to DPOR (but not compatible yet)
- Introspect the C/C++/Fortran app just like gdb (+some black magic)



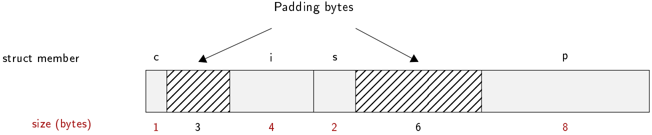
# OS-level Challenges of State Equality Detection

- Memory over-provisioning

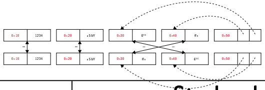



- Padding bytes: Data structure alignment

```
struct foo {
char c;
int i;
short s;
void *p;
}
```




- Irrelevant differences: system-level PID, fd, . . .
- Syntactic differences / semantic equalities:



## Solutions

| Issue | Heap solution | Stack solution |
|---|---|---|
| Overprovisioning | `memset 0` (customized `mmalloc`) | Stack pointer detection |
| Padding bytes | `memset 0` (customized `mmalloc`) | DWARF + libunwind |
| Irrelevant differences | Ignore explicit areas | DWARF + libunwind + ignore |
| Syntactic differences | Heuristic for semantic comparison | N/A (sequential access) |

# Some Results

## Wild safety bug in our Chord implementation (≈ 500 lines of C)

- Simulation: bug on large instances only; MC finds small trace (1s with DPOR)

## Mocked liveness bug

- Buggy centralized mutual exclusion: last client never obtains the CS
- About 100 lines – state snapshot size: 5Mib
- Verified with up to 7 processes (12,000 states, 9 minutes, 45Gb).

## Verifying MPICH3 complience tests

- Looking for assertion failures, deadlocks and non-progressive cycles
- 6 tests; ≈ 1300 LOCs (per test) – State snapshot size: ≈ 4MB
- With no reduction: no test concluded in a few hours
- With state equality: Exhaustive exploration up to 10 procs, but no error found
- With memory compaction: use only dozen of Gb in RAM, not hundreds
- We verified several MPI2 collectives too ☺ (but all good so far ☹)

# Verification of Protocol-wide Properties

## Motivation

- Clever checkpoint algorithms exist, provided that the application is nice enough
- *On communication determinism in parallel HPC applications*,
  F. Cappello, A. Guermouche and M. Snir (2010)
  - Manual inspection of 27 HPC applications, seeking for such properties

## Protocol-wide properties

- deterministic: On each node, send and receive events are always in same order
- send deterministic: ∀ node, send are always the same, no matter the recv order
- Not liveness, not even LTL: quantifies for all execution paths within property

## Status report: we can verify such properties in SimGrid

- Explore one path to learn the communication order, deduce the property
- Enforce that this order holds on all other execution path
- We reproduced the conclusions of previous paper on several benchmarks
  - All good ☹

# More on Formal Verification

## We've built a really cool tool

- We can verify many unmodified MPI applications (C/C++/Fortran)
- State space reduction: DPOR or State equality (not together yet)
- Properties: safety, liveness or protocol-wide

## Many remaining Research Leads

- Other reductions, HPC-specific properties, statistical model-checking, ...
- Interactive tool to get gdb-like info on each state in the execution graph

## We need more use cases

- We are done with all the ones provided by the practitioners we know
- We could make it even better with really relevant use cases
- We don't know what properties are relevant

# Agenda

- Introduction
- Modern Large Computing Facilities
- Computational Science of Computer Systems ($CS^2$)
- Simulation Models
  - CPU models
  - Modeling Communications
  - MPI Operations
- Emulation with SMPI
- Dynamic Verification of Distributed Applications
- Conclusion

# Take Away Messages

## Modern Computer Systems

- Tremendous societal impact: science, industry, R&D – 1% of energy world wide
- Energy is the new challenge (in addition to time performance)
- Large, Complex, Hierarchical, Heterogeneous, Dynamic $\leadsto$ challenging to study

## Experimental Methodologies

- Hard to have both Correctness and Performance in a given framework
- Simulation promising (and widely used), but models are hard
- Fine-grained models not better than coarse, hybrid, ad-hoc models.
- Simulation nicely combines with Dynamic Verification

## Some SimGrids' Success Stories

- Simulate many MPI applications out of the box (+faster than sota, +in parallel)
- Misprediction of BigDFT on Tibidabo turned out to be a hardware issue (!)
- Automated verification of assertions, liveness and protocol-wide properties
- SimGrid: many different aspects (models, systems, HPC, formal); Community

# Take Away Messages

## SimGrid will prove helpful to your research

- Versatile: Used in several communities (scheduling, GridRPC, HPC, P2P, Clouds)
- Accurate: Model limits known thanks to validation studies
- Sound: Easy to use, extensible, fast to execute, scalable to death, well tested
- Open: User-community much larger than contributors group; AGPL
- Around since over 10 years, and ready for at least 10 more years

**Welcome to the Age of (Sound) Computational Science**

- Discover: `http://simgrid.gforge.inria.fr/`
- Learn: 101 tutorials, user manuals and examples
- Join: user mailing list, #simgrid on irc.debian.org
  We even have some open positions ;)

`apt-get install simgrid` now! (or get the jarfile)