LINUX & Parallel Processing

Wouter Brissinck

Vrije Universiteit Brussel

Dienst INFO

wouter@info.vub.ac.be

+32 2 629 2965

Overview

- The Dream
- The Facts
- The Tools
- The Real Thing
- The Conclusion

History

- ‘60 **à** 1 computer N users
- ‘80 **à** N computers N users
- ‘90 **à** N computers 1 user

- past **à** problem = f(computer)
- future **à** computer = f(problem)

The Sequential Frustration

b Faster CPU : High development cost + time

b Speed of light : faster à smaller

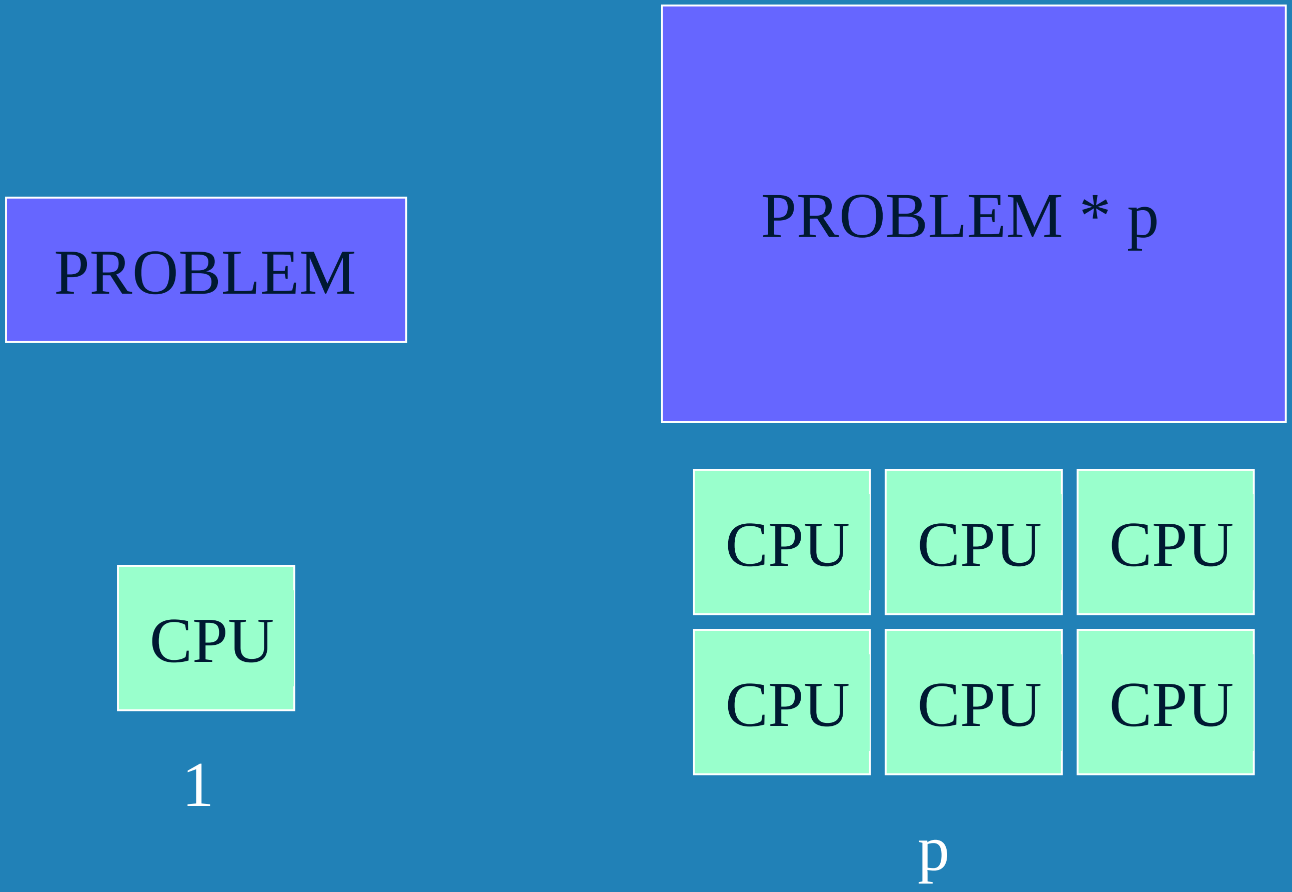
b Size of Si atom is the ultimate limit

Parallelism

- Offers scalable performance
- Solves as yet unsolvable problems
  - problem size
  - computation time

- Hardware cost
- Software development

Scalability



PROBLEM

CPU

1

PROBLEM * p

CPU CPU CPU

CPU CPU CPU

p

Scalability

- problem on 1 CPU takes T time
- p times larger problem on p CPUs takes T time

- Same technology (only more of it)
- Same code (if ...)
- Economy of scale

Hardware cost ?

- Parallel machines :
  - expensive to buy
  - expensive to maintain
  - expensive to upgrade
  - unreliable
  - lousy operating systems
- N.O.W :
  - you own one !
  - Its maintained for you
  - You just did, didn’t you ?
  - You rely on it every day !
  - How did you call LINUX ?

A network-of-workstations is a virtually free parallel machine !

Overview

- The Dream
- The Facts
- The Tools
- The Real Thing
- The Conclusion

Problems

b 1 painter à 100 days

b 100 painters à 1 day ?

b Synchronization
  - shared resources
  - dependencies

b Communication

Amdahl's Law

$$T_1 = aT + (1 - a)T$$

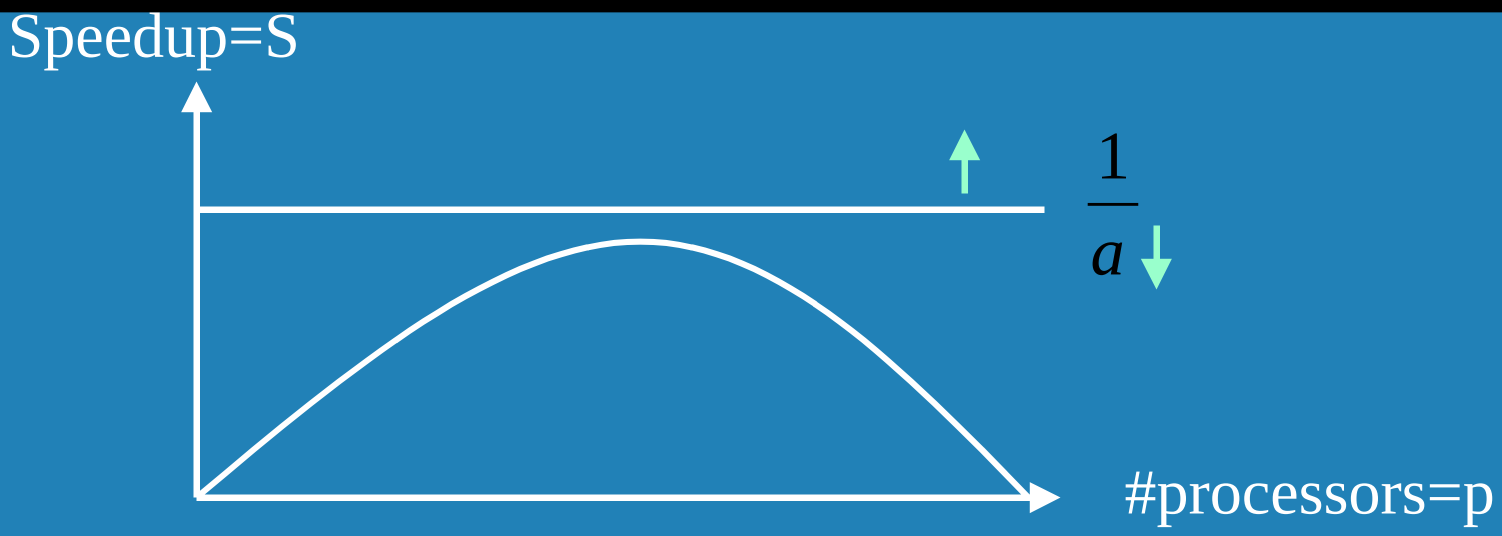
$$T_p = aT + \frac{(1-a)T}{p} + bpT$$

α = sequential fraction

β = communication cost

$$Speedup = \frac{T_1}{T_p} = \frac{1}{\frac{1-a}{p} + a + bp}$$

Amdahl’s Law



$$Speedup = S = \frac{1}{\frac{1-a}{p} + a + bp}$$

α = sequential fraction
β = communication cost

Granularity

b Granularity = $\dfrac{\mathit{Computation}}{\mathit{Communication}}$


N.O.W

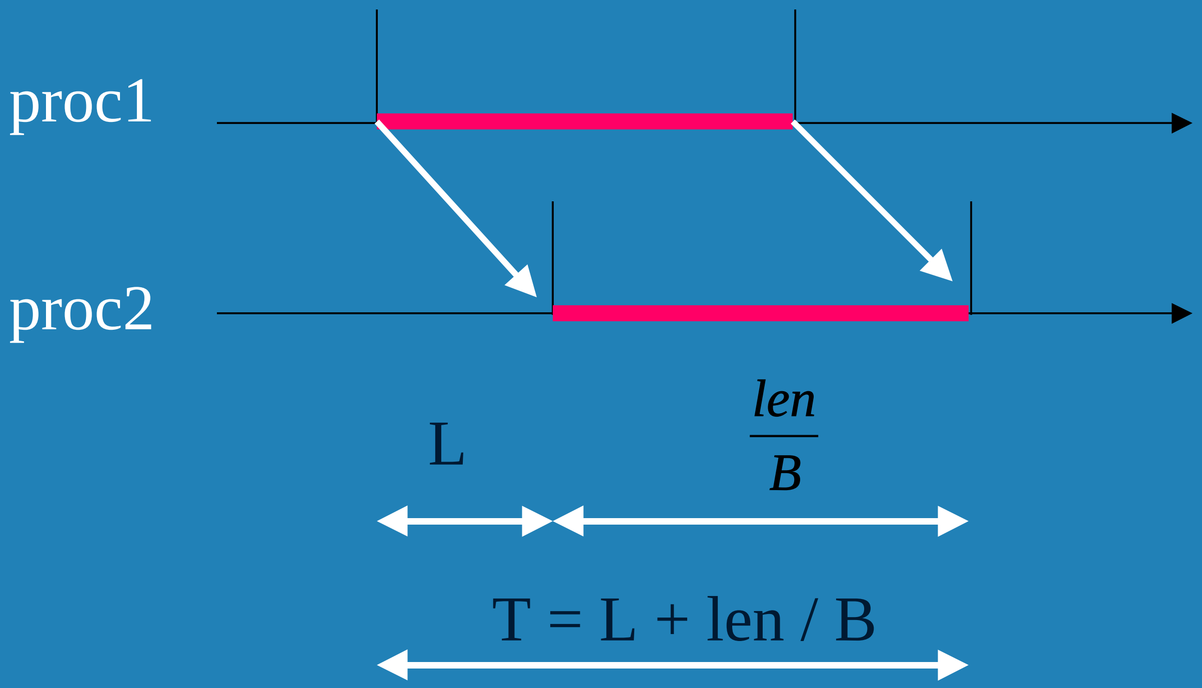
Dedicated MIMD

Pipelining in CPU

coarse

fine

Network latency

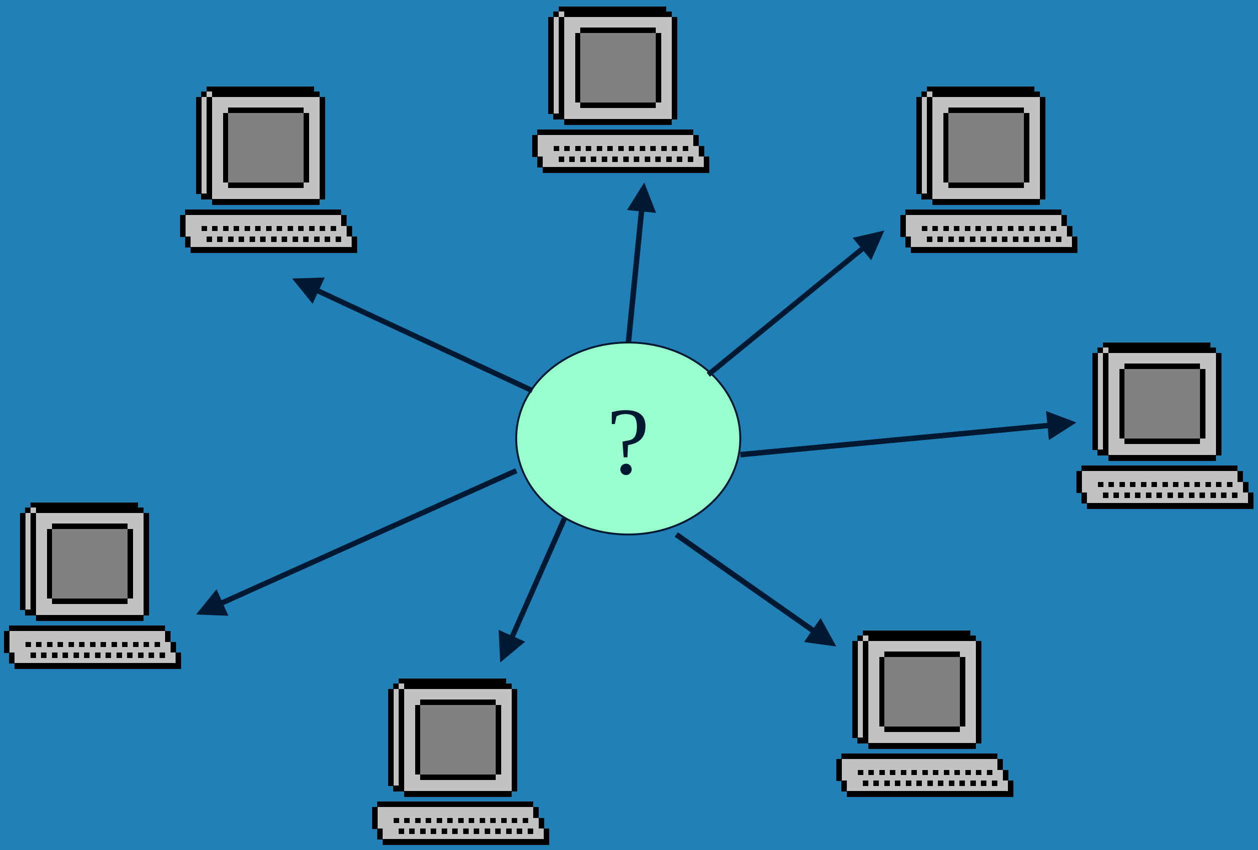


$\frac{len}{B}$

T = L + len / B

L=Latency(s) len=length(bits) B=Bandwidth(bits/s)

The N of N.O.W



Software Design

b Sequential specification à Parallel implementation

b Parallel specification à (less) parallel implementation

- Preferably : 1 process/processor

Software Design : classification

- Master - Slave
- Pipelining
- Systolic Array

- Functional parallelism
- Data parallelism

Overview

- The Dream
- The Facts
- The Tools
- The Real Thing
- The Conclusion

P.V.M

- PVM : overview
- PVM installation
- Using PVM
  - console
  - compiling
  - writing code
- Message passing
  - in C
  - in C++
- gdb and pvm

Parallel Virtual Machine

- Most popular Message Passing Engine
- Works on MANY platforms (LINUX, NT, SUN, CRAY, ...)
- Supports C, C++, fortran
- Allows heterogenious virtual machines
- FREE software (ftp.netlib.org)

Heterogenious NOWs

- PVM provides data format translation of messages
- Must compile application for each kind of machine
- Can take advantage of specific resources in the LAN

PVM features

- User-configured host pool
- Application = set of (unix) processes
  - automatically mapped
  - forced by programmer
- Message Passing model
  - strongly typed messages
  - asynchronous (blocking/non-blocking receive)
- Multiprocessor support

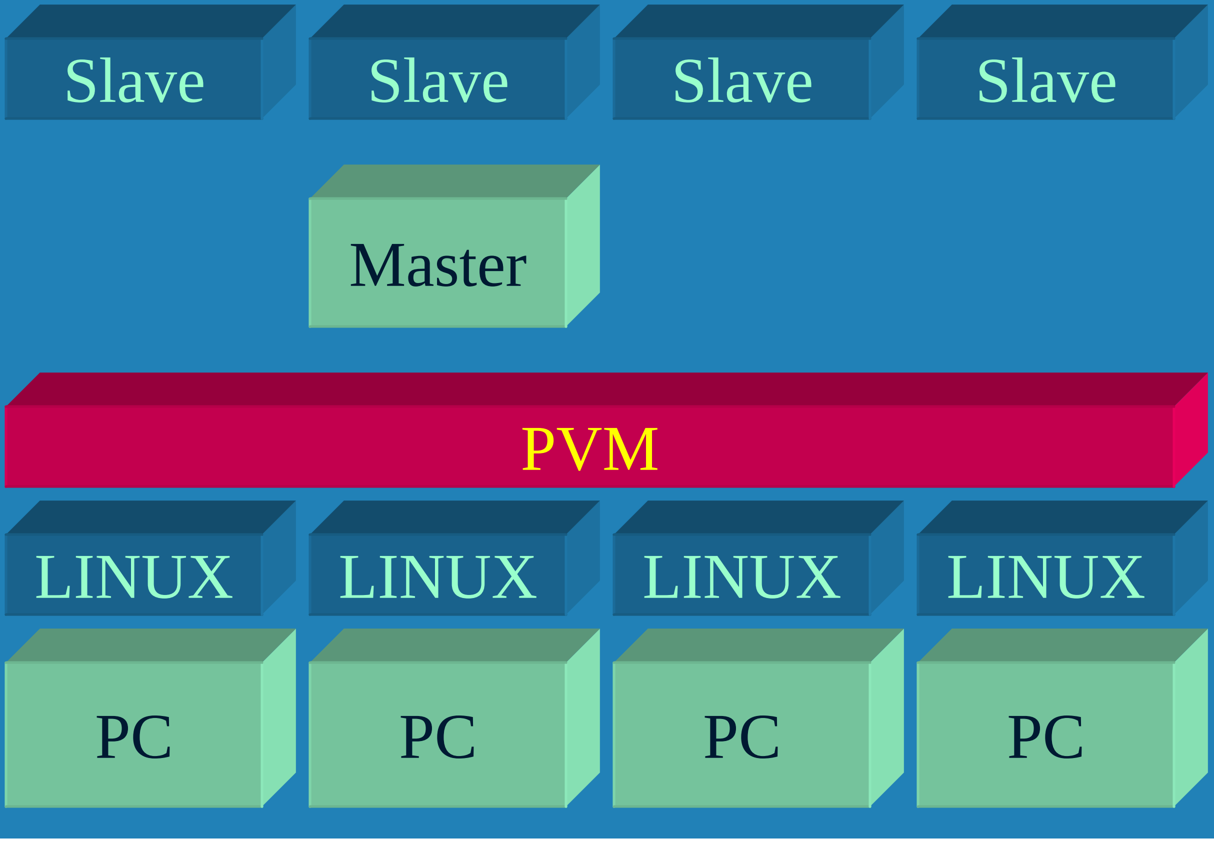
How PVM works

- Console for resource management
- Deamon for each host in the virtual machine
  - message passing
  - fault tolerance
- PVM library provides primitives for
  - task spawning
  - send/receive
  - VM info, ...

How PVM works

- Tasks are identified by TIDs (pvmd supplied)
- Group server provides for groups of tasks
  - Tasks can join and leave group at will
  - Groups can overlap
  - Useful for multicasts

N.O.W Setup



Slave Slave Slave Slave

Master

PVM

LINUX LINUX LINUX LINUX

PC PC PC PC

Example : master

```
#include "pvm3.h”
main(){
        int cc, tid;char buf[100];
        cc = pvm_spawn("hello_other",
                (char**)0, 0, "", 1, &tid);
        if (cc == 1) {
                cc = pvm_recv(-1, -1);
                pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
                pvm_upkstr(buf);
                printf("from t%x: %s\n", tid, buf);
        }
        pvm_exit();
}
```

Example : Slave

```
#include "pvm3.h"
main()
{
        int ptid;char buf[100];
        ptid = pvm_parent();
        strcpy(buf, "hello, world from ");
        gethostname(buf + strlen(buf), 64);

        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid, 1);

        pvm_exit();
}
```

PVM : installing

- Set environment variables (before build)
  - PVM_ROOT
  - PVM_ARCH
- Make default
- Make sure you can rsh without authentication

PVM and rsh

- PVM starts tasks using rsh
- Fails if a password is required
- .rhosts on remote account must mention the machine where the console is running
- Typical error : Can’t start pvmd

Typical situation

- All machines share the same user accounts
- Account has these directories/files :

```
/shared/pvm3/lib/LINUX/pvmd3
/shared/pvm3/lib/SUN4/pvmd3
~/pvm3/bin/LINUX/Eg.App
~/pvm3/bin/SUN4/Eg.App
```

- .rhosts file looks like this : (mymachine is the machine which runs the console)

```
mymachine
mymachine.my.domain
```

PVM Console

- add hostname : add a host to VM
- conf : show list of hosts
- ps -a : show all pvm processes running
  - Manually started (i.e. not spawned) tasks are shown by a ‘-’
- reset : kill all pvm tasks
- quit : leave console, but keep deamon alive
- halt : kill deamon

PVM Console : hostfile

- hostfile contains hosts to be added, with options

```
mymachine ep=$(SIM_ROOT)/bin/$(PVM_ARCH)
faraway lo=woutie pw=whoknows
* lo=wouter
```

- On machine without network (just loopback):

```
* ip=127.0.0.1
```

Compiling a program

- Application must be recompiled for each architecture
  - binary goes in `~/pvm3/bin/$(PVM_ARCH)`
  - aimk determines architecture, then calls ‘make’
- Must be linked with -lpvm3 and -lgpvm3
- Libraries in `$(PVM_ROOT)/lib/$(PVM_ARCH)`

The PVM library

- Enrolling and TIDs

```
int tid=pvm_mytid()

int tid=pvm_parent()

int info=pvm_exit()
```

The PVM library

- Spawning a task

```
int numt=pvm_spawn(char *task,
     char **argv, int flag, char* where,
     int ntask, int *tids)
```

- flag = PvmTaskDefault
  PvmTaskHost (‘where’ is host)
  PvmTaskDebug (use gdb to spawn)

- typical error : tids[i] = -7 : executable not found

The PVM library

- Message sending :
  - initialize a buffer (initsend)
  - pack the datastructure
  - send (send or mcast)
- Message receiving :
  - blocking/non-blocking receive
  - unpack (the way you packed)

The PVM library

- Use pvm_initsend() to initiate send

```
int bufid = pvm_initsend(int encoding)
```

- Use one of the several pack functions to pack

```
int info=pvm_pkint(int *np,int nitem, int
  stride)
```

- To send

```
int info=pvm_send( int tid, int msgtag)
int info=pvm_mcast(int *tids, int ntask, int
  msgtag)
```

The PVM library

- pvm_recv waits till a message with a given source-tid and tag arrives (-1 is wildcard) :

```
int bufid=pvm_recv (int tid, int msgtag)
```

- pvm_nrecv returns if no appropriate message has arrived

```
int bufid=pvm_nrecv (int tid, int msgtag)
```

- Use one of the several unpack functions to unpack (exactly the way you packed)

```
int info=pvm_upkint(int *np,int nitem, int
  stride)
```

Message passing : an example

```
struct picture
{
     int sizex,sizey;
     char* name;
     colourMap* cols;
};
```

Message passing : an example

```
void pkPicture(picture* p)
{
     pvm_pkint(&p->sizex,1,1);
     pvm_pkint(&p->sizey,1,1);
     pvm_pkstr(p->name);
     pkColourMap(p->cols);
};
```

Message passing : an example

```
picture* upkPicture()
{
     picture* p=new picture;
     pvm_upkint(&p->sizex,1,1);
     pvm_upkint(&p->sizey,1,1);
     p->name=new char[20];
     pvm_upkstr(p->name);
     p->cols=upkColourMap();
     return p;
};
```

Message passing : an example

```
//I’m A
picture myPic;
...
pvm_initsend(PvmDataDefault);
pkPicture(&myPic);
pvm_send(B,PICTAG);
```

Message passing : an example

```
//I’m B
picture* myPic;

pvm_recv(A,PICTAG);
myPic=upkPicture();
```

In C++ ?

- Messages contain data, not code
- No cute way to send and receive objects
- Provide constructors that build from receive buffer
- Receiving end must have a list of possible objects

PVM and GDB

- Spawn tasks with PvmTaskDebug
- Pvmd will call

`$PVM_ROOT/lib/debugger taskname`

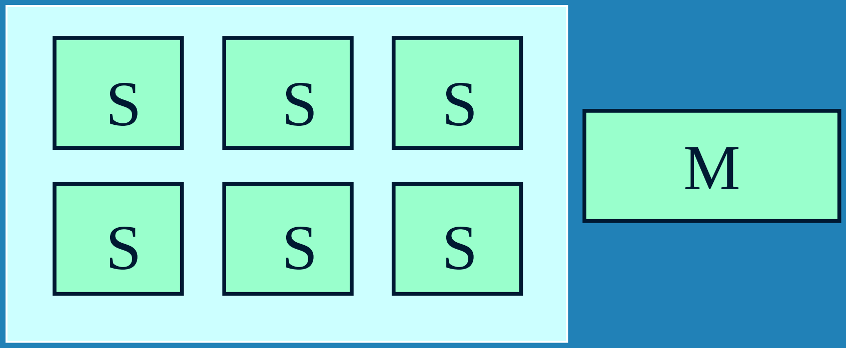
- Make sure this scripts starts an xterm with a debugger running ‘taskname’
- GREAT debugging tool !!

Overview

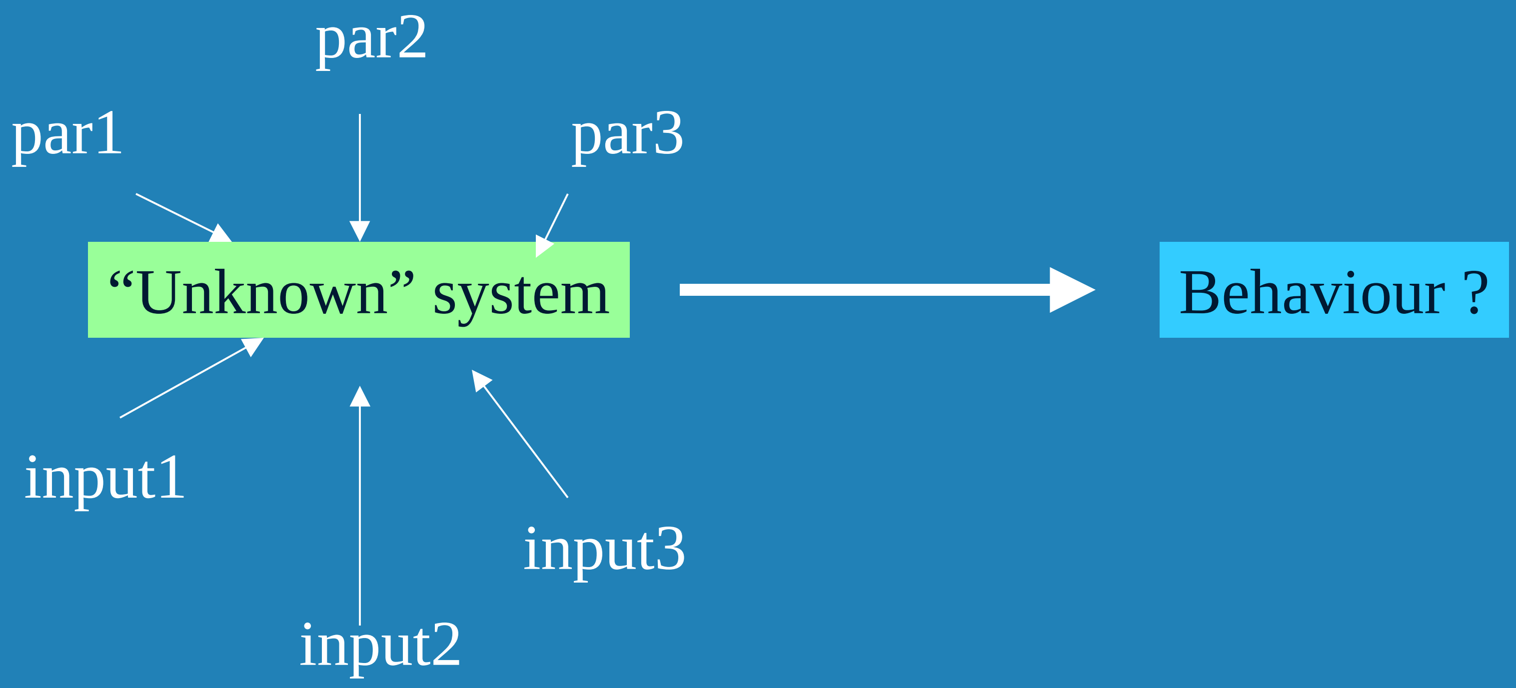
b The Dream
b The Facts
b The Tools
b The Real Thing
b The Conclusion

Ray-Tracing

- Ray-Tracing
  - master slave
  - Data Parallel



Simulation : the Problem



par2

par1

par3

“Unknown” system

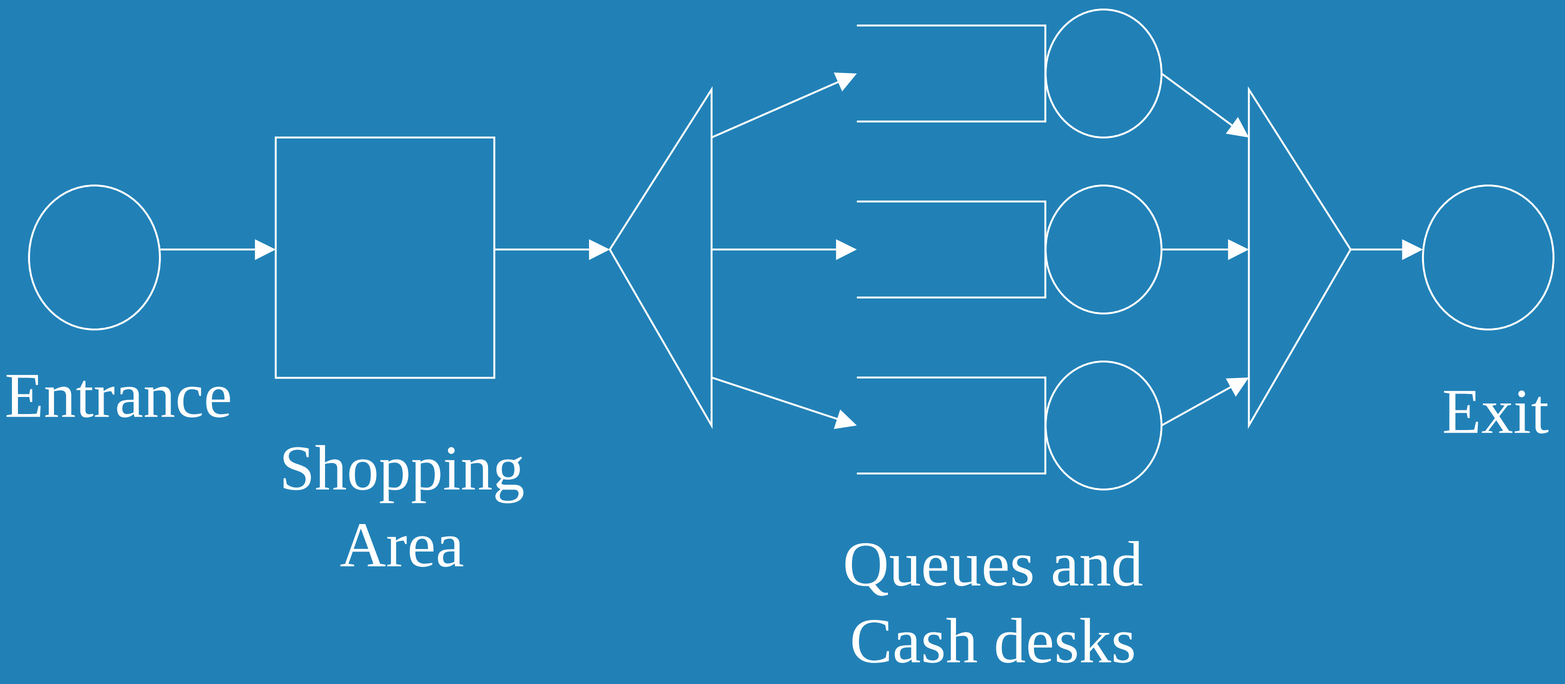
Behaviour ?

input1

input3

input2

Modeling : example

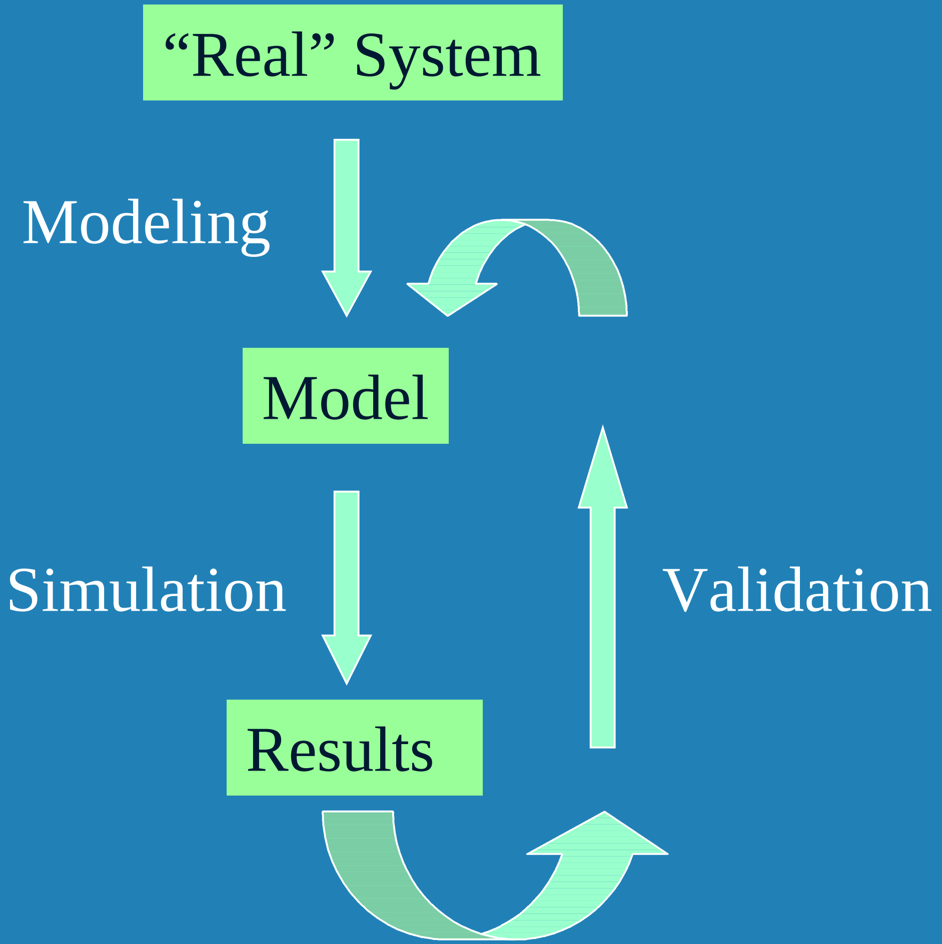


How many queues to achieve average queue length < 4 ?

Modeling : views

- Modeling viewpoints :
  - event view
  - process view
- Model classification
  - Logic gates
  - Queueing nets
  - Petri nets
  - ....

Simulation : Design Cycle



“Real” System

Modeling

Model

Simulation

Results

Validation

Simulation : Time Investment

- Modeling & Validation

  =

  Human Thinking

- Simulation

  =

  Machine Thinking

Simulation : Time Investment (2)

- Simple Model :
  - Short Simulation Time
  - Large Modeling Time
  - Large Validation Time

→ Solution ???

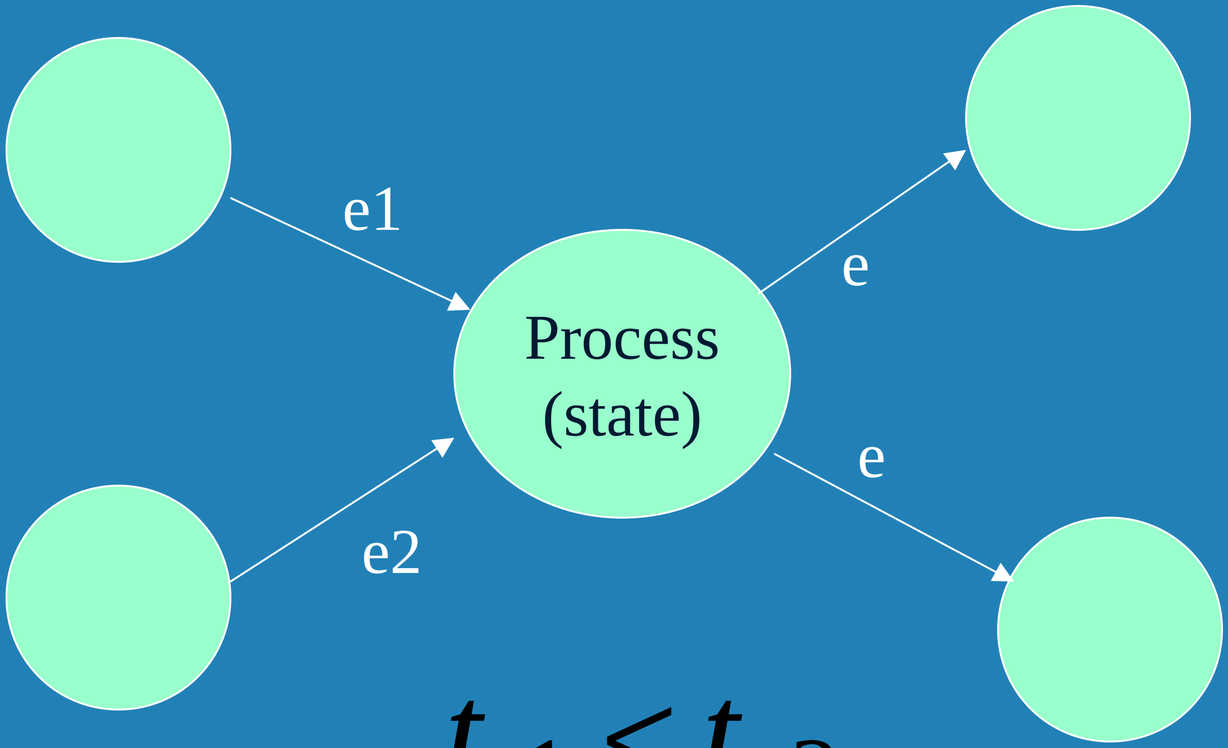
- Complex Model :
  - Large Simulation Time → Solution !!
  - Short Modeling Time
  - Short Validation Time

Parallel Simulators

- Two Methods :
  - Multiple-Run :
    - same simulation, different parameters/input stimuli
    - easy (stupid but effective)
  - “event-parallelism” :
    - distribute model over processors
    - hard : fine granular

Modeling : causality

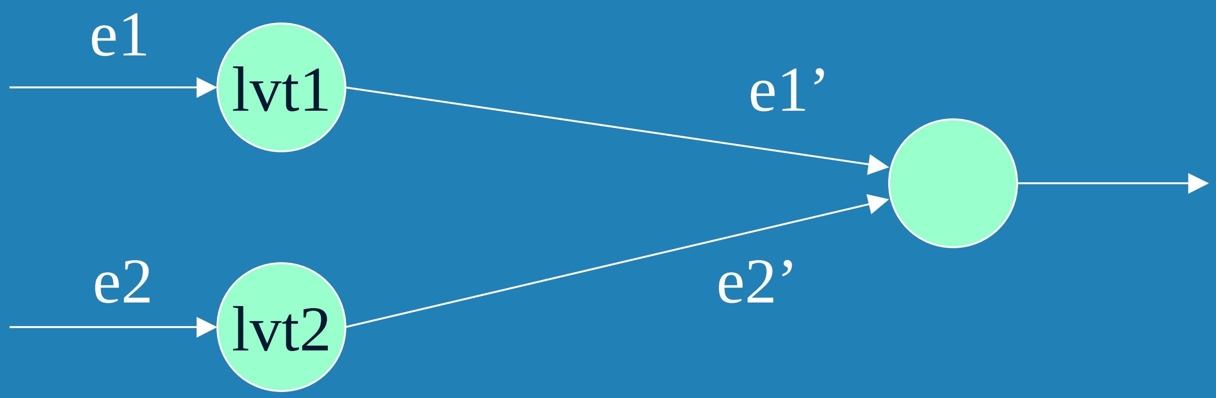
→ Global Event Ordering



$$t_{e1} < t_{e2}$$

Parallel Simulators : event parallelism

- Idea : simulate different events in parallel
- Condition : causal independence !
- Local Time ≠ Global Time



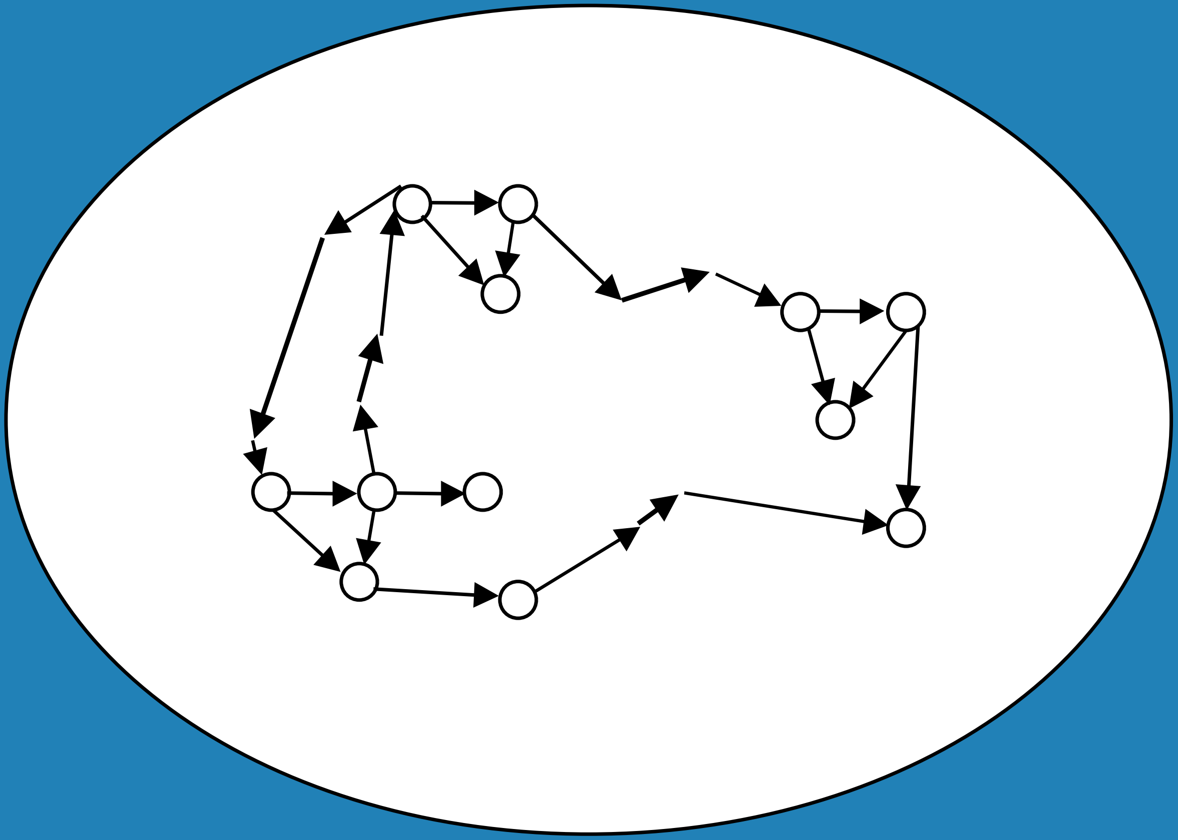
Parallel Simulators : event parallelism

- Very fine grain
- Strong synchronization (=limited parallelism)
- Strongly model dependent performance
- Hard to implement

→ Software TOOL

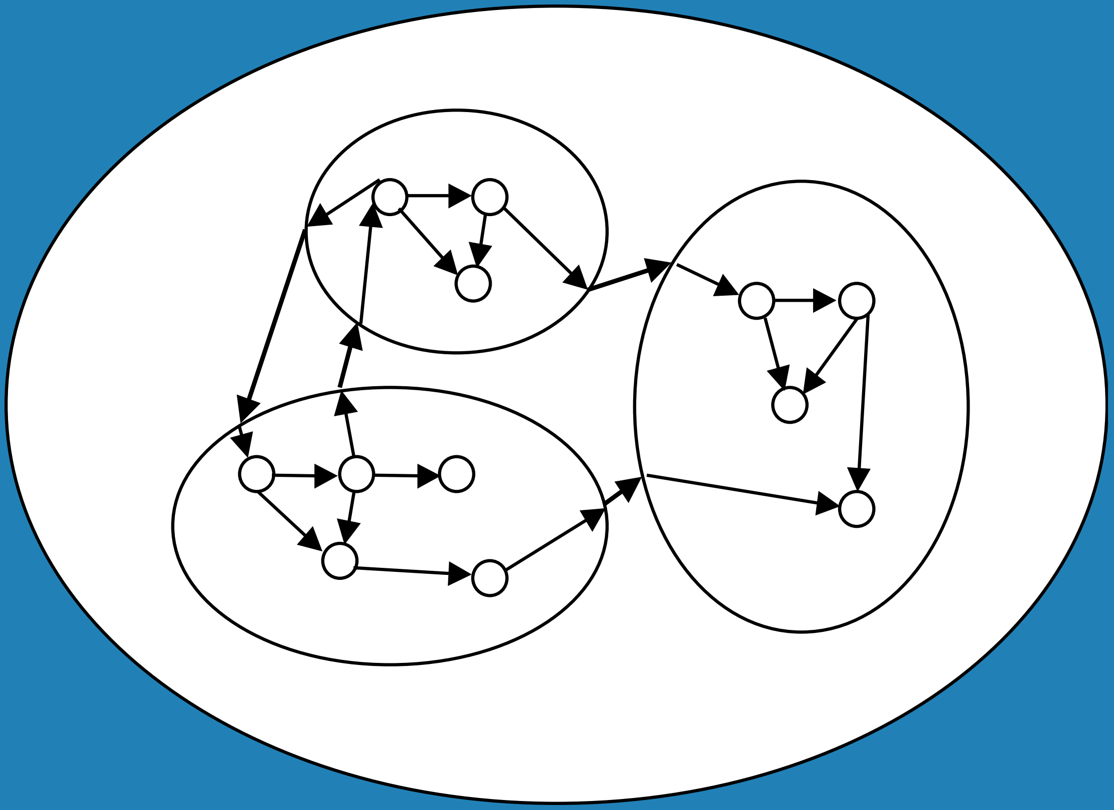
Parallel Simulators : Event Parallelism

Fine grain specification :



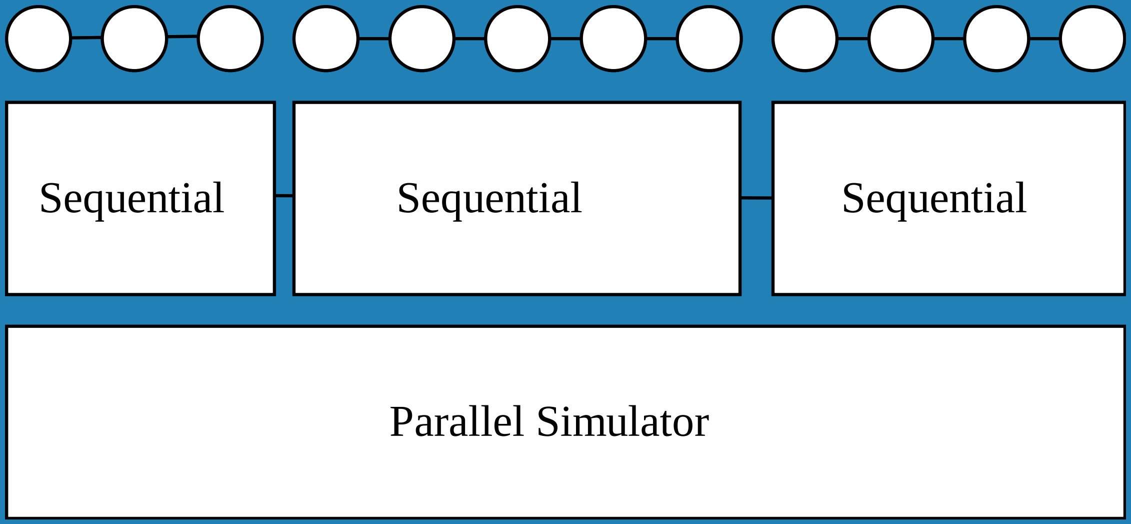
Parallel Simulators : Event Parallelism

Coarse grain implementation : (top view)

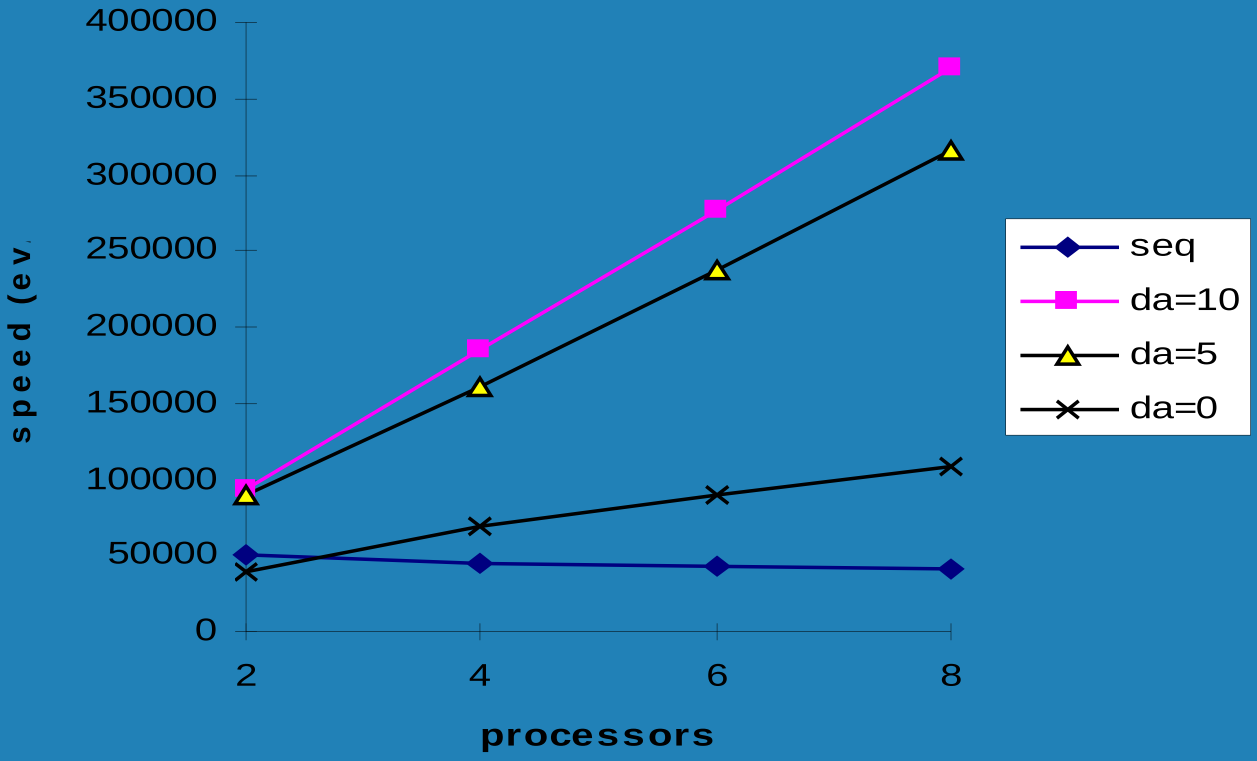


Parallel Simulators : Event Parallelism

Coarse grain implementation : (side view)



Parallel Simulators : some results



Overview

- The Dream
- The Facts
- The Tools
- The Real Thing
- The Conclusion

Conclusion

- NOW + LINUX + PVM = Scalability for free !
- Drawbacks :
  - Software is hard to implement
  - Performance is sensitive to
    - problem
    - machine
    - implementation
- Solution : software tools

References

- See our web-page at

  http://infoweb.vub.ac.be/~parallel