Algorithmique et Techniques de Base du Calcul Parallèle

Jean-Louis.Roch@imag.fr

Denis.Trystram@imag.fr

Thierry.Gautier@imag.fr

Bruno.Raffin@imag.fr

Projet CNRS-INRIA APACHE

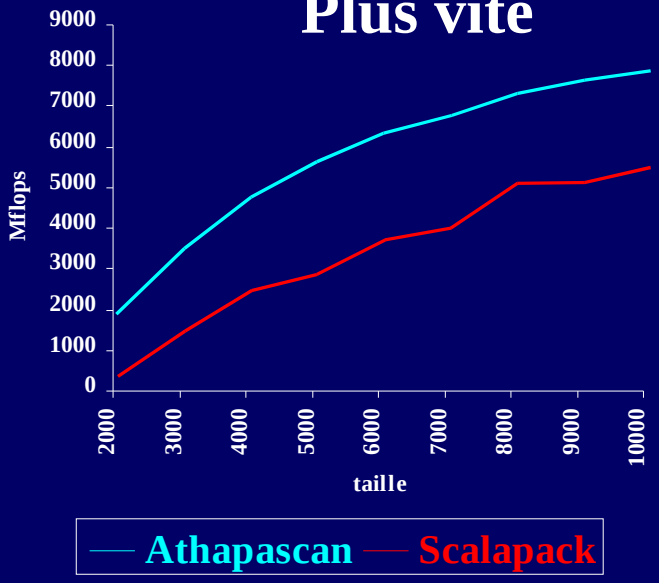
Laboratoire ID-IMAG

Grenoble - France

Objectif du calcul parallèle

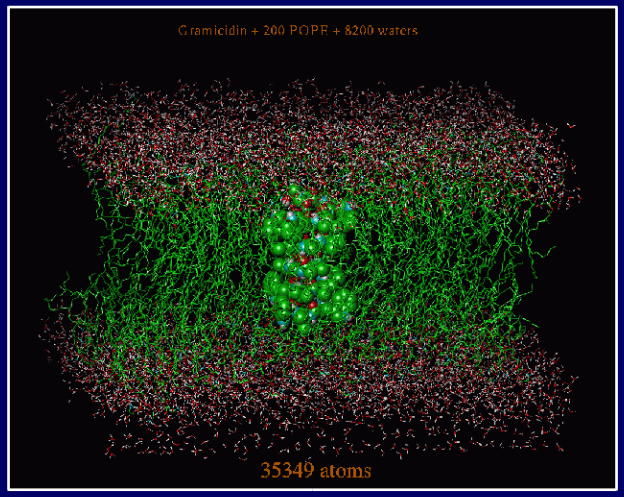
Performance

Plus vite



Grappe 16 x 4 processeurs

Plus loin



≠ Systèmes distribués : fiabilité, hétérogénéité

Plan du cours

- 14/10 [Roch] Introduction – Modèle de coût
- 21/10 [Roch-Gautier] Algo. sans com. - Cascading
- 28/10 [Roch] Algo. avec com. (1)
- 4/11 [Roch-Raffin] Algo avec com (2)
- 18/11 [Roch-Gautier] Simulation distribuée
- 25/11[Trystram-Roch] Ordt par liste.
- 2/12 [Trystram-Roch] Ordt par clustering
- 9/12 - Exposé d'articles

Plan

- Architectures pour le calcul parallèle
- Programmation parallèle: outils
- Techniques de parallélisation
- Analyse de complexité

Références pour ce cours:
[Designing&Building Parallel Programs - Ian Foster, Addisson-Wesley 1995]

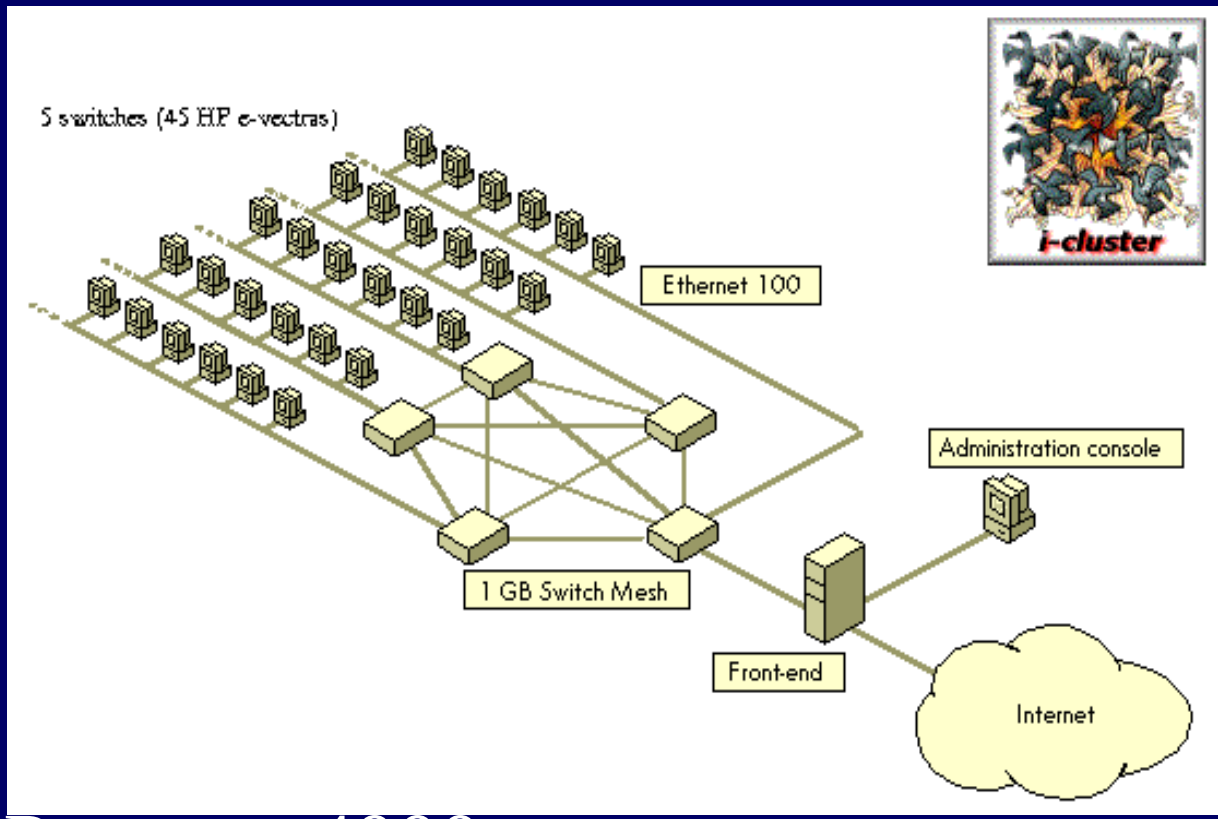
Architectures pour le calcul parallèle

Top 500

www.top500.org

- Architectures distribuées NUMA ex: NEC, IBM SP
- SMP : Symmetric multi-processor UMA ex: Sun, Hitachi
- Grappes : homogènes COW / hétérogènes NOW

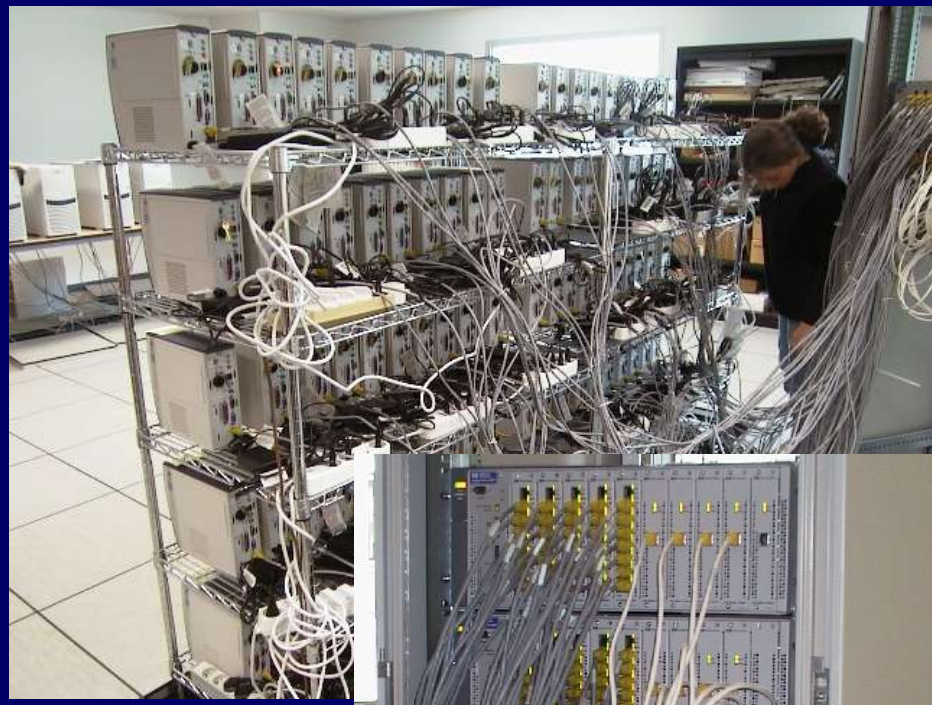
Exemple d'une grappe : le I-Cluster

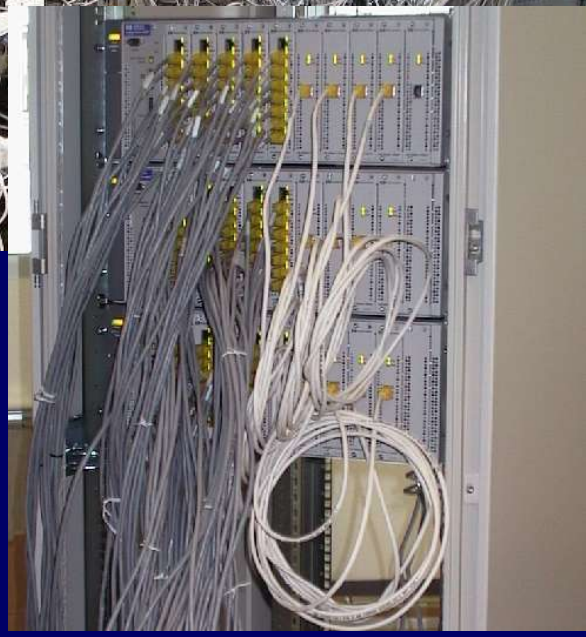


100 Pentium III (HP e-vectra)
733 MHz, 256 Mo
15 Go de disque
Réseau : switch HP Procurve4000
Nœud-Sw : Ethernet-100 ; Sw-Sw : Ethernet1000
Système : Linux 2.4.2

En pratique

http://icluster.imag.fr



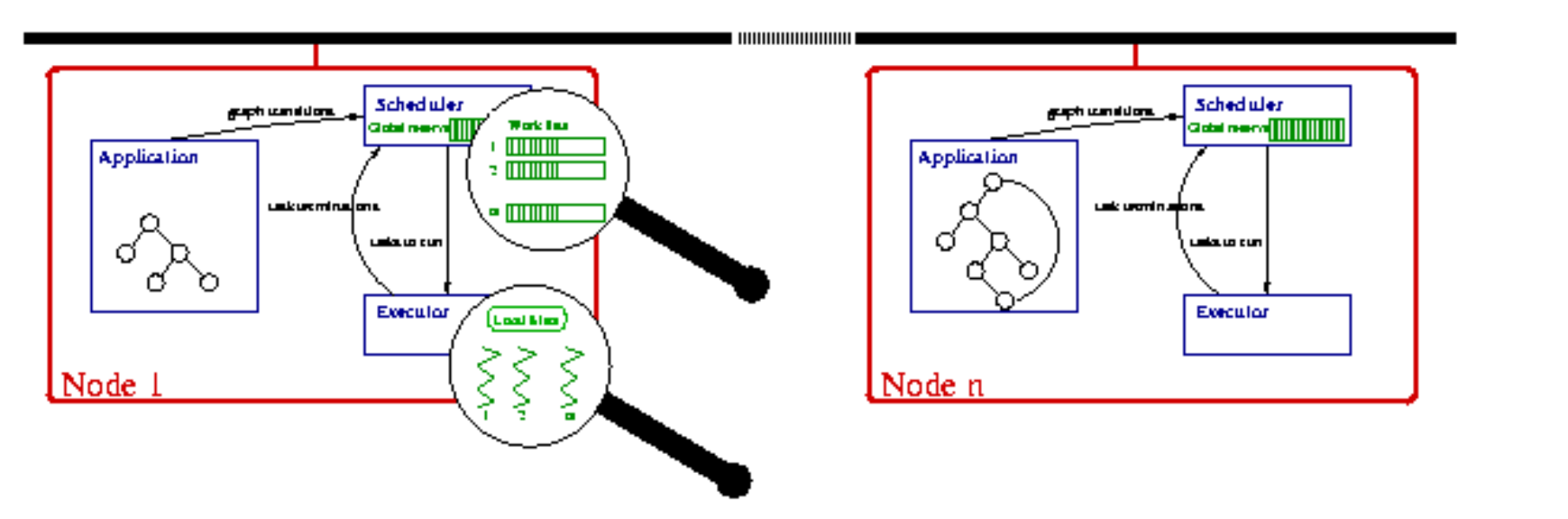


Déballage/montage/câblage 100 PCs :
3h20mn à 12 = 2mn /PC

Plan

- Architectures pour le calcul parallèle
- Programmation parallèle: outils
- Techniques de parallélisation
- Analyse de complexité

Outils de programmation de base



Exploitation de deux niveaux de parallélisme :

- au sein d'un noeud SMP : processus léger

  standard : Threads POSIX

- entre noeuds : communication

  standard : bibliothèque MPI

Concurrence et mémoire partagée

- Partage de mémoire : gérer les accès concurrents
- Processus “légers”
  - Ex: POSIX threads, Java
- Accès concurrents: pas d'atomicité
  - Synchronisation par verrou, sémaphore, moniteur
- Utilisation de librairies externes :
  - thread-safe vs thread aware

Processus communiquants

- Avant 1990 : sockets, xdr, etc
  CSP Communicating Sequential Processes / Occam
- 1990 : PVM : Parallel Virtual Machine

- 1995 : Standard : MPI : Message Passing Interface

  "MPI - The complete reference" M. Suiv&al : www.netlib.org/utk/papers/mpi_book.htm

  - Programmation : chaque processeur exécute son programme
    souvent SPMD: Single Program - Multiple data
  - Communication asynchrone, étiquetée (tag)
  - Groupe de communication : communicateur
    > communications collectives: MPI_Bcast, MPI_Barrier
  - Format : description représentation d'un objet (hétérogène)

Conclusion MPI

- Intérêt :
  - Efficace car proche de la machine
  - Portabilité sur machine avec processeurs identiques
  - Benchmark du Top500 : HPL : écrit en MPI
    + certains tunings possibles (notamment bcast)
- Inconvénient :
  - Programmation complexe:
    enchaînement des communications (send/receive)
  - Modèle de programmation SPMD: peu lisible

Interfaces de programmation parallèle de plus haut niveau

- But: séparer allocation des ressources/prog.
  - HPF Fortran [1990] : au dessus de MPI
    - Placement des données explicite
    - Communication implicite
  - Open-MP : au dessus de Threads
    - Regroupement des calculs explicites
    - Parallélisation des boucles implicites
  - Fonctionnels/Objets : Cilk, Nesl, Smarts, Athapascan

Plan

- Architectures pour le calcul parallèle
- Programmation parallèle: outils
- Techniques de parallélisation
- Analyse de complexité

Techniques de parallélisation

- Dégager le parallélisme maximal
  - Découpe de données -découpe de calculs
- Identifier les schémas de communication
  - Comm. Locales - réduction
  - Statique vs dynamique - structuré/non structuré
- Regrouper calcul/données
  - Augmenter la granularité
- Affecter les agrégats aux processeurs
  - Ordonnancement

Un exemple

Portage du benchmark LinPack HPL sur le i-cluster

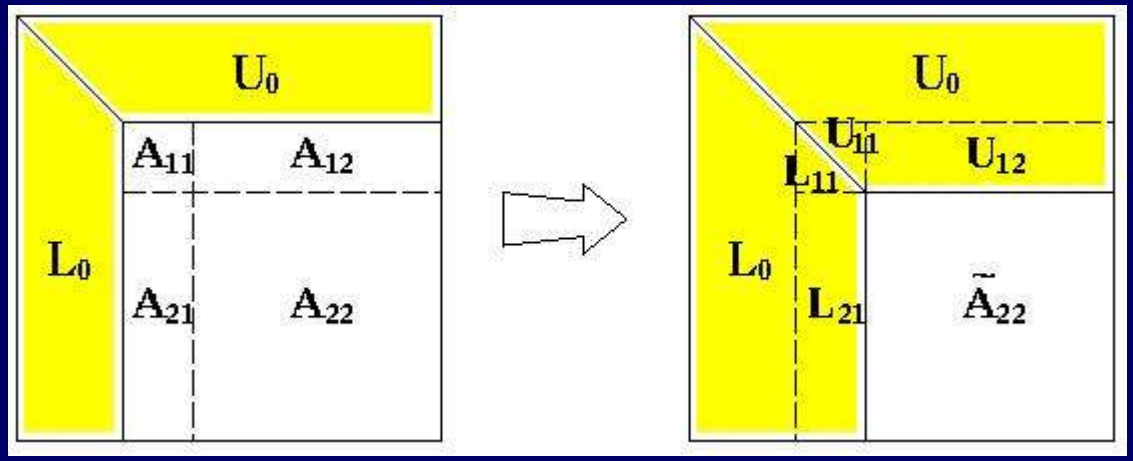
ou

vers le « Top 500 »

Nicolas Maillard -- I.D. Mai 2001

HPL : factorisation P.LU récursive

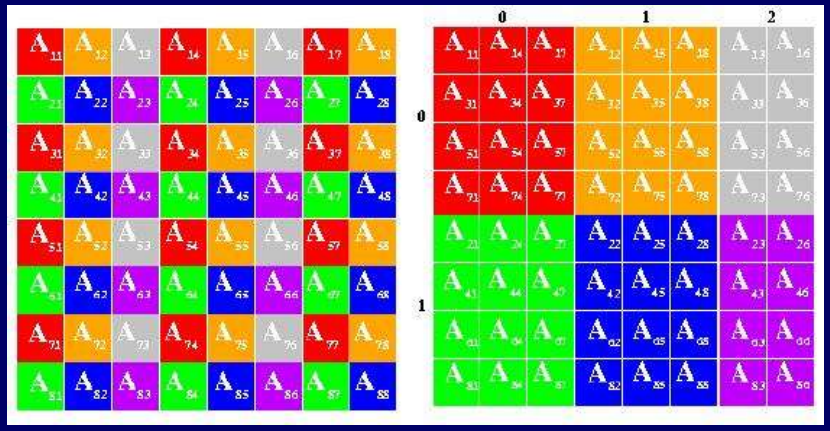
A =



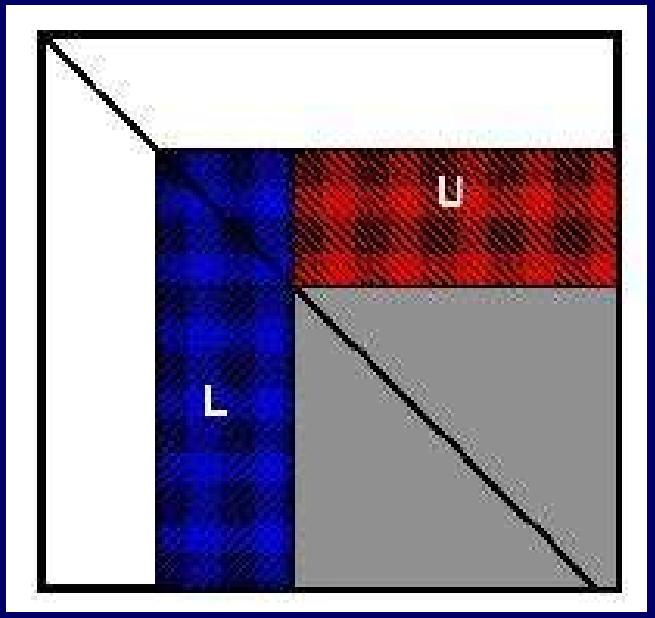
= P.L.U

$$A_{1,1} = L_{11}U_{11}$$
$$A_{21} = L_{21}U_{11}$$
$$A_{12} = L_{11}U_{12} \Leftrightarrow U_{12} = (L_{11})^{-1}A_{12}$$
$$A_{22} - L_{21}U_{12} = L_{22}U_{22}$$

Parallélisation: bloc-cyclique bi-dim



Mapping des blocs Nb x Nb sur une grille virtuelle de PxQ Pes (blocs cycliques)



- Nb colonnes calculées sur une colonne de Pes
- Mise à jour sur une ligne

Programme MPI « portable » : HPL
5 Janvier 2001 :28.6 Gf sur 96 processeurs

Tuning : paramètres de HPL

- Algo de factorisation séquentielle (blas)
- N, Nb : taille de A et d'un bloc
- Profondeur de « pipe »
- P, Q : taille de la grille de PEs
- Algo de diffusion (BCAST)

Flop-story (1)

Objectif top500 -> 70 GFlops le 15 Avril 2001

- 5 Janvier : 28.6 Gf sur 96 procs

- 15 Mars : départ : amélioration BLAS3 + BCAST
  22 Mars : 34.0 Gf sur 96 procs

- 30 Mars : +114 procs + nouveaux drivers + kernel
  7 Avril : de 40 à 50 Gf sur 210 procs :-(

- 8 Avril : 210=15x14 + Bcast=arbre binomial prof 2
  67 Gf sur 210 procs :-)
  10 Avril : **72 Gf** sur **210 procs** (Mapping nœuds/sw)

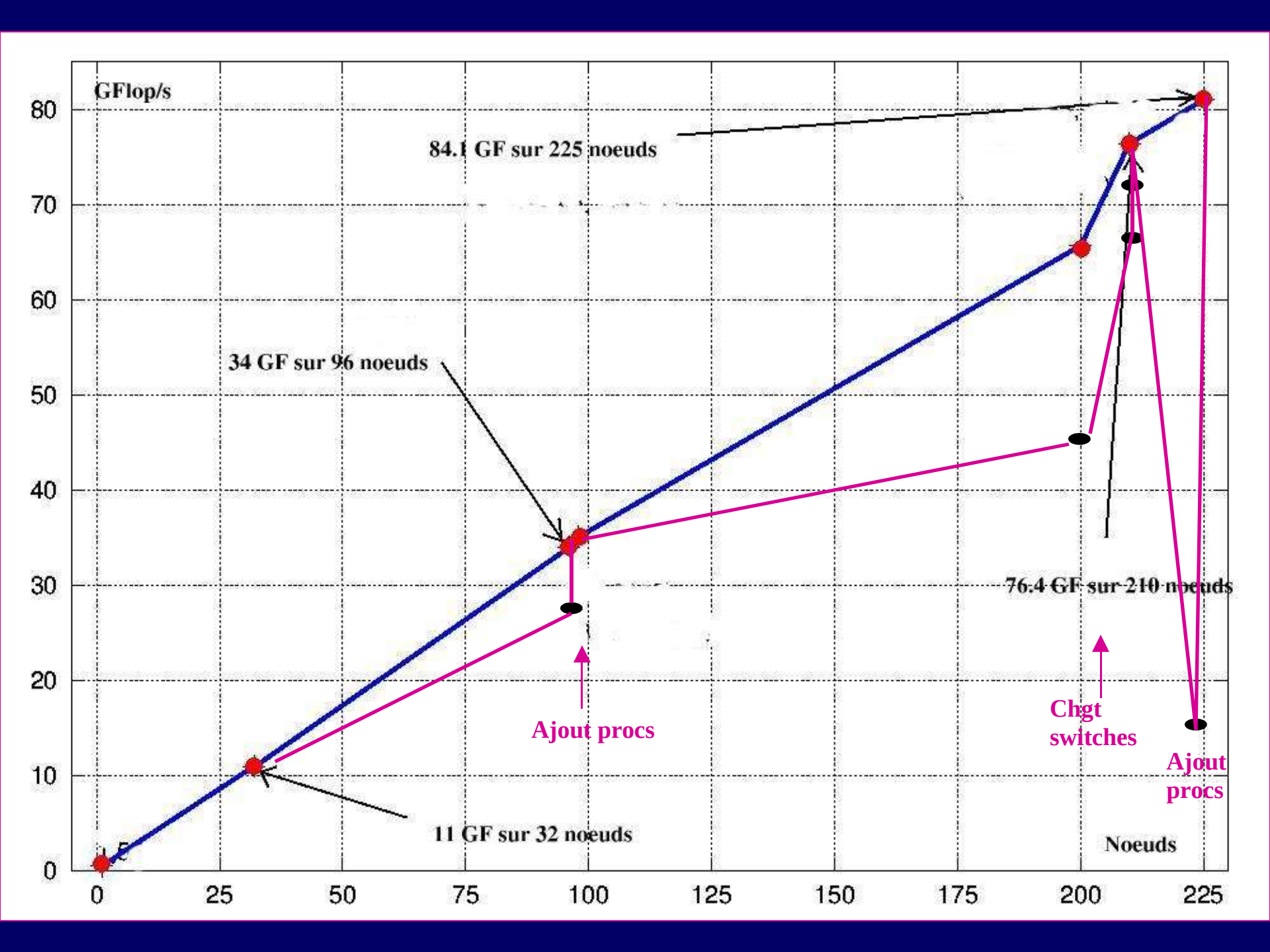
Flop-story (épisode 2)

- 12 Avril – 14h : nouvelles cartes réseaux + 5 procs
  – 17h : 76.4 Gf sur 210 procs (43 n/sw) :-)
  ...mais 32 Gf sur 215 = 43x5 procs

- 13 Avril : Projection à 81 Gf sur 225=15x15 nœuds…
  récupération et ajout de 10 PCs Compaq + cde HP
  14 Avril : 15 Gf sur 225 procs [hétérogénéité+swap…]

- **15 Avril** : Soumission **76.4 Gf** sur **210 procs**
  + estimation 81 Gf sur 225 procs...

- 20 Avril : Réception des 10 nœuds HP e-vectra
  81.4 Gf sur **225 procs**
  81.4 GF = 360 MF/nœud vs. Perf. crête = 680 MF


GFlop/s
84.1 GF sur 225 noeuds
34 GF sur 96 noeuds
76.4 GF sur 210 noeuds
11 GF sur 32 noeuds
Ajout procs
Chgt switches
Ajout procs
Noeuds
0
10
20
30
40
50
60
70
80
0
25
50
75
100
125
150
175
200
225

Conclusion : portabilité sur grappes

- Matériel standard : prix (réseau) + portabilité logicielle
- mais…
  - caractéristiques spécifiques
  - Evolue : ajout de processeurs, chgt de switchs…
  - Programme difficile à porter à une architecture hétérogène

Algorithme	<>	Programme
|		|
Gauss PLU par blocs		hpl : mpi + paramètres

Objectif du cours

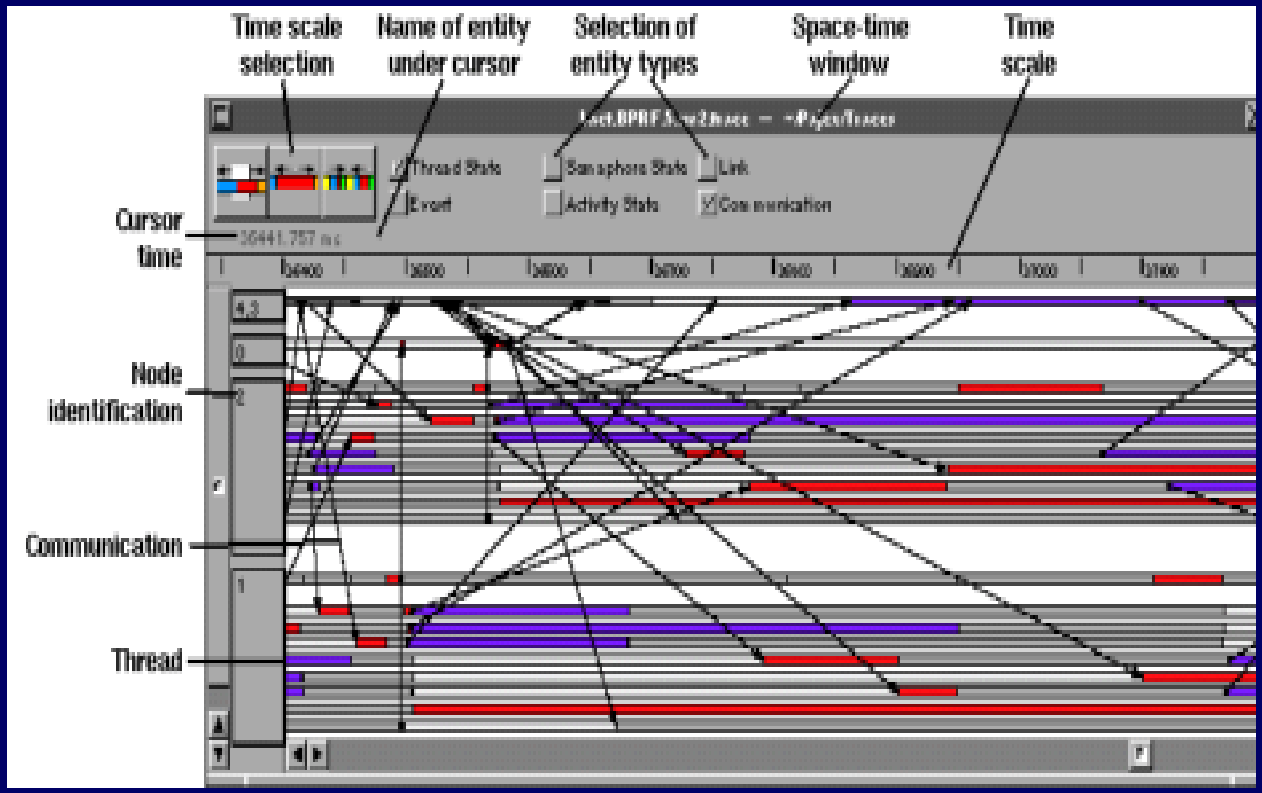
- Maîtrise des techniques algorithmiques pour la construction de programmes parallèles portables et efficaces
  - But: aller p fois plus vite/loin avec p processeurs
- Point critique : ordonnancement
  - Regroupement des calculs
  - Affectation des calculs aux processeurs

Plan

- Architectures pour le calcul parallèle
- Programmation parallèle: outils
- Techniques de parallélisation
- Analyse de coût / complexité

Evaluation des performances

- Diagramme de Gant



Analyse de complexité

- Mesure du coût de l'exécution d'un algorithme parallèle
- Portabilité : abstraction de l'architecture
  - Exécution sur une architecture symbolique
- Représentation de cette exécution : “graphe”
  - Précédence
  - Dépendance
  - Flot de données

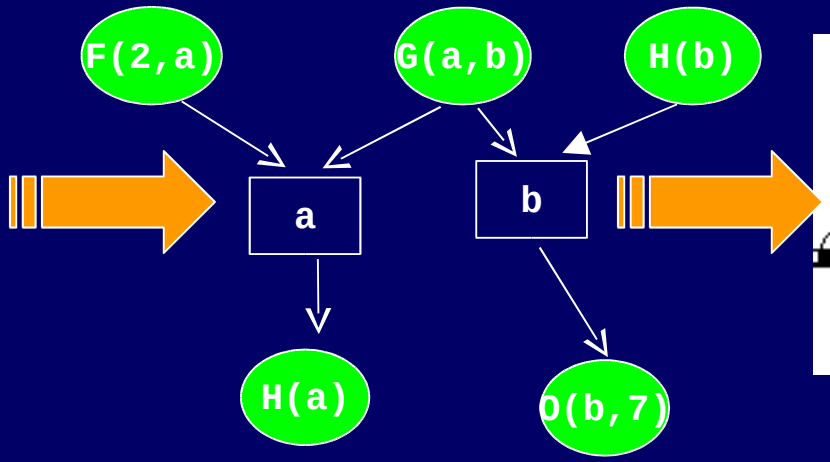
Entre programme et exécution : analyse du graphe de flot de données

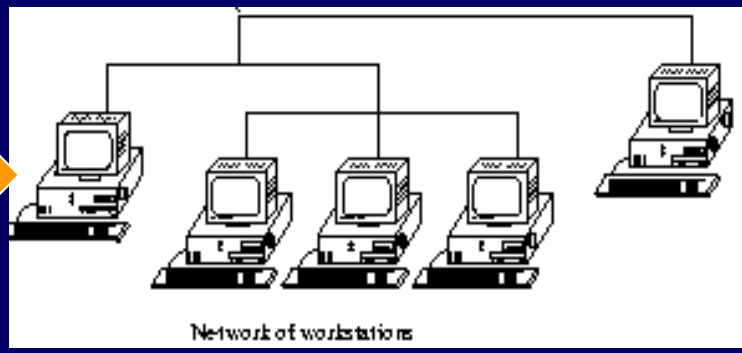
Algorithme parallèle

Exécution du programme

```
Class f {
operating()( shared_r_w<int> x ) {
x = sin(x->val ) * BB(taq ( x ) ) ;
}

void main(int argc, char* argv) {
Fork<f> ( shared<int> (3) ) ;
}
```





Exécution "symbolique"

=> Coût théorique: T_1, T_∞, ...

T_p = temps mesuré

Rem: Sisal, Cilk, Athapascan1,...

Coût d'un algorithme
Définitions et notations

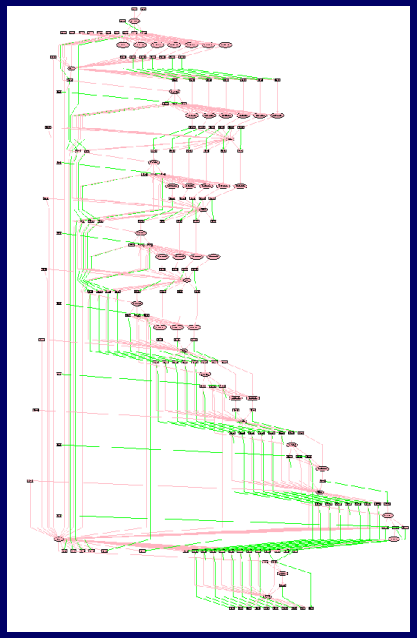
- **T_{seq}** : nbre opérations "meilleur" algo seq.
- **T_1** : nbre opérations total de l'algo par.
- **T_∞** : chemin critique en nbre opérations.
- **T_p** : Temps d'exécution sur p processeurs
- C_1 , C_∞ : idem pour accès mémoire
- **Accélération** : Speed-Up= T_{seq} / T_p ou T_1/T_p

  Efficacité = Tseq / p.Tp ou T1/p.Tp

  Loi d'Amdhal => tout paralléliser !

Exemple 1 : Gauss LU [HPL]

```
Factorization_LU_Par(matrix<bloc>& A, int n ) {
    int i, j, k;
    for( k = 0 ; k < n ; k++ ) {
        BlocFactorization()( A(k,k) ) ;
        for( i = k+1 ; i < n ; i++ )
            BlocSolve_trsm_sup>()( A(k,k), A(i,k) ) ;
        for( j = k+1; j < n ; j++ )
            BlocSolvetrsm_inf>()( A(k,k), A(k,j) ) ;
        for(i = k+1 ; i < n ; i++ )
            for( j = k+1; j < n ; j++ )
                BlocUpdate_gemm>()( A(i,k), A(k,j), A(i,j) ) ;
    }}
```



- Boucle (k=1..n, i=k..n, j=k..n) => $T_1(n) = \theta(n^3)$
- Dépendance sur k, mais i et j en parallèle : $T_\infty(n) = n$

Exemple 2 : Factoriel

```
Void Factoriel ( /* Input /  int i ,  /* Input */  int j ,    /* Output */  int& res)   {
    if (i == j) { res = i ; }
   else {
        int res1, res2 ;
        Factoriel(i, (i+j)/2, res1) ;
        Factoriel((i+j)/2+1, j,  res2) ;
        Multiplication( res1, res2, res) ;   /* Multiplication > res := res1* res2 */
    }
}
```

$T_1(n) = 2.T_1(n/2) + 1 = n-1$ multiplications

$T_\infty(n) = T_\infty(n/2) + 1 = \text{ceil}(\log_2 n)$ multiplications

Théorème fondamental

Ordonnancement glouton [Graham66] - Principe de Brent [80]

Ordt glouton (ou de liste) : minimiser l'inactivité

i.e. Il n'existe pas de tops où il existe une tâche prête et un processeur inactif

- Théorème [Principe de Brent] :
  L'ordt glouton d'un programme de coût séquentiel T_1 et de chemin critique T_∞ conduit à un temps d'exécution sur p processeurs identiques:

$$T_p < (T_1 / p) + T_\infty$$

Corollaire: Ordt work-stealing

- Ordt “work-stealing” (liste, ...) : lorsqu'un processeur devient inactif, il **vole** une tâche prête à un processeur qui en a (dès qu'il en existe)

- **Corollaire** : #opérations de vol < $p.T_\infty$
- **Application** : prise en compte du surcoût d'ordonnancement dans le temps d'exécution:

$$T_p < (T_1/p + T_\infty) + p.T_\infty.coût_{vol}$$

Conclusion (1/2)

- Si $T_\infty << T_1$: ***grain fin***

  portabilité intrinsèque grâce à un ordt work-stealing, “facile”

- NB : le coût doit prendre en compte le coût des opérations de ***description du parallélisme***
  - Création de tâches, dépendances de données

- Pour paralléliser à grain fin la description et obtenir $T_\infty = O(T_1)^\varepsilon$

  => intérêt du parallélisme “récursif”

Conclusion (2/2)

A

- Objectif algorithmique parallèle théorique :
  - Cadre : algorithmes séquentiels polynomiaux
  - $T_{seq} = n^{O(1)}$
  - $T_1 = T_{seq} + o(T_{seq}) \sim T_{seq}$
  - $T_\infty = \log^{O(1)}(n)$

- Prochain cours: construction de tels algorithmes

A

Exercice

- Préfixe : * opération associative
  - Entrée : $a_0, ..., a_n$
  - Sortie : $\pi_0, ..., \pi_n$ avec $\pi_k = a_0 * ... * a_k$ (k=0..n-1)
- Algo séquentiel : $T_{seq} = n . t_*$
- Algo parallèle récursif à grain fin, découpe en 2:
  - Ecrire l'algorithme (avec description)
  - Mq :$T_\infty = \theta(\log n)$
  - Mq $T_1 = \theta(n . \log n)$
  - Question subsidiaire: comment obtenir $T_1 = \theta(n)$