Multithreading & Multiprocessing dans ATLAS

Sébastien Binet

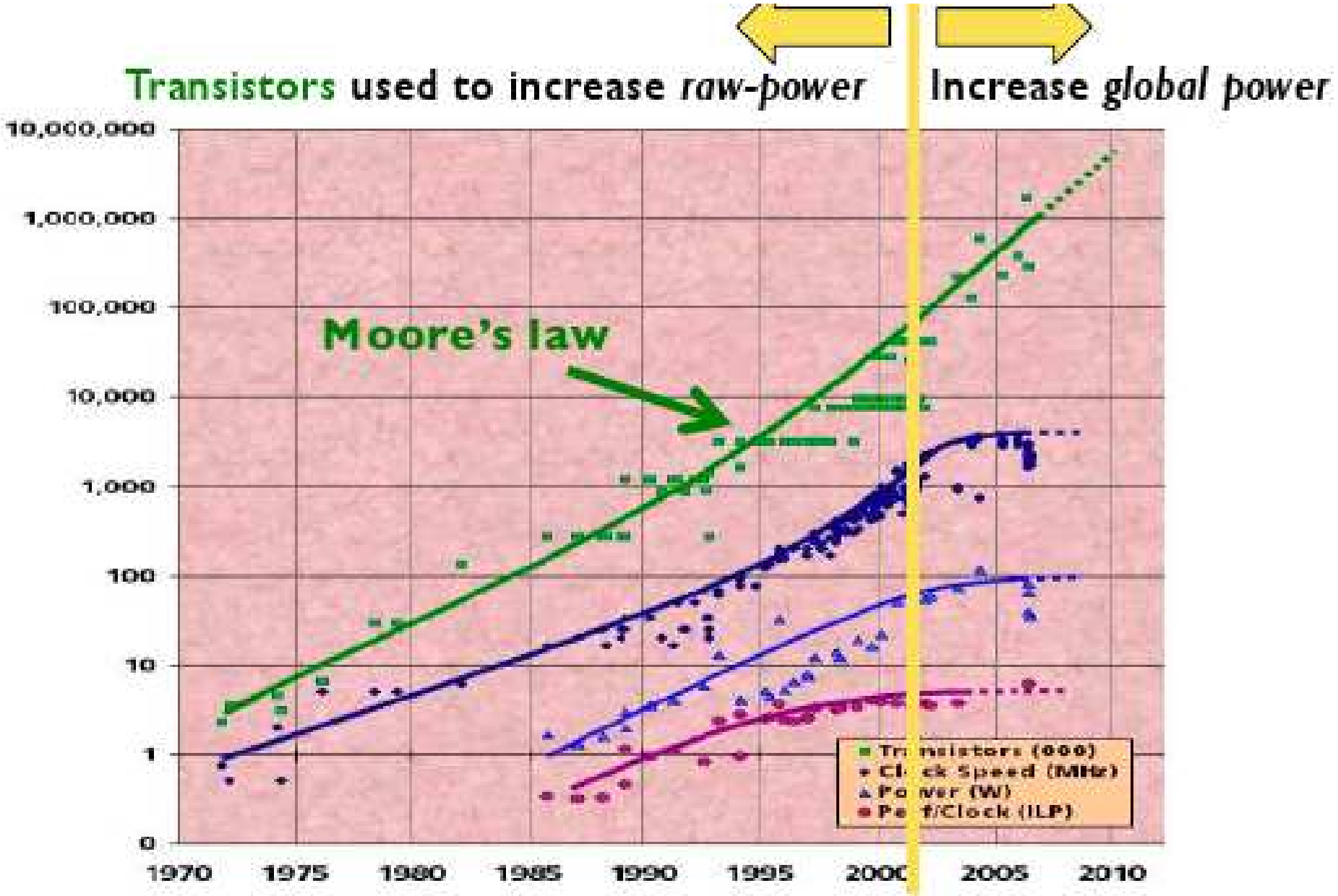
Laboratoire de l'Accélérateur Linéaire

15-09-2009

Outline

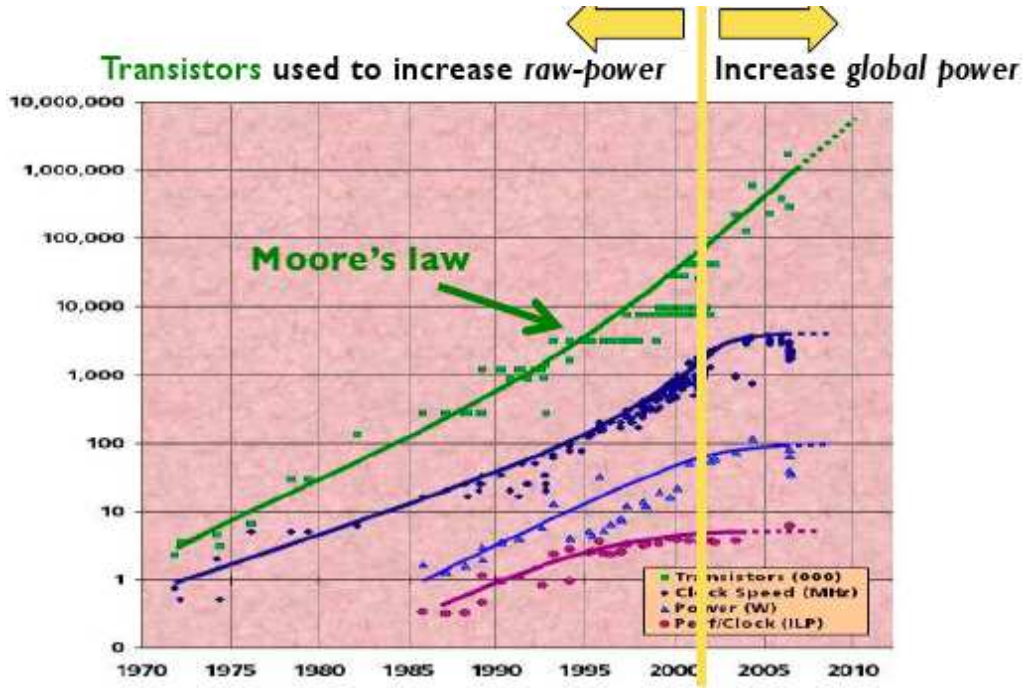
- Introduction
- Expérience avec multi-threading: `AthenaMT`
- Expérience avec multi-processing: `AthenaMP`
- Conclusions

Introduction



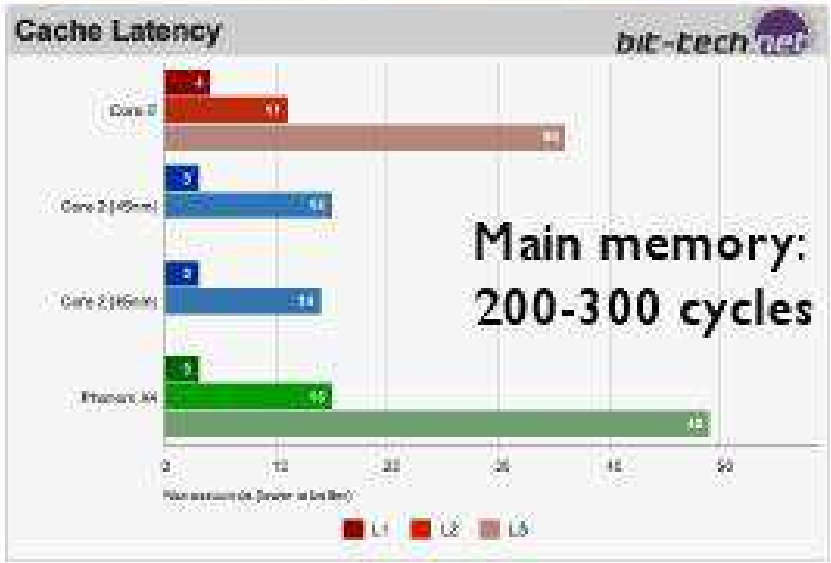
Les 3 barrières: *free lunch is over*

- Loi de Moore toujours observée au niveau *hardware*
- cependant la puissance de calcul *“effective”* perçue est mitigée: confrontation avec les 3 murs
  - *memory wall*
  - *power wall*
  - *ILP (instruction level parallelism) wall*



Memory wall

- augmentation des fréquences processeurs plus rapide que la mémoire
- mémoires cache plus grosses et plus rapides ont ralenti la tendance
- latence dans les accès mémoire: goulot d'étranglement
- mise en place de plusieurs niveaux hiérarchiques de mémoires

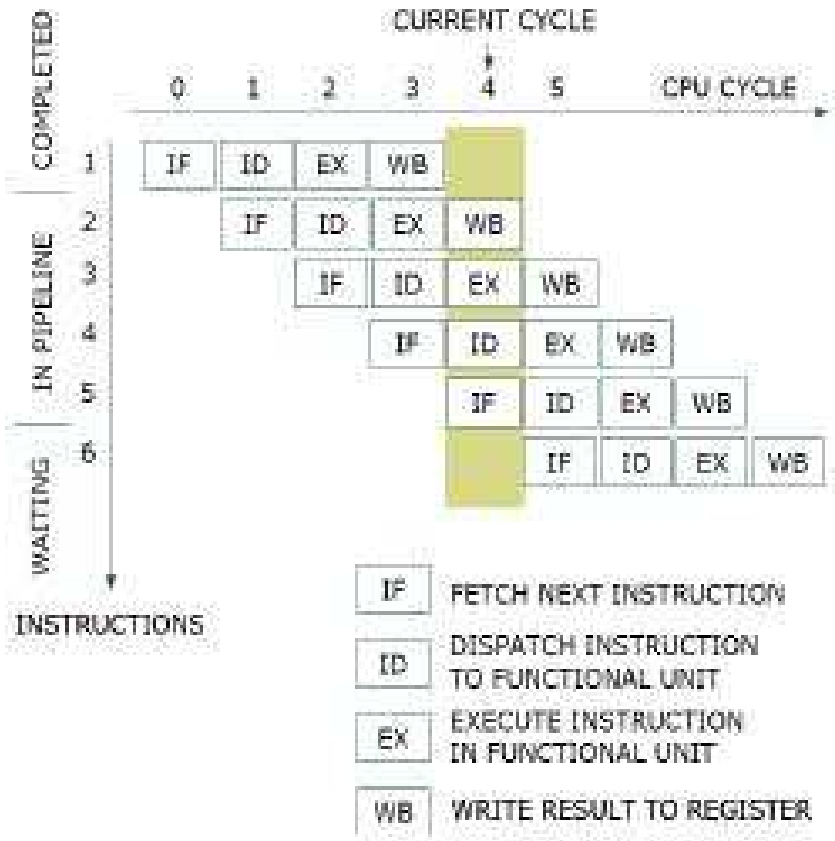


Power wall

- plus un processeur est rapide, plus il est gourmand en consommation
- relation non-linéaire:
  - augmentation de 73% en puissance donne seulement une amélioration de 13% en performances
  - *downclocker* un CPU de 13% $\Rightarrow$ économiser $\sim 1/2$ consommation
- la plupart des centres de calcul sont limités par:
  - la consommation électrique totale qu'ils peuvent s'offrir
  - la fraction dédiée au refroidissement/extraction dissipation

Architecture wall

- *pipelines* longs et larges
- techniques d'amélioration de l'`ILP`
  - *hardware branch prediction*
  - *hardware speculative execution*
  - *instruction reordering*
  - *JIT compilation*
  - *hardware threading*, . . .
- en pratique: problèmes d'inter-dépendances entre les instructions limitent l'`ILP`



Motivations: tendances *hardware*

- *CPU* $\Rightarrow$ multicores
  - chaque *CPU* peut comporter plusieurs (2 $\rightarrow\sim$ 1024) cœurs
  - chaque *core* est moins rapide individuellement que les *"anciens" CPU*
  - quantité de mémoire par *core* $\downarrow$
- $\uparrow$ #*CPU* cores $\Rightarrow$ $\uparrow$ parallélisme
  - programmes d'analyse/reconstruction:
    - parallélisme au niveau des événements
    - *embarassingly parallel*
  - parallélisation au niveau des algorithmes:
    - plus (+) profitable
    - plus difficile aussi (redesign du code)
- Loi d'Amdahl: $R_{speedup} = \frac{1}{(1-S)+\frac{S}{N_{CPU}}}$
- exploitation du parallélisme *via*:
  - multi-threading (`AthenaMT`)
  - multi-processes (`AthenaMP`)

Introduction - II

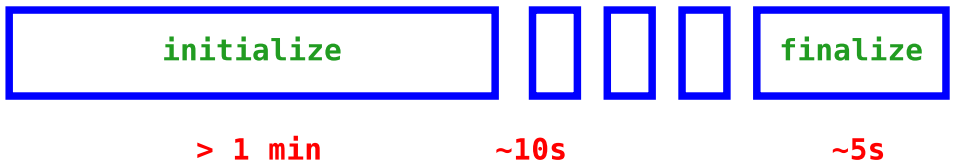
Contraintes

- ATLAS-*software*:
  - code écrit avant l'*ère multi-core*
  - reconstruction *online* et *offline*: *single-process* et *single-threaded*
- période de prise de données imminente
- experiences avec *multi-threading* et *multi-processing*:
  - les modifications apportées doivent aussi transparentes que possible

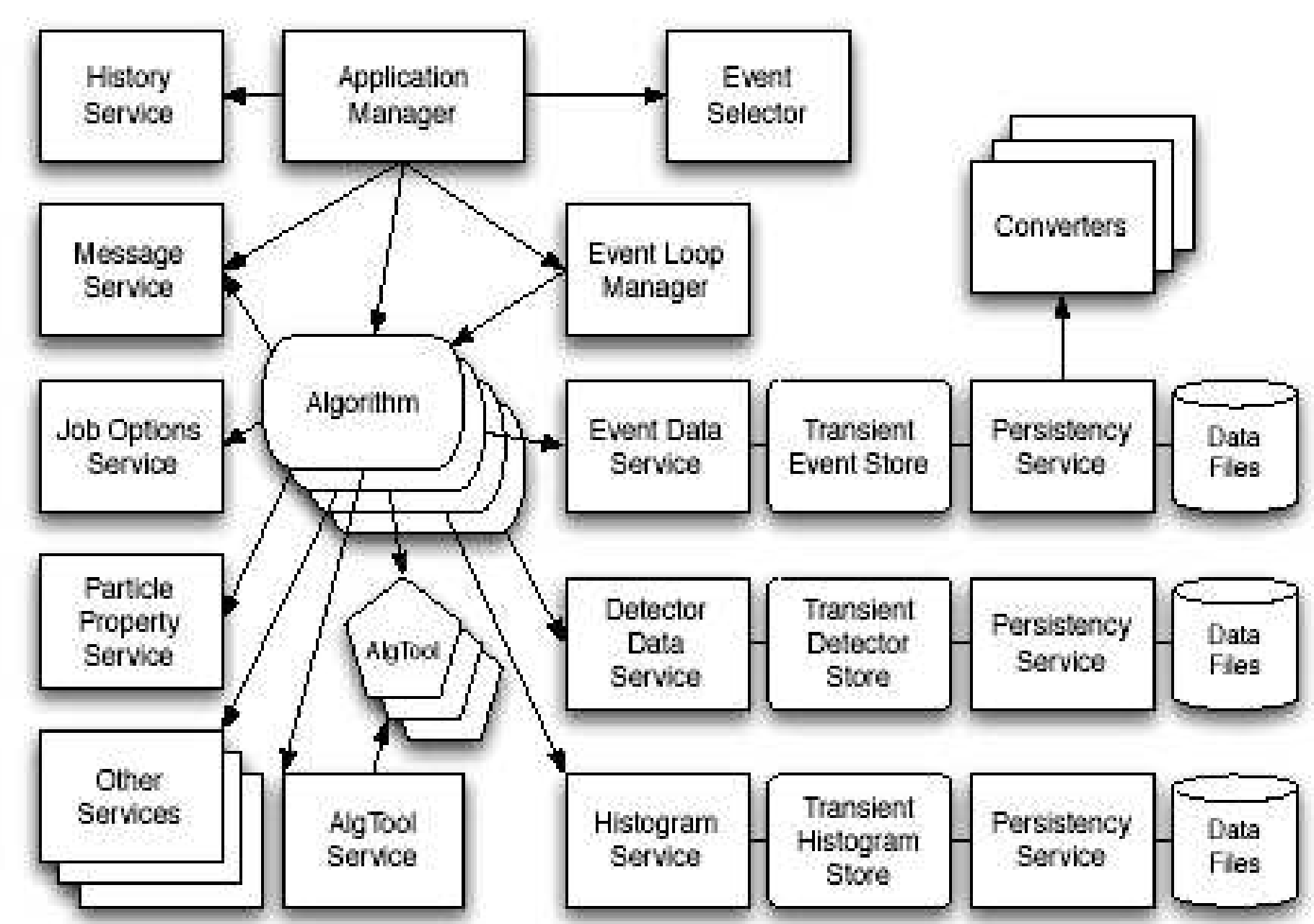
Introduction - III

Athena: fiche technique

- association de multiples composants élémentaires:
  - `Algorithm, AlgTool, Service,...`
- un seul fil d'execution (*single thread*)
- reconstruction (15.5.0):
  - $\sim$ 1.8 *Gb VMEM*
  - $\sim 10 - 15$ s/evt
- 3 phases principales:
  - `initialize`:
    - ★ lecture de la configuration
    - ★ instantiation des composants
    - ★ ouverture des fichiers d'input, création fichier(s) d'output
  - `execute`: boucle des événements
  - `finalize`: fermeture des fichiers, ...



Introduction - IV

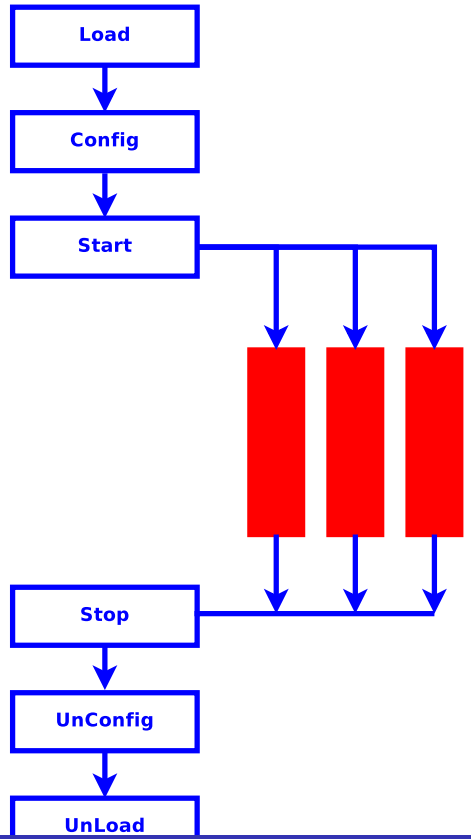


Multi-threading: AthenaMT

- partage de `.text (.so)`
- *context-switch* léger $\rightarrow$ *"lightweight processes"*
- partage automatique des ressources *hardware*
- $\Rightarrow$ application au système de déclenchement de niveau 2

Trigger L2 Processing Unit

- traitement evts.: *n* worker threads
- *framework* `Athena` modifié:
  - instantiation+*scheduling* des algos de traitement des données par *worker thread*
  - `pthreads`
- *testbench* initial: dual processor single core (bien avant l'ère des multi-cœurs)



Support du Multi-threading dans le *framework* Athena

- Multi-threading: tous les composants qui modifient des données doivent être *thread specific (TLS)* (*e.g.* `EventStore`)
  - certains composants peuvent être "read only"
  - $\Rightarrow$ communs à tous les *threads* (*e.g.* `GeometrySvc, DetectorStore`)
- les composants *TLS* sont identifiés par leur type et leur nom `(generic name)__(thread ID)`
  - *ex* instantiation d'un alg. de type `TriggerSteering` et nom `TrigStr` pour 2 *threads*:
    - `TriggerSteering/TrigStr__0`
    - `TriggerSteering/TrigStr__1`
- Hypothèses de départ:
  - Algorithmes sont toujours TLS
    - chaque *thread* possède un copie de l'algorithme, générée automatiquement à partir de la config. *single thread*
  - possibilité de spécifier au niveau de la configuration si un composant est *TLS* ou commun à tous les *threads*
- la version modifiée d'`Athena` peut être aussi utilisée dans l'environnement *offline* (*debugging*)

Expériences avec le *multi-threading*

- création *“d’applications de sélections d’évts” MT-aware*
  - et bénéficiant de l’approche *MT*
- problèmes techniques (plus ou moins) historiques (mais intéressants):
  - problèmes avec de vieilles versions de la STL (ex: `gcc-2.95`)
    - modèle d’allocation mémoire sous-optimal (ex: contention au niveau de `std::string`)
  - *thread-safety* de certaines bibliothèques externes (FORTRAN et C++)

Expériences avec multi-threading - II

Développement *software*

- dev. dans un environnement MT
  - entraînement spécial + (acquisitions de) nouvelles connaissances
- prise en compte du modèle MT $\Rightarrow$ event parallelism
  - creation d'un émulateur *athenaMT* (*debugging tool*)
- usuels problèmes de debugging (*races, synchronization problems, dead/live-locks,. . .*)
- réel besoin d'outils (libres) pour assister les dévs. dans le *debugging* et pour l'optimisation
- code de sélection des évts change rapidement du aux besoins en ϕ
  - besoin constant de ré-optimisation du code après modification

- Problème: préserver *thread safety* + code optimisé à travers les *releases* et avec une large communauté (hétérogène) de développeurs

Multi-processing: implémentation naïve

- lancer *n* instances d'une application sur une machine avec *n* cœurs
  - utilisation du code existant
  - *a priori*, pas de modification du code requise
  - bonne *scalabilité* avec le nombre de cœurs

Problème(s)

- augmentation de la demande en ressources avec le nbre de processus
- ↑ empreinte mémoire
- autres ressources de l'O/S: file descriptors, network sockets, . . .
- partage des ressources (+optimisation) - ex. sur les clusters de DAQ:
  - contrôle du nbr d'applications
  - nbr de connections réseau vers le système de *readout*
  - transfert des mêmes données de configuration *n* fois vers la même machine
  - recalcul *n* fois des mêmes données de configuration
  - optimisation *CPU*: utilisation du *CPU* pour l'*evt. processing* pendant un I/O-wait

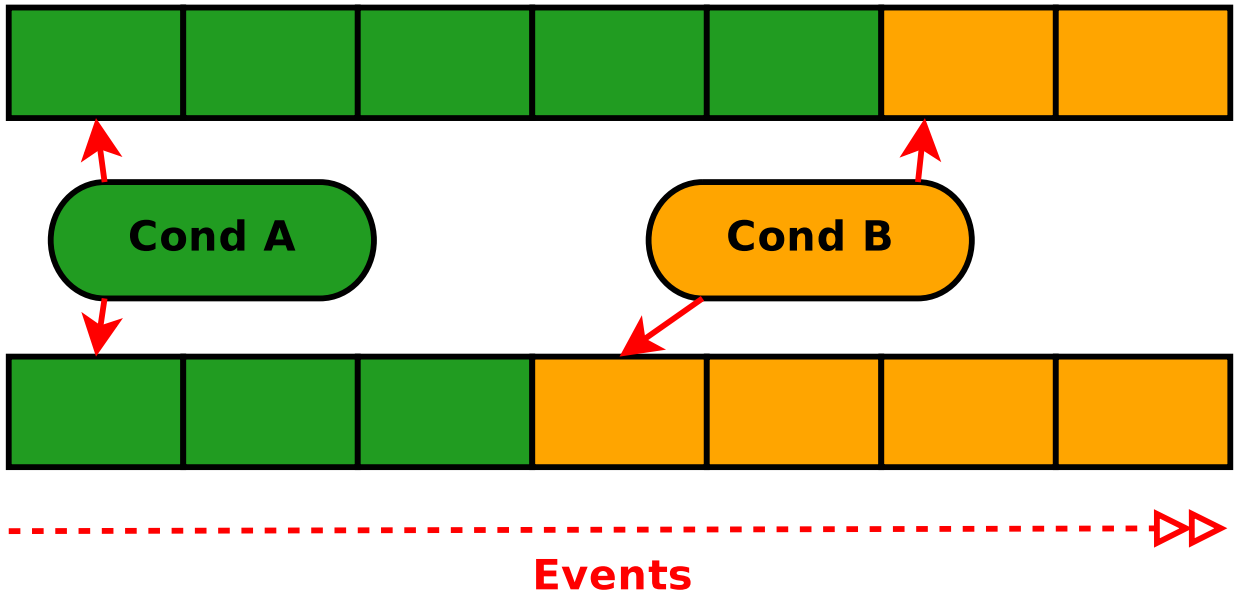
Partage de la mémoire *via* `fork+COW`

- Principe:
  - lancer plusieurs jobs (similaires) `Athena`, partageant autant de mémoire que possible
  - minimiser les modifications de code
    - ⋆ laisser l'O/S faire la plupart du travail
  - utiliser **`fork()`**
- `fork()`:
  - `fork()` clone un processus, y compris la totalité de son adressage mémoire
  - `fork()` sur un O/S moderne implémenté *via* *"Copy On Write"*
    - ⋆ page mémoire est partagée jusqu'à ce qu'un processus écrive dessus
    - ⋆ page mémoire est copiée et devient non-partagée
  - `fork()`er aussi tard que possible mais avant d'écrire sur disque
  - partage optimal et automatique de la mémoire entre sous-processus

Partage de la mémoire *via* `fork` - II

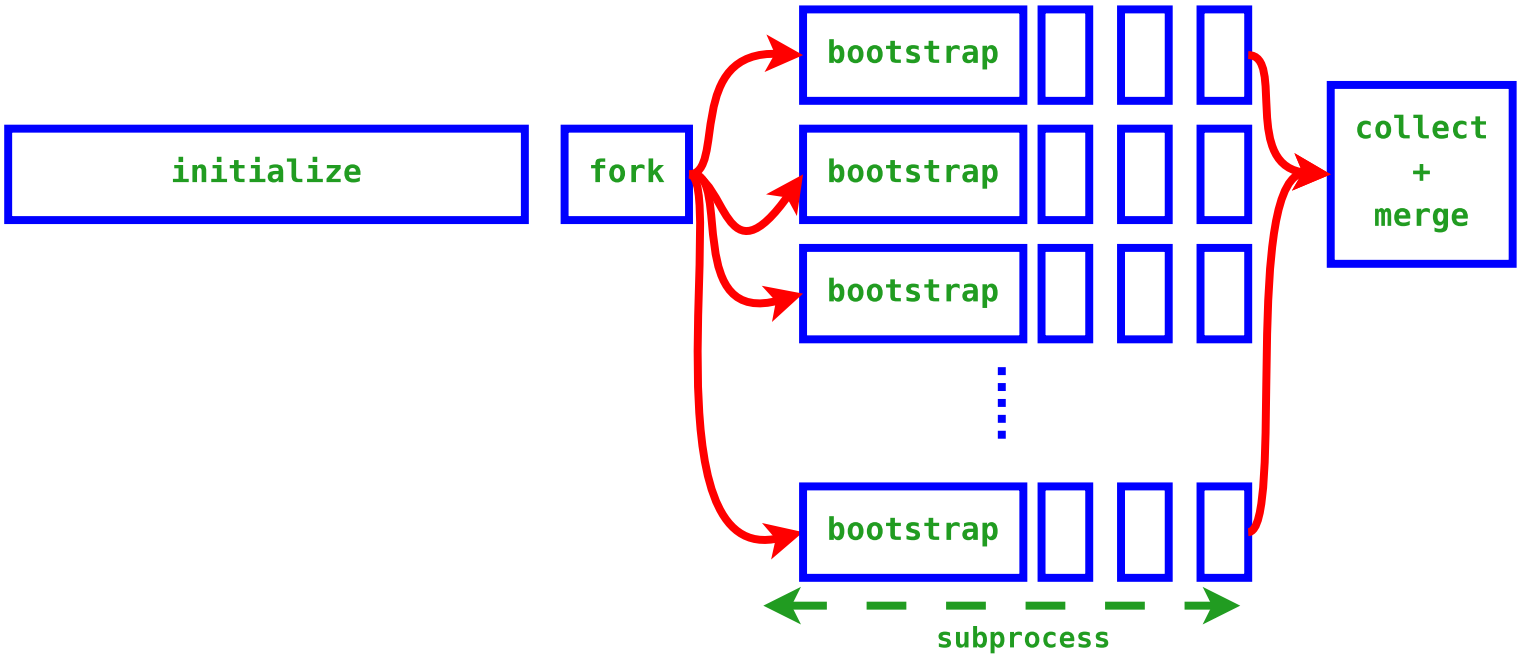
- avantages:
  - toute la mémoire qui peut être partagée le sera
  - modifications restreintes à qques *core packages* d'`Athena`
  - pas de problème de locks/mutexes

- désavantages
  - la mémoire ne peut pas être re-partagée une fois *non-partagée*
    - ★ problème éventuel pour les *conditions data*



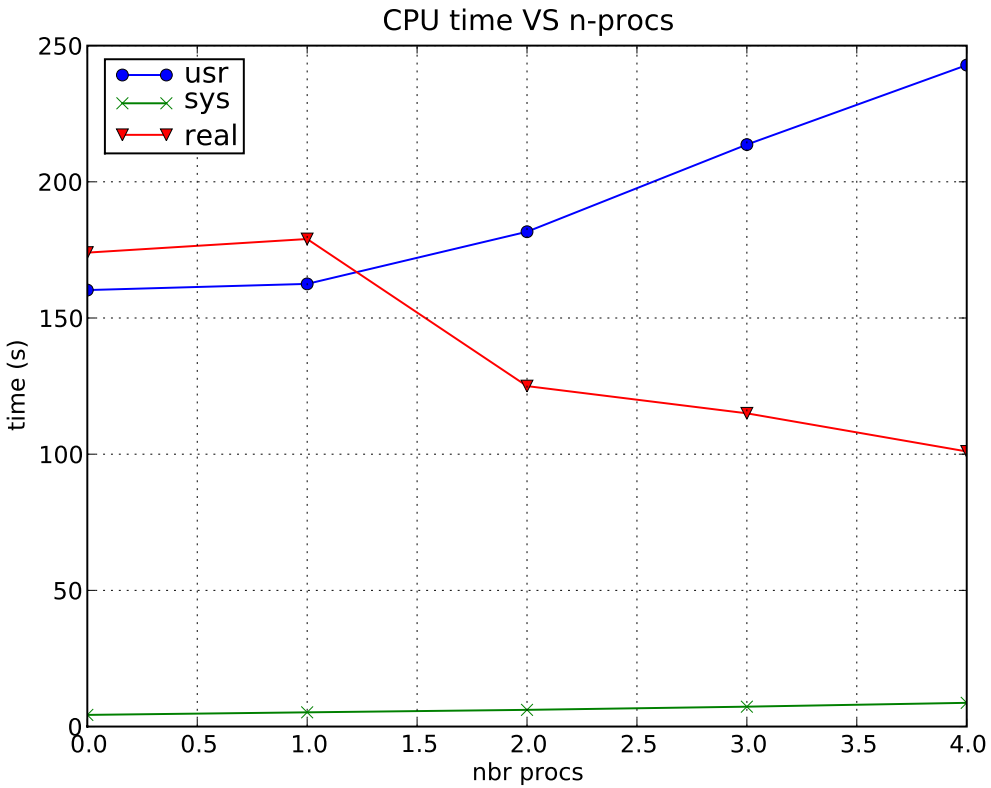
AthenaMP - implémentation

- minimise impact sur le code client
- utilisation du module python `multiprocessing` pour la gestion des processus
- encapsulation des modifications reliées à la gestion du parallélisme *via* un nouveau gestionnaire de la boucle des évts.
- modification des composants reliés à l'I/O



cpu

```
4procs 242.85s user 8.71s system 249% cpu 1:40.99 total
3procs 213.67s user 7.30s system 191% cpu 1:55.12 total
2procs 181.67s user 6.13s system 149% cpu 2:05.77 total
1procs 162.52s user 5.22s system 093% cpu 2:59.18 total
0procs 160.25s user 4.28s system 094% cpu 2:53.45 total
```

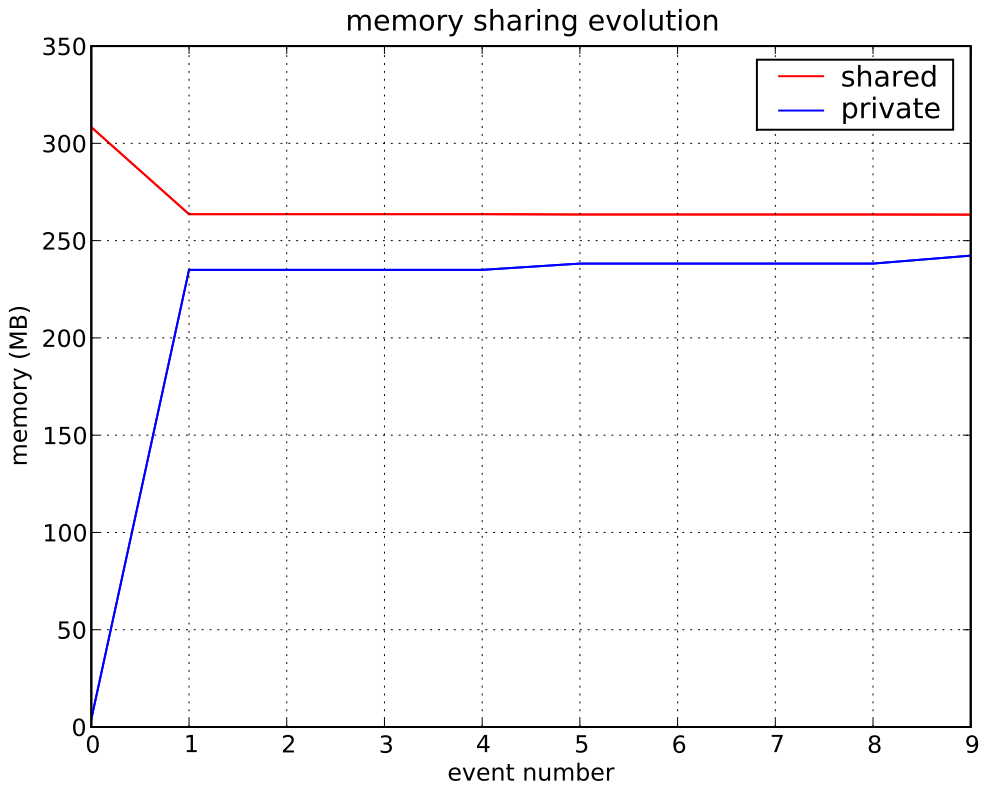


NB: 0 procs == no_MP

memory

process: $\sim$ 700*MB VMem* and $\sim$ 420*MB RSS*

```
(before) evt 0:  private:  004 MB | shared:  310 MB
(before) evt 1:  private:  235 MB | shared:  265 MB
...
(before) evt50:  private:  250 MB | shared:  263 MB
```



Expériences avec multi-processing `fork+COW`

- impact localisé
- modifications restreintes au *control framework* et aux composants-`I/O`
- développement relativement plus aisé:
  - pas de partage implicite (sans `fork+COW`)
  - pas de locks, races, ...

Problèmes

- caractère définitif de `COW` (ex: *conditions data*)
- générateurs de nombres aléatoires (cfg, graines, reproductibilité)
- I/O
  - traque des appels direct à `fopen()` (court-circuitant le *framework*)
  - merge des fichiers d'output
- Grille de calcul
  - sousmission de jobs-MP (overbooking)
  - `vmem` *accounting*
    - ⋆ double-comptage de la mémoire partagée entre les sous-processus `fork()`és

Multi-processing + KSM

- `Kernel Shared Memory (KSM)` est un module pour `GNU/Linux` permettant de partager (dynamiquement) des pages mémoires identiques entre 1 ou plusieurs processus
  - développé dans le contexte de la virtualisation *via* `KVM` pour partager la mémoire entre plusieurs VM
  - `KSM` scanne des zones mémoires étiquetés par une fonction dédiée, et collecte les pages identiques
- modification de `(tc)malloc` afin d'utiliser l'API de `KSM`
  - juste une seule ligne de code rajoutée

- application à un job de reconstruction:
  - aucune modification appliquée au code
  - 1.6 *GB* de *VMem* $\Rightarrow$ 1 *GB* partagés grâce à `KSM`

Conclusions

- expérimenté avec les 2 voies usuelles pour améliorer le parallélisme:
  - multi-threading: `AthenaMT`
  - multi-processing: `AthenaMP`
- approche `MP` (`COW+fork` et/ou `KSM`) semble plus prometteuse:
  - modifications du code localisées
  - maintenance plus aisée
  - plus facile à *debugguer*

Manycores: défis à l'horizon

- *scalabilité* limitée du parallélisme *evt-level* face aux $\sim$ 1024 cœurs des *manycores*
  - `I/O`, `I/O` *buffers*
  - utilisation mémoire
- $\Rightarrow$ parallélisation au niveau algorithmique
  - `pthreads`, `OpenMP`, `MPI`, `OpenCL`, `shmem`
  - ex: `Minuit2 + MPI`, `Geant4-MT`
- `parallel-I/O ?`

Backup

```
class MpEventLoopMgr (PyAthena.Svc):
    def executeRun (self, maxevt):
        """Process `maxevt` events as Run (beginRun->endRun)"""
        if self._ncpus <= 0:
            return self._evtloop_mgr.executeRun(maxevt)

        import multiprocessing as mp
        info ("nbr of workers: %i", self._ncpus)
        info ("master workdir: %s", self._wkdir)
        wrks = mp.Pool(processes=self._ncpus,
                       initializer=self._worker_bootstrap)
        results = wrks.map_async(func=batch_run,
                                 iterable=(maxevt,)*self._ncpus)
```

_worker_bootstrap

- function called after `fork`
- change work dir
- reopen file descriptors
- tickle the `IoComponentMgr`

```
class IIoComponentMgr {
  /** allow a @c IIoComponent to register itself with this
   *  manager so appropriate actions can be taken when e.g.
   *  a @c fork(2) has been issued (usually handled by calling
   *  IIoComponent::io_reinit on every registered component)
   */
  virtual
  StatusCode io_register (IIoComponent* iocomponent) = 0;

  /** @brief: reinitialize the I/O subsystem.
   *  This effectively calls IIoComponent::io_reinit on all
   *  the registered @c IIoComponent.
   */
  virtual StatusCode io_reinitialize () = 0;

  /** @brief: finalize the I/O subsystem.
   *  Hook to allow to e.g. give a chance to I/O subsystems to
   *  merge output files.
   */
  virtual StatusCode io_finalize () = 0;
};
```

```
class MpEventLoopMgr (PyAthena.Svc):
    def executeRun (self, maxevt):
        """Process `maxevt` events as Run (beginRun->endRun)"""
        if self._ncpus <= 0:
            return self._evtloop_mgr.executeRun(maxevt)

        import multiprocessing as mp
        info ("nbr of workers: %i", self._ncpus)
        info ("master workdir: %s", self._wkdir)
        wrks = mp.Pool(processes=self._ncpus,
                       initializer=self._worker_bootstrap)
        results = wrks.map_async(func=batch_run,
                                 iterable=(maxevt,)*self._ncpus)
```

batch_run

- inject a filter algorithm in front of alg-sequence
  - accept/reject events based on local process-id and current event number
- effectively implement a round-robin filter
- call the `executeRun` of the wrapped event loop manager

```
class MpEventLoopMgr (PyAthena.Svc):
    def finalize (self): ...
```

- tickle `IIoComponentMgr::io_finalize` (when a forked process)
- master will run the merge of output files
  - usually trivial for ROOT files containing histos and ntuples
  - trickier for ROOT/POOL files
    - have to take care of POOL links/references

AthenaMP with pile-up