AthenaMP:
Athena on steroids

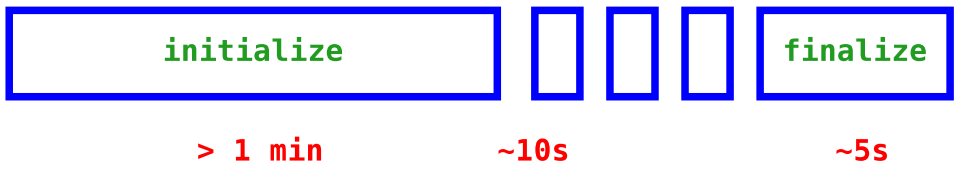
Sébastien Binet

Laboratoire
de
l'Accélérateur Linéaire

18-12-2008

Athena: fiche technique

- *framework* basé sur GAUDI
- association de multiples composants élémentaires:
  - `Algorithm, AlgTool, Service,...`
- un seul fil d'execution (*single thread*)
- reconstruction (14.5.0):
  - `rdotoesdnotrigger`:
    - ★ $\sim$ 1.6 *Gb VMEM*
    - ★ $\sim$ 10 *s*/*evt* ($\sim$ 24000 *kSi2k*)
  - `rdotoesd`:
    - ★ $\sim$ 2.1 *Gb VMEM*
    - ★ $\sim$ 15 *s*/*evt* ($\sim$ 37000 *kSi2k*)
  - `esdtoaod`:
    - ★ $\sim$ 1.3 *Gb VMEM*
    - ★ $\sim$ 1 *s*/*evt* ($\sim$ 2500 *kSi2k*)



Motivations - II

hardware: tendance générale

- *CPU* $\Rightarrow$ multicores
  - chaque *CPU* peut comporter plusieurs (2 $\rightarrow\sim$ 1024) unités de calcul
- chaque *core* est moins rapide individuellement que les *'anciens' CPU*
- mémoire plus restreinte (pour `Athena`)

Pour tirer parti des machines de la prochaine génération,
il faut paralléliser `Athena`

Problème

- solution usuelle: créer plusieurs *threads* d'execution
- **MAIS:** 90% du code n'est pas *'thread-safe'*
- extrêmement dur de coder *'multi-threaded'*
  - (et vous pensiez qu'`Athena/C++` était compliqué ?)

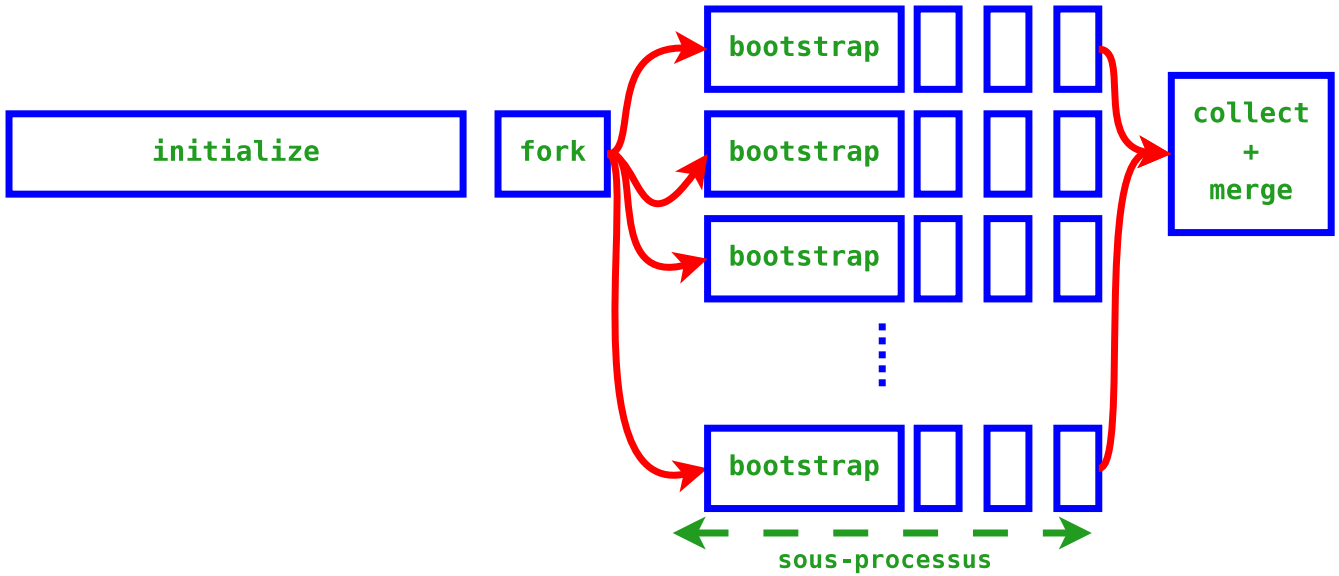
Solution

Idée originale - *proof of concept* (Scott Snyder)

- utiliser plusieurs *processus*
  - par défaut, pas d'interférences entre processus
  - chaque processus a sa plage d'adresses mémoires
- pas de problème au niveau code client
  - pas de *races, deadlocks,...*
  - généralement, pas de modifications à apporter
- impact limité à certains composants *'core'*
  - *I/O* (`THistSvc, AthenaPoolCnvSvc, ...`)

Solution - II

- recette **AthenaMP** (multiple-processes):
  - créer une instance d'`Athena`
  - procédure normale jusqu'à `::initialize`
  - processus parent `fork` *n* sous-processus
    - `Linux` partage efficacement la mémoire qui peut être partagée entre les différents processus
    - pas d'explosion d'utilisation de la mémoire
    - *swapping* limité
  - chaque sous-processus analyse *m* événements,
  - puis appel de `::finalize`
  - processus parent collecte fichiers d'*output* + *merge*



Implémentation (S. Binet)

- création d'un nouveau event loop manager: `MpEventLoopMgr`
  - encapsulation des détails MP
  - activation/désactivation *via* joboptions
- utilisation d'un module `python` (`multiprocessing`, dans la librairie standard de py-2.6) pour la gestion des sous-processus
  - *pool* de sous-processus (= *ncores*)
  - gestion des fichiers d'entrée et de sortie
- création d'un script pour *merger* les fichiers POOL et ROOT
  - de loin la partie la plus délicate (POOL, `ElementLinks`, `...`)

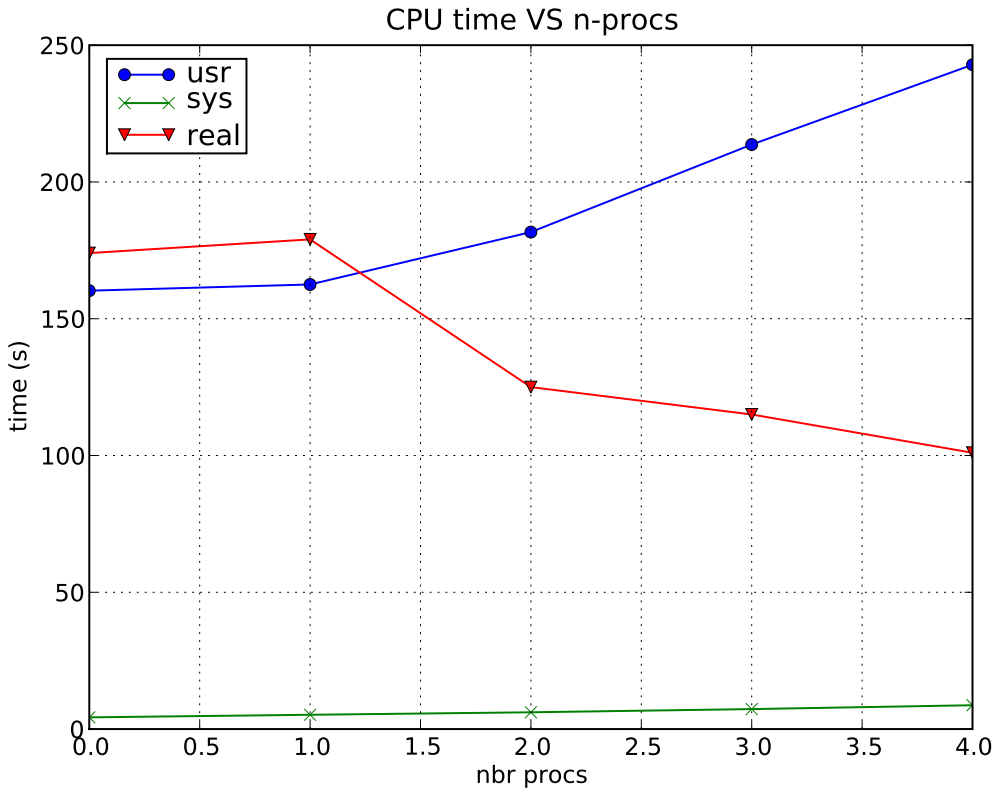
Status

- *alpha* code
- testé avec `rdotoesdnotrigger` et `esdtoaod`
- tourne **OK** mais tests plus approfondis nécessaires
  - développement des outils de validation en cours (`PyDumper`)

Résultats préliminaires

cpu - RecExCommon/esdtoaod

```
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
```

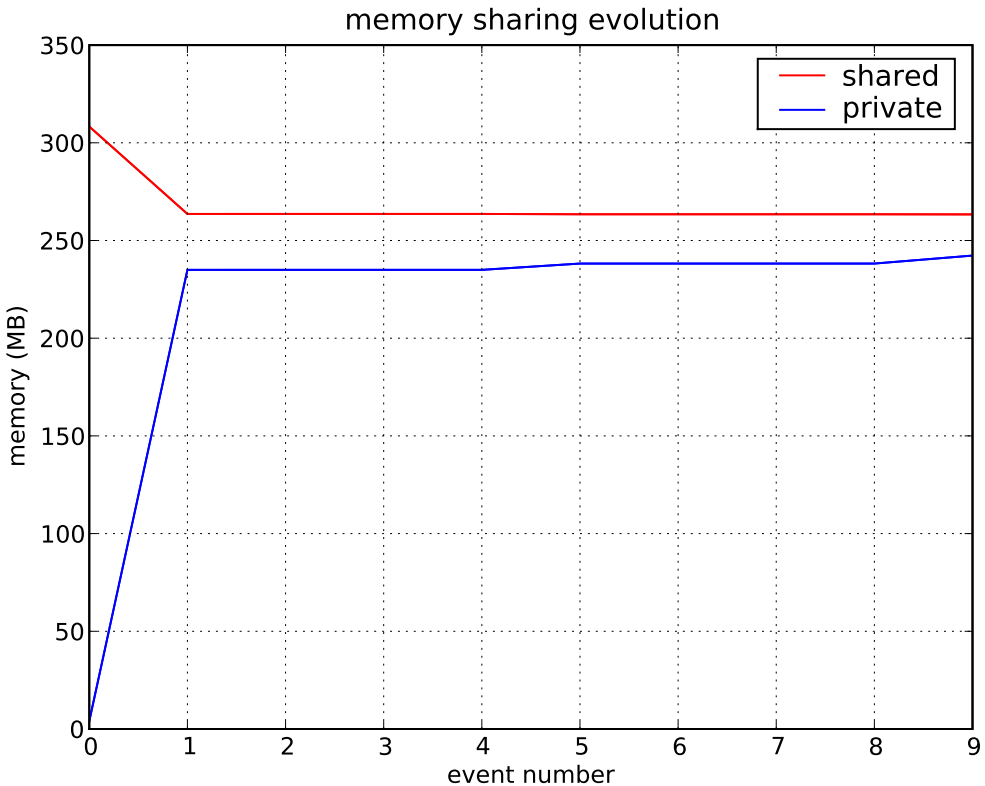


Résultats préliminaires - II

memory - RecExCommon/esdtoaod

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
```



Prospects

- développement d'un prompt interactif et *MP*
  - à la PROOF
  - réduction latence de l'analyse *'laptop'*
- déploiement sur la grille
  - simples utilisateurs
  - production ?
  - problèmes d'interoperabilité avec les *batch systems*

Plus de tests:

- backnavigation (*AOD* $\rightarrow$ *ESD* $\rightarrow$ *RDO*)
- physics plots
- random numbers reproduceability
- reproduceability
- debugging tools
- ...

Backup. . .

AthenaMP implementation

rational

- avoid client changes
- shove the MP-stuff inside Athena instead of putting it as a layer on top of it
- use the python module multiprocessing (backported from 2.6) for the process management
- write a new event loop manager as a usual Gaudi component to encapsulate the parallelism handling
- modify the I/O-related components appropriately

```
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)
        workers = mp.Pool (processes=self._ncpus,
                           initializer=self._worker_bootstrap)
        results = workers.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 (this is usually handled
   *  by calling @c IIoComponent::io_reinit on every registered
   *  component)
   */
  virtual
  StatusCode io_register (IIoComponent* iocomponent) = 0;

  /** @brief: reinitialize the I/O subsystem.
   *  This effectively calls @c 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 IIoComponent
{
  /** callback method to reinitialize the internal state of
   *  the component for I/O purposes (e.g. upon @c fork(2))
   */
  virtual
  StatusCode io_reinit () = 0;
};
```

- implemented by `THistSvc`, `AthenaPoolSvc`, ...
- reopen input ROOT files
- open output ROOT files
  - created in the worker's own directory
  - take care of migrating all the objects of *'already opened for writting'* ROOT files to the new ones

```
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)
        workers = mp.Pool (processes=self._ncpus,
                           initializer=self._worker_bootstrap)
        results = workers.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
    - ★ take care of POOL links/references
    - ★ actually just a few integers here and there to offset by the right amount
    - ★ needs some modifications in the `AthenaPOOL` layer to enable usage of the fast-merge mode (*à la* `hadd`)
    - ★ right now: pedestrian/manual approach (slower)
  - I wish there were a general `pool_merge` command !