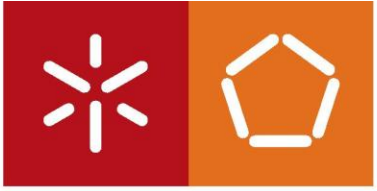
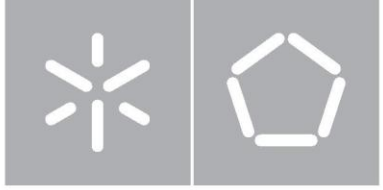


**Universidade do Minho**
Escola de Engenharia

Rafael Caldeira Silva

# Estudo de viabilidade de paralelização de códigos de análise de dados em PROOF

Outubro de 2014



**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Rafael Caldeira Silva

# Estudo de viabilidade de paralelização de códigos de análise de dados em PROOF

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

**Professor António Manual Pina**
**Professor Nuno Filipe da Silva Fernandes de Castro**

Outubro de 2014

*"A ciência sem a religião é manca,*
*a religião sem a ciência é cega."*

*Albert Einstein*

*"Feliz o homem que atinge a sabedoria;*
*feliz aquele que adquire inteligência"*

*Provérbios 3:13*

*À Graziela,*

# Agradecimentos

Antes de tudo, a Deus, pela força para continuar nos momentos de desânimo e pela orientação e provisão durante este percurso.

À Graziela, minha noiva, pela incansável paciência e compreensão, que mesmo distante ofereceu constante ajuda e suporte na escrita desta dissertação, te amo.

Aos meus pais, Roberto & Elizabete, pelo apoio incondicional e pelos bons conselhos, sempre.

Ao LIP-Minho, representados pelo Prof. António Onofre, Prof. Nuno Castro e Juanpe Araque, pela proposta de desenvolvimento deste trabalho, pelo apoio durante o percurso, pelos bons momentos passados e pelo financiamento do mesmo.

Ao meu orientador Prof. António Pina, e ao meu co-orientador Prof. Nuno Castro, pela paciêcia e orientação ao longo deste trabalho.

Ao Carlos Eduardo, meu irmão, pela eterna amizade e companheirismo de todos os momentos.

Ao meu grande amigo Pedro Santos, coordenador da *Revista PROGRAMAR*, pelos bons conselhos, e pelo incansável apoio. Obrigado pelo empréstimo Sara.

Aos meus amigos e companheiros de trabalho, André, Diogo e Samuel, pela amizade, pelos bons momentos, boas conversas e bons conselhos.

Ao Prof. Alberto Proença, pela orientação dada na fase inicial deste trabalho.E também em conjunto com o Prof. João Luís Sobral e o Prof. Rui Ralha, pela excelente formação anterior que foi de extrema valia durante este trabalho, e com certeza continuará sendo durante a minha carreira.

À comunidade do ROOT, em especial ao Dr Gerardo Ganis pelo excelente trabalho realizado no PROOF, e pelo tempo despendido a responder as minhas perguntas.

Aos meus grandes amigos "*patras*", adquiridos durante o meu percurso académico, pela amizade, pelas histórias que nunca serão esquecidas e pelo apoio desde que nos conhecemos.

# Resumo

Esta dissertação surge no contexto das análises de dados gerados pelo LHC (*Large Hadron Collider*), do esperado crescimento do volume de dados produzidos depois da atualização de 2013-2014 e do atual paradigma pseudo-paralelo destas aplicações no LIP-Minho (Laboratório de Instrumentação e física experimental de Partículas, delegação Minho).

O trabalho surgiu como um estudo da utilização do PROOF (*Parallel ROOT Facilities*) como plataforma para habilitar a extração automática de paralelismo nas aplicações de análises de dados do LIP-Minho.

Na consideração que as análises em estudo têm uma estrutura semelhante que é susceptível de ser paralelizada, partimos de um caso de estudo para a familiarização e experimentação do ambiente PROOF.

Face às dificuldades de adaptação da aplicação para utilização do sistema PROOF, desenvolvemos e testamos uma nova estrutura de classes, chamada *event*, que pode eliminar uma série de problemas na fase de desenvolvimento. Esta proposta é suportada por um gerador de código esqueleto de aplicações deste tipo, o *makeEvent*.

Os testes efetuados comprovam a possibilidade de usar a estrutura *event* como alternativa à API *TSelector*, sem perda de desempenho e com a possibilidade de alcançar *speedups* superlineares no ambiente de *cluster* utilizado.

No caso de códigos de análise de dados com alguma dimensão e complexidade, o processo de adaptação para um modelo compatível com o sistema PROOF pode ser uma tarefa morosa e exigente que pode não ser trivial. Por este motivo, propomos como trabalho futuro a criação de uma biblioteca que trate das tarefas habituais no processo de análise dos dados. Prevê-se também que a aplicação *makeEvent* permita a seleção apenas dos *branches* utilizados na classe *event*, reduzindo significativamente o tempo de execução de análises de dados que carregam desnecessariamente todos os *branches* de uma *tree*.

A conclusão a que chegamos é a da viabilidade da utilização da estrutura *event*, e consequentemente do *makeEvent*, como uma alternativa possível para a extração de paralelismo automático das análises de dados em estudo, recorrendo à plataforma PROOF.

# Abstract

This dissertation comes in the context of the analysis of data generated by the LHC (Large Hadron Collider), the expected growth of the produced data volume after the machine upgrade in 2013-2014 and the current pseudo-parallel paradigm of these applications at LIP-Minho (Laboratório de Instrumentação e física experimental de Partículas, Minho delegation).

The work came as a study of the use of PROOF (Parallel ROOT Facilities) as a platform to enable automatic extraction of parallelism in data analysis applications at LIP-Minho.

Knowing that the analysis in study have a similar structure that is capable of being parallelized, we start from a case study for familiarization and testing the PROOF environment.

Given the difficulties of porting the application to use the PROOF system, we developed and tested a new class structure, named event, which can eliminate a series of problems in the development phase. This proposal was supported by a generator of skeleton code of applications of this type, makeEvent.

The tests performed show the possibility of using the event structure as an alternative to *TSelector* API without loss of performance and with the possibility of reaching superlinear speedups in the cluster environment used.

In the case of data analysis with considerable size and complexity, the process of adaptation to a level compatible with the PROOF system can be a time consuming and demanding task, most likely non trivial. For this reason, we propose as future work to create a library that handles the common tasks in the data analysis process. It is also envisaged that the makeEvent application will allow one to select only the branches used in the event class, significantly reducing the execution time data analysis that needlessly load all the branches of a tree.

The conclusion we reached is the viability of using the event structure, and consequently the makeEvent, as a possible alternative for the automatic extraction of parallelism of data analysis in study, using the PROOF platform.

# Índice

# Lista de Figuras

# Lista de Tabelas

# 1 Introdução

## 1.1 O Percurso e o estudo dos dados do LHC

### 1.1.1 Acontecimentos em ATLAS

O LHC (Large Hadron Collider) construído no CERN (Organização Europeia para a Pesquisa Nuclear) na fronteira Franco-Suíça, a cerca de 200 metros de profundidade, é atualmente o maior acelerador de partículas à escala mundial. Construído para comprovar a validade do modelo padrão da física de partículas e abrir portas para a descoberta de novos fenômenos, consiste de um sistema de imãs supercondutores ao longo de um túnel circular com 27 km de perímetro.

No LHC, feixes de partículas com altas energias são acelerados a velocidades muito próximas da velocidade da luz até colidirem num dos gigantescos detectores, sendo os quatro principais: ATLAS, CMS, LHCb e ALICE. Estes feixes colidem cerca de 600 milhões de vezes por segundo, sendo o resultado de cada colisão chamado de acontecimento, ou em inglês *event*. Todas estas colisões geram um fluxo de dados de aproximadamente 1 PB/s por detector [1]. Além disto, é esperado que a quantidade de dados aumente em grandes proporções depois do *upgrade* de 2013-2014 [2].

Em 2013, a colaboração ATLAS (responsável por um dos detectores de partículas) armazenou cerca de 140 PB de informação, sendo espetável que a média anual, de 20PB [3], aumente numa ordem de grandeza, após a renovação do LHC, agendado para reabertura no início de 2015 [4].

Além dos dados reais captados pelos detectores é também necessário gerar e processar dados simulados, através do método de *Monte Carlo*, para poder estudar a resposta do detector a diferentes cenários e processos físicos [5]. A geração e simulação de acontecimentos baseda em métodos de *Monte Carlo* é computacionalmente intensiva.

### 1.1.2 Recursos computacionais

Para armazenar e processar os dados recolhidos pelos detectores do LHC são necessários recursos computacionais em grande escala habitualmente inexistentes numa única localização central [1]. A alternativa são as abordagens de computação distribuída. Foi por este motivo que, recorrendo a tecnologias de *grid* pré-existentes, foi criada a Worldwide LHC Computing Grid (WLCG) [1], que engloba múltiplos centros computacionais, espalhados geograficamente por todo o mundo [1]. Cada um destes centros computacionais, mantidos e geridos localmente, implementam uma mesma camada computacional, que facilita a administração da rede de comunicações e aumenta a tolerância a falhas (por ser uma solução distribuída) [1].

A WLCG está dividida em quatro camadas de recursos distintas: *Tier* 0, *Tier* 1, *Tier* 2 e *Tier* 3, responsáveis por armazenar os dados detectados pelos detectores, o resultado de reconstruções destes dados e o resultado de diversas análises e, ao mesmo tempo, disponibilizar toda esta informação à comunidade científica da WLCG.

### 1.1.3 Visão global do processamento de dados em ATLAS

O detector ATLAS capta aproximadamente 100 milhões de acontecimentos por segundo [6]. No entanto, a maior parte destes acontecimentos não são interessantes do ponto de vista da física de partículas. Por isso, foi implementado um sistema de *triggers* [7] que reduz o fluxo original para aproximadamente 200 acontecimentos por segundo [8].

Estes acontecimentos são armazenados no formato *Raw Data Object* (RDO) que contém o resultado das interações das partículas com as diversas camadas do detector são automaticamente armazenados no *Tier* 0. São também armazenados permanentemente no *Tier* 0 os ficheiros com o formato *Event Summary Data* (ESD) criados pelo primeiro passo de reconstrução.

Cada um dos doze centros computacionais que compõem o *Tier 1* é responsável por armazenar uma parte proporcional de uma cópia dos dados presentes no *Tier* 0 e por executar o passo seguinte do processamento dos dados que consiste em criar ficheiros em que a representação dos acontecimentos contém apenas o resultado da reconstrução. Esta conversão cria ficheiros num formato chamado Analysis Object Data (AOD). Estes centros 1 são também responsáveis pelo armazenamento de uma parte proporcional dos dados simulados gerados no *Tier 2*.

Os centros computacionais no *Tier 2* são responsáveis pela geração de acontecimentos simulados, execução de análises e armazenamento local dos dados necessários obtidos dos centros computacionais *Tier 1*.

As instalações *Tier* 3 são centros computacionais locais ou mesmo computadores pessoais utilizados para aceder à WLCG [1], [9], [10].

Um caso particular de *Tier* 3 é o *cluster* lip.di.uminho.pt, do pólo LIP-Minho, do Laboratório de Instrumentação e Física Experimental de Partículas (LIP) baseado em *Scientific Linux 5.7* e *Rocks 5.4.3,* que utiliza o PBS (*Portable Batch System*) como escalonador de trabalhos.

### 1.1.4 Processamento de dados no LIP-Minho

No LIP-Minho, os dados são inicialmente copiados para um sistema local no formato disponibilizado pelo sistema ROOT [11] e distribuídos por múltiplos ficheiros de tamanhos variados (totalizando algumas dezenas de *terabytes*) [12], [13]. Cada ficheiro

é constituído por uma estrutura de dados chamada TTree, que contém dados que descrevem um conjunto de acontecimentos, reais ou simulados, todos no mesmo formato.

### 1.1.5 O sistema ROOT

O sistema ROOT consiste num conjunto de *frameworks* desenvolvidas no CERN para suportar eficientemente a análise e o armazenamento de dados, numa escala de *petabytes*. Algumas das principais funcionalidades disponibilizadas pelo sistema aos utilizadores são enumeradas e explicadas de forma sucinta, abaixo:

1. Um sistema interativo de interpretação, compilação e execução de código C++ em tempo real. Baseando-se no CINT [14], ou a partir da versão 6 no *cling*[15], o ROOT é capaz de interpretar e executar código fonte C++. No caso do CINT, o código carregado em *runtime* é armazenado em dicionários, que são gerados automaticamente. Além disto, também é possível criar bibliotecas dinâmicas com um compilador e usar utilitários como *rootcint* e *makecint* para criar os dicionários e transformá-los em bibliotecas de código, acessíveis a partir do sistema ROOT; este processo pode ser automatizado através do ACLiC (*Automatic Compiler of Libraries for CINT*). No caso do *cling*, é utilizado um mecanismo de compilação *just-in-time* baseado nas bibliotecas LLVM.
2. Uma biblioteca de gestão de Entrada/Saída de dados capaz de armazenar e recuperar virtualmente qualquer objeto que o sistema suporte, ou possa vir a suportar em *runtime*, num ficheiro com formato próprio do ROOT com possibilidade de compressão automática em disco, independente da plataforma de execução.
3. Estruturas de dados desenvolvidas e optimizadas a pensar no armazenamento e análise de dados do LHC, tal como a classe *TFile*, capaz de tratar ficheiros dispersos em sistemas de ficheiros locais, partilhados, ou remotos recorrendo a uma série de *plugins* fornecidos com a *framework* ou passíveis de serem adicionados pelo utilizador, capazes de obter o ficheiro em tempo real. Ou ainda a estrutura de dados TTree, desenvolvida para armazenamento e análise estatística de grandes volumes de dados.
4. Bibliotecas de linguagem C++ que disponibilizam diversas funções matemáticas, estatísticas e físicas, incluindo funções de cálculo, minimização e ajustamento de dados, que são de extrema importância para o trabalho em causa, nomeadamente no uso para fins de investigação na comunidade científica de física.
5. Uma *framework* para criação, gestão, visualização e análise de histogramas de dados, com uma ou mais dimensões, que podem ser facilmente exportados diretamente para diversos formatos de imagem, tais como o PDF (*Portable Document Format*) e o PS (PostScript), ou ainda salvaguardados num formato

próprio, suportado pela *framework* ROOT, para posterior reanálise dos dados.

6. O sistema *Parallel ROOT Facilities* (PROOF) que tira partido da computação distribuída para o aproveitamento dos recursos computacionais disponíveis, minimizando o tempo total de execução das análises.

A Figura 1.1 ilustra o modelo computacional descrito acima, desde os dados originais, passando pelas várias fases de processamento na WLCG, pelas análises de dados suportadas pelo ROOT e, por fim, pela publicação dos resultados.

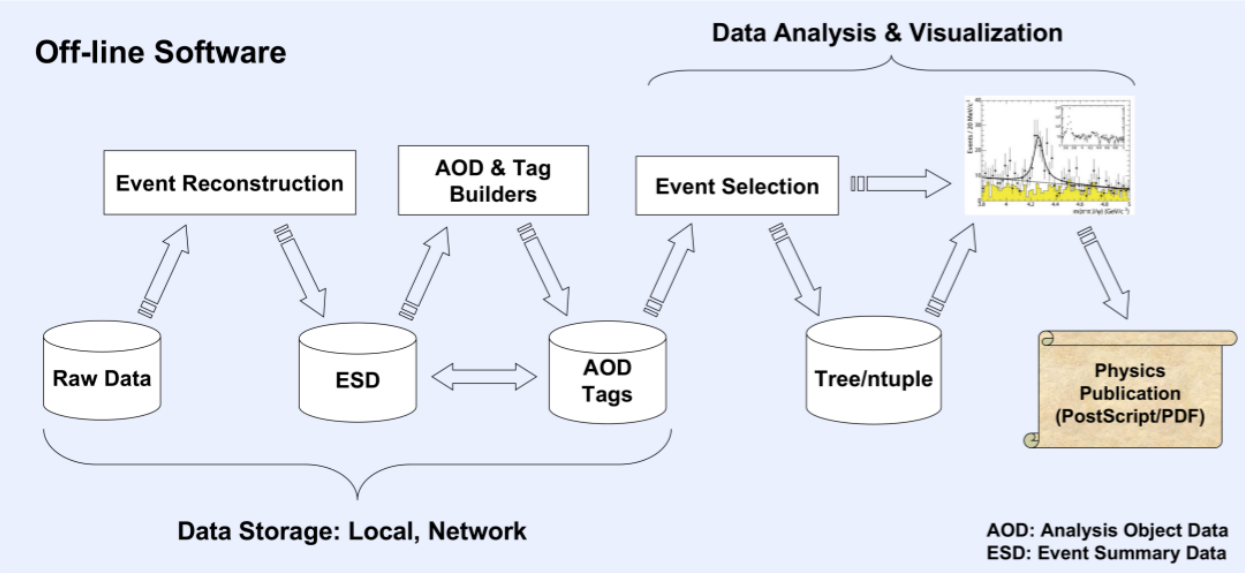



*Figura 1.1 Modelo de processamento de dados do LHC (fonte [35])*

### 1.1.6 A estrutura de dados TTree

Para usar o ROOT em análise de dados é necessário compreender os conceitos fundamentais relacionados com as *trees* e a respetiva implementação no sistema, também chamada TTree. Outros conceitos e termos relacionados são ilustrados na Figura 1.2.

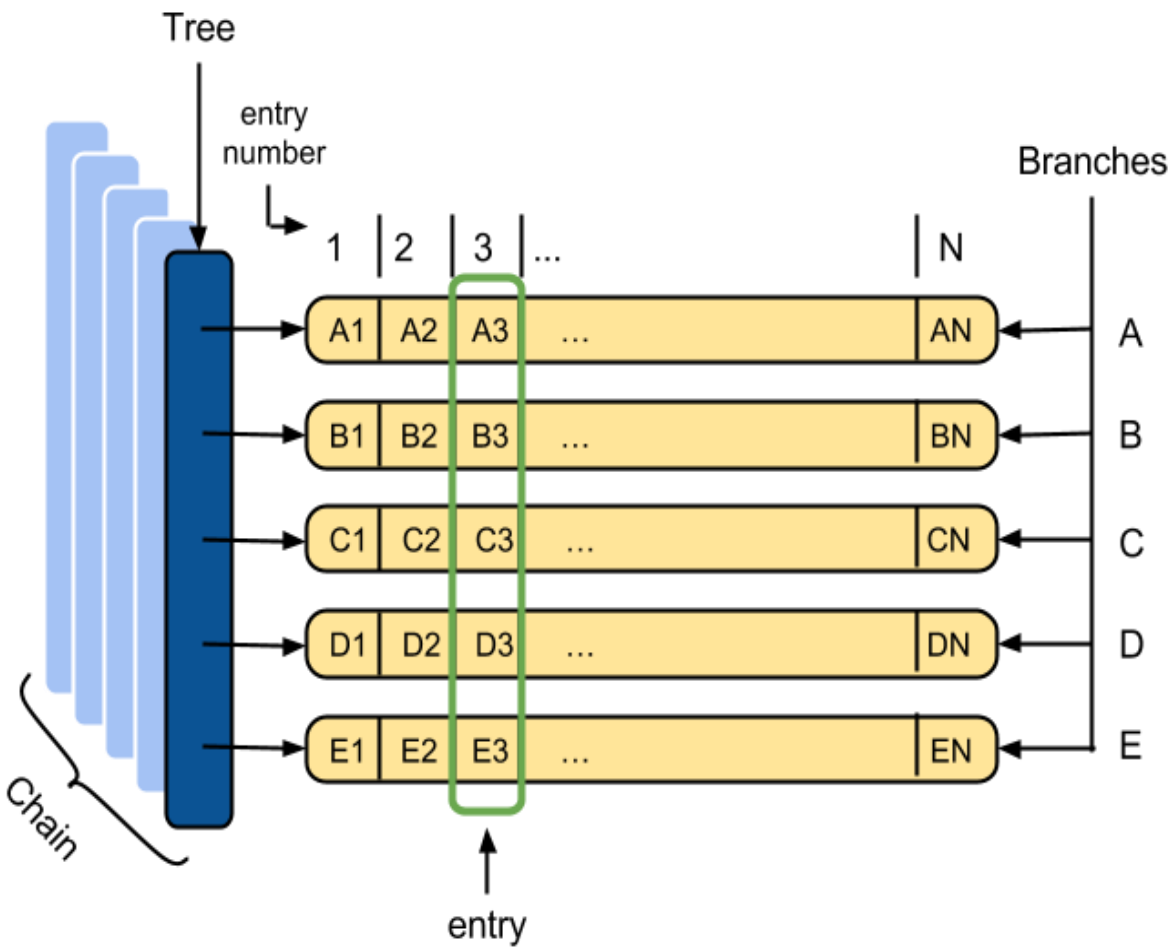



*Figura 1.2 Principais conceitos associados à estrutura de dados TTree*

- *Branch* é uma lista de elementos com a mesma estrutura, sempre como parte de uma *tree*.

- *Tree* é uma lista de *branches* independentes.

- *Chain* é uma estrutura de dados temporária (armazenada em RAM) que permite ao utilizador manipular um conjunto de *trees* com a mesma estrutura.

- *Entry* é um conjunto de propriedades (cada propriedade é armazenada num *Branch* a que corresponde) associada a um acontecimento (*event*). Por exemplo, na figura 1.2 estão representadas N *entrys* (1,2,3...N), cada *entry* possui 5 propriedades (A, B, C, D e E). Assim, podemos dizer que a *entry* número 3, tem “D3” como valor de D.

### 1.1.7 Análises

Em cada análise é necessário processar vários ficheiros ROOT com o objetivo

de extrair dados de saída que permitam chegar a conclusões relevantes. Os tipos de dados de saída podem ser variados, mas o mais habitual é serem valores numéricos, histogramas ou outras *trees*. Por norma a quantidade de dados de saída é muito inferior aos dados de entrada.

As análises de dados são aplicações que descrevem o conjunto de dados que devem ser aceites e o processamento correspondente. Por isso, em geral, é possível identificar no código das análises a seguinte estrutura:

➔ Inicialização: criação das estruturas de dados necessárias para a leitura e armazenamento dos dados de entrada e a correspondente iniciação dos componentes específicos de cada análise.

➔ Processamento: iteração sobre todos os acontecimentos para extrair os dados de saída, o que corresponde normalmente a "filtrar" os acontecimentos em vários níveis de corte, cada vez mais restritivos. O contributo de cada acontecimento para os dados de saída é uma função do número de cortes que atravessou.

➔ Finalização: corresponde ao processamento dos dados de saída, sendo que normalmente os dados são armazenados em disco para futura visualização.

### 1.1.8 Geração do código das análises

Partindo do princípio que as aplicações de análise têm uma estrutura semelhante, o sistema ROOT oferece métodos para gerar automaticamente um código esqueleto para as análises, tendo como base a estrutura dos ficheiros que serão processados, isto é: as estruturas de dados e as manipulações habituais. Assim, parte da tarefa monótona de escrever o código necessário para manipulação das estruturas de dados, é substituída pela evocação dos métodos: *TTree::MakeClass*, *TTree::MakeSelector*.

A título de exemplo, os ficheiros com código esqueleto podem ser gerados através dos seguintes comandos da interface de linha do ROOT:

```
1. $ root -l <filename>.root
2. root[] TTree *t = (TTree *)_file0->Get("tree_name");
3. root[] t->MakeClass();
4. root[] .q
```

O método *TTree::MakeClass* permite gerar uma classe genérica capaz de processar uma *TTree* (ou *TChain*). A classe gerada contém os habituais métodos construtores e destrutores, usados pelas classes em C++, e ainda os métodos *Init*, *GetEntry* e Loop. A função destes métodos é resumidamente o seguinte:

➔ O construtor e o destrutor iniciam e terminam, respectivamente, a análise e que podem ser implementados pelo utilizador.

➔ O método *Init* é chamado para inicializar uma *tree*, isto é: inicializar a zero os objetos que armazenam os elementos (folhas) de uma *entry*, associar cada *branch* da *tree* a um objecto que o represente. Este método não carece de alterações do utilizador.

➔ O método *GetEntry* é um *wrapper* para o método GetEntry com o mesmo nome na TTree correspondente. Este método permite ler do ficheiro de entrada de dados a *entry* correspondente e escrever os dados nas variáveis associadas. Este método não carece de alterações do utilizador.

➔ Finalmente, o método *Loop* itera sobre todas as *entries*, chamando o *GetEntry* em cada passo e, seguidamente, faz o processamento do acontecimento. Este método tem que ser completado pelo utilizador com o código específico de processamento de cada análise.

O método *TTree::MakeSelector* é equivalente ao *TTree::MakeClass*, mas gera código derivado da classe *TSelector* [16], o que vai permitir a execução da análise utilizando o ambiente de execução paralelo do PROOF [17]. Os métodos principais da nova classe são apresentados a seguir:

➔ *Init*: usado para a inicialização dos *TBranchs*, é chamado quando é necessário inicializar uma TTree.

➔ *SlaveBegin*: usado para inicializações, é chamado antes da iteração sobre as *entries*, em cada processo trabalhador (*worker*), inicializado anteriormente.

➔ *Notify*: chamado nos processos trabalhadores antes da execução da primeira *entry* de cada *tree* de uma *chain*.

➔ *Process*: usado nos processos trabalhadores para carregar os dados através do método *GetEntry* e seguidamente processá-los. É chamado uma vez para cada *entry*/acontecimento, que é carregado.

➔ *SlaveTerminate*: chamado antes da finalização de cada processo trabalhador (*worker*).

➔ *Terminate*: chamado uma única vez no processo cliente, depois da junção, executada pelo processo *master,* das saídas produzidas por cada um dos processos trabalhadores. Normalmente é utilizado para armazenar permanentemente os resultados da análise em disco, para futuras utilizações.

Estão disponíveis outras alternativas para análises de dados baseadas em

sessões interativas, nomeadamente, o método *TTree::MakeCode* (obsoleto), e *TTree::MakeProxy*, além do *TTree::Draw* que permite a seleção e visualização de variáveis interativamente.

O método *TTree::MakeCode* gera uma macro que pode ser interpretada pelo ROOT (utilizando o CINT), mas como não é compatível com compiladores de C/C++ é considerada obsoleta (o seu uso não é recomendado).

## 1.2 Paralelismo

Nos últimos anos a evolução dos processadores foi confrontada com as limitações ao aumento da frequência do ciclo do relógio. Com efeito, cada vez que se aumenta a frequência do relógio, cresce o consumo de energia e o aquecimento aumenta de forma proporcional, o que sugere a aproximação de limites físicos [18].

Assim, como resultante do melhoramento do processo de fabrico dos integrados, em consonância com a lei de Moore que estabelece que o número de transístores duplica cada dois anos[19], [20], a resposta no mercado dos processadores passou por introduzir mais processadores no mesmo *chip* (multi-núcleo), aumentando a capacidade de processamento do *chip*, sem sofrer os problemas de eficiência energética e controlo de temperatura associados ao aumento da frequência do ciclo de relógio dos processadores “convencionais”. Esta alternativa engenhosa, de aumentar o desempenho do processamento, pela via do paralelismo por *hardware* tem vindo a impor-se tanto no mercado doméstico, como no das máquinas de elevada exigência.

O paralelismo oferece a grande vantagem de reduzir o tempo de processamento de grandes volumes de dados e cálculos matemáticos complexos, pela via do processamento paralelo com recurso a múltiplos nós, com um ou mais processadores por nó[21].

O tempo de processamento das análises é normalmente proporcional à quantidade de dados de entrada, podendo tornar-se um factor limitativo em análises computacionalmente intensivas ou que lidam com grandes volumes de dados. Assim considerando a consolidação das tecnologias associadas ao paralelismo ao longo dos últimos anos e a existência de sistemas de computação paralela acessíveis à comunidade científica, pode-se considerar inevitável o recurso ao paralelismo para minimizar o tempo consumido no processamento de dados.

A Figura 1.3 ilustra a redução no tempo total de execução através da utilização de paralelismo.

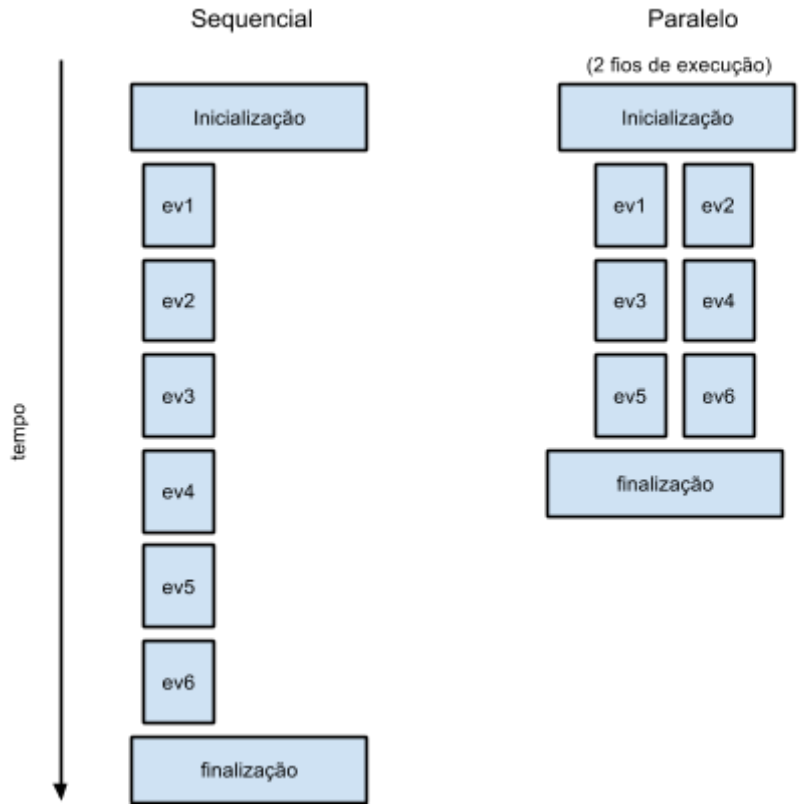



*Figura 1.3 Diagrama ilustrativo das diferenças entre o processamento sequencial e paralelo*

Tendo em conta que os dados analisados são, por definição, um conjunto de acontecimentos independentes, o processamento de cada acontecimento pode, em teoria, ser feito de forma independente, sendo apenas necessário que ao findar da aplicação o resultado contenha a contribuição, mesmo que nula, de todos os acontecimentos processados. Este é um tipo de problema conhecido como embaraçosamente paralelo e que consiste normalmente na divisão de trabalhos entre todas as entidades de processamento disponíveis, no processamento independente do trabalho enviado a cada entidade e, por fim, na junção dos resultados para obtenção do resultado final.

### 1.2.1 Limites do processamento paralelo

Quando um mesmo problema tem a oportunidade de ser executado com o dobro dos recursos computacionais é expectável que o tempo de execução total seja reduzido para a metade. No entanto, isto só é verdade em casos muito específicos explicados mais à frente.

Segundo a lei de Amdahl [22], o tempo total de execução T, utilizando n entidades de processamento, pode ser reduzido para o limite teórico $\mathrm{T}^{(n)}$, dependendo da fracção B do problema que não pode ser resolvida em paralelo, de acordo com a

seguinte equação:

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

*Equação 1. Tempo total de execução, segundo a lei de Amdahl*

Não obstante à melhoria máxima teórica prevista pela Lei de Amdahl, o ganho oferecido pelo paralelismo pode ser limitado por vários outros factores. Um dos principais factores limitativos do ganho de desempenho através do paralelismo é a largura de banda de acesso aos dados, uma vez que a capacidade de cálculo dos processadores atuais supera habitualmente a largura de banda de carregamento dos dados, quer estejam na memória, em discos ou na rede.

O desfasamento da velocidade do processador em comparação com os dispositivos de armazenamento é tão grande que em muitos tipos de problema não é possível manter as unidades de processamento ocupadas durante todo o tempo, porque mesmo com grandes optimizações nos padrões de acesso à memória, com intuito de maximizar a utilização dos vários níveis de memória cache, e com esforços para maximizar a largura de banda dos sistemas de discos rígidos, os processadores acabam por passar a maior parte do tempo ociosos à espera de receber dados para processar.

A situação é agravada quando os acesso aos dados estão em sistemas de armazenamento permanente em discos magnéticos comuns (tais como do tipo IDE, SCSI, SAS ou SATA) mesmo quando ligados em RAID. A investigação em entrada/saída de dados tem vindo a proporcionar avanços significativos nos sistemas de armazenamento de informação [23].

Aos factores acima referidos deve-se acrescentar a sobrecarga em tempo de processamento resultante da execução de código associado à gestão do paralelismo que pode ser tanto mais relevante quanto menor for a granularidade do paralelismo. Por outras palavras, devem ser tidas em conta as possíveis perdas resultantes da implementação do paralelismo, ou seja, na generalidade dos casos é preferível aumentar o grão da secção paralela, por oposição ao paralelismo de grão-fino [24], [25]. Outro factores relevantes surgem quando a zona paralela contém sincronizações, seja por limitações da implementação, ou por requisitos do problema que vêm a traduzir-se em “engarrafamentos” no desempenho, impedindo assim a obtenção de resultados mais próximos dos que seriam os teoricamente expectáveis, segundo a lei de Amdahl.

### 1.2.2 Processamento paralelo no LIP-Minho

Com o intuito de utilizar ao máximo os recursos nos *clusters* computacionais

disponíveis, e tendo em conta que atualmente as análises não têm mecanismos próprios para a execução de código em paralelo ao nível do acontecimento, tem-se recorrido à paralelização através de gestores de recurso de processamentos em lote do tipo do *Portable Batch* System (PBS) [26] que permitem o lançamento em paralelo de múltiplas aplicações sequenciais. No processamento em lotes, os dados de entrada são divididos em subconjuntos menores e criadas múltiplas tarefas para processar cada um daqueles subconjuntos de dados, sendo depois necessária uma fase de junção dos vários resultados parciais numa saída final.

A grande desvantagem desta abordagem é a eventual divisão não balanceada de trabalho entre as tarefas, tanto a nível da quantidade de dados quanto ao nível do interesse do acontecimento (o que dita a quantidade de instruções). Esta divisão não balanceada pode resultar num desfasamento nos tempos de processamento das várias tarefas e consequentemente na não utilização da totalidade dos recursos computacionais disponíveis.

### 1.2.3 Mecanismos paralelos do ROOT

Tendo em conta a necessidade de paralelismo ao executar análises, foi desenvolvida uma extensão ao ROOT, chamada PROOF, que disponibiliza um conjunto de ferramentas e bibliotecas para a criação de um ambiente paralelo otimizado para execução de análises. O PROOF (*Parallel ROOT Facilities*) [16], [17] foi construído a pensar em três objetivos principais:

➔ **Transparência:** os códigos de análise desenhados para serem executados em modo sequencial no ROOT deverão sofrer o mínimo possível de alterações para serem convertidos para o PROOF.

➔ **Escalabilidade:** o modelo de execução não deve impor nenhuma restrição relativa ao número de unidades de processamento utilizadas para processar a análise.

➔ **Adaptabilidade:** a capacidade para se adaptar a diversos ambientes distribuídos, tendo em conta as unidades de processamento disponíveis, as falhas de rede, etc.

O PROOF implementa uma arquitetura *client*, *multi-master*, *slave* que obedece ao esquema que abaixo se reproduz na Figura 1.4:

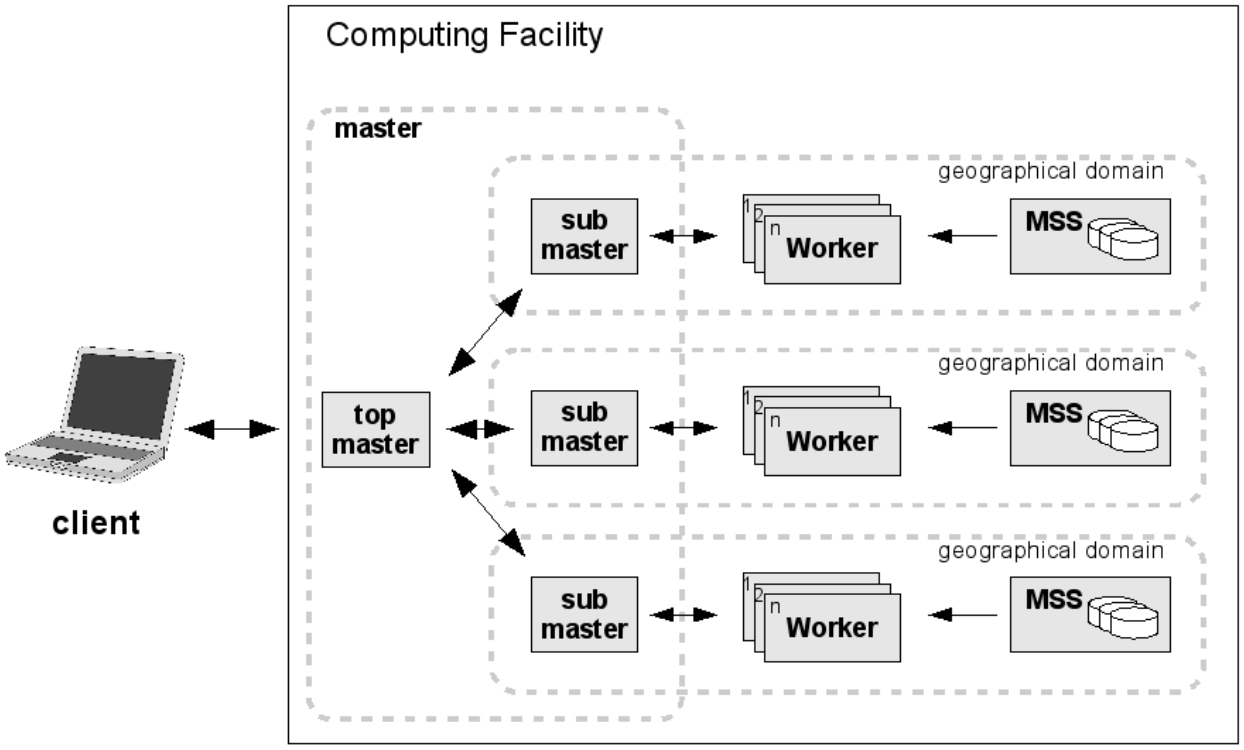



*Figura 1.4 Visão geral da arquitetura do PROOF: "Multi-Tier Master-Worker" (fonte: http://root.cern.ch/drupal/content/multi-tier-master-worker-architecture)*

Na Figura 1.4 o cliente (*client*) corresponde à sessão ROOT iniciada pelo utilizador que iniciou o processamento, o mestre (*master*) é o processo responsável por distribuir o trabalho entre os vários trabalhadores (*workers*) e os trabalhadores são os responsáveis pelo processamento dos acontecimentos. É assumido que os trabalhadores têm acesso aos ficheiros de entrada através da URL do ficheiro.

A utilização do PROOF passa pela implementação de um algoritmo utilizando a *framework TSelector*. O PROOF é o responsável pelo escalonamento do *selector*, quer seja guiado por dados ou simplesmente baseado em iterações independentes.

No presente trabalho a ênfase é posta no processamento guiado por dados, através de uma *TChain*. Neste caso, o utilizador pode escolher entre três maneiras diferentes de se processar um *TSelector*:

➔ Nome: passar o nome de uma classe derivada do *TSelector* já conhecida pelo sistema ROOT. O método correspondente da API da *TChain* tem a seguinte assinatura:

```
Long64_t Process(const char *selector, Option_t *option = "",
Long64_t nentries = -1, Long64_t firstentry = 0)
```

➔ Ficheiro: implementa uma classe derivada do *TSelector*, com o mesmo

nome de ficheiro. Esta funcionalidade também está disponível no método anterior.

➔ Objeto: o utilizador passa uma instância de uma classe derivada do *TSelector*, sendo o PROOF responsável por serializar a classe e enviá-la aos processos relevantes. O método implementado na API da *TChain* tem a seguinte assinatura:

```
Long64_t Process(TSelector *selector, Option_t *option = "",
   Long64_t nentries = -1, Long64_t firstentry = 0)
```

Qualquer classe derivada da classe *TSelector* pode ser executada utilizando o PROOF, no entanto, apenas classes cientes do modelo de memória distribuída na sua implementação da API do *TSelector* irão produzir os resultados corretos. Ou seja, é necessário que o utilizador esteja consciente de que os métodos que são executados nos processos trabalhadores não têm acesso imediato às modificações feitas pelo método *Begin*, assim como, o método *Terminate* não tem acesso aos resultados computacionais dos métodos que não foram executados no cliente.

Para que o utilizador não tenha que se preocupar com estas transações de dados, o PROOF disponibiliza a *OutputList*, onde o utilizador deve registar o endereço das variáveis que devem ser enviadas para o cliente, ficando disponível antes da execução do método *Terminate*. Este processo é feito utilizando o sistema de serialização do ROOT.

Assim como a *OutputList*, também existe a *InputList*, que permite ao utilizador enviar objetos criados no cliente, no método *Begin*, para os nós trabalhadores. No entanto, são raros os casos que justifiquem o uso deste mecanismo, uma vez que, sendo as inicializações feitas nos nós trabalhadores, poupa-se o tempo de serialização, transferência e reconstrução dos dados.

Depois dos dados de saída estarem disponíveis no cliente, este possui uma instância de cada objeto presente na *OutputList*, por cada nó trabalhador, sendo necessário juntá-las num único objeto, para garantir a consistência dos dados. O processo de junção é efetuado pelo PROOF assumindo que todos as classes têm um método com a seguinte assinatura:

```
Long64_t Merge(TCollection *);
```

Este método está implementado na maioria das classes disponibilizadas pelo ROOT.

Ao utilizar o PROOF, apenas os objetos que estiverem disponíveis no sistema ROOT, em *runtime*, e tiverem sido preparados para serem enviados para os processos trabalhadores podem ser utilizados nos trabalhadores.

No caso de o algoritmo consistir apenas num conjunto de código fonte e ficheiro *header* o processamento pode ser feito utilizando os ficheiros com o código fonte. De modo geral, esta solução não é suficiente porque algoritmos mais complexos são convenientemente implementados utilizando mais ficheiros e até mesmo bibliotecas externas. Nestes casos é necessário utilizar um ficheiro PAR (PROOF ARchive) que consiste num arquivo *tar* com uma estrutura própria que encapsula o código e os mecanismos de compilação e carregamento que os *workers* necessitam para efetuar todo o processamento dos dados de uma análise.

### 1.2.4 Ambientes de execução do PROOF

A *framework* PROOF dispõe de dois ambientes de execução diferentes: o PROOF lite ou um *cluster* computacional habilitado a receber cargas de trabalho do PROOF.

O PROOF Lite é destinado a uso em computadores pessoais e estações de trabalho. Quando uma sessão PROOF-Lite é criada, o PROOF inicia um processo trabalhador para cada processador disponível, sendo o processo que iniciou a sessão quem irá executar o papel de cliente e mestre. No entanto, normalmente é desejável utilizar um *cluster* computacional para processamento dos dados. Neste caso, cada nó do cluster deve ser configurado para receber dados e trabalhos através dos protocolos escolhidos pela *framework* PROOF.

Reconhecendo a morosidade da tarefa de configuração de um *cluster* para este aceitar trabalho através do PROOF, o centro de pesquisa GSI desenvolveu o *PROOF on Demand* (PoD), originalmente conhecido como gLitePROOF, que utiliza um gestor de recursos já disponível no *cluster* para dinamicamente criar nós trabalhadores, com pouca ou nenhuma configuração por parte do utilizador.

Atualmente o PoD é capaz de criar nós trabalhadores utilizando *plugins* que comunicam com os seguintes gestores de recursos: LSF (*Load Sharing Facility*), PBS Pro/OpenPBS/Torque (*Portable Batch System*), *Grid Engine* (*Oracle/Sun Grid Engine*), Condor, LoadLeveler (*IBM Tivoli Workload Scheduler LoadLeveler*) e gLite; além destes, também é possível utilizar o *plugin* de SSH (Secure Shell).

Uma das grandes vantagens da utilização do PoD é a possibilidade do utilizador não necessitar de privilégios de administrador para instalar, configurar ou utilizar o *cluster* com o PROOF, dando autonomia ao utilizador. O PoD tem tido um papel fundamental em novas tecnologias baseadas em PROOF, tal como a criação de um PROOF *cluster* “instantâneo” utilizando serviços de *cloud* [27].

### 1.2.5 PROOF *benchmarking*

O ROOT disponibiliza testes de *benchmarking* para o PROOF [28], [29], encapsulados na classe *TProofBench,* desenvolvida por Sangsu Ryu, que medem a capacidade de processamento do cluster em uma sessão PROOF (PROOF-lite ou PoD). O TProofBench disponibiliza dois testes principais; o teste *cycle-driven* utiliza uma análise padrão que cria 16 histogramas de uma dimensão com 30000 números aleatórios para cada processo trabalhador, que visa testar as capacidades de resposta do sistema a nível de processamento de dados; o teste *data-driven* baseia na leitura de *trees* com a estrutura Event (disponível nos exemplos do ROOT) com o objetivo de testar a largura de banda de acesso aos ficheiros. Os resultados destes testes de *bechmarking* no *cluster* lip.di.uminho.pt estão reproduzidos na Figura 1.5.

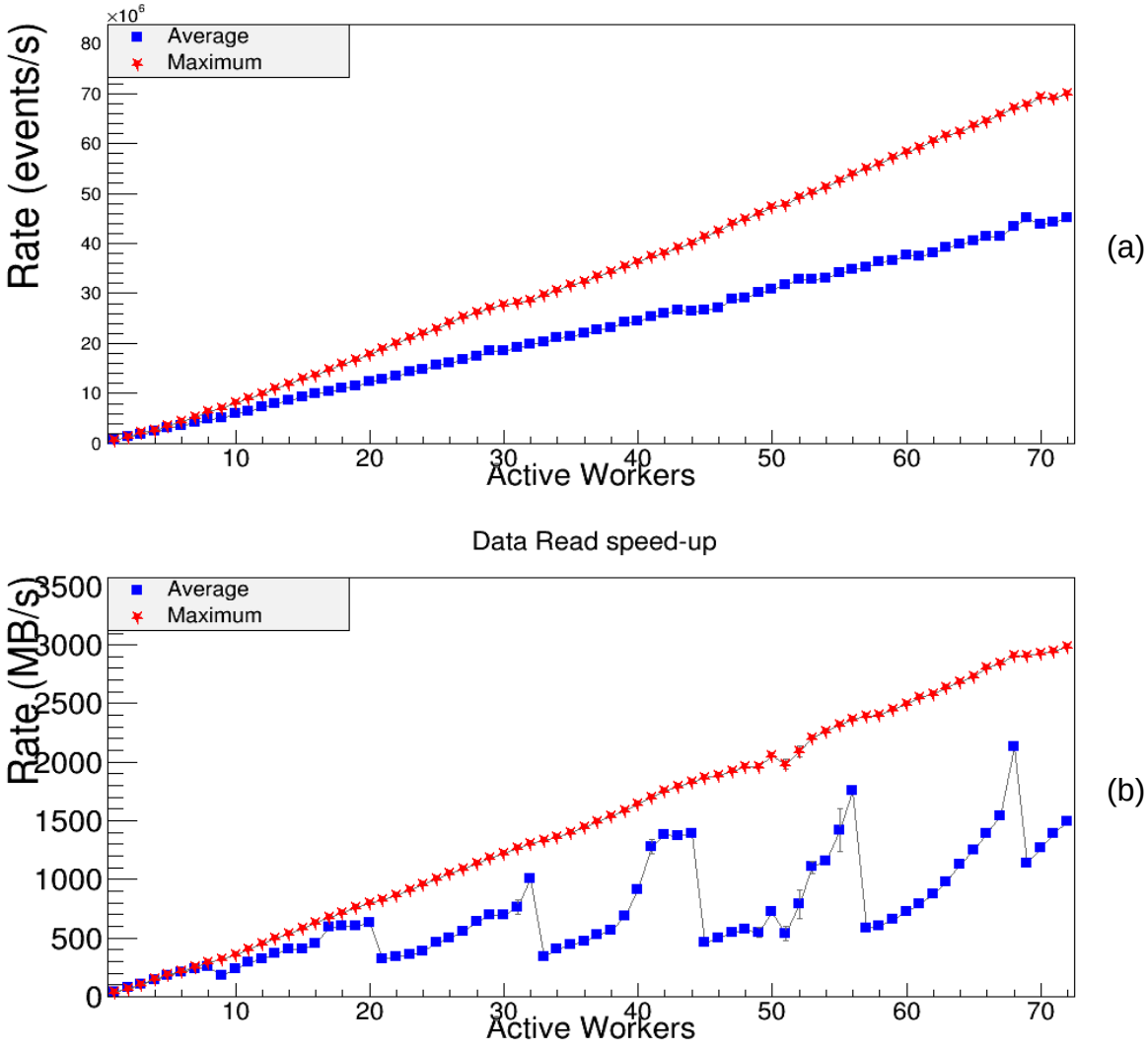



*Figura 1.5 Resultados da execução dos testes TProofBench no cluster lip.di.uminho.pt, utilizando 72 processos trabalhadores nos nós compute-0-{0,3,5,6,9,10,11,12}. a) Teste "cycle-driven" b) Teste "data-driven"*

O primeiro teste mostra que o PROOF não demonstra constrangimentos de

escalabilidade ao nível do processamento de dados e análises maioritariamente computacionais, sendo a quantidade de acontecimentos processada por segundo diretamente proporcional à quantidade de processos trabalhadores ativos. O segundo teste demonstra que a largura de banda do sistema de ficheiros não é sempre proporcional à quantidade de processos trabalhadores ativos, notando-se um efeito de altos e baixos na taxa média de leitura, provavelmente derivado dos mecanismos de *caching* do sistema de armazenamento; este efeito não existe nos melhores casos.

## 1.3 Motivação, objetivos e metodologia

A evolução dos sistemas computacionais pessoais e científicos, que já não se tornam mais rápidos apenas porque a CPU é capaz de fazer mais coisas na mesma unidade de tempo (aumento da frequência), conduziu a que os sistemas tenham cada vez mais núcleos de processamento, mais processadores, e no caso das comunidades de investigação, mais computadores a funcionar em conjunto em *commodity clusters*, ou supercomputadores de alta gama.

Tendo em conta a enorme quantidade de dados habitualmente processada, as análises de dados poderão ser limitadas pela largura de banda de acesso aos dados de entrada e saída (*I/O bound*). Esta limitação, especialmente em aplicações não paralelas, pode tomar proporções que tornem a pertinência dos resultados questionável. Por isso é importante utilizar um sistema de processamento capaz de optimizar esta largura de banda, o que, na maioria dos casos, implica a utilização eficiente de vários sistemas de armazenamento em simultâneo.

Considerando que os utilizadores já estão familiarizados com as *frameworks* do sistema ROOT e que esta se apresenta como uma solução especificamente desenhada e otimizada para a paralelização de aplicações baseadas em processamento de acontecimentos embaraçosamente paralelos, e considerando também os mais de dez anos de desenvolvimento do PROOF[16] e a comunidade científica que oferece suporte ao mesmo, o PROOF foi escolhido como ferramenta de paralelização para este trabalho.

### 1.3.1 Objetivos

Na presente dissertação apresenta-se um estudo sobre a utilização do PROOF para habilitar o paralelismo na execução de aplicações de análise de dados, provenientes do detector ATLAS, ao nível do acontecimento, tendo como objetivo a minimização do tempo total de execução da aplicação.

Pretende-se estudar a utilização do PROOF com vista a criar as condições para automatização do processo de paralelização de análises do LIP.

### 1.3.2 Metodologia

O problema foi abordado de uma maneira sistemática e prática. Assim, escolheu-se uma análise simples para servir de estudo de caso e ajudar com a familiarização do ambiente ROOT e da sua *framework* paralela, o PROOF.

Esta análise foi adaptada para utilizar a API da classe *TSelector*, de modo a tornar possível o funcionamento com o ambiente de execução do PROOF, tendo sido necessário verificar a consistência dos resultados finais em comparação com os resultados da análise original, já que o *standard output* normal do programa é inevitavelmente alterado pelo PROOF.

Depois dos resultados verificados, é desejável verificar a escalabilidade da análise, utilizando o ambiente de execução do PROOF, verificando se a hipótese inicial é válida, ou seja, se o PROOF se adapta e é tão escalável quanto o verificado nos testes do *TProofBench*. De acordo com os resultados de desempenho deste teste e com a experiência obtida com o PROOF, será proposta uma forma de integração do PROOF que seja mais facilmente adaptada nas análises mais complexas.

Os dados de entrada utilizados foram ficheiros ROOT que continham *trees* com uma estrutura chamada “mini”, a mesma utilizada nas pesquisas de novos quarks vectoriais em topologias dileptónicas [30]. Os ficheiros ficam armazenados num repositório acessível por NFS (*Network File System*) por todos os nós de computação. Os ficheiros têm tamanhos variados (desde alguns KB até alguns GB). Para ter um sistema de testes uniforme os ficheiros foram agrupados em grupos com tamanhos aproximados aos múltiplos de 5 GB. Cada grupo de ficheiro ficou representado num ficheiro de texto que contém o caminho absoluto para todos os ficheiros deste grupo. O agrupamento dos ficheiros, que é um problema conhecido como *bin-packing*, foi feito utilizando uma solução publicada em [31].

Os tempos de execução foram medidos com o comando *time* da *bash* (*Bourne Again SHell*) que mede o tempo de execução utilizado pelo processo em modo utilizador, o tempo das chamadas ao sistema deste processo e o tempo total de execução. Só foi considerado o tempo total de execução. Este temporizador utiliza a precisão de micro-segundos, que é o máximo disponível para temporizadores baseados em *software* na versão do *kernel* instalado no ambiente de testes. De modo a evitar anomalias derivados de factores externos, nomeadamente alterações no tempo de execução derivadas da interação com outros processos a serem executados em simultâneo, interações com o sistema operativo, interações com dispositivos de *hardware*, entre outras, foi considerada a mediana de 5 execuções, para cada medida.

Feitas estas escolhas seguiu-se para a paralelização da análise escolhida para caso de estudo, utilizando o PROOF.

# 2 Estruturas do código de uma análise

## 2.1 Versões de código

### 2.1.1 Versão original

*VLQValidation* é uma aplicação de análise, cujo código foi desenvolvido a partir do código esqueleto gerado pelo método *TTree::MakeClass*. A análise compreende três partes fundamentais:

1. Na fase inicial é inicializada uma estrutura de dados que gere histogramas, chamada *SmartHistos*, e feita a leitura de um ficheiro de texto, passado como argumento à aplicação, para conhecer os ficheiros ROOT que contêm os dados de entrada de forma proceder à inicialização de uma TChain com aqueles ficheiros;

2. Seguidamente, na fase de processamento, são preenchidos vários histogramas mediante certas condições.

3. Na fase final, os histogramas são guardados num ficheiro ROOT para posterior visualização.

Na Tabela 2.1 reproduz-se uma visão SWOT das caraterísticas gerais da aplicação. O ambiente interno refere os factores relacionados com a estrutura da análise derivada do esqueleto produzido pelo método *MakeClass*, enquanto que o ambiente externo diz respeito aos factores com reflexo no utilizador, no sistema operativo, etc...

*Tabela 2.1 Análise SWOT das principais características da versão original do caso de estudo.*

| | Ajuda | Atrapalha |
|---|---|---|
| Ambiente Interno | Estabilidade de código. | Pseudo-paralelismo: i) divisão dos dados em conjuntos menores; ii) execução de várias instâncias da aplicação em paralelo, a subconjuntos dos dados de entrada; iii) junção dos resultados parciais. |
| Ambiente Externo | Curva de aprendizagem inexistente, visto já estar em uso. | |

O “pseudo-paralelismo” referido é fruto de uma distribuição de trabalho estática e manual que pode levar a uma repartição desigual de trabalho entre os nós de computação, tanto a nível da quantidade dos dados quanto ao nível de interesse destes acontecimentos, resultando desta forma num desbalanceamento no escalonamento e consequente desperdício de rescursos computacionais, atrasando o término final da aplicação.

### 2.1.2 Versão “selector”

De modo a poder ter uma abordagem compatível com a utilização do PROOF como ambiente de execução, criou-se um novo esqueleto de código que estende a API da classe *TSelector*, executando para o efeito o método *TChain::MakeSelector*, sobre uma *TChain* que continha os mesmos ficheiros de dados de entrada da análise original. A classe gerada foi adaptada para substituir a parte de leitura de dados da analise original e integrada na análise.

Esta nova versão do código de análise, a que chamamos “selector”, quando executada utilizado o ambiente de execução do PROOF ainda não produz os resultados corretos. Isto apesar de ser derivada da classe *TSelector, de* conter o código para leitura da *OutputList* e os registos dos objetos considerados *output* dos trabalhadores na *OutputList* e destes objetos passarem a ter como *superclasse* a classe *TObject*.

### 2.1.3 Versão “selector 2”

A incompatibilidade da versão “selector” com o ambiente de execução do PROOF surge da impossibilidade do PROOF serializar, transmitir, reconstruir e juntar automaticamente os objetos presentes na *OutputList* quando estes objetos não são nativos do sistema ROOT. Assim, para se poderem produzir resultados consistentes foram efectuadas modificações do código de análise, da forma que se descreve a seguir, até se alcançar uma solução estável a que chamamos “selector 2”.

A análise foi reestruturada diversas vezes com várias configurações:

i. Utilização do método *TChain::Process(TSelector *)* sobre uma instância de *TSelector*, já usado na versão “selector”, mas desta vez mudando as regras de compilação e ligação para incluir os mecanismos de geração automática de dicionários do ROOT do *rootcint*;

ii. Utilização do método *TChain::Process(const char *)*, referindo ao ficheiro que continha a implementação da classe derivada do *TSelector* pelo nome, depois de compilar e carregar a classe *SmartHistos*, utilizando a funcionalidade *gROOT::ProcessLine* combinada com o comando ROOT *”.L SmartHistos.cpp+”*;

iii. Utilização de dois ficheiros PAR, um para a classe *SmartHistos* e outro para o código de análise, sendo o código compilado juntamente com os dicionários gerados pelo *rootcint* e carregado em tempo de execução para ser usado como referência no método *TChain::Process*;

iv. Utilização de apenas um ficheiro PAR com todo o código da análise, também compilando com os dicionários gerados pelo *rootcint*, e referindo-me ao selector pelo nome;

v. E finalmente, referindo o selector a ser executado pelo nome, e carregando todo o código relativo à análise através de um ficheiro PAR, utilizando uma chamada ao método *TROOT::ProcessLine(".L ficheiro.cpp+")* no *SETUP.C,* tanto para a implementação da classe *SmartHistos* como para a classe *proof2*.

Várias das opções acima foram testadas com quatro configurações distintas da classe *SmartHistos*: armazenando os histogramas sempre em um *std::map<TH1 *,std::string>*, ou copiando os histogramas para um *"C array"* antes da serialização, utilizando ou não as primitivas *ClassDef/ClassImp*.

A configuração que mostrou evidências de que os dados computados pelos processos trabalhadores estavam a chegar corretamente no processo mestre foi a opção v utilizando a classe *SmartHistos* com as primitivas *ClassDef*/*ClassImp* e movendo os histogramas para um "C array" antes da serialização.

Foi ainda necessário criar um algoritmo de junção da classe *SmartHistos*, para que o PROOF consiga disponibilizar apenas uma instância desta classe à parte final da análise (de notar que quando se utiliza apenas classes do ROOT, este passo não é necessário porque este método já está presente). Este algoritmo foi implementado utilizando a assinatura *Long64_t SMartHistos::Merge(TCollection *)*[1], que recorre ao método *Bool_t TH1::Add(const TH1*)*[2] para adicionar os histogramas que chegam dos nós trabalhadores aos histogramas correspondentes no nó principal (*master*).

É importante notar que o conteúdo da saída padrão (*standard output*) da aplicação teve de ser alterada porque cada processo trabalhador imprime numa saída diferente, para além dos dados, informação relevante acerca do estado da sessão PROOF.

A análise SWOT para a versão "selector 2", é apresentada na

[1] Não confundir com a assinatura obsoleta *void Merge(TCollection *)*[34]

[2] O método *Long64_t TH1::Merge(TCollection*) também foi testado, mas nem sempre conseguimos obter resultados consistentes.*

Tabela 2.2, que resume os argumentos discutidos nesta secção assim como sumariza a discussão apresentada na secção 2.1.2.1.

*Tabela 2.2 Análise SWOT da versão "selector 2" do caso de estudo.*

| | Ajuda | Atrapalha |
|---|---|---|
| Ambiente Interno | Permite o processamento paralelo, utilizando escalonamento dinâmico. | Implementação da análise em uma API centralizada, com o intuito de executar num ambiente de memória distribuída. |
| Ambiente Externo | | Curva de aprendizagem |

#### 2.1.3.1 Discussão da API da classe *TSelector*

Grande parte das dificuldades sentidas até chegar a uma versão "selector 2" compatível com o PROOF, decorrem do facto da API da classe *TSelector* centralizar numa única classe todo o código relativo às diferentes fases da análise, sendo que as respectivas instâncias executam uma parte do código no processo principal e outra parte nos processos trabalhadores, em diferentes nós de um sistema computacional distribuído.

Por esta razão um mesmo código de análise pode à partida produzir resultados corretos quando executa sequencialmente, mas gerar resultados incorretos quando corre num ambiente de memória distribuída como o PROOF, em que múltiplos processos interatuam em um ou mais nós de computação. A escolha de um modelo de processamento centralizado para execução em um sistema de memória distribuída, é comummente aceite como uma decisão contra-produtiva.

A solução alcançada, apesar de efetiva, viola os princípios de transparência da programação, uma vez que é necessário: i) registar os objetos desejados na *OutputList*, ii) garantir que os mesmos são serializáveis, iii) garantir que é possível agregar a informação de objetos com a mesma estrutura, e por fim, iv) recuperar o objeto.

Numa outra dimensão, verificou-se que, contrariamente às especificações oficiais, a utilização do ambiente PROOF-lite ou *PROOF on Demand* pode conduzir a resultados diferentes, pelo que nem sempre uma aplicação que corre consistentemente no PROOF-lite irá produzir resultados corretos quando executada numa sessão *PROOF on Demand* [32].

### 2.1.4 Nova abordagem - versão “event”

Face às dificuldades sentidas com a implementação da análise utilizando a API disponibilizada pelo PROOF, propusemo-nos conceber uma nova estrutura de classes que aliviasse os problemas da API centralizada da classe *TSelector* e da serialização e reconstrução de código.

Em primeiro lugar, foram encapsulados numa estrutura com nome *event* as variáveis de classe correspondentes aos dados de entrada (*input*), lidos através da chamada *fChain->GetTree()->GetEntry(entry,getall)*, sendo que a parte de tratamento dos dados foi movida para um método *event::Process*.

Seguidamente, os dados de saída (*output*) da análise, foram encapsulados numa classe chamada *eventOut* derivada da classe *TNamed*, nomeadamente foram encapsulados uma instância da classe *SmartHistos* (que contém todos os histogramas da aplicação) e nalguns casos variáveis para debug. Este passo força o sistema ROOT a fazer uso dos dicionários de forma direta apenas na classe *eventOut*, sendo que as classes encapsuladas não precisarão de derivar da classe *TObject* ou das primitivas *ClassDef*/*ClassImp*, removendo a atenção do utilizador do modelo centralizado da API *TSelector.*

De modo a implantar este modelo utilizando o PROOF, foi escrita uma classe *selector*, que contém o esqueleto “normal” (gerado pelo método TTree::*MakeSelector*), o código relativo a leitura de dados e transformação destes dados num *event* e o código relativo à gestão do controlo do fluxo das classes *event* e *eventOut*.

A Figura 2.1 apresenta o diagrama de classes da nova estrutura de classes.

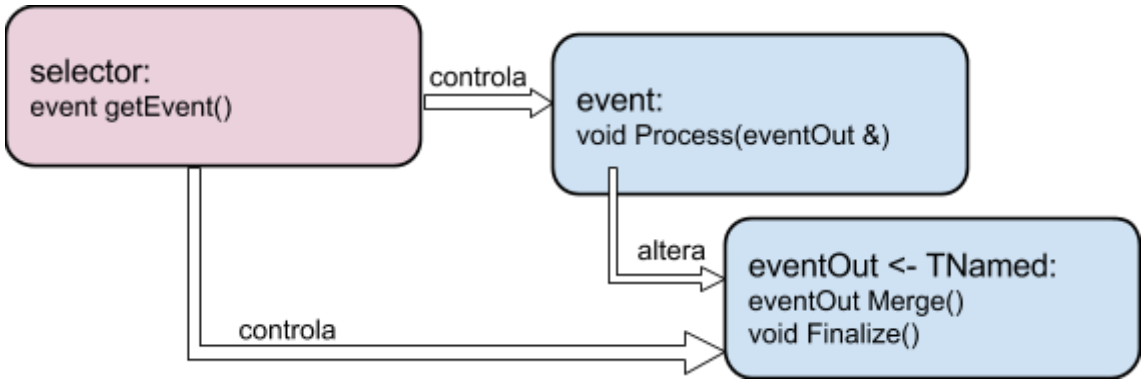



*Figura 2.1 Diagrama de classe da nova estrutura de classes*

Foi ainda definido um componente externo responsável por identificar os ficheiros de *entrada de dados*, iniciar a sessão PROOF e chamar o processamento da

análise, baseado na biblioteca *boost::program_options* [33].

A análise SWOT para esta nova abordagem é resumida na Tabela 2.1, que sumariza a discussão apresentada nesta secção.

*Tabela 2.3 Análise SWOT da versão event do caso de estudo.*

| | Ajuda | Atrapalha |
|---|---|---|
| Ambiente Interno | Redução do âmbito das classes | |
| Ambiente Externo | Redução dos problemas associados à transmissão dos dados entre processos. Melhorias associadas à manutenção do código. | Curva de aprendizagem |

## 2.2 Automatização da geração de esqueletos de código de análises

Este trabalho propõe uma nova estrutura de classes capaz de explorar as capacidades do PROOF, que ultrapassa as dificuldades associadas à utilização da API centralizada do *TSelector*, mas que disponibiliza facilidades equivalentes de geração automática do código esqueleto das análises.

A nova API baseia-se nas classes *event* e *eventOut* apresentadas na secção 2.1.4, que interagem entre si, e numa terceira classe, *selector*, que ao mesmo tempo que as controla estabelece a interface com o PROOF. Apesar dos nomes destas classes poderem ser escolhidos e alterados pelo utilizador, a estrutura das mesmas deve ser mantida para garantir a interface com o PROOF.

A classe *selector* e o código esqueleto das outras classes são gerados automaticamente a partir de uma *TTree*, não sendo necessário modificar a classe *selector*. Para o efeito, foi desenvolvida a aplicação *makeEvent* que recebe como argumentos: i) o caminho para um ficheiro ROOT que tenha uma *TTree*, ii) o nome da *TTree*, iii) o nome da classe que irá armazenar a estrutura do acontecimento e iii) o nome desejado para a classe que irá moldar o resultado do processamento dos acontecimentos. Por exemplo:

```
makeEvent ./file.root mini event event_out
```

A classe *selector* tem a estrutura padrão gerada pelo método *MakeSelector* a

que se acrescentou código para a criação: i) de uma instância da classe *eventOut*, ii) de uma instância da classe *event* para cada acontecimento, durante a leitura de cada acontecimento, e iii) chamadas aos métodos de finalização *BeforeStream e Terminate.*

Com a nova abordagem o fluxo de execução de uma análise tem os seguintes passos:

➔ O PROOF inicia em cada processo trabalhador uma instância do selector.

➔ O método selector::SlaveBegin inicia uma instância da *eventOut* e regista-a na *OutputList*.

➔ O método *selector::Process* é chamado para cada acontecimento e por sua vez este método chamará o método *selector::GetEvent* para criar uma instância da classe *event*, e depois chamar o método *event::Process.*

➔ Sempre que um processo trabalhador completa o processamento dos acontecimentos que lhe foram atribuídos, é chamado o método *selector::SlaveTerminate* que poderá eventualmente chamar o método *eventOut::BeforeStream*, no caso deste método ter sido definido.

➔ O método *eventOut::Merge* será chamado automaticamente pelo PROOF para combinar os resultados dos vários processos trabalhadores numa saída única.

➔ Finalmente, o método *eventOut::Terminate* é chamado no *selector::Terminate* para efetuar as operações sobre o resultado final antes de terminar a aplicação.

A classe *event* representa um acontecimento. Com base nos campos de dados dos acontecimentos, os métodos da classe deverão implementar o algoritmo que processa os dados com o objetivo de contribuir para o resultado global da análise, que irá ser modelado por instâncias da classe *eventOut*. A título de exemplo, as partes mais relevantes do ficheiro *event.h* são reproduzidas a seguir.

```
1.  class event {
2.  public:
3.          // Fixed size dimensions of array or collections stored in the TTree if any.
4.          static const Int_t kMaxfTracks= 614;
5.
6.          // Event properties
7.   //Event           *event;
8.     UInt_t          fUniqueID;
9.     ...
10.
11.         // Methods
12.         event(eventOut *_out=0);
13.         virtual ~event();
14.         virtual Int_t Version() const { return 0; }
15.         virtual void Process();
16.
17.         ClassDef(event,0); // Event event definition
18.
19.         // Variables definition
20.         eventOut*out;
21.
22. };
```

*Listagem 2.1 Partes mais relevantes do ficheiro event.h gerado automaticamente pela aplicação makeEvent*

A classe *eventOut* modela o output da análise. São os métodos desta classe que o programador usa para construir o resultado da análise. Em particular, deverá ser implementado o método *void Merge(eventOut *)*, usado para combinar o resultado de duas instâncias desta classe e o método *void Terminate()* usado para tratar os dados de saída antes da finalização da aplicação. Opcionalmente, o programador poderá criar o método *void BeforeStream()* para efetuar operações nos dados de saída antes de estes serem transmitidos aos processos trabalhadores. A API da classe é gerada é a seguinte:

```
1.  class eventOut : public TNamed  {
2.
3.  public:
4.          eventOut();
5.          virtual ~eventOut();
6.
7.          virtual void Merge(eventOut*);
8.          Long64_t Merge(TCollection *list);
9.          virtual void Print(const Option_t *op="")const;
10.         virtual void BeforeStream();
11.         virtual void Terminate();
12.
13.         ClassDef(eventOut,1);
14.
15.         // User variables bellow
16.         // One should use ROOT streamers annotations like:
17.         // Int_t ex1;           // This will be streamed
18.         // Float_t ex2;         //! This will not be streamed
19.         // Double32_t *ex3;     //-> This will be streamed without check if ex3 is a valid
    pointer
20.         // MyClass *ex4;        //[ex1] This will be streamed as an array of lenght ex1
21.         // read more in: http://goo.gl/SVwPSw
22.
23. };
```

*Listagem 2.2 Partes mais relevantes do ficheiro eventOut.h gerado automaticamente pela aplicação makeEvent*

O anexo 7.1 mostra o *output* completo da aplicação *makeEvent*, utilizando a *tree* Event disponibilizada pelo sistema ROOT na pasta *tests/tree* na diretoria de instalação.

# 3 Testes e proposta de adaptação

Ao longo deste trabalho foram apresentadas diversas alternativas de desenvolvimento de códigos de análises físicas, com vista a melhorar o desempenho através do paralelismo ao nível do acontecimento, utilizando a *framework* PROOF. A utilização de um ambiente paralelo já é, por si, vantajosa, na medida em que tira partido dos recursos disponíveis ao utilizador. No entanto, a estrutura de análise proposta implica num acréscimo no número de instruções computacionais por acontecimento, visto que é acrescentada uma cópia para cada objeto lido, o que pode degradar o desempenho. Desta forma, torna-se necessário a avaliação do desempenho da nova estrutura de análise.

## 3.1 Desempenho das versões paralelas da análise VLQValidation

A abordagem baseada na versão *event* apresenta melhorias no processo de desenvolvimento, nomeadamente no que diz respeito ao encapsulamento dos dados e na redução drástica dos problemas associados à serialização automática. No entanto, a mudança de paradigma tem que ter em conta não apenas o ganho de desempenho entre versões, mas também o ganho em tempo de desenvolvimento.

Assim, foram executados testes comparativos entre as versões “selector 2” e “event” com o intuito de comparar a diferença no tempo de execução e *speedup*. A tabela 3.1 representa os tempos de execução destas versões utilizando *entradas* de dados com 25 GB e 30 GB. Utilizando estes dados, pode-se calcular o *speedup* comparativo destas versões, pelo que a Figura 3.1 representa o gráfico do *speedup* em função do número de trabalhadores.

*Tabela 3.1 Tempos de execução (segundos) das versões "selector 2" e "event", utilizando PROOF-lite, no nó compute-0-11 do cluster lip.di.uminho.pt, 2 x Intel Xeon E5-2630 (6 cores x 2 threads @ 1.2 GHz) e 32 GB de memória RAM*

| #Processos trabalhadores | Selector2~25GB | Selector2~30GB | Event~25GB | Event~30GB |
|---|---|---|---|---|
| 1 | 6075,691 | 7095,839 | 6082,950 | 7085,156 |
| 2 | 2864,720 | 3354,340 | 2862,209 | 3360,772 |
| 3 | 1827,117 | 2246,843 | 1913,314 | 2254,183 |
| 4 | 1383,087 | 1684,129 | 1440,417 | 1636,934 |
| 5 | 1088,642 | 1275,463 | 1086,403 | 1305,029 |
| 6 | 914,822 | 1015,382 | 879,091 | 1019,883 |
| 7 | 752,498 | 875,109 | 747,435 | 874,396 |
| 8 | 657,388 | 805,646 | 653,620 | 763,943 |
| 9 | 584,571 | 713,116 | 582,384 | 684,290 |
| 10 | 526,661 | 615,067 | 527,147 | 614,537 |
| 11 | 481,401 | 561,673 | 481,010 | 560,648 |
| 12 | 441,997 | 515,675 | 441,849 | 515,158 |
| 13 | 431,934 | 505,817 | 432,974 | 505,231 |
| 14 | 423,168 | 492,587 | 423,154 | 494,341 |
| 15 | 413,944 | 483,226 | 413,270 | 483,297 |
| 16 | 406,142 | 473,678 | 406,297 | 473,173 |
| 17 | 397,066 | 463,176 | 397,910 | 463,453 |
| 18 | 390,166 | 454,598 | 390,777 | 454,733 |
| 19 | 383,654 | 445,372 | 383,138 | 446,050 |
| 20 | 375,884 | 438,421 | 375,431 | 437,967 |
| 21 | 368,733 | 429,735 | 369,330 | 431,403 |
| 22 | 362,401 | 422,626 | 362,140 | 422,484 |
| 23 | 356,638 | 414,655 | 357,077 | 414,002 |
| 24 | 354,787 | | 354,555 | |

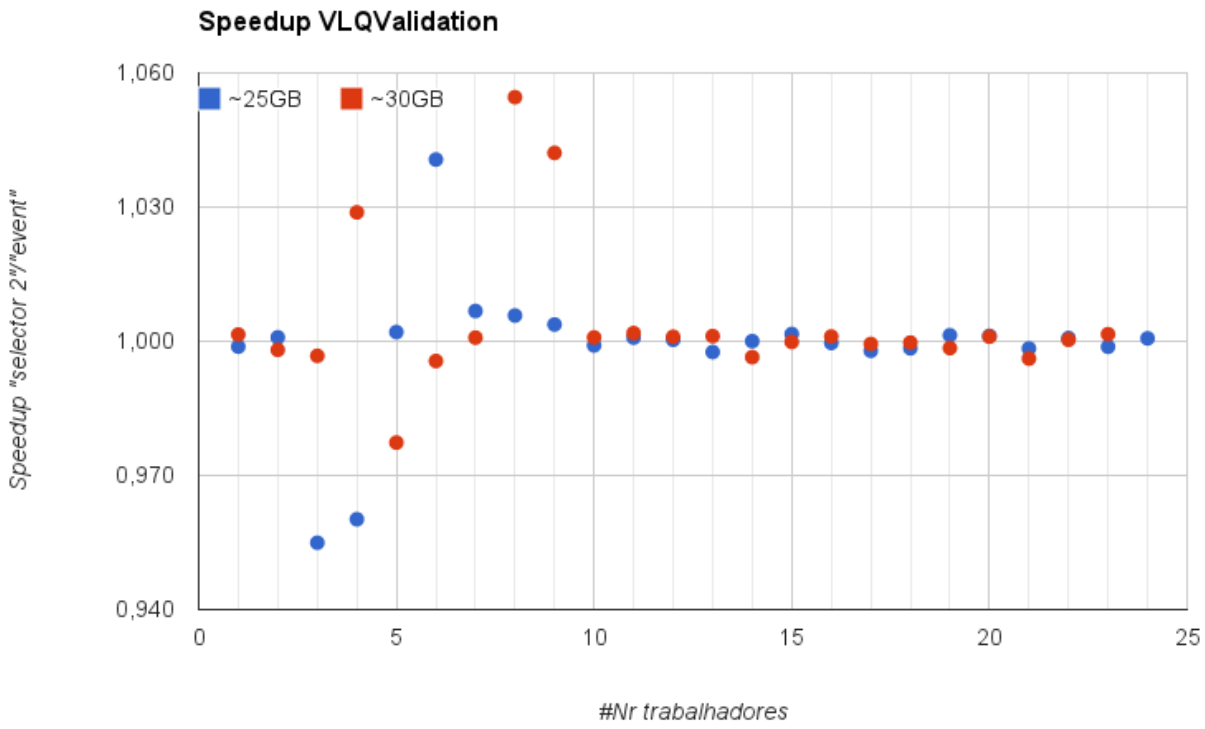



*Figura 3.1 Speedup da versão event em relação à versão selector2, tempos de acordo com a tabela 3.1*

No gráfico acima quanto mais próximo de 1 estiverem os valores no eixo Y maior a equivalência entre as versões comparadas, sendo que os valores maiores do que 1 indicam que a versão *event* foi mais rápida.

Como se pode reparar na maior parte dos casos a variação não ultrapassa 1%, sendo que nos casos de divergência (valor máximo de 1.055, ou seja 5.5%), não há consistência nem para a execução mais rápida, nem para a mais lenta. As anomalias advêm possivelmente de factores externos à aplicação que congestionam a rede, ou do nó *front-end* responsável pelo acesso aos dados através do serviço de NFS. Quando há congestão de um destes recursos a aplicação demora mais tempo para ler os dados de *entrada*, o que resulta em resultados finais piores, em casos pontuais e em qualquer uma das versões.

## 3.2 Estudo de escalabilidade da versão *event*

Ainda com base nos dados da tabela 3.1, podemos calcular o *speedup* associado ao número de processos trabalhadores utilizados no processamento dos dados, e consequentemente a eficiência do processamento paralelo.

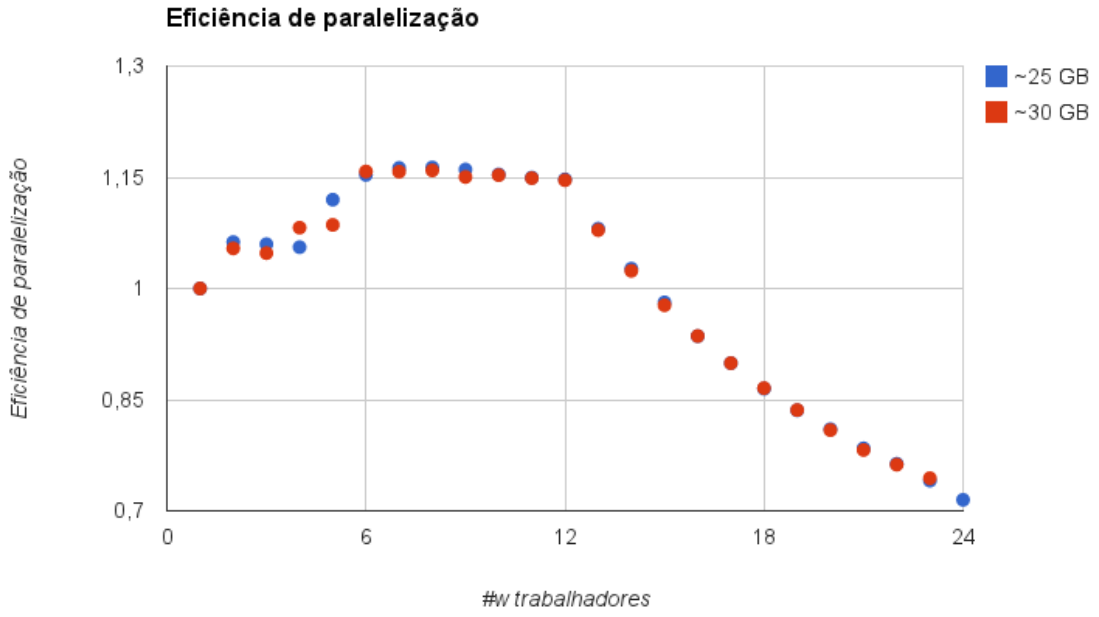



*Figura 3.2 Eficiência do processamento paralelo, valores comparativos a execução com um trabalhador, utilizando a versão event do caso de estudo*

No gráfico acima nota-se, com clara distinção, dois segmentos de pontos:

i. Eficiência acima de 1 utilizando de 2 a 12 processos trabalhadores; indicando *speedup*s superlineares, provavelmente ocasionados pela sobrecarga de instruções associadas ao PROOF utilizando apenas um processo trabalhador, o que causa degradação de desempenho neste caso e consequente melhoria nos testes subsequentes.
ii. Eficência decrescente a partir de 12 processos trabalhadores, possivelmente decorrentes do fato do número de processos trabalhadores ultrapassar o número de núcleos reais de processamento (12), passando a utilizar todas as *threads* de processamento disponibilizadas ao sistema operativo através da tecnologia *hyperthreadning*.

## 3.3 Adaptação da estrutura *event* nas análises do LIP

Tendo em conta o modelo atualmente utilizado no LIP-Minho, a utilização do PROOF em códigos de análise já existentes do LIP passariam pela respectiva reescrita utilizando uma API compatível com o *TSelector*. A reescrita é um processo moroso, especialmente para análises mais complexas, pelo que o investimento temporal deve ser avaliado em comparação aos possíveis ganhos de desempenho.

No caso de desenvolvimento de novas análises estamos convencidos que a abordagem proposta neste trabalho é compensadora, no sentido em que um pequeno investimento para superar a curva de aprendizagem e a integração com as ferramentas desejáveis para a análise em questão, pode resultar no paralelismo automático da análise; de notar que no investimento de uma nova análise pode ser feito o reaproveitamento de código, o que se traduz na diminuição da curva de aprendizagem.

Para as análises mais complexas, o processo de adaptação pode ser minimizado se for desenvolvida uma *framework* modular que auxilie o utilizador nas tarefas mais comuns, tais como, a gestão dos níveis de cortes, o tratamento estatístico de dados, a gestão de histogramas, gestão de múltiplas execuções para análise de erros sistemáticos e a provisão de valores comuns (constantes, unidades de medida, etc).

# 4 Trabalho Futuro

Em termos de trabalho futuro prevê-se que, a exemplo do método *TTree::MakeProxy*, a próxima versão da aplicação *makeEvent* altere a lógica da classe derivada do *TSelector* para carregar apenas os *branches* utilizados na classe *event*. Esta modificação poderia permitir reduzir significativamente o tempo de execução de análises que carregam todos *branches* de uma *tree* (ação padrão), mas não necessitam de todos estes dados.

Apesar de ser possível usar o PROOF para extração automática de paralelismo em aplicações de análise de dados de LHC já em ambiente de produção no LIP-Minho, esta solução pode ser contraproducente. Em alternativa sugerimos a criação de um conjunto de bibliotecas com as funcionalidades normalmente requeridas por estas análises, tal como é descrito na secção 3.3.

A modularidade da nova abordagem que propomos na estrutura de classes *event* permite tornar a interface do utilizador independente do *TSelector*, desta forma facilitando os testes de desempenho comparativos do PROOF com outras *frameworks* que implementam paralelismo em ambiente distribuído, sendo as mais notáveis a *Message Passing Interface* (MPI) e o MapReduce.

# 5 Conclusões

As aplicações de análises de dados do LIP-Minho têm uma estrutura semelhante que é passível de ser paralelizada. O PROOF apresenta-se como uma solução atrativa uma vez que permite alcançar *speedups* máximos que crescem linearmente com a quantidade de trabalhadores. Tanto no teste *cycle-driven* do *benchmark TProofBench,* quanto na análise testada, foi possível obter *speedups* superlineares.

Apesar da curva de aprendizagem ter-se demonstrado acentuada, por causa das dificuldades apresentadas na secção 2.1.3, para um utilizador inexperiente, as dificuldades foram minimizadas com a nova estrutura de classes, gerada pela aplicação makeEvent, que provou ser tão eficiente quando as análises baseadas apenas na API baseada na classe *TSelector*.

É nossa convicção que a utilização do PROOF é adequado para a paralelização de análises do LIP-Minho, e que os conceitos associados à abordagem *makeEvent* poderão vir a ter um papel relevante nos custos de desenvolvimento de novas aplicações e na redução dos tempos de processamento dos dados das análises.

# 6 Referências


[1] J. Shiers, “The Worldwide LHC Computing Grid (worldwide LCG)” *Computer Physics Communications*, vol. 177, no. 1–2, pp. 219–223, Jul. 2007.

[2] F. Gianotti, M. L. Mangano, T. Virdee, S. Abdullin, G. Azuelos, A. Ball, D. Barberis, A. Belyaev, P. Bloch, M. Bosman, L. Casagrande, D. Cavalli, P. Chumney, S. Cittolin, S. Dasu, A. Roeck, N. Ellis, P. Farthouat, D. Fournier, J.-B. Hansen, I. Hinchliffe, M. Hohlfeld, M. Huhtinen, K. Jakobs, C. Joram, F. Mazzucato, G. Mikenberg, A. Miagkov, M. Moretti, S. Moretti, T. Niinikoski, A. Nikitenko, A. Nisati, F. Paige, S. Palestini, C. G. Papadopoulos, F. Piccinini, R. Pittau, G. Polesello, E. Richter-Was, P. Sharp, S. R. Slabospitsky, W. H. Smith, S. Stapnes, G. Tonelli, E. Tsesmelis, Z. Usubov, L. Vacavant, J. Bij, A. Watson, and M. Wielers, “Physics potential and experimental challenges of the LHC luminosity upgrade,” *The European Physical Journal C*, vol. 39, no. 3, pp. 293–333, Feb. 2005.

[3] G. Dimitrov, T. Maeno, and V. Garonne, “Next generation database relational solutions for ATLAS distributed computing”, Oct. 2013.

[4] CERN press office, “The first LHC protons run ends with new milestone,” 17-Dec-2012.

[5] The ATLAS Collaboration, “The ATLAS Simulation Infrastructure,” *The European Physical Journal C*, vol. 70, no. 3, pp. 823–874, Sep. 2010.

[6] “Atlas Fact Sheet.” [Online]. Available: http://www.atlas.ch/pdf/ATLAS_fact_sheets.pdf.

[7] The ATLAS Collaboration, “The ATLAS Experiment at the CERN Large Hadron Collider,” *Journal of Instrumentation*, vol. 3, no. 08, pp. S08003–S08003, Aug. 2008.

[8] The ATLAS Collaboration, “Performance of the ATLAS Trigger System in 2010,” *The European Physical Journal C*, vol. 72, no. 1, p. 1849, Jan. 2012.

[9] E. Moyse and F. Akesson, “Event data model in ATLAS,” *Proceedings of CHEP 2004 Conference*, 2004.

[10] “*Tier* centres.” [Online]. Available: http://wlcg-public.web.cern.ch/*Tier*-centres, [Accessed: 24-Oct-2014].

[11] S. Yacoob, “The ATLAS Computing Model,” *ACM SIGMETRICS*, 2013. [Online]. Available: https://cds.cern.ch/record/1645234/.

[12] R. Brun and F. Rademakers, “ROOT — An object oriented data analysis framework,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1–2, pp. 81–86, Apr. 1997.

[13] B. Bellenot, “ROOT : a Data Storage and Analysis Framework,” 2008.

[14] The ROOT Team, “CINT.” [Online]. Available: http://root.cern.ch/drupal/content/cint. [Accessed: 24-Oct-2014].

[15] The ROOT Team, “Cling.” [Online]. Available: http://root.cern.ch/drupal/content/cling. [Accessed: 24-Oct-2014].

[16] M. Ballintijn, R. Brun, F. Rademakers, and G. Roland, “The PROOF Distributed Parallel Analysis Framework based on ROOT,” pp. 1–6, Jun. 2003.

[17] M. Ballintijn, M. Biskup, R. Brun, P. Canal, D. Feichtinger, G. Ganis, G. Kickinger, A. Peters, and F. Rademakers, “Parallel interactive data analysis with PROOF,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 559, no. 1, pp. 13–16, Apr. 2006.

[18] T. Karnik, S. Borkar, and V. De, “Sub-90nm Technologies--Challenges and Opportunities for CAD,” pp. 0–3, 2002.

[19] S. E. Thompson and S. Parthasarathy, “Moore’s law: the future of Si microelectronics,” *Materials Today*, vol. 9, no. 6, pp. 20–25, Jun. 2006.

[20] B. G. E. Moore, “Cramming more components onto integrated circuits,” vol. 38, no. 8, 1975.

[21] M. Eddy and B. B. N. Technologies, “Advantages of Parallel Processing Effects of Communications Time,” no. February, 2000.

[22] G. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” *Proceedings of the April 18-20, 1967*.

[23] R. Barve, E. Shriver, and P. Gibbons, “Modeling and optimizing I/O throughput of multiple disks on a bus”, *ACM SIGMETRICS*, 1999.

[24] H. Jin and R. Van der Wijngaart, “Performance characteristics of the multi-zone NAS parallel benchmarks,” *Parallel and Distributed Processing Symposium*, 2004.

[25] N. Ioannou and M. Cintra, “Complementing user-level coarse-grain parallelism with implicit speculative parallelism,” *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[26] D. G. Feitelson and L. Rudolph, Eds., *Job Scheduling Strategies for Parallel Processing*, vol. 949. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995.

[27] D. Berzano, J. Blomer, P. Buncic, I. Charalampidis, G. Ganis, G. Lestaris, and R. Meusel, “PROOF as a Service on the Cloud : a Virtual Analysis Facility based on the CernVM ecosystem”, 2014.

[28] S. Ryu and G. Ganis, “The PROOF benchmark suite measuring PROOF performance,” *Journal of Physics: Conference Series*, 2012.

[29] G. Ganis and S. Panitkin, “Evaluating Google Compute Engine with PROOF”, *Journal of Physics: Conference Series*, 2014.

[30] The ATLAS Collaboration, “Search for pair and single production of new heavy quarks that decay to a $Z$ boson and a third-generation quark in $pp$ collisions at $\sqrt{s}=8$ TeV with the ATLAS detector”, p. 36, Sep. 2014.

[31] “How to split a filename list in 5GB sets?” [Online]. Available: http://unix.stackexchange.com/a/122796/30320. [Accessed: 02-Apr-2014].

[32] R. Silva, “podXlite,” 2014. [Online]. Available: http://bitbucket.org/RSFalcon7/podxlite/. [Accessed: 27-Oct-2014].

[33] V. Prus, “Boost.Program_options.” [Online]. Available: http://www.boost.org/doc/libs/1_56_0/doc/html/program_options.html. [Accessed: 27-Oct-2014].

[34] The ROOT Team, “Merging customized classes.” [Online]. Available: http://root.cern.ch/drupal/content/merging-customized-classes. [Accessed: 27-Oct-2014].

[35] Bertrand Bellenot, “ROOT: a Data Storage and Analysis Framework” [Online]. Avaliable: http://www.math.ch/colloqnum08/posters/02_Bellenot.pdf. [Accessed: 27-Oct-2014].

# 7 Anexos

## 7.1 Exemplos de código esqueleto gerados automaticamente pela aplicação makeEvent

As listagens abaixo exemplificam o código esqueleto gerado pela aplicação makeEvent, utilizando a *tree Event* disponibilizada pelo sistema ROOT em *$ROOTSYS/tests/tree*.

---

```
23. #ifndef event_h
24. #define event_h
25.
26. /**
27.  * This file has been automatically generated
28.  * by void makeEvent_h(TTree*, std::string, std::string, bool)
29.  * at Fri Oct 10 00:50:30 2014
30.  */
31.
32. // Default include files
33. #include <exception>
34. #include <Rtypes.h>
35.
36. #include "eventOut.h"
37.
38. // event include files
39.
40. // Header file for the classes stored in the TTree if any.
41. #include <TObject.h>
42. #include <TBits.h>
43.
44. // event namespace definitions
45. using namespace std;
46.
47. class InvalidEventOut : public std::exception   {
48.         virtual const char* what() const throw()        {
49.                 return "Invalid output object.";
50.         }
51. };
52.
53. class event {
54. public:
55.
56.         // Fixed size dimensions of array or collections stored in the TTree if any.
57.         static const Int_t kMaxfTracks= 614;
58.
59.         // Event properties
60.  //Event           *event;
61.    UInt_t          fUniqueID;
62.    UInt_t          fBits;
63.    Char_t          fType[20];
64.    char*           fEventName;
65.    Int_t           fNtrack;
66.    Int_t           fNseg;
67.    Int_t           fNvertex;
```

```
68.    UInt_t          fFlag;
69.    Double32_t      fTemperature;
70.    Int_t           fMeasures[10];
71.    Double32_t      fMatrix[4][4];
72.    Double32_t      fClosestDistance[20];   //[fNvertex]
73.    Int_t           fEvtHdr_fEvtNum;
74.    Int_t           fEvtHdr_fRun;
75.    Int_t           fEvtHdr_fDate;
76.    Int_t           fTracks_;
77.    UInt_t          fTracks_fUniqueID[kMaxfTracks];   //[fTracks_]
78.    UInt_t          fTracks_fBits[kMaxfTracks];   //[fTracks_]
79.    Float_t         fTracks_fPx[kMaxfTracks];   //[fTracks_]
80.    Float_t         fTracks_fPy[kMaxfTracks];   //[fTracks_]
81.    Float_t         fTracks_fPz[kMaxfTracks];   //[fTracks_]
82.    Float_t         fTracks_fRandom[kMaxfTracks];   //[fTracks_]
83.    Float16_t       fTracks_fMass2[kMaxfTracks];   //[fTracks_]
84.    Float16_t       fTracks_fBx[kMaxfTracks];   //[fTracks_]
85.    Float16_t       fTracks_fBy[kMaxfTracks];   //[fTracks_]
86.    Float_t         fTracks_fMeanCharge[kMaxfTracks];   //[fTracks_]
87.    Float16_t       fTracks_fXfirst[kMaxfTracks];   //[fTracks_]
88.    Float16_t       fTracks_fXlast[kMaxfTracks];   //[fTracks_]
89.    Float16_t       fTracks_fYfirst[kMaxfTracks];   //[fTracks_]
90.    Float16_t       fTracks_fYlast[kMaxfTracks];   //[fTracks_]
91.    Float16_t       fTracks_fZfirst[kMaxfTracks];   //[fTracks_]
92.    Float16_t       fTracks_fZlast[kMaxfTracks];   //[fTracks_]
93.    Double32_t      fTracks_fCharge[kMaxfTracks];   //[fTracks_]
94.    Double32_t      fTracks_fVertex[kMaxfTracks][3];   //[fTracks_]
95.    Int_t           fTracks_fNpoint[kMaxfTracks];   //[fTracks_]
96.    Short_t         fTracks_fValid[kMaxfTracks];   //[fTracks_]
97.    Int_t           fTracks_fNsp[kMaxfTracks];   //[fTracks_]
98.    Double32_t     *fTracks_fPointValue[kMaxfTracks];   //[fTracks_fNsp]
99.    TBits           fTracks_fTriggerBits[kMaxfTracks];
100.         TRef            fLastTrack;
101.         TRef            fWebHistogram;
102.         UInt_t          fTriggerBits_fUniqueID;
103.         UInt_t          fTriggerBits_fBits;
104.         UInt_t          fTriggerBits_fNbits;
105.         UInt_t          fTriggerBits_fNbytes;
106.         UChar_t         fTriggerBits_fAllBits[1];   //[fTriggerBits.fNbytes]
107.         Bool_t          fIsValid;
108.
109.              // Methods
110.              event(eventOut *_out=0);
111.              virtual ~event();
112.              virtual Int_t Version() const { return 0; }
113.              virtual void Process();
114.
115.              ClassDef(event,0); // Event event definition
116.
117.              // Variables definition
118.              eventOut*out;
119.
120.      };
121.
122.      #endif
```

*Listagem 7.1 event.h*

```
1.  #ifndef event_cpp
2.  #define event_cpp
3.
4.  /**
5.   * This file has been automatically generated
6.   * by void makeEvent_cpp(TTree*, std::string, std::string)
7.   * at Fri Oct 10 00:50:30 2014
8.   */
9.
10. // Default include files
11. #include <iostream>
12. #include "event.h"
13.
14. /**
15.  * This method should initialize user define variables
16.  * The selector will automatically initialize all the
17.  * tree vars after the constructor
18.  */
19. event::event(eventOut* _out)    {
20.         if(_out)
21.                 out = _out;
22.         else    {
23.                 throw InvalidEventOut();
24.         }
25.
26. }
27. event::~event() { }
28.
29. /**
30.  * This method should compute and add the contribute of
31.  * this event to the eventOut object
32.  */
33. void event::Process()   {
34.
35. }
36.
37.
38. #endif
```

*Listagem 7.2 event.cpp*

```
24. #ifndef eventOut_h
25. #define eventOut_h
26.
27. /**
28.  * This file has been automatically generated
29.  * by void makeEventOut_h(std::string)
30.  * at Fri Oct 10 00:50:30 2014
31.  */
32.
33. // Default include files
34. #include <TNamed.h>
35.
36. // User include files
37.
38. class eventOut : public TNamed  {
```

```
39.
40. public:
41.         eventOut();
42.         virtual ~eventOut();
43.
44.         virtual void Merge(eventOut*);
45.         Long64_t Merge(TCollection *list);
46.         virtual void Print(const Option_t *op="")const;
47.         virtual void BeforeStream();
48.         virtual void Terminate();
49.
50.         ClassDef(eventOut,1);
51.
52.         // User variables bellow
53.         // One should use ROOT streamers annotations like:
54.         // Int_t ex1;           // This will be streamed
55.         // Float_t ex2;         //! This will not be streamed
56.         // Double32_t *ex3;     //-> This will be streamed without check if ex3 is a valid
    pointer
57.         // MyClass *ex4;        //[ex1] This will be streamed as an array of lenght ex1
58.         // read more in: http://goo.gl/SVwPSw
59.
60.
61. };
62. #endif
```

*Listagem 7.3 eventOut.h*

```
1.  #ifndef eventOut_cpp
2.  #define eventOut_cpp
3.
4.  /**
5.   * This file has been automatically generated
6.   * by void makeEventOut_cpp(std::string)
7.   * at Fri Oct 10 00:50:30 2014
8.   */
9.
10. // Default include files
11. #include <iostream>
12. #include "eventOut.h"
13.
14. #endif
```

*Listagem 7.4 eventOut.cpp*

```
1.  #ifndef selector_h
2.  #define selector_h
3.
4.  /**
5.   * This file has been automatically generated
6.   * by void makeSelector_h(TTree*, bool, std::string, std::string)
7.   * at Fri Oct 10 00:50:30 2014
8.   */
9.
10. // Default include files
11. #include <TROOT.h>
```

```
#include <TChain.h>
#include <TFile.h>
#include <TSelector.h>
#include "event.h"

// user include files

// Header file for the classes stored in the TTree if any.
#include <TObject.h>
#include <TBits.h>

using namespace std;
        // Fixed size dimensions of array or collections stored in the TTree if any.
        static const Int_t kMaxfTracks= 614;

class selector : public TSelector      {
public:
        TTree   *fChain;        //!pointer to the analyzed TTree or TChain

        // Declaration of lead types
 //Event           *event;
   UInt_t          fUniqueID;
   UInt_t          fBits;
   Char_t          fType[20];
   char*           fEventName;
   Int_t           fNtrack;
   Int_t           fNseg;
   Int_t           fNvertex;
   UInt_t          fFlag;
   Double32_t      fTemperature;
   Int_t           fMeasures[10];
   Double32_t      fMatrix[4][4];
   Double32_t      fClosestDistance[20];   //[fNvertex]
   Int_t           fEvtHdr_fEvtNum;
   Int_t           fEvtHdr_fRun;
   Int_t           fEvtHdr_fDate;
   Int_t           fTracks_;
   UInt_t          fTracks_fUniqueID[kMaxfTracks];   //[fTracks_]
   UInt_t          fTracks_fBits[kMaxfTracks];   //[fTracks_]
   Float_t         fTracks_fPx[kMaxfTracks];   //[fTracks_]
   Float_t         fTracks_fPy[kMaxfTracks];   //[fTracks_]
   Float_t         fTracks_fPz[kMaxfTracks];   //[fTracks_]
   Float_t         fTracks_fRandom[kMaxfTracks];   //[fTracks_]
   Float16_t       fTracks_fMass2[kMaxfTracks];   //[fTracks_]
   Float16_t       fTracks_fBx[kMaxfTracks];   //[fTracks_]
   Float16_t       fTracks_fBy[kMaxfTracks];   //[fTracks_]
   Float_t         fTracks_fMeanCharge[kMaxfTracks];   //[fTracks_]
   Float16_t       fTracks_fXfirst[kMaxfTracks];   //[fTracks_]
   Float16_t       fTracks_fXlast[kMaxfTracks];   //[fTracks_]
   Float16_t       fTracks_fYfirst[kMaxfTracks];   //[fTracks_]
   Float16_t       fTracks_fYlast[kMaxfTracks];   //[fTracks_]
   Float16_t       fTracks_fZfirst[kMaxfTracks];   //[fTracks_]
   Float16_t       fTracks_fZlast[kMaxfTracks];   //[fTracks_]
   Double32_t      fTracks_fCharge[kMaxfTracks];   //[fTracks_]
   Double32_t      fTracks_fVertex[kMaxfTracks][3];   //[fTracks_]
   Int_t           fTracks_fNpoint[kMaxfTracks];   //[fTracks_]
   Short_t         fTracks_fValid[kMaxfTracks];   //[fTracks_]
   Int_t           fTracks_fNsp[kMaxfTracks];   //[fTracks_]
   Double32_t     *fTracks_fPointValue[kMaxfTracks];   //[fTracks_fNsp]
```

```
71.    TBits           fTracks_fTriggerBits[kMaxfTracks];
72.    TRef            fLastTrack;
73.    TRef            fWebHistogram;
74.    UInt_t          fTriggerBits_fUniqueID;
75.    UInt_t          fTriggerBits_fBits;
76.    UInt_t          fTriggerBits_fNbits;
77.    UInt_t          fTriggerBits_fNbytes;
78.    UChar_t         fTriggerBits_fAllBits[1];   //[fTriggerBits.fNbytes]
79.    Bool_t          fIsValid;
80.
81.    // List of branches
82.    TBranch        *b_event_fUniqueID;   //!
83.    TBranch        *b_event_fBits;   //!
84.    TBranch        *b_event_fType;   //!
85.    TBranch        *b_event_fEventName;   //!
86.    TBranch        *b_event_fNtrack;   //!
87.    TBranch        *b_event_fNseg;   //!
88.    TBranch        *b_event_fNvertex;   //!
89.    TBranch        *b_event_fFlag;   //!
90.    TBranch        *b_event_fTemperature;   //!
91.    TBranch        *b_event_fMeasures;   //!
92.    TBranch        *b_event_fMatrix;   //!
93.    TBranch        *b_fClosestDistance;   //!
94.    TBranch        *b_event_fEvtHdr_fEvtNum;   //!
95.    TBranch        *b_event_fEvtHdr_fRun;   //!
96.    TBranch        *b_event_fEvtHdr_fDate;   //!
97.    TBranch        *b_event_fTracks_;   //!
98.    TBranch        *b_fTracks_fUniqueID;   //!
99.    TBranch        *b_fTracks_fBits;   //!
100.       TBranch        *b_fTracks_fPx;   //!
101.       TBranch        *b_fTracks_fPy;   //!
102.       TBranch        *b_fTracks_fPz;   //!
103.       TBranch        *b_fTracks_fRandom;   //!
104.       TBranch        *b_fTracks_fMass2;   //!
105.       TBranch        *b_fTracks_fBx;   //!
106.       TBranch        *b_fTracks_fBy;   //!
107.       TBranch        *b_fTracks_fMeanCharge;   //!
108.       TBranch        *b_fTracks_fXfirst;   //!
109.       TBranch        *b_fTracks_fXlast;   //!
110.       TBranch        *b_fTracks_fYfirst;   //!
111.       TBranch        *b_fTracks_fYlast;   //!
112.       TBranch        *b_fTracks_fZfirst;   //!
113.       TBranch        *b_fTracks_fZlast;   //!
114.       TBranch        *b_fTracks_fCharge;   //!
115.       TBranch        *b_fTracks_fVertex;   //!
116.       TBranch        *b_fTracks_fNpoint;   //!
117.       TBranch        *b_fTracks_fValid;   //!
118.       TBranch        *b_fTracks_fNsp;   //!
119.       TBranch        *b_fTracks_fPointValue;   //!
120.       TBranch        *b_fTracks_fTriggerBits;   //!
121.       TBranch        *b_event_fLastTrack;   //!
122.       TBranch        *b_event_fWebHistogram;   //!
123.       TBranch        *b_event_fTriggerBits_fUniqueID;   //!
124.       TBranch        *b_event_fTriggerBits_fBits;   //!
125.       TBranch        *b_event_fTriggerBits_fNbits;   //!
126.       TBranch        *b_event_fTriggerBits_fNbytes;   //!
127.       TBranch        *b_fTriggerBits_fAllBits;   //!
128.       TBranch        *b_event_fIsValid;   //!
129.            selector(TTree * =0) : fChain(0) { }
130.            ~selector()     { }
```

```
131.           Int_t   Version() const { return 2; }
132.           void    Begin(TTree *tree);
133.           void    SlaveBegin(TTree *tree);
134.           void    Init(TTree *tree);
135.           Bool_t  Notify();
136.           event   GetEvent(Long64_t entry);
137.           Bool_t  Process(Long64_t entry);
138.           Int_t   GetEntry(Long64_t entry,Int_t getall = 0)       { return fChain ? fChain-
    >GetTree()->GetEntry(entry,getall) : 0; }
139.           void    SetOption(const char *option) { fOption = option; }
140.           void    SetObject(TObject *obj)         { fObject = obj; }
141.           void    SetInputList(TList *input)      { fInput = input; }
142.           TList * GetOutputList() const { return fOutput; }
143.           void    SlaveTerminate();
144.           void    Terminate();
145.
146.           ClassDef(selector,1);
147.
148.           eventOut *out; // Output of the workers, it must be streameable throught ROOT
    streamer
149.    };
150.    #endif
151.
152.    #ifdef selector_cpp
153.    void selector::Init(TTree *tree)
154.    {
155.       // The Init() function is called when the selector needs to initialize
156.       // a new tree or chain. Typically here the branch addresses and branch
157.       // pointers of the tree will be set.
158.       // It is normally not necessary to make changes to the generated
159.       // code, but the routine can be extended by the user if needed.
160.       // Init() will be called many times when running on PROOF
161.       // (once per file to be processed).
162.
163.       // Set array pointer
164.       for(int i=0; i<kMaxfTracks; ++i) fTracks_fPointValue[i] = 0;
165.
166.       // Set branch addresses and branch pointers
167.       if (!tree) return;
168.       fChain = tree;
169.       fChain->SetMakeClass(1);
170.
171.       fChain->SetBranchAddress("fUniqueID", &fUniqueID, &b_event_fUniqueID);
172.       fChain->SetBranchAddress("fBits", &fBits, &b_event_fBits);
173.       fChain->SetBranchAddress("fType[20]", fType, &b_event_fType);
174.       fChain->SetBranchAddress("fEventName", &fEventName, &b_event_fEventName);
175.       fChain->SetBranchAddress("fNtrack", &fNtrack, &b_event_fNtrack);
176.       fChain->SetBranchAddress("fNseg", &fNseg, &b_event_fNseg);
177.       fChain->SetBranchAddress("fNvertex", &fNvertex, &b_event_fNvertex);
178.       fChain->SetBranchAddress("fFlag", &fFlag, &b_event_fFlag);
179.       fChain->SetBranchAddress("fTemperature", &fTemperature, &b_event_fTemperature);
180.       fChain->SetBranchAddress("fMeasures[10]", fMeasures, &b_event_fMeasures);
181.       fChain->SetBranchAddress("fMatrix[4][4]", fMatrix, &b_event_fMatrix);
182.       fChain->SetBranchAddress("fClosestDistance", fClosestDistance, &b_fClosestDistance);
183.       fChain-
    >SetBranchAddress("fEvtHdr.fEvtNum", &fEvtHdr_fEvtNum, &b_event_fEvtHdr_fEvtNum);
184.       fChain->SetBranchAddress("fEvtHdr.fRun", &fEvtHdr_fRun, &b_event_fEvtHdr_fRun);
185.       fChain->SetBranchAddress("fEvtHdr.fDate", &fEvtHdr_fDate, &b_event_fEvtHdr_fDate);
186.       fChain->SetBranchAddress("fTracks", &fTracks_, &b_event_fTracks_);
```

```
187.       fChain->SetBranchAddress("fTracks.fUniqueID", fTracks_fUniqueID, &b_fTracks_fUniqueID);
188.       fChain->SetBranchAddress("fTracks.fBits", fTracks_fBits, &b_fTracks_fBits);
189.       fChain->SetBranchAddress("fTracks.fPx", fTracks_fPx, &b_fTracks_fPx);
190.       fChain->SetBranchAddress("fTracks.fPy", fTracks_fPy, &b_fTracks_fPy);
191.       fChain->SetBranchAddress("fTracks.fPz", fTracks_fPz, &b_fTracks_fPz);
192.       fChain->SetBranchAddress("fTracks.fRandom", fTracks_fRandom, &b_fTracks_fRandom);
193.       fChain->SetBranchAddress("fTracks.fMass2", fTracks_fMass2, &b_fTracks_fMass2);
194.       fChain->SetBranchAddress("fTracks.fBx", fTracks_fBx, &b_fTracks_fBx);
195.       fChain->SetBranchAddress("fTracks.fBy", fTracks_fBy, &b_fTracks_fBy);
196.       fChain->SetBranchAddress("fTracks.fMeanCharge",
    fTracks_fMeanCharge, &b_fTracks_fMeanCharge);
197.       fChain->SetBranchAddress("fTracks.fXfirst", fTracks_fXfirst, &b_fTracks_fXfirst);
198.       fChain->SetBranchAddress("fTracks.fXlast", fTracks_fXlast, &b_fTracks_fXlast);
199.       fChain->SetBranchAddress("fTracks.fYfirst", fTracks_fYfirst, &b_fTracks_fYfirst);
200.       fChain->SetBranchAddress("fTracks.fYlast", fTracks_fYlast, &b_fTracks_fYlast);
201.       fChain->SetBranchAddress("fTracks.fZfirst", fTracks_fZfirst, &b_fTracks_fZfirst);
202.       fChain->SetBranchAddress("fTracks.fZlast", fTracks_fZlast, &b_fTracks_fZlast);
203.       fChain->SetBranchAddress("fTracks.fCharge", fTracks_fCharge, &b_fTracks_fCharge);
204.       fChain->SetBranchAddress("fTracks.fVertex[3]", fTracks_fVertex, &b_fTracks_fVertex);
205.       fChain->SetBranchAddress("fTracks.fNpoint", fTracks_fNpoint, &b_fTracks_fNpoint);
206.       fChain->SetBranchAddress("fTracks.fValid", fTracks_fValid, &b_fTracks_fValid);
207.       fChain->SetBranchAddress("fTracks.fNsp", fTracks_fNsp, &b_fTracks_fNsp);
208.       fChain->SetBranchAddress("fTracks.fPointValue",
    fTracks_fPointValue, &b_fTracks_fPointValue);
209.       fChain->SetBranchAddress("fTracks.fTriggerBits",
    fTracks_fTriggerBits, &b_fTracks_fTriggerBits);
210.       fChain->SetBranchAddress("fLastTrack", &fLastTrack, &b_event_fLastTrack);
211.       fChain->SetBranchAddress("fWebHistogram", &fWebHistogram, &b_event_fWebHistogram);
212.       fChain-
    >SetBranchAddress("fTriggerBits.fUniqueID", &fTriggerBits_fUniqueID, &b_event_fTriggerBits_fUn
    iqueID);
213.       fChain-
    >SetBranchAddress("fTriggerBits.fBits", &fTriggerBits_fBits, &b_event_fTriggerBits_fBits);
214.       fChain-
    >SetBranchAddress("fTriggerBits.fNbits", &fTriggerBits_fNbits, &b_event_fTriggerBits_fNbits);
215.       fChain-
    >SetBranchAddress("fTriggerBits.fNbytes", &fTriggerBits_fNbytes, &b_event_fTriggerBits_fNbytes
    );
216.       fChain-
    >SetBranchAddress("fTriggerBits.fAllBits", &fTriggerBits_fAllBits, &b_fTriggerBits_fAllBits);
217.       fChain->SetBranchAddress("fIsValid", &fIsValid, &b_event_fIsValid);
218.    }
219.
220.    Bool_t selector::Notify()
221.    {
222.       // The Notify() function is called when a new file is opened. This
223.       // can be either for a new TTree in a TChain or when when a new TTree
224.       // is started when using PROOF. It is normally not necessary to make changes
225.       // to the generated code, but the routine can be extended by the
226.       // user if needed. The return value is currently not used.
227.
228.       return kTRUE;
229.    }
230.
231.    #endif
```

*Listagem 7.5 selector.h*

```
1.  #ifndef selector_cpp
2.  #define selector_cpp
3.
4.  /**
5.   * This file has been automatically generated
6.   * by void makeSelector_cpp(TTree*, std::string)
7.   * at Fri Oct 10 00:50:30 2014
8.   */
9.
10. #define selector_cxx
11. // This class is derived from the ROOT class TSelector.
12. // For more information on the TSelector framework see
13. // $ROOTSYS/README/README.SELECTOR or the ROOT User Manual.
14.
15. // The following methods are defined in this file:
16. //    Begin():        called every time a loop on the tree starts,
17. //                    a convenient place to create your histograms.
18. //    SlaveBegin():   called after Begin(), when on PROOF called only on the
19. //                    slave servers.
20. //    Process():      called for each event, in this function you decide what
21. //                    to read and fill your histograms.
22. //    SlaveTerminate: called at the end of the loop on the tree, when on PROOF
23. //                    called only on the slave servers.
24. //    Terminate():    called at the end of the loop on the tree,
25. //                    a convenient place to draw/fit your histograms.
26. //
27. // To use this file, try the following session on your Tree T:
28. //
29. // root> T->Process("selector.C")
30. // root> T->Process("selector.C","some options")
31. // root> T->Process("selector.C+")
32. //
33.
34. #include "selector.h"
35. #include <TH2.h>
36. #include <TStyle.h>
37.
38.
39. void selector::Begin(TTree * /*tree*/)
40. {
41.    // The Begin() function is called at the start of the query.
42.    // When running with PROOF Begin() is only called on the client.
43.    // The tree argument is deprecated (on PROOF 0 is passed).
44.
45.    TString option = GetOption();
46.
47. }
48.
49. void selector::SlaveBegin(TTree * /*tree*/)
50. {
51.    // The SlaveBegin() function is called after the Begin() function.
52.    // When running with PROOF SlaveBegin() is called on each slave server.
53.    // The tree argument is deprecated (on PROOF 0 is passed).
54.
55.    TString option = GetOption();
56.
57. }
58.
```

```
59. event selector::GetEvent(Long64_t entry)        {
60.
61.         GetEntry(entry);
62.         event ev(out);
63.         {
64.                 // Copy of leafs
65.                  //ev.event = event; // makeEvent.cpp:1375
66.                 ev.fUniqueID = fUniqueID; // makeEvent.cpp:1579
67.                 ev.fBits = fBits; // makeEvent.cpp:1579
68.                 for(int i=0; i<20; ++i) {
69.                         // Char_t fType[20]
70.                         ev.fType[i] = fType[i]; // makeEvent.cpp:1575
71.                 }
72.                 ev.fEventName = fEventName; // makeEvent.cpp:1579
73.                 ev.fNtrack = fNtrack; // makeEvent.cpp:1579
74.                 ev.fNseg = fNseg; // makeEvent.cpp:1579
75.                 ev.fNvertex = fNvertex; // makeEvent.cpp:1579
76.                 ev.fFlag = fFlag; // makeEvent.cpp:1579
77.                 ev.fTemperature = fTemperature; // makeEvent.cpp:1579
78.                 for(int i=0; i<10; ++i) {
79.                         // Int_t fMeasures[10]
80.                         ev.fMeasures[i] = fMeasures[i]; // makeEvent.cpp:1575
81.                 }
82.                 for(int i=0; i<4; ++i)  {
83.                         for(int j=0; j<4; ++j)  {
84.                                 // Double32_t fMatrix[4][4]
85.                                 ev.fMatrix[i][j] = fMatrix[i][j]; // makeEvent.cpp:1571
86.                         }
87.                 }
88.                 for(int i=0; i<20; ++i) {
89.                         // Double32_t  fClosestDistance[20]
90.                         ev.fClosestDistance[i] = fClosestDistance[i]; // makeEvent.cpp:1519
91.                 }
92.                 ev.fEvtHdr_fEvtNum = fEvtHdr_fEvtNum; // makeEvent.cpp:1579
93.                 ev.fEvtHdr_fRun = fEvtHdr_fRun; // makeEvent.cpp:1579
94.                 ev.fEvtHdr_fDate = fEvtHdr_fDate; // makeEvent.cpp:1579
95.                 ev.fTracks_ = fTracks_; // makeEvent.cpp:1361
96.                 for(int i=0; i<kMaxfTracks; ++i)        {
97.                         // UInt_t  fTracks_fUniqueID[kMaxfTracks]
98.                         ev.fTracks_fUniqueID[i] = fTracks_fUniqueID[i];  // makeEvent.cpp:1503
99.                 }
100.                for(int i=0; i<kMaxfTracks; ++i)        {
101.                        // UInt_t  fTracks_fBits[kMaxfTracks]
102.                        ev.fTracks_fBits[i] = fTracks_fBits[i];  // makeEvent.cpp:1503
103.                }
104.                for(int i=0; i<kMaxfTracks; ++i)        {
105.                        // Float_t  fTracks_fPx[kMaxfTracks]
106.                        ev.fTracks_fPx[i] = fTracks_fPx[i];  // makeEvent.cpp:1503
107.                }
108.                for(int i=0; i<kMaxfTracks; ++i)        {
109.                        // Float_t  fTracks_fPy[kMaxfTracks]
110.                        ev.fTracks_fPy[i] = fTracks_fPy[i];  // makeEvent.cpp:1503
111.                }
112.                for(int i=0; i<kMaxfTracks; ++i)        {
113.                        // Float_t  fTracks_fPz[kMaxfTracks]
114.                        ev.fTracks_fPz[i] = fTracks_fPz[i];  // makeEvent.cpp:1503
115.                }
```

```
116.                    for(int i=0; i<kMaxfTracks; ++i)        {
117.                            // Float_t  fTracks_fRandom[kMaxfTracks]
118.                            ev.fTracks_fRandom[i] = fTracks_fRandom[i];  //
    makeEvent.cpp:1503
119.                    }
120.                    for(int i=0; i<kMaxfTracks; ++i)        {
121.                            // Float16_t  fTracks_fMass2[kMaxfTracks]
122.                            ev.fTracks_fMass2[i] = fTracks_fMass2[i];  //
    makeEvent.cpp:1503
123.                    }
124.                    for(int i=0; i<kMaxfTracks; ++i)        {
125.                            // Float16_t  fTracks_fBx[kMaxfTracks]
126.                            ev.fTracks_fBx[i] = fTracks_fBx[i];  // makeEvent.cpp:1503
127.                    }
128.                    for(int i=0; i<kMaxfTracks; ++i)        {
129.                            // Float16_t  fTracks_fBy[kMaxfTracks]
130.                            ev.fTracks_fBy[i] = fTracks_fBy[i];  // makeEvent.cpp:1503
131.                    }
132.                    for(int i=0; i<kMaxfTracks; ++i)        {
133.                            // Float_t  fTracks_fMeanCharge[kMaxfTracks]
134.                            ev.fTracks_fMeanCharge[i] = fTracks_fMeanCharge[i];  //
    makeEvent.cpp:1503
135.                    }
136.                    for(int i=0; i<kMaxfTracks; ++i)        {
137.                            // Float16_t  fTracks_fXfirst[kMaxfTracks]
138.                            ev.fTracks_fXfirst[i] = fTracks_fXfirst[i];  //
    makeEvent.cpp:1503
139.                    }
140.                    for(int i=0; i<kMaxfTracks; ++i)        {
141.                            // Float16_t  fTracks_fXlast[kMaxfTracks]
142.                            ev.fTracks_fXlast[i] = fTracks_fXlast[i];  //
    makeEvent.cpp:1503
143.                    }
144.                    for(int i=0; i<kMaxfTracks; ++i)        {
145.                            // Float16_t  fTracks_fYfirst[kMaxfTracks]
146.                            ev.fTracks_fYfirst[i] = fTracks_fYfirst[i];  //
    makeEvent.cpp:1503
147.                    }
148.                    for(int i=0; i<kMaxfTracks; ++i)        {
149.                            // Float16_t  fTracks_fYlast[kMaxfTracks]
150.                            ev.fTracks_fYlast[i] = fTracks_fYlast[i];  //
    makeEvent.cpp:1503
151.                    }
152.                    for(int i=0; i<kMaxfTracks; ++i)        {
153.                            // Float16_t  fTracks_fZfirst[kMaxfTracks]
154.                            ev.fTracks_fZfirst[i] = fTracks_fZfirst[i];  //
    makeEvent.cpp:1503
155.                    }
156.                    for(int i=0; i<kMaxfTracks; ++i)        {
157.                            // Float16_t  fTracks_fZlast[kMaxfTracks]
158.                            ev.fTracks_fZlast[i] = fTracks_fZlast[i];  //
    makeEvent.cpp:1503
159.                    }
160.                    for(int i=0; i<kMaxfTracks; ++i)        {
161.                            // Double32_t  fTracks_fCharge[kMaxfTracks]
162.                            ev.fTracks_fCharge[i] = fTracks_fCharge[i];  //
    makeEvent.cpp:1503
```

```
163.                     }
164.                     for(int i=0; i<kMaxfTracks; ++i)        {
165.                             for(int j=0; j<3; ++j)  {
166.                                     // Double32_t  fTracks_fVertex[kMaxfTracks][3];
    //[fTracks_]
167.                                     ev.fTracks_fVertex[i][j] = fTracks_fVertex[i][j]; //
    makeEvent.cpp:1465
168.                             }
169.                     }
170.                     for(int i=0; i<kMaxfTracks; ++i)        {
171.                             // Int_t  fTracks_fNpoint[kMaxfTracks]
172.                             ev.fTracks_fNpoint[i] = fTracks_fNpoint[i];  //
    makeEvent.cpp:1503
173.                     }
174.                     for(int i=0; i<kMaxfTracks; ++i)        {
175.                             // Short_t  fTracks_fValid[kMaxfTracks]
176.                             ev.fTracks_fValid[i] = fTracks_fValid[i];  //
    makeEvent.cpp:1503
177.                     }
178.                     for(int i=0; i<kMaxfTracks; ++i)        {
179.                             // Int_t  fTracks_fNsp[kMaxfTracks]
180.                             ev.fTracks_fNsp[i] = fTracks_fNsp[i];  // makeEvent.cpp:1503
181.                     }
182.                     // TODO::       handle with 'stars' at makeEvent.cpp:1506
183.                     for(int i=0; i<kMaxfTracks; ++i)        {
184.                             // Double32_t *fTracks_fPointValue[kMaxfTracks];
    //[fTracks_fNsp]
185.                             #warning Soft copying of pointer, it might not be safe to
    process this var in parallel
186.                             ev.fTracks_fPointValue[i] = fTracks_fPointValue[i]; //
    makeEvent.cpp:1510
187.                     }
188.                     for(int i=0; i<kMaxfTracks; ++i)        {
189.                             ev.fTracks_fTriggerBits[i] = fTracks_fTriggerBits[i]; //
    makeEvent.cpp:1409
190.                     }
191.                     ev.fLastTrack = fLastTrack; // makeEvent.cpp:1413
192.                     ev.fWebHistogram = fWebHistogram; // makeEvent.cpp:1413
193.                     ev.fTriggerBits_fUniqueID = fTriggerBits_fUniqueID; //
    makeEvent.cpp:1579
194.                     ev.fTriggerBits_fBits = fTriggerBits_fBits; // makeEvent.cpp:1579
195.                     ev.fTriggerBits_fNbits = fTriggerBits_fNbits; // makeEvent.cpp:1579
196.                     ev.fTriggerBits_fNbytes = fTriggerBits_fNbytes; // makeEvent.cpp:1579
197.                     for(int i=0; i<1; ++i)  {
198.                             // UChar_t  fTriggerBits_fAllBits[1]
199.                             ev.fTriggerBits_fAllBits[i] = fTriggerBits_fAllBits[i]; //
    makeEvent.cpp:1519
200.                     }
201.                     ev.fIsValid = fIsValid; // makeEvent.cpp:1579
202.             }
203.             return ev;
204.     }
205.
206.
207.     Bool_t selector::Process(Long64_t entry)
208.     {
209.        // The Process() function is called for each entry in the tree (or possibly
210.        // keyed object in the case of PROOF) to be processed. The entry argument
```

```
211.          // specifies which entry in the currently loaded tree is to be processed.
212.          // It can be passed to either selector::GetEntry() or TBranch::GetEntry()
213.          // to read either all or the required parts of the data. When processing
214.          // keyed objects with PROOF, the object is already loaded and is available
215.          // via the fObject pointer.
216.          //
217.          // This function should contain the "body" of the analysis. It can contain
218.          // simple or elaborate selection criteria, run algorithms on the data
219.          // of the event and typically fill histograms.
220.          //
221.          // The processing can be stopped by calling Abort().
222.          //
223.          // Use fStatus to set the return value of TTree::Process().
224.          //
225.          // The return value is currently not used.
226.
227.
228.          return kTRUE;
229.       }
230.
231.       void selector::SlaveTerminate()
232.       {
233.          // The SlaveTerminate() function is called after all entries or objects
234.          // have been processed. When running with PROOF SlaveTerminate() is called
235.          // on each slave server.
236.
237.       }
238.
239.       void selector::Terminate()
240.       {
241.          // The Terminate() function is the last function to be called during
242.          // a query. It always runs on the client, it can be used to present
243.          // the results graphically or save the results to file.
244.
245.       }
246.       #endif
```

*Listagem 7.6 selector.cpp*