Fizyka wysokich energii: Proof on Demand

Krótki opis usługi

Usługa *Proof on Demand* (PoD) przeznaczona jest do prowadzenia końcowych analiz fizycznych HEP. Pozwala na równoległe przetwarzanie danych w środowisku ROOT wykorzystując podział danych wejściowych i ich przetwarzanie przez niezależne procesy oraz zapewnia scalanie wyjściowych wyników. W ten sposób czas przetwarzania dużych ilości danych może być skrócony o czynnik 1/n, gdzie n oznacza liczbę uruchomionych procesów roboczych PoD. Aplikacja PoD składa się z serwera oraz z procesów roboczych. Podstawową metodą jest wykorzystanie procesów uruchamianych w systemie zadań wsadowych. Procesy robocze zgłaszają swoją gotowość do serwera PoD i od tego momentu mogą być wykorzystane do analizy danych. Wymagana jest podstawowa znajomość posługiwania się pakietem ROOT.

Aktywowanie usługi

Aby móc korzystać z usługi PoD należy mieć założone konto w Portalu PL-Grid i aktywować usługi:

1. Dostęp do klastra w ośrodku obliczeniowym (np. ZEUS)
2. Dostęp do UI w ośrodku obliczeniowym (np. Cyfronet, ICM, PCSS)
3. Platforma dziedzinowa HEPGrid: usługa Proof on Demand

Usługi aktywuje się w Portalu Użytkownika, zgodnie z opisem.

Ograniczenia w korzystaniu

Aplikacji nie należy uruchamiać na serwerach dostępowych UI ze względu na potencjalnie duże obciążenie podczas intensywnych obliczeń. Instrukcja poprawnego użycia PoD opisana została na początku sekcji *Pierwsze kroki.*

Autorzy aplikacji PoD dostarczają skrypt inicjalizacyjny jedynie dla powłoki bash. Jeżeli użytkownik używa innej powłoki domyślnej przy logowaniu, konieczne jest przejście do powłoki bash.

Aplikacja przygotowana została dla systemu SL6 (Scientific Linux 6). Ze względu na malejące zasoby dostępne w systemie SL5, udostępniona została tylko wersja dla SL6. Zarówno serwer PoD jak i procesy robocze muszą być uruchamiane na zasobach SL6, tak jak jest to opisane w sekcji *Pi erwsze kroki.*

Pierwsze kroki

Przykład dla klastra Zeus

Inicjalizacja (każdorazowo przed uruchomieniem PoD)

Należy zalogować się na UI a następnie uruchomić sesję interakcyjną za pomocą komendy

```
qsub –I –X –l nodes=1:sl6
```

Należy przejść do powłoki *bash*, jeżeli nie jest to domyślna powłoka użytkownika.

```
bash
```

Usługa PoD udostępniana jest w formie modułu. Należy aktywować odpowiedni moduł:

```
module add plgrid/apps/pod
```

Inicjalizacja PoD polega na wykonaniu skryptu konfiguracyjnego przygotowanego dla powłoki bash:

```
source $PODINI/podini.sh
```

Po tych krokach środowisko pracy PoD jest w pełni skonfigurowane.

Jednorazowa konfiguracja (tylko przy pierwszym uruchomieniu Pod)

Po pierwszej inicjalizacji w katalogu $HOME utworzony zostanie podkatalog .PoD a w nim PoD.cfg. Należy zmodyfikować jedna linię w sekcji [pbs_plugin] :

```
options_file=$POD_LOCATION/etc/Job.pbs.option
```

na

```
options_file=$HOME/.PoD/etc/Job.pbs.option
```

a następnie utworzyc zbiór `$HOME/.PoD/etc/Job.pbs.option` o zawartości:

```
-l nodes=1:sl6
```

Dodanie tej opcji zapewnia uruchomienie procesu roboczego na zasobach SL6.

Uruchomienie

Server PoD jest uruchamiany za pomocą komendy

```
pod-server start
```

a procesy robocze uruchamiane są za pomocą polecenia:

```
pod-submit -r pbs -q l_short -n 5
```

gdzie –n 5 oznacza żądaną liczbę procesów roboczych. Sprawdzenie aktywnych procesów roboczych umożliwia polecenie:

```
pod-info –n
```

Należy pamiętać że procesy robocze uruchamiane są jako zadania systemu kolejkowego PBS. Szybkość ich pojawienia się zależy od dostępności wyspecyfikowanej kolejki PBS. Status zadania można zbadać za pomocą standardowej komendy PBS „*qstat –u username*" (zadania PoD mają przypisaną nazwę *pod*).

Pakiet ROOT zawiera klasę do testów pakietu PROOF. Należy uruchomić aplikacje ROOT za pomocą polecenia

```
root
```

A następnie zadeklarować obiekt do testów PROOF i uruchomić test

```
root[] TProofBench pb(gSystem->GetFromPipe("pod-info -c"))
```

```
root[] pb.RunCPU()
```

Pomyślne wykonanie testu świadczy o poprawnym działaniu PoD.

Ustawienie grantu do obliczeń PoD innego niż domyślny

Aby przeprowadzić obliczenia z wykorzystaniem innego grantu niż domyślny dodać do linii w zbiorze

```
$HOME/.PoD/etc/Job.pbs.option
```

opcję -A moj grant:

```
–l nodes=1:sl6 -A moj_grant
```

gdzie *moj_grant* jest nazwą grantu pod którym wykonywane będą obliczenia. Sesję *pod-server* należy także utworzyć z odpowiednim parametrem:

```
qsub –I –X –l nodes=1:sl6 -A moj_grant
```

Zaawansowane użycie

Uruchomienie roota

```
root
```

Powiadomienie root'a o PoD i podłączenie PROOFa do analiz wielowątkowych:

```
TProof *p = TProof::Open("pod://")
```

Przygotowanie zadania do analizy równoległej

Tworzenie zadania

Aby uruchomić zadanie obsługiwane przez PROOF, klasa wykonywalna musi dziedziczyć po bazowej klasie TSelector. Aby uzyskać pliki (nagłówkowy i z ciałem klasy) służące do analizy, w tym celu najprościej skorzystać z predefinowanej komendy root'a, która dla danego drzewa tworzy potrzebne pliki. Jest to metoda MakeSelector.

W przykładzie poniżej otwieramy plik o nazwie 'kr_r015577_f000.root' znajdujący się w katalogu, w którym otworzyliśmy root'a (linia 1). Następnie uzyskujemy dostęp do drzewa w nim zapisanego o nazwie 'ClusterKr' (linia 2). Dla danego drzewa tworzymy pliki z zadaniem o nazwie 'MySelector' (linia 3); sa to pliki MySelector.h oraz MySelector.C utworzone w katalogu, w którym się znajdowaliśmy uruchamiając root'a. W linii 4 listujemy te pliki.

```
TFile *f=TFile::Open("PATH_TO_ROOT_FILE/kr_r015577_f000.root")
TTree *t =(TTree*)f->Get("ClusterKr")
t->MakeSelector("MySelector")
.!ls MySelector*
```

gdzie PATH_TO_ROOT_FILE oznacza ścieżkę do katalogu w którym znajduje się zbiór z danymi.

Współpraca z plikami

Istnieje kilka sposobów uruchamiania zadania z wykorzystaniem danych lub tez bez. a) Zadanie, które nie wykorzystuje danych zewnętrznych lecz należy daną czynność wykonać n razy np.: n=1000. Wówczas uruchamiamy:

```
p->Load("MySelector.C+")
MySelector *mysel = new MySelector(arg1, arg2)
p->Process(mysel, 1000)
```

lub bezposrednio:

```
p->Process("MySelector.C+", 1000);
```

b) Procesowanie na łańcuchu danych (uwaga: nazwę pliku należy podać ze ścieżką bezwzględną aby pliki były widoczne dla procesów na węzłach roboczych!)

```
TChain chain("ClusterKr")
chain.Add("PATH_TO_DATA/kr_r015577_f000.root");
chain.Add("PATH_TO_DATA/kr_r015577_f001.root");
chain.Add("PATH_TO_DATA/kr_r015577_f002.root");
chain.Add("PATH_TO_DATA/kr_r015577_f003.root");
chain.SetProof();
chain.Process("MySelector.C+")
```

gdzie PATH_TO_DATA oznacza bezwzględną ścieżkę do katalogu gdzie znajdują się dane.

c) Procesowanie na kolekcji danych. Do tworzenia kolekcji danych mamy dwie możliwości. Albo tworzymy plik tekstowy ze ścieżkami do plików użytych do utworzenia kolekcji przed uruchomieniem roota, albo dodajemy pliki ręcznie już w sesji root'a lub w skrypcie root'a. Pierwsze rozwiązanie wygląda następująco:

```
ls `pwd`/kr*root>kr_r015577.list
root
TFileCollection *mycollection = new TFileCollection("nazwa", "tytul","kr_r015577.list")
```

gdzie kolekcja występuje pod nazwa nazwa, tytulem *tytul*, i jest utworzona z plików znajdujących się pliku 'kr_r015577.list' w bieżącym katalogu. Musimy jeszcze ustawić nazwę drzewa, z którego będziemy korzystać w analizie. Robimy to metodą

```
mycollection->SetDefaultTreeName("ClusterKr");
```

gdzie jako drzewa użyliśmy 'ClusterKr'.

Drugie rozwiązanie wygląda następująco (uwaga: nazwę pliku należy podać ze ścieżką bezwzględną aby pliki były widoczne przez węzły robocze!):

```
TFileCollection *krypton = new TFileCollection("nazwa","tytul");
krypton->Add("PATH_TO_DATA/kr_r015577_f000.root");
krypton->Add("PATH_TO_DATA/kr_r015577_f001.root");
krypton->Add("PATH_TO_DATA/kr_r015577_f002.root");
krypton->Add("PATH_TO_DATA/kr_r015577_f003.root");
```

Tutaj dodajemy do kolekcji kolejne pliki metoda Add(). Pola nazwy i tytułu mogą być łańcuchami pustymi, tj.: "".

Procesowanie w PROOF'ie uruchamiamy podając nazwę kolekcji lub wskaźnik do niej.

```
p->Process("nazwa","MySelector.C+")
```

lub

```
p->Process(mycollection ,"MySelector.C+")
```

d) Procesowanie na zarejestrowanym zestawie danych. Aby sprawdzić jakie zestawy danych są już udostępnione w sesji PROOFa wpisujemy polecenie:

```
p->ShowDataSets()
```

W naszym przypadku dostajemy np.:

```
Dataset repository: /home/username/.proof/datasets
Dataset URI                              | # Files | Default tree | # Events |   Disk   | Staged
/default/username/krypton                |      13 | /ClusterKr   | 2.64e+07 |   681 MB |  100 %
/default/username/new_kr                 |       4 | /ClusterKr   | 8.205e+06|   210 MB |  100 %
```

gdzie *username* oznacza login użytkownika.

Aby zarejestrować zestaw danych należy użyć metody RegisterDataSet("nazwa_zestawu_danych_w_PROOF",kolekcja), np.:

```
p->RegisterDataSet("nowa",krypton)
```

Taka kolekcja nie posiada jednak zarejestrowanej nazwy drzewa i liczby przypadków. To co dostaniemy po wyświetleniu dostępnych zbiorów danych to:

```
Dataset repository: /people/username/.proof/datasets
Dataset URI                              | # Files | Default tree | # Events |   Disk   | Staged
/default/username/krypton                |      13 | /ClusterKr   | 2.64e+07 |   681 MB |  100 %
/default/username/new_kr                 |       4 | /ClusterKr   | 8.205e+06|   210 MB |  100 %
/default/username/nowa                   |       2 |        N/A              |    95 MB |    0 %
```

W celu weryfikacji zbioru danych należy

```
p->VerifyDataSet("nowa")
```

Aby dokładnie zapoznać się z zestawem danych używamy metody ShowDataSet(), np.:

```
p->ShowDataSet("nowa","MF")
```

Do usuwania predefiniowanych zestawów danych dostępnych w PROOFie służy:

```
p->RemoveDataSet("nowa")
```

W końcu, aby procesować zestaw danych selektorem uruchamiamy, np.:

```
p->Process("nowa","MySelector.C+")
```

Jeżeli chcemy procesować zestaw danych, który jest zarejestrowany lecz nie zweryfikowany musimy ustawić opcje:

```
p->SetParameter("PROOF_LookupOpt", "all")
```

Kończenie pracy

Aby zakończyć pracę, wydajemy komendę:

```
pod-server stop
```

po której zamykane są; wszystkie węzły obliczenioweg (*WN - worker nodes*) i serwer PoD. Zamknięcie sesji odbywa się także w przypadku braku aktywności interakcyjnej w czasie 30 minut serwera.

Przykład selektora oraz pliki z danymi znajdują się w katalogu:

```
$PLG_GROUPS_SHARED/podsoft/example/
```

Aby uruchomić przykład, należy skopiować cały podkatalog z example do własnego katalogu roboczego.

Kolejność uruchamiania metod w makrze opartym na klasie bazowej TSelector.

Dostępne są dwa sposoby uruchamiania makr opartych na klasie bazowej TSelector. Pierwszy jest tzw. sposobem lokalnym (standardowym, liniowym), gdzie zapytanie wykonywane jest sekwencyjnie. Kolejność wywoływania metod jest pokazana na grafie poniżej:

```
Begin()
SlaveBegin()
Init()
   Notify()
      Process()
      ...
      Process()
   ...
   Notify()
      Process()
      ...
      Process()
SlaveTerminate()
Terminate()
```

Drugi sposób to działania rozproszone, uwzględniające obliczenia równoległe, w których używany jest PROOF. Na grafie poniżej pokazano, które metody są wykonywane na węźle głównym, a które na roboczych.

```
+++ CLIENT Session +++       +++ (n) WORKERS +++
Begin()
                             SlaveBegin()
                             Init()
                                 Notify()
                                 Process()
                                 ...
                                 Process()
                                 ...
                             Init()
                                 Notify()
                                 Process()
                                 ...
                                 Process()
                                 ...
                             SlaveTerminate()
Terminate()
```

Poniżej przedstawiono opis i znaczenie poszczególnych funkcji.

Begin(), SlaveBegin()

Funkcja Begin() jest wołana na samym początku. Zawsze działa w sesji klienta ROOT. Funkcja SlaveBegin() jest wywoływana w sesji klienta (opcja pierwsza liniowa) lub, w przypadku analizy wielowątkowej z udziałem PROOF na każdym z węzłów (WN). Wszystkie inicjalizacje, które są potrzebne dla funkcji Process() należy zatem umieścić w SlaveBegin(). Kod, który potrzebuje dostępu do lokalnego środowiska klienta, np. grafiki lub systemu plików należy umieścić w Begin (). Podczas pracy z PROOFem lista wejściowa (fInput) jest rozprowadzana do WN-ow po zakończeniu funkcji Begin(), a przed wywolaniem SlaveBegin(). W ten sposób obiekty z klienta stają się dostępne dla instancji TSelector'a na węzłach roboczych (WN). Argument *tree* jest przestarzały. (W przypadku PROOF'a argument *tree* jest niedostępny na kliencie i przekazywane jest 0. Funkcja Init() powinna być używana aby zaimplementować operacje zależne od argumentu *tree*.)

Sygnatura:

```
virtual void Begin(TTree *tree); virtual void SlaveBegin(TTree *tree);
```

Init()

Funkcja Init() jest wywoływana kiedy selektor potrzebuje inicjalizowac nowe drzewo (tree) lub łańcuch (chain). Zwykle zostają tu ustawione adresy gałęzi drzewa (branch addresses). Zwykle nie jest konieczne wprowadzanie zmian do automatycznie wygenerowanego kodu, lecz funkcja może zostać rozszerzona przez użytkownika jeżeli zachodzi taka potrzeba. Funkcja Init() będzie wywoływana wiele razy podczas pracy z PROOF'em.

Sygnatura:

```
virtual void Init(TTree *tree);
```

Notify()

Funkcja Notify() jest wywoływana za każdym razem kiedy otwierany jest nowy plik. Może to nastąpić dla nowego drzewa TTree w łańcuchu TChain lub kiedy nowe drzewo TTree jest wczytywane na początku pracy z PROOF'em. Zwykle w tym miejscu zostają zwrócone wskaźniki do gałęzi (branch pointers). Zwykle nie jest konieczne wprowadzanie zmian do automatycznie wygenerowanego kodu, lecz funkcja może zostać rozszerzona przez użytkownika jeżeli zachodzi taka potrzeba.

Sygnatura:

```
virtual Bool_t Notify();
```

Process()

Funkcja Process() jest uruchamiana dla każdego przypadka (entry) zapisanego w drzewie (tree) (lub możliwego obiektu kluczowego w przypadku PROOF'a), które jest procesowane. Argument 'entry' tejże funkcji specyfikuje, który przypadek w obecnie załadowanym drzewie będzie przeprocesowany. Może on zostać przekazany albo TTree::GetEntry(), albo TBranch::GetEntry() aby wczytać odpowiednio wszystkie części danych albo tylko wymagane części danych. Kiedy procesowany jest obiekt kluczowy w PROOF'ie obiekt ten jest już załadowany i jest dostępny przez wskaźnik fObject. Funkcja Process() powinna zawierać ciało całej analizy. Może zawierać proste lub bardziej wysublimowane kryteria selekcji, algorytmy analizy dla danego przypadku i typowo zawiera wypełnianie histogramów.

Sygnatura:

```
virtual Bool_t Process(Int_t entry);
```

SlaveTerminate(), Terminate()

Funkcja SlaveTerminate() zostaje wywoływana kiedy wszystkie przypadki lub obiekty zostały przeprocesowane. Kiedy pracujemy z PROOF'em jest wykonywana przez każdy WN. Może zostać użyta do post-processingu zanim częściowe rezultaty z każdego WN zostaną scalone (merging). Po zakończeniu SlaveTerminate() obiekty z list 'fOutput' na WN'ach są łączone przez system PROOF i zwracane do sesji klienta ROOT. Funkcja Terminate() jest ostania z uruchamianych w całym zadaniu. Zawsze jest uruchamiana na kliencie ROOT'a. Może zostać użyta do prezentacji graficznej rezultatów lub do zapisania ich do pliku.

Sygnatura:

```
virtual void SlaveTerminate(); virtual void Terminate();.
```

Gdzie szukać dalszych informacji?

Strony projektu PoD, gdzie można znaleźć podręcznik użytkownika w zakładce *Documentation*.

Instrukcje przygotowania i uruchamiania skryptów Proof znajdują się na stronach pakietu Proof