Klaster obliczeniowy

Wydanie 2.3

Politechnika Łódzka, Instytut Fizyki

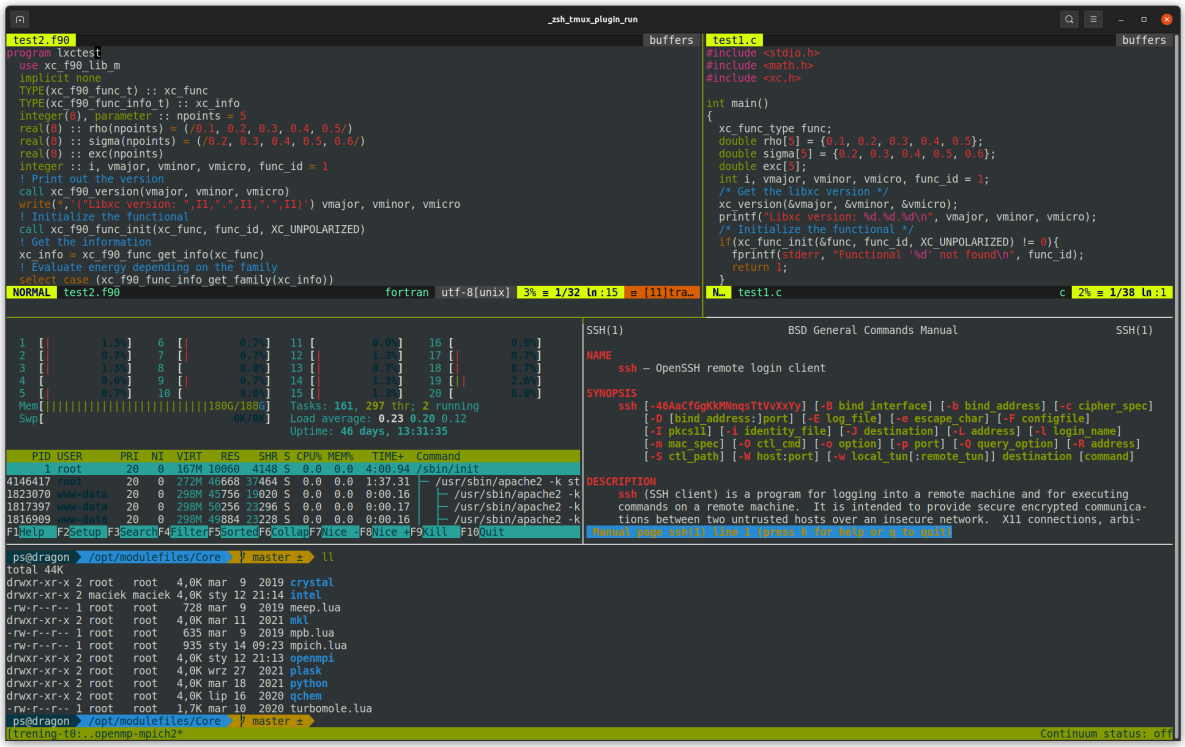
19 lut 2024

Spis treści

1 Użycie systemu SLURM

1.1 Wprowadzenie

Podstawowym zadaniem klastra `dragon` jest uruchamianie długo działających zadań (trwających godziny lub dni). W tym celu wykorzystuje się system zarządzania zadaniami SLURM, który umieszcza zadania w kolejce i przydziela im zasoby obliczeniowe, dbając o to, aby wszystkie zadania były wykonywane sprawiedliwie i efektywnie. W tym rozdziale opisane są podstawowe koncepcje związane z przesyłaniem zadań do klastra, takie jak jak zalogować się, jak przesyłać zadania, jak zarządzać zadaniami i jak określić wymagane zasoby obliczeniowe.



Rys. 1: Przykładowa sesja terminala ssh pokazuje możliwości programu `tmux` oraz konfigurowalny znak zachęty `zsh`.

1.2 Logowanie

Do klastra `dragon` można zalogować się zdalnie za pomocą protokołu `SSH`. W tym celu należy użyć programu `PuTTY` na Windowsie lub komendy `ssh` w systemie Linux. Pliki niezbędne do obliczeń można przesyłać za pomocą programu `WinSCP` na Windowsie lub komendy `scp` w systemie Linux (należy zapoznać się z ich podręcznikami). Adres klastra to `dragon.phys.p.lodz.pl`. Należy używać niestandardowego portu **6010**.

Do zalogowania należy użyć nazwy użytkownika i hasła, które zostały dostarczone przez administratora. Po pierwszym logowaniu zaleca się zmianę hasła za pomocą polecenia `passwd`. Wszystkie pliki można umieścić w katalogu domowym. Będą one dostępne ze wszystkich węzłów obliczeniowych poprzez wewnętrzny sieciowy system plików klastra.

Ostrzeżenie: Przy połączeniu SSH z klastrem `dragon`, należy użyć niestandardowego portu **6010** (zamiast domyślnego).

1.3 Uruchamianie zadań

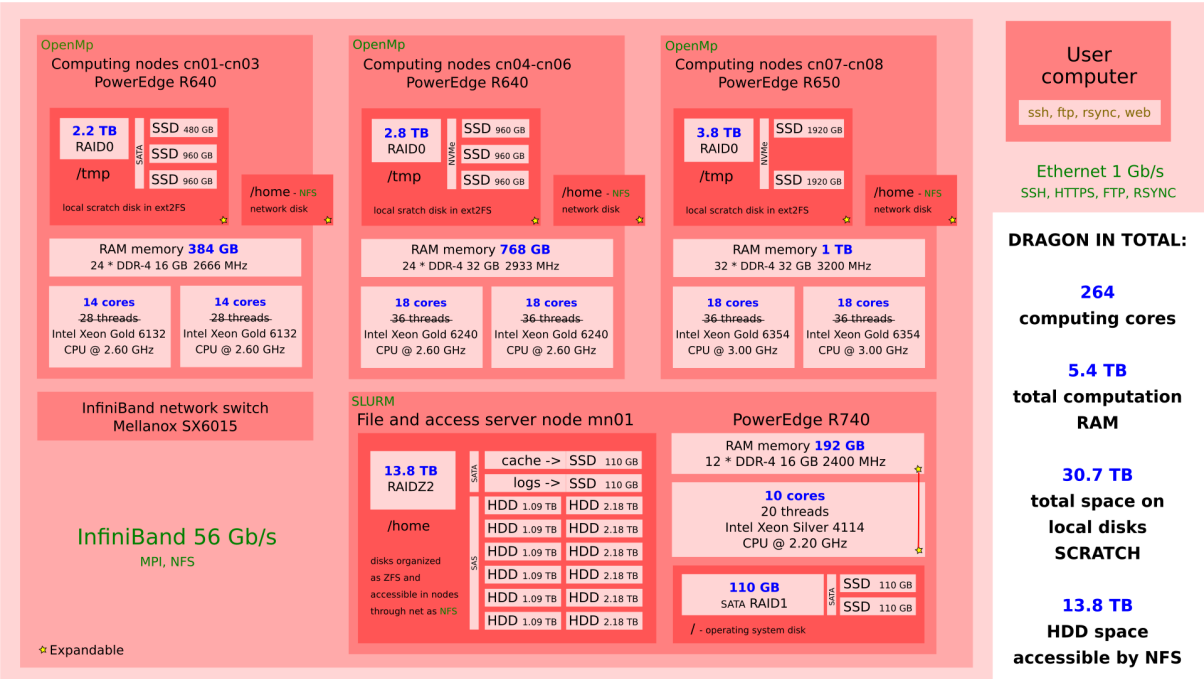
Nie należy uruchamiać aplikacji obliczeniowych bezpośrednio z wiersza poleceń podczas połączenia z klastrem. Może to spowolnić działanie serwera dla innych użytkowników, a uruchomione programy zostaną automatycznie ograniczone lub zakończone. Programy powinny być uruchamiane za pomocą systemu kolejkowego SLURM na węzłach obliczeniowych. Poniżej znajduje się krótki opis różnic między węzłem logowania a węzłami obliczeniowymi.

Węzeł logowania

Gdy użytkownik łączy się z klastrem, jest on połączony z jednym komputerem współdzielonym ze wszystkimi innymi użytkownikami, zwanym „węzłem logowania". Celem węzła jest przesyłanie zadań, kopiowanie danych, edytowanie skryptów itp. Nie wolno uruchamiać żadnych obliczeń bezpośrednio na tym węźle.

Węzły obliczeniowe

Komputery te wykonują ciężką pracę związaną z prowadzeniem obliczeń. Jednak użytkownik nie wchodzi z nimi w bezpośrednią interakcję. Zamiast tego, poleca on systemowi kolejkowania SLURM uruchomienie żądanych aplikacji, zaś SLURM znajduje dostępne węzły obliczeniowe i uruchamia na nich obliczenia.



Rys. 2: Szczegóły organizacji klastra `dragon`.

Zasoby obliczeniowe

Klaster HPC składa się z pewnej liczby węzłów obliczeniowych, z których każdy posiada zestaw procesorów, pamięci i (ewentualnie w przyszłości) układów GPU. Użytkownik przesyła zadania które określają jakie programy mają zostać uruchomione na tych węzłach, wraz z opisem wymaganych zasobów obliczeniowych.

Użytkownik może przesyłać zarówno zadania szeregowe, jak i równoległe. Zadanie szeregowe jest wykonywane na pojedynczym węźle przy użyciu dokładnie jednego procesora CPU. Zadania równoległe mogą współdzielić pamięć na pojedynczym węźle lub działać jako oddzielne procesy komunikujące się ze sobą. W celu wybrania typu uruchamianego zadania, należy określić ilość procesów (*task*) i liczbę procesorów dla każdego zadania. Poniższa tabela podsumowuje możliwe typy zadań.

Typ zadania	Liczba procesów	Liczba procesorów/proces
Szeregowe	1	1
Współdzielone równoległe (np. OpenMP)	1	2-64
Oddzielne równoległe (np. MPI)	2-64	1-64

Należy pamiętać, że liczba zadań pomnożona przez liczbę procesorów na zadanie może nie może przekraczać 64, chyba że nie zadania uruchamiane są w *kolejce* `free`.

Ponadto w przypadku klastra `dragon` istnieje większe rozdrobnienie dostępnych zasobów. Każdy użytkownik ma możliwość wykorzystania do 1080G pamięci RAM, a zasoby zależą od od używanej kolejki, jak to określono w poniższej tabeli:

Nazwa	Priorytet	PreemptMode	GrpTRES	MaxTRES	MaxTRESPerNode	MaxWall
normal	5	cluster		cpu=64		7-00:00:00
tiny	20	cluster		node=1	cpu=2,mem=2G	00:45:00
urgent	100	cluster	cpu=48			2-00:00:00
free	0	requeue				
long	3	cluster	cpu=64			28-00:00:00

Nadzorca zadań i menedżer zasobów

Nadzorca zadań i menedżer zasobów współpracują ze sobą w celu uruchamiania zadań na klastrze HPC. Nadzorca zadań jest jest odpowiedzialny za znalezienie i przydzielenie zasobów, które umożliwią wykonanie zadań w jak najkrótszym dostępnym czasie. Gdy zadanie jest zaplanowane do uruchomienia nadzorca instruuje menedżera zasobów, aby uruchomił program(y) na z wykorzystaniem przydzielonych zasobów. Jest to określane po prostu jako „uruchomienie zadania".

Można określić warunki uruchamiania zadań. Na przykład, jednym z warunków jest zakończenie (pomyślne lub niepomyślne) wcześniej przesłanego zadania. Inne warunki obejmują dostępność określonej licencji lub dostęp do określonego systemu plików.

Zadania wsadowe

Aby przesłać skrypt wsadowy do SLURM, należy użyć polecenia `sbatch`. Jest ono zaprojektowane w taki sposób, aby umożliwić użytkownikowi przesłanie zadań wsadowych do systemu kolejkowania, jeśli żądają one zasobów, które nie mogą być dostarczone zgodnie ze specyfikacją. Daje to możliwość weryfikacji zadania i ponownego przesłania go z niezbędnymi poprawkami.

Zadania interaktywne

Zadanie interaktywne to zadanie, które udostępnia użytkownikowi wiersz poleceń (zamiast uruchomienie skryptu), gdy zostanie uruchomione. Aby przesłać interaktywne zadanie do do SLURM należy użyć polecenia `salloc`. Gdy żądane zasoby zostaną przydzielone, pojawi się wiersz poleceń i będzie można uruchomić program(y) za pomocą polecenia `srun`.

W poniższy przykładzie żądany jest węzeł obliczeniowy dla jednego zadania i alokowane jest 2GB pamięci RAM oraz 4 procesory dla zadania interaktywnego, które może trwać do jednej godziny. Następnie pokazane jest przykładowe uruchomienie polecenia `your_application` na przydzielonym węźle:

```
user@dragon:~$ salloc --time=1:00:00 --mem=2G -n1 -c4
salloc: Przyznano przydział zadania 1234
[1234] user@dragon:~$ srun your_application
```

Anatomia zadania wsadowego

Zadania wsadowe wykonywane są na węzłach obliczeniowych bez udziału użytkownika. W zadaniach tych definiuje się wymagane zasoby obliczeniowe, programy do uruchomienia oraz wszelkie dane wejściowe i opcje wyjściowe. Użytkownik przesyła zadanie, zwykle w formie skryptu zadania wsadowego, do nadzorcy zadań. Jest to preferowany sposób korzystania z klastra.

Skrypt zadania wsadowego składa się z czterech głównych elementów:

- Określenia interpretera używanego do wykonywania skryptu,
- dyrektyw `#SBATCH`, które definiują domyślne parametry zadania.
- definicji zmiennych środowiskowych i modułów (jeśli to konieczne).
- komend uruchamiających właściwe obliczenia z ich argumentami wejściowymi i opcjami.

Oto przykład skryptu wsadowego, w którym zdefiniowano uruchomienie 8 zadań na dwóch węzłach, przydzielając łącznie 2GB pamięci RAM na każdy węzeł. Obliczenia mogą trwać maksymalnie jedną godzinę. Jest on przypisany do *kolejki normal*. Kiedy zadanie jest uruchamiane, ładowany jest *moduł* `my_module` i uruchamiany program `my_application` na przydzielonych węzłach:

```
#!/bin/bash
#SBATCH -n 8 -N 2
#SBATCH --time=1:00:00
#SBATCH --mem=2G
#SBATCH --qos=normal
module add my_module
srun my_application
```

Gdy zadanie zostanie zaplanowane do uruchomienia, menedżer zasobów wykona skrypt zadania wsadowego na pierwszym przydzielonym węźle. Polecenie `srun` uruchomi dodatkowe zadania na wszystkich przydzielonych węzłach.

Konkretne instrukcje dotyczące uruchamiania dostępnych programów są przedstawione *w osobnym rozdziale*.

Określenie wymaganych zasobów

Uruchamiając zadania na klastrze należy ręcznie określić wymagane zasoby. Jeśli użytkownik zapomni tego zrobić, zadanie otrzyma 1KB pamięci RAM i będzie mogło działać tylko przez 1 minutę. Zasoby określa się jako argumenty wiersza poleceń programu do `sbatch` lub `salloc` lub w dyrektywie `#SBATCH` skryptu.

Podstawowe zasoby są następujące:

Opcja długa (krótka)	Typ zasobu
`--time` (`-t`)	Całkowity czas działania zadania. Zadanie zostanie zakończone po upływie tego czasu.
`-mem`	Pamięć wymagana dla wszystkich zadań na każdym węźle. Ilość RAM dla każdego zadania zależy od liczby zadań przypisanych do pojedynczego węzła.
`-mem-per-cpu`	Pamięć wymagana na jeden przydzielony procesor. Każde zadanie otrzyma określoną ilość pamięci RAM pomnożoną przez liczbę procesorów na zadanie. Ta opcja oraz `--mem` wzajemnie się wykluczają.
`--ntasks` (`-n`)	Maksymalna liczba zadań, które mogą być uruchomione jednocześnie dla tego zadania zadania jednocześnie. Same zadania są uruchamiane za pomocą komendy `srun`. Domyślnie jest to jedno zadanie.
`--cpus-per-task` (`-c`)	Liczba procesorów na pojedyncze zadanie. Domyślnie jest to jeden. Należy zwiększyć tę liczbę dla obliczeń równoległych w pamięci współdzielonej. SLURM automatycznie ustawia zmienną środowiskową `OMP_NUM_THREADS` na tę wartość.
`--nodes` (`-N`)	Liczba odrębnych węzłów przydzielonych do zadania. Ogólnie należy unikać określania tej opcji. Zamiast tego należy ustawić `--ntasks` oraz `--cpus-per-task`.

Podczas przesyłania zadań należy dokładnie rozważyć wymagane zasoby. Określenie zbyt niskich wartości naraża zadanie na ryzyko przerwania, zaś ustawienie zbyt wysokich wartości marnuje zasoby klastra i zmniejsza priorytet zadania. Co więcej, małe zadania mogą być uruchamiane wcześniej w celu wypełnienia wolnych obszarów zasobach klastra, przy użyciu tzw. algorytmu *backfill*.

Jak oszacować, ile pamięci potrzebuje zadanie?

Trudno jest z góry oszacować dokładne zapotrzebowanie na pamięć dla zadań. Zazwyczaj najlepszą rzeczą, jaką można zrobić, jest wykorzystanie informacji z podobnych, wcześniej ukończonych zadań. Poniższe polecenie wydrukuje podsumowanie wymaganych i wykorzystanych zasobów dla uruchomionych i zakończonych zadań wsadowych:

```
$ seff JOBID
```

Wynik tego polecenia wygląda następująco:

```
$ seff 1592
Job ID: 1592
Cluster: dragon
User/Group: user/user
State: COMPLETED (exit code 0)
Cores: 48
Nodes: 4 (12 cores per node)
Tasks: 24 (2 cores per task)
Elapsed Time: 00:08:02
CPU Utilized: 04:36:28
CPU Efficiency: 71.70% of 06:25:36 core-walltime
Max Memory Utilized: 2.74 GB (up to 934.75 MB/node)
Memory Efficiency: 68.46% of 4.00 GB (1.00 GB/node)
```

Z powyższych danych można odczytać całkowity czas trwania zadania oraz wykorzystaną pamięć. Dla nowego zadania należy ustawić opcję `--time` na wartość nieco wyższą niż widoczna (w powyższym przykładzie rozsądną wartością wydaje się `--time=10:00`, tj. 10 minut). Podobnie można oszacować wymaganą pamięć. Należy jednak pamiętać, że w przypadku zadań wieloprocesowych nie można przewidzieć, ile węzłów zostanie przydzielonych dla zadania, chyba że zostanie to wyraźnie określone. Dlatego bardziej rozsądne może być określenie ilości pamięcie przypadającej na jeden na CPU niż na węzeł. W powyższym przykładzie zadanie wykorzystało 2.74GB pamięci RAM dla 48 rdzeni (58 MB na rdzeń). W związku z tym całkiem rozsądnie byłoby podać `--mem-per-cpu=64MB`.

Innym ważnym wynikiem polecenia `seff` jest wydajność procesora. Jeśli jej wartość jest niska, oznacza to, że wymagana jest zbyt duża liczba procesorów. Warto wtedy zmniejszyć liczbę procesorów na zadanie lub liczbę zadań, aby utrzymać wydajność jak najbliżej 100%.

Trzeba pamiętać, że nowe zadanie może mieć inne potrzeby. Jeżeli wymagany czas zostanie przeszacowany, zadanie może oczekiwać w kolejce dłużej niż to konieczne, ale żadne zasoby nie zostaną zmarnowane.

Z kolei, jeżeli zapotrzebowanie na pamięć zostanie oszacowane zbyt wysoko, zasoby prawdopodobnie zostaną zmarnowane. Dzieje się tak, z tego powodu, że jeżeli zadanie wykorzystuje tylko 4 rdzenie procesora, ale całą pamięć dostępną w węźle, to żadne inne zadanie nie zmieści się na nim i w efekcie wszystkie pozostałe rdzenie będą bezczynne.

Należy pamiętać, że jeśli zadanie POTRZEBUJE dużej ilości pamięci, to rezerwacja całej pamięci dostępnej w węźle jest całkowicie uzasadniona. Niemniej nie należy rezerwować jej „na wszelki wypadek" lub przy braku jakiegokolwiek oszacowania rzeczywistego zapotrzebowania na pamięć. Oszacowanie takie można zrobić na podstawie podobnych poprzednich zadań, korzystając z opisanego powyżej polecenia `seff`. W tym celu potrzebne są identyfikatory zadań, które można znaleźć w plikach wyjściowych zadań lub za pomocą polecenia `squeue` albo `sacct`.

Nazwy zadań i pliki wyjściowe

Wszystkie zadania otrzymują zarówno identyfikator zadania, jak i nazwę, w celu ich łatwiejszej identyfikacji w systemie wsadowym. Domyślną nazwą nadawaną zadaniu jest nazwa pliku skryptu zadania, co może utrudnić identyfikację zadania, w przypadku używania standardowej nazwy dla wielu skryptów. Dowolnemu zadaniu można nadać własną nazwę za pomocą opcji `-J`:

```
#SBATCH -J test_parametrow
```

Spowoduje to nazwanie zadania „test_parametrow".

Domyślnie - dla zadań przesłanych z `sbatch` - dane wyjściowe, które zadanie zapisuje do *stdout* i *stderr* są zapisywane do pliku o nazwie `slurm_%j.out`. Znak *%j* w nazwie pliku zostanie zastąpiony numerem zadania przypisanym przez SLURM. Zapewnia to, że plik wyjściowy zadania jest unikalny i nie będzie kolidował z plikami wyjściowymi innych zadań.

W wielu przypadkach domyślna nazwa pliku nie jest wygodna. Być może potrzebna jest nazwa pliku, która jest bardziej opisowa dla faktycznie uruchomionego zadania, lub zawierająca ważne metadane, takie jak parametry fizyczne. Nazwę plików wyjściowych można zmienić za pomocą opcji `-o` i `-e` programu `sbatch`. Opcja `-o` określa nazwę pliku zawierającego *stdout*, zaś opcja `-e` pliku zawierającego *stderr*. Dobrą praktyką jest dołączenie łańcucha `%j` do nazw plików. Zapobiegnie to nadpisywaniu plików plików wyjściowych. Poniżej znajduje się przykład:

```
#SBATCH -o calcflow-%j.out
#SBATCH -e calcflow-%j.err
```

Możesz podać tę samą nazwę pliku dla obu opcji, aby zapisać *stdout* i *stderr* do tego samego pliku. Również jeśli po prostu pominięta zostanie opcja `-e`, to *stderr* zostanie zapisany do tego samego pliku co *stdout*.

W przypadku używania *tablic zadań*, można zastąpić identyfikator zadania `%j` znakiem `%a`, który daje identyfikator całej tablicy i `%A` dla numeru elementu tablicy:

```
#SBATCH -o myarray-%a_%A.out
```

Partycje

Zadania SLURM są wykonywane na węzłach obliczeniowych. Ilość i rodzaje zasobów (procesory, pamięć, pamięć dyskowa itp.) mogą być zróżnicowane dla różnych węzłów obliczeniowych. Aby uzyskać przegląd węzłów i sprawdzić ich stan, należy zalogować się do klastra i wykonać polecenie:

```
$ sinfo
```

Partycje są używane przez SLURM do grupowania węzłów o identycznej lub podobnej konfiguracji. W przypadku klastra `dragon` — gdzie węzły są identyczne — nie ma potrzeby zwracać szczególnej uwagi na partycje. Każde zadanie zostanie przypisane do partycji najlepiej dostosowanej do zdefiniowanych wymogów.

Aby ręcznie wybrać partycję, należy użyć opcji `-p` poleceń `sbatch` i `salloc` lub w dyrektywie `#SBATCH` skryptu uruchomieniowego.

Priorytety zadań

Po przesłaniu zadania wsadowego lub zażądaniu zasobów dla zadania interaktywnego, zadanie to niekoniecznie jest uruchamiane natychmiast. Zamiast tego zadania zostaną uporządkowane w kolejce oczekujących zadań w oparciu o szereg czynników. Nadzorca zawsze będzie dążył do uruchomienia zadania znajdującego się na szczycie kolejki. Jeśli jednak zadanie zadanie nie wymaga wielu zasobów, może ono zostać uruchomione wcześniej, o ile nie opóźni rozpoczęcia żadnego zadania o wyższym priorytecie (jest to tzw. *back-fill*, czyli konserwatywne wypełnianie).

Czynniki wpływające na priorytet zadania można zobaczyć wywołując polecenie `sprio`. Czynniki te obejmują:

- Fair-share: liczba wskazująca, w jakim stopniu użytkownik korzystał ostatnio z klastra. Jest to najważniejszy czynnik, ponieważ zapewnia, że wszyscy użytkownicy będą mogli rozpocząć swoje zadania w rozsądnym czasie. Bieżące użycie można sprawdzić sprawdzić za pomocą polecenia `sshare`:

  ```
  $ sshare
     Account       User  RawShares  NormShares  EffectvUsage  FairShare
  ---------- ---------- ---------- ----------- ------------- ----------
  root                                 1.000000      1.000000   0.500000
   physics                       1    0.500000      1.000000   0.250000
    physics    username          1    0.125000      0.874713   0.007825
  ```

- Rozmiar zadania: liczba proporcjonalna do ilości zasobów obliczeniowych, których zażądało zadanie zasobów obliczeniowych. Mniejsze i krótsze zadania będą miały wyższy priorytet.
- Wiek: liczba proporcjonalna do czasu, jaki upłynął od przesłania zadania do kolejki.
- QOS: liczba przypisana do wybranej kolejki. Poniżej znajdują się szczegółowe informacje.

QOS

Podczas przesyłania zadania można wybrać jedną z kilku dostępnych kolejek (QOS). Pozwala to na na rozróżnienie między zadaniami o niższym i wyższym priorytecie. Na klastrze `dragon` użytkownik ma do wyboru następujące QOS:

`normal`
: Dla tych co nie mają pojęcia, o co chodzi z tym QOS i chcą po prostu uruchomić swoje obliczenia.

  Jest to domyślna kolejka i powinna być używana przez większość czasu. W jej ramach można uruchamiać zadania trwające do 7 dni i zużywające do 64 procesorów.

`tiny`
: Do zadań, które można by wykonać na laptopie, ale po co zużywać baterię.

  Ta kolejka jest przeznaczona do małych zadań seryjnych. Zadania są ograniczone do jednego procesora, 2GB RAM i mogą trwać tylko 45 minut. Powinna być używana do wstępnych obliczeń lub kompilacji nowego kodu. Ponieważ takie małe zadania nie powinny znacząco opóźniać innych, mają one zwiększony priorytet.

free

Coś dla skner!

Do zadań, które zużyją wszystkie dostępne zasoby, ale nie przejmują się tym, że mają najniższy możliwy priorytet. Tutaj nie ma ograniczeń innych niż fizyczne możliwości klastra. Ponadto zadania w tej kolejce nie zwiększają statystyk użycia klastra przez użytkownika. Innymi słowy, można wykonywać dowolne obliczenia za darmo!

Minusem jest to, że ta kolejka ma tak niski priorytet, że zadanie może zostać przerwane w dowolnym momencie (nazywa się to wywłaszczeniem zadania). Należy wziąć tę możliwość pod uwagę, poprzez częste zapisywanie częściowych wyników na dysk i projektowanie kodu w taki sposób, aby można go było wznowić od ostatniego zapisanego stanu po ponownym uruchomieniu. Gdy zadanie zostanie wywłaszczone, zostanie ono automatycznie umieszczone w kolejce i ponownie uruchomione, gdy zasoby będą znowu dostępne.

W tej kolejce ukończenie projektu badawczego może zająć trochę czasu, ale jest całkowicie darmowe!

urgent

Dla tych, którzy nie potrafią planować. Jeśli musisz wysłać artykuł albo zgłoszenie na konferencję na wczoraj, to jest to kolejka dla Ciebie!

Wiemy, jak działa nauka i planowanie może być trudne, szczególnie gdy należy oczekiwać nieoczekiwanego. Zatem w sytuacji krytycznej, zamiast panikować, można skorzystać z tej kolejki. Jest to jednak dość uciążliwe dla innych, którzy próbują spokojnie pracować w swoim tempie. W związku z tym obowiązują następujące zasady:

- Zadania w tej kolejce otrzymują ogromny priorytet, który daje niemal gwarancję przeskoczenia każdego innego zadania czekającego w kolejce.
- Do współczynnik użycia klastra jest naliczane 10 razy więcej (znormalizowanych) zasobów w porównaniu do kolejek `normal` lub `tiny`. Z tego powodu inne zadania użytkownika będą musiały czekać dłużej niż zwykle na swoją kolej. Obejmuje to także zadania w kolejce `urgent`, więc przesyłanie tutaj wielu zadań jest bezcelowe.
- Jeśli administratorzy serwera zauważą, że użytkownik korzysta z tej kolejki częściej niż raz na jakiś czas, dostep do niej zostanie zablokowany. Po tym niemiłym zdarzeniu, może on być przywrócony wyłączone po przekazaniu administratorom (obecnie Maciej i Piotr) odpowiednich łapówek w postaci piwa, ciasta lub innego eleganckiego napoju.

long

Czasami uzyskanie pożądanych rezultatów może zająć dużo czasu.

Ta kolejka jest przeznaczona do długich zadań (do 28 dni). Ma obniżony priorytet, a jeden użytkownik może zająć maksymalnie 64 procesory w tej kolejce. Ale można za to uruchomić obliczenia i wyjechać na wakacje do Hiszpanii. Gotowe wyniki będą czekały po powrocie. Trzeba tylko pamiętać, aby kupić każdemu administratorowi serwera butelkę dobrego wina (dlaczego to nigdy się jeszcze nie zdarzyło?).

Aby określić kolejkę podczas wysyłania zadania za pomcoą polecenia `sbatch` lub `salloc`, należy użyć przełącznika `--qos=` w linii poleceń lub w deklaracji `#SBATCH`.

1.4 Sprawdzanie statusu zadania

Status zadania można sprawdzić za pomocą polecenia `squeue`.

```
$ squeue
```

Najczęściej używanymi argumentami są `-u nazwa_użytkownika`, co pozwala na wyświetlanie tylko zadań danego użytkownika oraz `-j jobid` dla pokazania zadań o określonym identyfikatorze. Dodanie parametru `-l` pozwoli zobaczyć więcej szczegółowych informacji o zadaniach.

Alternatywnie, można użyć polecenia `sacct`. Polecenie to pokazuje przydatne informacje o bieżącym i przeszłym wykorzystaniu zasobów wykorzystaniu zasobów. Na przykład, aby uzyskać statystyki zadań własnych zadań, które rozpoczęły się po 2017-04-14 11:00:00 i zakończyły przed 2017-04-14 23:59:59 należy napisać:

```
$ sacct -S 2017-04-14T11:00:00 -E 2017-04-14T23:59:59
```

Aby uzyskać statystyki dla określonego zadania:

```
$ sacct -j 1234
```

1.5 Usuwanie zadań

Komenda `scancel` przerywa zadanie usuwając je z kolejki lub zabijając procesy zadania, jeśli zostało ono już uruchomione:

```
$ scancel 1234
```

Usuwanie wszystkich zadań za jednym razem:

```
$ scancel -u nazwa_użytkownika
```

1.6 Zmienne środowiskowe dostępne dla skryptów wykonawczych

W skryptach można używać wielu zmiennych środowiskowych, które są automatycznie ustawiane przez SLURM. Najbardziej przydatne z nich są wymienione poniżej:

Nazwa zmiennej	Opis
SLURM_ARRAY_TASK_COUNT	Całkowita liczba zadań w tablicy zadań.
SLURM_ARRAY_TASK_ID	Numer identyfikacyjny (indeks) w tablicy zadań.
SLURM_ARRAY_TASK_MAX	Maksymalny numer ID (indeksu) w tablicy zadań.
SLURM_ARRAY_TASK_MIN	Minimalny numer ID (indeksu) w tablicy zadań.
SLURM_ARRAY_TASK_STEP	Rozmiar kroku indeksu w tablicy zadań.
SLURM_ARRAY_JOB_ID	Numer identyfikacyjny zadania głównego w tablicy zadań.
SLURM_CLUSTER_NAME	Nazwa klastra, w którym wykonywane jest zadanie.
SLURM_CPUS_ON_NODE	Liczba procesorów w przydzielonym węźle.
SLURM_CPUS_PER_TASK	Liczba żądanych procesorów na zadanie. Ustawiane tylko jeśli `--cpus-per-task` jest określona.
SLURM_JOB_ID	Identyfikator zadania.
SLURM_JOB_CPUS_PER_NODE	Liczba procesorów dostępnych dla zadania w tym węźle.
SLURM_JOB_DEPENDENCY	Ustawione na wartość opcji *Wzajemne zależności zadań*.
SLURM_JOB_NAME	Nazwa zadania.
SLURM_JOB_NODELIST	Lista węzłów przypisanych do zadania.
SLURM_JOB_NUM_NODES	Całkowita liczba węzłów w alokacji zasobów zadania zadania.
SLURM_JOB_PARTITION	Nazwa partycji, w której uruchomione jest zadanie.
SLURM_JOB_QOS	Kolejka (QOS) alokacji zadania.
SLURM_MEM_PER_CPU	To samo co `--mem-per-cpu`
SLURM_MEM_PER_NODE	Taki sam jak `--mem`
SLURM_NODEID	Identyfikator przydzielonych węzłów.
SLURM_NTASKS	Taki sam jak `-n`, `--ntasks`
SLURM_PROCID	Numer MPI (lub względny identyfikator procesu) bieżącego procesu
SLURM_RESTART_COUNT	Liczba ponownych uruchomień zadania, jeśli zadanie zostało ponownie uruchomione z powodu awarii systemu lub zostało ponownie umieszczone w kolejce.
SLURM_SUBMIT_DIR	Katalog, z którego wywołano `sbatch`.
SLURM_SUBMIT_HOST	Nazwa hosta komputera, z którego wywołano `sbatch`. sbatch.
SLURM_TASKS_PER_NODE	Liczba zadań uruchomionych na każdym węźle. Wartości są oddzielone przecinkami i w tej samej kolejności co `SLURM_NODELIST`. Jeśli dwa lub więcej kolejnych węzłów mają mieć taką samą liczbę zadań, po tej liczbie występuje „`(x#)`”, gdzie „#„ jest liczbą powtórzeń. Na przykład, „`SLURM_TASKS_PER_NODE=2(x3),1`” oznacza, że pierwsze trzy węzły wykonają pierwsze trzy węzły wykonają po trzy zadania, a czwarty węzeł wykona jedno zadanie.
SLURM_TASK_PID	Identyfikator procesu uruchamianego zadania.
SLURMD_NODENAME	Nazwa węzła uruchamiającego skrypt zadania.

1.7 Tablice zadań

Aby przesłać dużą liczbę podobnych zadań, istnieją dwa możliwe podejścia. Po pierwsze, możliwe jest napisanie skryptu powłoki, który wielokrotnie wywoła polecenia `sbatch`.

Drugim podejściem - które jest prostsze i potencjalnie bardziej wydajne - jest przesłanie **tablicy zadań** przy użyciu jednego skryptu SLURM i pojedynczego wywołania polecenia `sbatch`. Tablice zadań pozwalają na uruchomienie dużej liczby podobnych zadań i zapewniają mechanizm, który pozwala użytkownikom klastra na odwoływanie się do całego zestawu zadań, tak jakby było to pojedyncze zadanie.

Przesyłanie tablic zadań

Tablice zadań są przesyłane z pomocą opcji `-a` lub `--array` w wywołaniu polecenia `sbatch` lub przez dołączenie dyrektywy `#SBATCH -a` do skryptu SLURM. Opcja `-a` przyjmuje rozdzielaną przecinkami listę numerów ID zadań lub jedną albo więcej par numerów ID zadań rozdzielonych myślnikiem.

Każde zadanie w tablicy zadań zostanie uruchomione z tym samym skryptem SLURM i w identycznym środowisku — z wyjątkiem wartości jego identyfikatora w tablicy. Wartość tego identyfikatora jest zapisywana w zmiennej środowiskowej `SLURM_ARRAY_TASK_ID`.

Na przykład, jeśli potrzebna jest tablica zadań z 10 elementami, ponumerowanymi od 1 do 10, polecenie przesłania byłoby następujące:

```
$ sbatch -a 1-10 array_script.sh
```

Można również określić wyraźne indeksy tablicy oddzielone przecinkami, albo dodać krok do określonego zakresu, używając „`:`”. W związku z tym poniższe polecenia są równoważne:

```
$ sbatch --array 0-20:4 array_script.sh
$ sbatch -a 0,4,8,12,16,20 array_script.sh
```

Ma końcu opcji `-a` może być dodany opcjonalny parametr, określający maksymalną liczbę elementów tablicy mogących, które mogą być uruchomione jednocześnie. Limit slotów jest określany przez dołączenie „`%`” do opcji `-a` po którym następuje wartość limitu. Na przykład, dwunastoelementową tablicę zadań z niesekwencyjnymi identyfikatorami i limitem slotów wynoszącym 3 można określić w następujący sposób:

```
$ sbatch -a 1-3,5-7,9-11,13-15%3 array_script.sh
```

Każde zadanie zawarte w tablicy zadań ma własną unikalną wartość indeksu w tablicy przechowywaną w zmiennej środowiskowej `SLURM_ARRAY_TASK_ID`. Wartość taj est dostępna w skrypcie zadania, tak jak każda inna zmienna środowiskowa. Jeśli zadanie uruchomiło skrypt powłoki `bash`, informacje o identyfikatorze tablicy zadania można wydrukować za pomocą następującego polecenia:

```
echo "Identyfikator tablicy elementu bieżącego zadania: ${SLURM_ARRAY_TASK_ID}"
```

Dostosowywanie danych dla tablicy zadań

Bardziej użytecznym wykorzystaniem indeksu w tablicy, pokazującym *prawdziwą* moc tablic, byłoby użycie go jako bezpośredniego lub pośredniego indeksu do danych przetwarzanych przez tablicę zadań.

Jedną z możliwości osiągnięcia tego celu jest użycie wartości `SLURM_ARRAY_TASK_ID` w celu zapewnienia niestandardowego zestawu parametrów wejściowych. Aby aby to zrobić, należy utworzyć plik tekstowy zawierający wiele wierszy, z których każdy składa się z szeregu wartości oddzielonych spacjami. W tym podejściu każdy wiersz w pliku danych zawiera parametry wejściowe wymagane przez jedno zadanie z tablicy. Skrypt SLURM należy zmodyfikować tak, aby zawierał on polecenie wczytujące właściwy wiersz pliku danych na podstawie parametru `SLURM_ARRAY_TASK_ID`. Istnieje wiele sposobów na odczytanie odpowiedniego wiersza z pliku danych. Poniżej pokazana jest przykładowa implementacja przy założeniu, że plik danych został nazwany `data.dat` i znajduje się on w tym samym katalogu, co skrypt uruchamiany dla każdego elementu tablicy zadań:

```
PARAMETERS=$(sed "${SLURM_ARRAY_TASK_ID}q;d" data.dat)
```

Zakładając, że program wykonywalny dla zadań w tej tablicy ma nazwę `command.sh`, skrypt SLURM powinien wyglądać jak poniżej:

```
./command.sh ${PARAMETERS}
```

Alternatywne podejście jest możliwe, jeśli unikalne parametry wejściowe wymagane przez każde zadanie w tablicy można obliczyć arytmetycznie. Na przykład, jeśli każda instancja skryptu `command.sh` wykonuje pętlę na jakimś zakresem wartości, w skrypcie SLURM można obliczyć maksymalne i minimalne wartości potrzebne dla każdego

zadania bezpośrednio na podstawie wartości w zmiennej środowiskowej SLURM_ARRAY_TASK_ID. Jeśli zakres w każdym zadaniu obejmuje 1000 wartości, można by to zrobić, włączając do skryptu SLURM następujące polecenia:

```
MAX=$(echo "${SLURM_ARRAY_TASK_ID}*1000" | bc)
MIN=$(echo "(${SLURM_ARRAY_TASK_ID}-1)*1000" | bc)
```

W tym podejściu plik danych, o którym mowa powyżej (data.dat) nie jest potrzebny, a wywołanie skryptu SLURM uruchamiające polecenie command.sh może wyglądać następująco:

```
command.sh ${MIN} ${MAX}
```

1.8 Wzajemne zależności zadań

Czasami przydatne jest uzależnienie zadania od innego zadania. Zależności pomiędzy zadaniami są używane do odroczenia rozpoczęcia zadania do momentu spełnienia określonych wymagań. Są one określane za pomocą opcji --dependency (krócej -d) w linii poleceń wywołania sbatch lub w dyrektywie #SBATCH.

```
$ sbatch --dependency=<type:job_id[:job_id][,type:job_id[:job_id]]> ....
```

Istnieją następujące typy zależności:

Zależność	Opis
after:jobid[:jobid...]	zadanie może rozpocząć się po tym, jak określone zadania zostały rozpoczęte
afterany:jobid[:jobid...]	zadanie może rozpocząć się po tym, jak określone zadania zostały zakończone
afternotok:jobid[:jobid...]	zadanie może rozpocząć się po tym, jak określone zadania nie powiodły się
afterok:jobid[:jobid...]	zadanie może rozpocząć się po tym, jak określone zadania zostały ukończone pomyślnie
singleton	zadania mogą rozpocząć wykonywanie po wszystkich wcześniej uruchomionych zadaniach o tej samej nazwie i nazwie użytkownika. Jest to przydatne do zestawiania wyników grupy zadań lub do wysłania powiadomienia na koniec grupy zadań.

Aby skonfigurować potoki przy użyciu zależności zadań, najbardziej przydatnymi typami są afterany, afterok i singleton. Najprostszym sposobem jest użycie zależności afterok dla pojedynczych, następujących po sobie zadań. Na przykład:

```
$ sbatch job1.sh
12345
$ sbatch --dependency=afterok:12345 job2.sh
```

Teraz, gdy job1 zakończy się kodem wyjścia równym zero, job2 stanie się kwalifikuje się do planowania. Jeśli jednak job1 nie powiedzie się (zakończy się niezerowym kodem wyjścia), job2 kodem wyjścia), ``job2 nie zostanie zaplanowane, ale pozostanie w kolejce i musi być zostać anulowane ręcznie.

Alternatywnie, można użyć zależności afterany i sprawdzić, czy pomyślnego wykonania warunków wstępnych można wykonać w samym skrypcie zadania.

1.9 Uruchamianie zadań MPI

Zadania MPI są natywnie obsługiwane przez SLURM. Na obu klastrach dostępne są dwa warianty MPI są dostępne: MPICH i OpenMPI. Można wybrać jeden z nich poprzez załadowanie ref:*module <modules>*, używając jednego z poniższych poleceń:

```
$ module add openmpi
```

lub

```
$ module add mpich
```

Zalecany sposób uruchamiania aplikacji różni się w zależności od wybranego MPI. Jeśli używasz OpenMPI, powinieneś użyć `mpirun` w skrypcie wsadowym skrypcie wsadowym lub interaktywnej powłoce. W przypadku MPICH można po prostu uruchomić program za pomocą polecenia `run`. W obu przypadkach nie trzeba ręcznie określać liczby procesów ani węzłów. Te informacje są automatycznie dostarczane przez SLURM, w zależności od liczby przydzielonych zadań.

Przykłady:

```
#!/bin/bash
#SBATCH -n 8
#SBATCH --time=1:00:00
#SBATCH --mem=2G
#SBATCH --qos=normal
module add openmpi
mpirun aplikacja_openmpi
```

```
#!/bin/bash
#SBATCH -n 8
#SBATCH --time=1:00:00
#SBATCH --mem=2G
#SBATCH --qos=normal
module add mpich
srun aplikacja_mpich
```

1.10 Podstawowe polecenia SLURM

Akcja	Polecenie
Kolejkowania zadań	`sbatch`
Usunięcie zadania	`scancel`
Lista wszystkich zadań w kolejce	`squeue`
Lista wszystkich węzłów	`sinfo`
Wyświetlanie informacji o węzłach	`scontrol show nodes`
Czas rozpoczęcia zadania	`queue --start`
Informacje o zadaniach	`scontrol show job`

Najważniejsze zmienne środowiskowe

Opis Zmienna	
Identyfikator zadania	`$SLURM_JOB_ID`
Katalog, z którego wywołano `sbatch`	`$SLURM_SUBMIT_DIR`
Host na którym wywołano `sbatch`	`$SLURM_SUBMIT_HOST`
Lista węzłów	`$SLURM_JOB_NODELIST`
Indeks w tablicy zadań	`$SLURM_ARRAY_TASK_ID`

Specyfikacja zadań

Opis	Dyrektywa
Dyrektywa w skrypcie	`#SBATCH`
Kolejka	`-p [partycja] --qos=[qos]`
Limit czasu wykonania	`-t [dni-hh:mm:ss]`
Plik wyjściowy	`-o [nazwa_pliku]`
Plik błędów	`-e [nazwa_pliku]`
Połączenie stdout/err	(użyj `-o` bez `-e`)
Powiadamianie	`--mail-type=[zdarzenia]`
Adres e-mail	`--mail-user=[adres]`
Nazwa zadania	`-J [nazwa]`
Restart zadania	`--requeue --no-requeue`
Rozmiar pamięci	`-mem=[M|G|T] -mem-per-cpu=[M|G|T]`
Liczba węzłów	`-N [min[-max]]`
Liczba procesorów	`-n [ilość] -c [ilość]`
Zależności zadania	`-d [stan:id_zadania]`
Tablice zadań	`-a [specyfikacja_tablicy]`
Zasoby ogólne	`--gres=[zasób]`

2 Podręcznik użytkownika modułów środowiskowych

Moduły środowiskowe to narzędzie do dynamicznej modyfikacji środowiska użytkownika za pomocą *plików modułów*. Dla każdego z dostępnych pakietów oprogramowania zainstalowanych w systemie istnieje odpowiedni plik modułu. Zatem, użytkownicy mogą użyć polecenia `module`, aby sprawdzić dostępne oprogramowanie w systemie. Polecenie to można też użyć, aby dynamicznie ustawić lub usunąć zmienne środowiskowe powłoki takie jak `PATH`, `INCLUDE`, `LD_LIBRARY_PATH`, `MANPATH` itp. dla odpowiedniego pakietu/wersji oprogramowania. Użytkownicy mogą umieścić polecenie `module` w swoim skrypcie uruchamiającym obliczenia, aby załadować odpowiednie moduły w razie potrzeby. Alternatywnie, można dodać polecenie modułu do swojego `~/.bashrc` `~/.bashrc`.

2.1 Sprawdzenie dostępnych modułów/pakietów

Aby sprawdzić jakie są dostępne moduły, należy wydać polecenie:

```
$ module avail
```

Przykład:

```
$ moduł avail

------------------------------------------------- /opt/modulefiles/Core ------------
↪-----------------------------------------------------------------
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
crystal/09.1.0.1        intel/2019.1.144          mkl/2019.1.144          mpich
↪              plask/openblas        qchem/mpich
crystal/09.2.0.1        intel/2021.1.1   (L,D)    mkl/2020.0.166          openmpi/2.1.
↪0             python/default        qchem/openmp
crystal/17.1.0.2 (D)    meep             (L)       mkl/2021.1.1   (L,D)    openmpi/4.0.
↪3rc4 (D)      python/2.7            qchem/serial (D)
intel/2013.2.146        mkl/default               mpb            (L)      plask/mkl
↪     (L,D)    python/3.8     (D)    turbomole

------------------------------------------ /usr/share/lmod/lmod/modulefiles/Core ----
↪-----------------------------------------------------------------
lmod/6.6    settarg/6.6

  Where:
    L:  Module is loaded
    D:  Default Module
```

2.2 Wyświetlenie listy załadowanych modułów/pakietów

Polecenie do wyświetlenia listy załadowanych modułów/pakietów to `module list`

```
$ module list

  Currently Loaded Modules:
    1) intel/2021.1.1   2) meep   3) mpb   4) mkl/2021.1.1   5) plask/mkl
```

2.3 Załadowanie modułu w celu ustawienia zmiennych środowiskowych

Polecenie akywujące moduł i ustawiające zmienne środowiskowe dla odpowiedniego pakietu oprogramowania to `module load [plik_modułu]`. Po jego użyciu, można sprawdzić załadowane moduły używając polecenia `module list`.

```
$ module load crystal
$ module list

  Currently Loaded Modules:
    1) intel/2021.1.1   2) meep   3) mpb   4) mkl/2021.1.1   5) plask/mkl   6)
↪crystal/17.1.0.2
```

Aby wyświetlić, jakie zmienne środowiskowe zostały ustawione w module, należy użyć polecenia `module display [modulename]`:

```
$ module display crystal

-------------------------------------------------------------------------------------
↪-------------------------------------------------------
    /opt/modulefiles/Core/crystal/17.1.0.2.lua:
-------------------------------------------------------------------------------------
↪-------------------------------------------------------
help([[
  This module loads CRYSTAL.
]])
whatis("Name: crystal ")
whatis("Description: This module loads CRYSTAL. ")
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
whatis("Version: 17.1.0.2 ")
setenv("CRY_EXEDIR","/opt/crystal/Linux-ifort17_XE_emt64/v1.0.2")
setenv("CRY_SCRDIR","/tmp")
setenv("CRY_MPIBIN","/opt/openmpi-2.1.0/bin/mpirun")
prepend_path("PATH","/opt/crystal/bin")
family("crystal")
```

Aby wyświetlić podstawowe informacje pomocy dla modułu należy użyć polecenia `module help`:

```
$ module help [plik_modułu]
```

2.4 Usuwanie/dezaktywacja modułów

Polecenie do usunięcia zmiennych środowiskowych ustawionych przez dany moduł to `module unload [plik_modułu]`. Po jego użyciu, można sprawdzić pozostałe załadowane moduły używając polecenia `module list`.

```
$ module unload crystal
$ module list

Currently Loaded Modules:
  1) intel/2021.1.1   2) meep   3) mpb   4) mkl/2021.1.1   5) plask/mkl
```

2.5 Usunięcie wszystkich załadowanych modułów

Polecenie pozwalające na usunięcie wszystkich załadowanych modułów to `module purge`.

```
$ module purge
```

2.6 Zapisywanie i przywracanie załadowanych modułów

Polecenie do zapisywania załadowanych modułów to `module save [nazwa_pliku]`. To polecenie zapisze aktywne moduły/pakiety w pliku `filename` w katalogu `~/.lmod.d`. Można załadować zapisane moduły używając `module restore [nazwa_pliku]`. Jeśli nazwa pliku zostanie w tym przypadku pominięta, to zostanie użyty domyślny plik `~/.lmod.d/default`.

```
$ module save
$ module list

Currently Loaded Modules:
  1) intel/2021.1.1   2) meep   3) mpb   4) mkl/2021.1.1   5) plask/mkl

$ module purge
$ module list

Currently Loaded Modules:

$ module restore
$ module list

Currently Loaded Modules:
  1) intel/2021.1.1   2) meep   3) mpb   4) mkl/2021.1.1   5) plask/mkl
```

2.7 Wyszukiwanie modułu

Polecenie `module spider` wyszukuje moduły pasujące do wyrażenia regularnego. Jeśli wyrażenie regularne zostanie pominięte, wówczas wylistowane zostaną wszystkie moduły. Wynikiem działania tego polecenia jest tabela z następującymi kolumnami:

- *Module*: Nazwa modułu.
- *Version*: Wersja modułu.
- *Description*: Krótki opis modułu.
- *Keywords*: Lista słów kluczowych używanych do wyszukiwania modułu.
- *URL*: Adres URL, za pomocą którego można uzyskać więcej informacji o module.

Komenda `module spider` jest potężnym narzędziem, które może być użyte do wyszukiwania modułów. Należy zauważyć, że komenda nie jest ona tym samym co polecenie `module avail`, które po prostu wyświetla listę modułów dostępnych dla użytkownika. Z kolei, polecenie `module spider` wyszukuje moduły pasujące do wyrażenie regularnego, lub — jeśli wyrażenie regularne zostanie pominięte — wszystkie moduły.

Komenda `module spider` jest przydatna, do znalezienia modułu dla określonego pakietu oprogramowania. Na przykład, aby znaleźć moduł dla pakietu FFTW, można użyć następującego polecenia polecenia:

```
$ module spider FFTW
```

Spowoduje to wyświetlenie listy wszystkich modułów dostępnych dla FFTW. Polecenie `module spider` nie rozróżnia wielkości liter, więc można również użyć następującego polecenia:

```
$ module spider fftw
```

W przypadku, gdy znana jest nazwa modułu, ale nie wiadomo, jaka wersja jest dostępna, można użyć następującego polecenia

```
$ module spider fftw/3.3
```

Spowoduje to wyświetlenie listy wszystkich modułów dostępnych dla FFTW 3.3.

2.8 Wygodne narzędzie `ml`

Dla wszystkich osób, które nie potrafią poprawnie wpisać komendy *mdoule*, *moduel*, lub *module* lmod ma rozwiązanie! Zamiast niej można użyć prostszego polecenia `ml`. Jako, że diw najczęściej używane komendy to *module list* i *module load* <*coś*>, `ml` pozwala na łatwe użycie każdej z nich:

```
$ ml
```

oznacza `module list`. Zaś

```
$ ml foo
```

oznacza `module load foo`. Z kolei

```
$ ml -bar
```

oznacza `module unload bar`. Nie powinien być zaskoczeniem fakt, że dwie powyższe formy można łączyć:

```
$ ml foo -bar
```

oznacza `module unload bar; module load foo`. Ponadto, dostępne są wszystkie standardowe polecenia komendy `module`:

```
$ ml save
$ ml avail
$ ml spider
$ ml show foo
```

Jeśli ktokolwiek kiedykolwiek będzie musiał załadować moduł o nazwie *spider*, to może napisać:

```
$ ml load spider
```

3 Dostępne oprogramowanie

3.1 PLaSK

PLaSK jest dostępny poprzez moduł `plask`. Możliwe jest wybranie wersji skompilowanej z użycuem biblioteki OpenBLAS lub Intel MKL. Aby wybrać właściwą wersję, należy wpisać odpowiednio `module load plask/openblas` lub `module load plask/mkl`.

Dostępna jest również większoś poprzednich wersji PLaSKa. Aby móc z nich korzystać, należy wpisać `module load oldplask`, a następnie można wybrać stare wersje programu za pomocą polecenia `module load plask/version`.

Przykład:

```
$ module load oldplask
$ module load plask/2019.01.01
```

Aby zobaczyć wszystkie dostępne stare wersje, należy wpisać:

```
moduł load oldplask
moduł dostępny
```

3.2 MPB i Meep

MPB

MPB to darmowy pakiet oprogramowania o otwartym kodzie źródłowym do badania kryształów fotonicznych. Aby z niego skorzystać, należy załadować moduł `mpb`. W celu użycia użyć wersji MPI programy MPB (MPICH lub OpenMPI), wystarczy załadować odpowiedni moduł MPI oprócz `mpb` i rozpocząć obliczenia za pomocą `mpirun`. W takim przypadku należy użyć pliku wykonywalnego `mpb-mpi`.

Przykład:

```
#!/bin/bash
#SBATCH --ntasks=8
#SBATCH --time=1:00:00
#SBATCH --mem=2G
#SBATCH --qos=normal

moduł add mpich
moduł add mpb

srun mpb-mpi plik_wejściowy.scm
```

Aby użyć interfejsu Pythona do MPB należy załadować moduł `meep`.

Meep

Meep to darmowy i otwarty pakiet oprogramowania do symulacji elektromagnetycznych za pomocą metody FDTD (finite-difference time-domain, metoda różnic skończonych w dziedzinie czasu). Aby z niego skorzystać, należy załadować moduł `meep`. W celu użycia programu Meep w wersji MPI (MPICH lub OpenMPI), należy po prostu załadować odpowiedni moduł MPI jako dodatek do `meep` i rozpocząć obliczenia za pomocą `mpirun`.

Przykład:

```
#!/bin/bash
#SBATCH --ntasks=8
#SBATCH --time=6:00:00
#SBATCH --mem=8G
#SBATCH --qos=normal

moduł add mpich
moduł add meep

srun python plik_wejściowy_meep.py
```