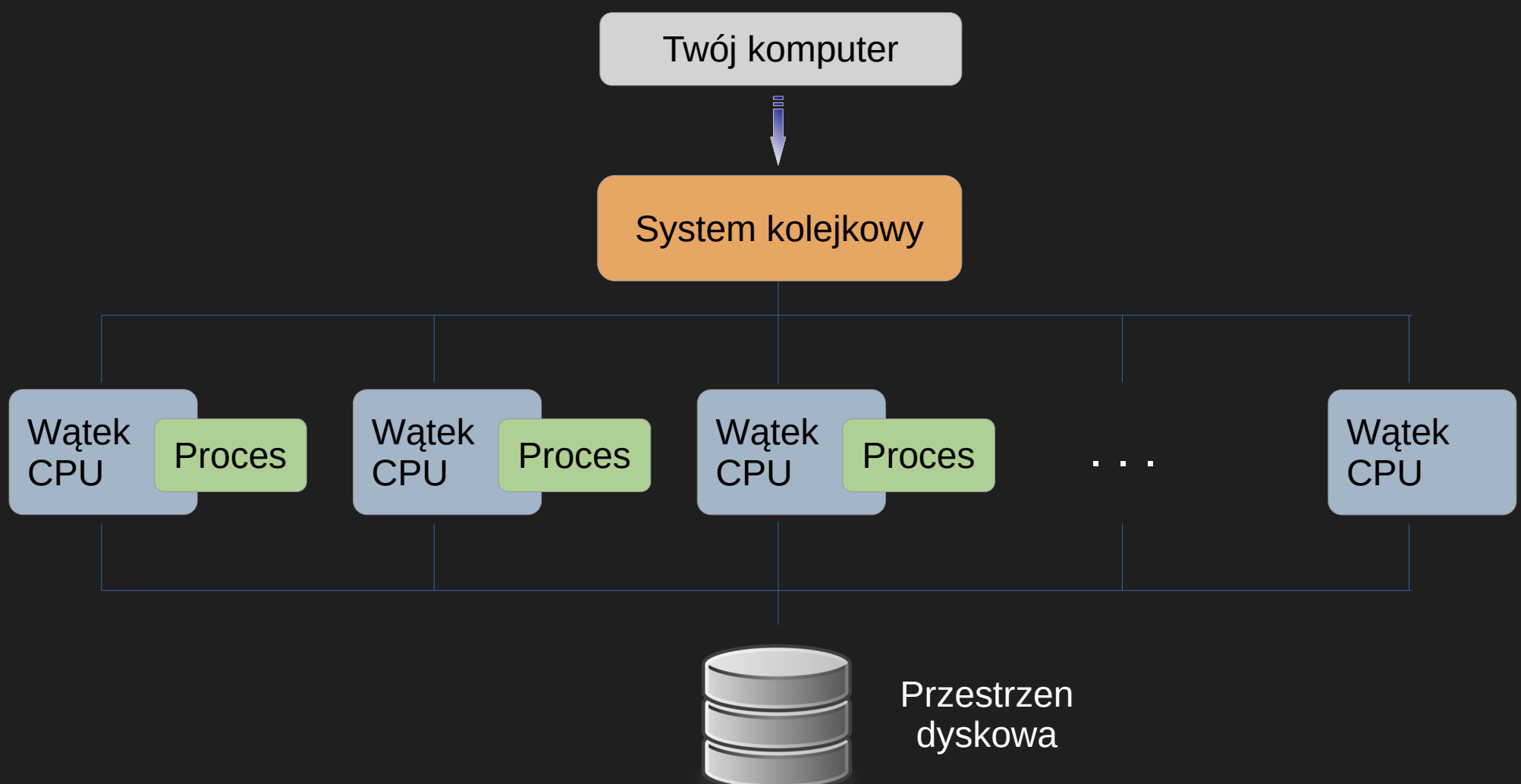


Narzędzia Wspierające Programowanie

System kolejkowy na farmie komputerowej

- Systemy kolejkowe są odpowiedzią na problem takich zadań obliczeniowych, których czas wykonywania staje się uciążliwie długi.
- Udostępnia się grupę wielu komputerów (wiele wątków CPU), z dużą, wspólną przestrzenią dyskową – nazywamy to **farmą komputerową**.
- Sednem zadania programisty jest podział długiego zadania na kolejne segmenty. Kod obliczeniowy (zwykle ten sam) jest włączany na grupie wątków CPU. Każdy proces analizuje niezależnie swój segment. Pliki wejściowe danych (jeśli są) i te wyjściowe mają unikalne nazwy/lokalizacje. Pliki wyjściowe zostaną “sklejone” na dalszych etapach analizy. Uwaga: nie oczekujemy żadnych okien graficznych (pliki graficzne można obejrzeć post-factum).
- Farmę zarządza “**system kolejkowy**” (**job scheduler**). Tworzy on **kolejki (queues)** dla **zadań (jobs)**: zadanie (np. program) do wykonania podaje się do kolejki, a gdy zwalnia się okno w zasobach, to system włącza proces (uruchamia kod). Manager zarządza zasobami w czasie rzeczywistym. Obsługując dany proces, kontroluje jego wyjście (stdout, stderr).



- Popularne systemy kolejkowe: np. PBS (tu omówimy) , SLURM.

Narzędzia Wspierające Programowanie

Farma komputerowa

- Omówimy tu system kolejkowy **PBS**. Podstawowe komendy, które będziemy używać:

qsub {opcje} plik_wykonywalny - podanie zadania z plikiem do kolejki

qstat ... - sprawdzenie statusu zadań w kolejce

qdel - skasowanie zadania

- ▶ Opanujmy najpierw obsługę jednego procesu. Zaczniemy od prostej aplikacji `sleep {N sek}`. i opakujemy ją w skrypt o nazwie `sleep1.sh`:

```
#!/bin/bash
echo I will sleep for 5 seconds.
date ; sleep 5 ; date
```

Wstawmy zadanie z tym skryptem do kolejki:

```
$ qsub sleep1.sh
```

Po chwili zadanie się wykona, a na naszej ścieżce pojawią się dwa pliki wyjściowe:

```
sleep1.sh.o{jobNo}    ← zawartość strumienia stdout. Tu tkwi wypis skryptu.
sleep1.sh.e{jobNo}    ← zawartość strumienia stderr. To jest puste.
```

- ▶ **Argumenty wejścia.** Niestety, nie działa tu bashowski mechanizm:
`./skrypt.sh opt1 opt2 ...` → `$1` , `$2` , ... w skrypcie

Przypuśćmy, że chcemy, aby w skrypcie-zadaniu pojawiła się zmienna `var` o wartości 123. Wprowadzamy ją opcją `-v` w poleceniu `qsub`:

```
$ qsub -v var=123 myJob.sh
```

Zmodyfikujmy nasz skrypt do `sleep2.sh`, aby liczbę sekund uśpienia podawać z zewnątrz:

```
if [ -z "$Seconds" ] ; then
    echo Usage: qsub -v Seconds={HowMany} ./sleep2.sh
    exit
fi
```

```
echo I will sleep for $Seconds seconds.
date ; sleep $Seconds ; date
```

Teraz wywołujemy: `$ qsub -v Seconds=5 ./sleep2.sh`

● Strumienie

Jak widać, każde zadanie kieruje strumienie `stdout` i `stderr` o plików (potocznie “logfiles”).
Przejmijmy kontrolę nad nazwami tych plików, poprzez opcje `-o` oraz `-e`:

```
$ qsub -o sleeper.log -e sleeper.err -v Seconds=5 ./sleeper2.sh
```

● Status zadań (qstat)

W każdej chwili można podejrzeć status swoich zadań:

```
$ qstat -u {mójLogin}
```

Otrzymamy mniej więcej taki wypis:

kruk-host.ift-klaster.lan:

Job ID	Username	Queue	Jobname	SessID	NDS	TSK	Req'd Memory	Req'd Time	S	Elap Time
7023484.kruk-hos	kpias	batch	sleeper2.sh	--	1	1	800mb	672:0	Q	--

- ⤴ Jak widać, proces wszedł do kolejki „batch” i otrzymał nr identyfikacyjny (Job ID).
Widzimy też informacje o limicie przydzielonej pamięci i czasie działania.
Pod literą S mamy symbol statusu:

- Q = queued (czeka w kolejce)
- R = running (działa)
- S = suspended (jest zawieszony)
- C = completed (ukończony)

Nb. napisanie tylko `qstat` – wypisze wszystkie zadania na farmie.

● Kolejki (queues) .

Farma posiada co najmniej 1 kolejkę (queue). Podawane jej zadania włączane są w kolejności nadejścia. Aby wylistować kolejki na farmie, piszemy:

```
$ qstat -Q
```

Kolejka ma zdefiniowane parametry, np. liczba CPU, max. pamięć dla zadania, jago max. czas itp. :

```
$ qmgr -c "list queue {nazwa kolejki}"
```

Nb. ten wypis nie pokazał parametru `max_running`. Oznacza on: max. liczbę procesów na raz. Możliwe, że manager nie ustawił takiego limitu.

Często manager farmy definiuje kilka kolejek, opisując np. na stronie www ich przeznaczenie. Np. kolejka A: do krótkich procesów, ale wyższy priorytet dostępu do CPU, kolejka B: do długich procesów, ale niższy priorytet (dłuższe oczekiwanie na włączenie) .

Aby podać zadanie na daną kolejkę, do `qsub` dodajemy opcję `-q {kolejka}` :

```
$ qsub -q batch -o sleeper.log -e sleeper.err -v Seconds=20 ./sleeper2.sh
```

● Ścieżka pracy (working directory)

Wstawmy nasz skrypt do podfolderu:

```
$ mkdir folder ; cp sleeper2.sh folder/ ; cd folder
```

Teraz wprowadźmy do treści skryptu `pwd` i puśćmy na farmę (↑ komenda powyżej). W logu widać, że byliśmy w katalogu domowym, a nie w miejscu podania zadania. Aby wymusić konkretną ścieżkę pracy, do `qsub` dodajemy `-d {ścieżka}` :

```
$ qsub -d $PWD -q batch -o sleeper.log -e sleeper.err -v Seconds=20 ./sleeper2.sh
```

● Zasoby (resources)

Można podać oczekiwany limit, jakiego nasze zadanie ma nie przekroczyć.

W tym celu dodamy do `qsub` opcję `-l` z argumentami. Np. dla limitu czasu 1' i pamięci 10 MB:

```
$ qsub -l cput=0:01:00,mem=10mb -d $PWD -q batch -o sleeper.log -e sleeper.err -v Seconds=20 ./sleeper2.sh
```

● **Recepty na rozwlekłe opcje w qsub.**

Zrobiło się dużo opcji. Wpisywanie ich jest uciążliwe, jak i możemy je zapomnieć.
Na ten problem są dwie recepty:

- ① Jeśli podawanym zadaniem jest skrypt, to na jego początku można wpisać te opcje, dodając linie rozpoczynające się od #PBS . Na bazie sleeper2 napiszmy sleeper3.sh :

```
#!/bin/bash
#PBS -q batch
#PBS -l cput=0:01:00,mem=10mb
#PBS -d /home/2/kpias/folder          ← tu podmień na swoją ścieżkę
#PBS -o sleeper.log
#PBS -e sleeper.err

if [ -z "$Seconds" ] ; then
    echo Usage: qsub -v Seconds={HowMany} sleeper2.sh
    exit
fi

echo I will sleep for $Seconds seconds.
date ; sleep $Seconds ; date
```

Teraz podajmy powyższy skrypt do kolejki. (Nb. w przypadku konfliktu, priorytet ma qsub).

```
$ qsub -v Seconds=5 sleeper3.sh
```

- ② Albo możemy utworzyć specjalny skrypt do podawania zadań ([submission script](#)) .
Dla czytelności, warto umieścić atrybuty w zmiennych i nakazać ich wypis,
a nawet wypisać pełne qsub z opcjami. Napiszmy taki “pełny” skrypt, submitSleeper.sh :

```
#!/bin/bash
if [ $# -ne 1 ] ; then
    echo Usage: ./submitSleeper.sh {No. of seconds}
    exit
fi

args="Seconds=$1"
logFile=./sleeper.log
errFile=./sleeper.err
queueName=batch
jobScript=./sleeper2.sh
jobWorkDir=$PWD
resources="cput=0:01:00,mem=10mb"

echo -e "* Submitting $jobScript to queue: $queueName with args: $args "
echo -e "* LogFiles      : [stdout= $logFile] [stderr= $errFile]"
echo -e "* Job's workdir : $jobWorkDir "
echo -e "* Max. resources: $resources"

cmd="qsub -q $queueName -v $args -l $resources -d $jobWorkDir -o $logFile -e $errFile $jobScript"
echo -e "\n$cmd \n"
eval "$cmd"

exit
```

● Wydobycie dalszych informacji.

- ① Włączając program, farma dodaje tam [zmienne z informacjami](#) , np.:

```
PBS_O_WORKDIR :    ścieżka pracy
PBS_O_HOME    :    katalog domowy użytkownika, który podał zadanie
PBS_JOBNAME    :    nazwa zadania
PBS_JOBID     :    numer zadania
```

- ② Zapoznając się z daną farmą, warto poznać podstawowe informacje o CPU(s), przestrzeni dyskowej, dystrybucji systemu itd.

W tym celu napiszmy skrypt `extractProperties.sh` :

```
#!/bin/bash
#PBS -d /home/2/kpias/folder
#PBS -o ./properties.log -j oe

echo "PBS_O_WORKDIR= $PBS_O_WORKDIR"
echo "PBS_O_HOME    = $PBS_O_HOME"
echo "PBS_JOBNAME    = $PBS_JOBNAME"
echo "PBS_JOBID     = $PBS_JOBID"

echo -e "\n* Memory report (free -h ) : \n"
free -h

echo -e "\n* Report on used disk (df -H . ) : \n"
df -H .

echo -e "\n No. of CPUs: `nproc` "
echo -e "\n No. of bogomips on this machine: `cat /proc/cpuinfo | grep bogomips | head -1 | awk '{print $3}'` "
echo -e "\n Linux distribution is: `cat /etc/issue` "
```

i wywołajmy:

```
$ qsub ./extractProperties.sh
```

● Porada

Podczas działania procesów nie można widzieć grafiki na bieżąco, a gdyby się kod wyrócił, nie dostajecie tej wiadomości „od razu” na terminal.

Dlatego ważne jest sprawdzanie w kodach sytuacji nietypowych (np. brak ścieżki, parametru itp) i przechwytywanie monitów do plików log.

- **Serializacja zadań** . Skrócenie czasu obliczeń poprzez podzielenie ich na segmenty (**runs**) do równoległego wykonania - to główny powód, dla którego używamy farm.

Potrzebujemy przemyśleć **strukturę zadań**.

- ▶ Dla czytelności, utwórzmy osobne katalogi dla:
 - danych wejściowych: `$ mkdir input`
 - logów: `$ mkdir log`
 - danych wyjściowych: `$ mkdir output`
- ▶ Ściągniemy proste dane: `$ wget http://www.fuw.edu.pl/~kpias/nwp/people2.dat`
 - ↳ Po nagłówku, mamy tu 20 wpisów o osobach. Przyjmijmy, że 3 ostatnie kolumny - to oceny. A celem naszego zadania - jest wystawienie średniej oceny każdej z osób.
- ▶ Najpierw jednak podzielmy dane na 4 osobne pliki `input_n.dat` po 5 wpisów. Zróbmy to w skrypcie `split.sh`, który umieści pliki w katalogu `input`.

```
#!/bin/bash
rm -f input/input_?.dat
LineNo=0

while IFS= read -r Line; do

    if (( ++LineNo ≤ 2 )) ; then
        continue
    fi
    (( OutIndex = (LineNo - 3) / 5 + 1 ))
    echo "$Line" >> input/input_${OutIndex}.dat

done < people2.dat
```

- ▶ Teraz uformujmy pojedyncze zadanie. Powinno ono znać nr indeksu segmentu (runu) danych. Niech kod analizuje plik wejściowy: `input/input_{nr}.dat`. Pamiętajmy o bezpieczniku: jeśli nie ma takiego pliku, to daj monit do `stderr` i zakończ skrypt. Wyznaczaniem średniej z 3 ostatnich kolumn niech zajmie się `awk`. Każda linia niech będzie zapisywana do `output/output_{nr}.dat`. Zapiszmy więc `task.sh`:

```
#!/bin/bash

inFile=input/input_${Run}.dat
outFile=output/output_${Run}.dat

if [ ! -e $inFile ] ; then
    echo -e "\nInput file $inFile not found. Escaping. \n" >&2
    exit
fi

echo -e "\nAnalysing input file:  $inFile \n"

awk ' { average = ($5 + $6 + $7) / 3 ;
      print $1"\t"$2"\t" average } ' $inFile > $outFile
```


● Serializacja zadań c.d.

- ▶ Potrzebujemy teraz napisać skrypt do podawania zadania (submission script). Powinien on:
 - przyjmować przedział indeksów segmentów (runów) danych
 - ↳ (bezpiecznik na złą liczbę arg. Wejścia)
 - ustalić parametry podawania zadania wspólne dla każdego segmentu (runu)
 - ↳ (i czytelnie je wyświetlić)
 - wykonać pętlę po segmentach (runach). W każdym kroku:
 - wstawić nr kroku do zmiennej przekazywanej do `task.sh` i do nazw plików logu
 - uformować komendę podania zadania
 - czytelnie wyświetlić informacje, powyższą komendę najlepiej też
 - podać zadanie do kolejki farmy

Przykład takiego skryptu (nazwijmy go: `submitTasks.sh`)

```
#!/bin/bash

if [ $# -ne 2 ] ; then
    echo Usage: ./submitTask.sh {Run From} {Run To}
    exit
fi

queueName=batch
jobScript=./task.sh
jobWorkDir=$PWD
resources="cput=0:01:00,mem=10mb"

echo -e "* Submitting $jobScript to queue: $queueName in runs [ $1 : $2 ]\n"
echo -e "* Job's workdir : $jobWorkDir "
echo -e "* Max. resources: $resources \n"

for i in $(seq $1 $2) ; do
    logFile=./log/task${i}.log
    errFile=./log/task${i}.err
    args="Run=$i"

    command="export Run=$i ; ./task.sh "          ← przetestuj przed podaniem na farmę

    # command="qsub -q $queueName -v $args -l $resources -d $jobWorkDir -o $logFile
    # -e $errFile $jobScript"

    echo -e "\n| Run : $i "
    echo -e "| LogFiles : [stdout= $logFile ] [stderr= $errFile ] "
    echo -e "| $command \n"

    eval "$command"
done

exit
```


☉ Przykład realnych obliczeń na batch-farmie: symulator **SMASH** zderzeń jąder atomowych.

- Przejdźmy na katalog domowy i wykonajmy:

```
mkdir smash ; cd smash
cp -p /home/2/kpias/smash-sim/serialize/* .
```

- Potrzebujemy:
 - submission script'u, w pętli podającego kolejne zadania na farmę (`submitTasks.sh`).
 - Z powodów technicznych, zadaniem nie może być surowy kod wykonywalny Smash'a. Jako zadania potrzebujemy skryptu (`task.sh`), który dokona kilku ustawień i pod koniec włączy Smash'a, ponadto podając mu kilka opcji.
- Zajrzyjmy wpierw do `task.sh`. Przyjmujemy tu, że submission script dostarczy mu do środka zmienną `Run` z numerem runu (segmentu).

Pojedyncze włączenie kodu `smash` potrzebuje:

- udostępnić ścieżkę, na której jest biblioteka `libpythia8.so`, której potrzebuje kod `smash`
- **wejściowego pliku sterującego**, definiującego fizykę symulacji (`-i config.yaml` , może być ten sam dla każdego kroku pętli)
- podania ścieżki, na której `smash` ma osadzać dane wynikowe (`-o $outDir`)
- warto przechwycić logi z kodu `smash` (`stdout` i `stderr`) , do plików na osobnej ścieżce (`1>$log 2>$err`)

Uwaga: w symulacjach często trzeba podać ziarno randomizacji, po to, aby dane z różnych włączeń kodu nie były identyczne. Bywa, że użytkownik ma je podać w pliku wejściowym. Tu, w pliku `config.yaml` , linia `Randomseed: -1` oznacza, że kod sam je zrandomizuje.

Jeśli trzeba to zrobić ręcznie, to:

- tworzymy wzorec pliku sterującego, a w linii dla ziarna, wstawiamy umówiony placeholder, np.: `Randomseed: _RandomPlaceholder_`
- dla każdego runu tworzymy dedykowaną mu kopię pliku sterującego
- w kopii automatycznie podmieniamy placeholder na wylosowane ziarno:

```
rnd=$(( (RANDOM<<14) | RANDOM ))
sed -i "s/_RandomPlaceholder_/$rnd/g" {plik sterujący}
```

- Zajrzyjmy teraz do `submitTasks.sh` . Widzimy tu niemal to samo, co na poprzedniej stronie.

🕒 Przykład SMASH c.d.

- Podajmy teraz 5 runów Smash'a do kolejki:

```
$ ./submitTasks.sh 1 5
```

- Możemy teraz podejrzeć kolejkę: `qstat -u {WaszLogin}`

- Jeśli chcemy zatrzymać run, piszemy: `qdel {nr runu}`

- Podejrzymy teraz log'i:
`less smash_run{nr}.log`
`less smash_run{nr}.err`

- Teraz sprawdzimy katalog `output_{nr}`:
`cd output_{nr}`
`less particle_lists.oscar.unfinished`

🕒 Dalsza analiza danych

- Pliki wyjściowe z symulacji można połączyć w 1 duży.

👍 1 plik do dalszej analizy

👎 podwojenie zużycia miejsca na dysku

- Alternatywnie, opracowuje się dane w pętli po plikach wyjściowych.

Np. na ścieżce jest skrypt `checkNparticles.awk`. Wywołajmy go dla wszystkich danych na raz:

```
$ ./checkNparticles.awk output_*/particle_lists.oscar
```