Narzędzia Wspierające Programowanie

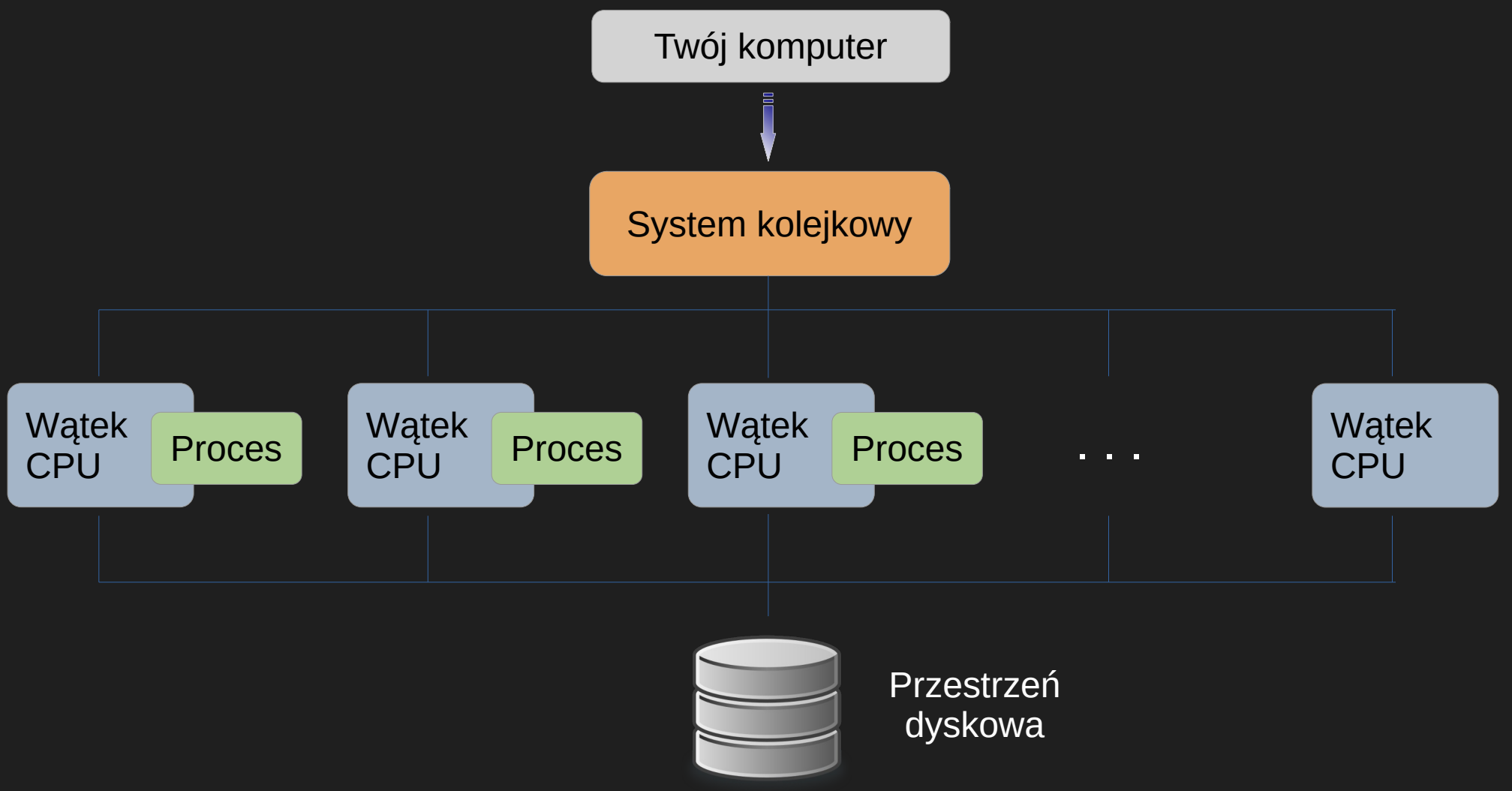
System kolejkowy na farmie komputerowej

- 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 odróżnialne i zserializowane nazwy/lokacje.
  Pliki wyjściowe zostaną "sklejone" na dalszych etapach analizy.
  Uwaga: nie oczekujmy żadnych okien graficznych (pliki graficzne można obejrzeć post-factum).

- Farmą zarządza "system kolejkowy" (job scheduler).
  Tworzy on kolejki (queues) dla zadań (jobs) :
  zadanie (np. program) do wykonania podaje się do kolejki zadań oczekujących,
  a gdy zwalnia się okno w zasobach, to system włącza proces (uruchamia kod) .
  Manager zarządza zasobami w czasie rzeczywistym.
  Obsługując dany proces, kontroluje też 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 opakujmy ją w skrypt o nazwie `sleeper1.sh` :

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

  Wstawmy zadanie z tym skryptem do kolejki:

```
$ qsub sleeper1.sh
```

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

  `sleeper1.sh.o{jobNo}` ← zawartość strumienia `stdout`. Tu tkwi wypis skryptu.
  `sleeper1.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 `sleeper2.sh` , aby liczbę sekund uśpienia podawać z zewnątrz:

```
#!/bin/bash
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 wywołujemy: `$ qsub -v Seconds=5 ./sleeper2.sh`

Strumienie

Jak widać, każde zadanie kieruje strumienie `stdout` i `stderr` do plików (potocznie “logfiles”) .
Przejmiemy teraz kontrolę nad nazwami tych plików, używając opcji `-o` i `-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:
                                                                  Req'd  Req'd   Elap
Job ID                  Username Queue    Jobname          SessID NDS   TSK Memory Time  S Time
----------------------- -------- -------- ---------------- ------ ----- --- ------ ----- - ----
7023484.kruk-hos        kpias    batch    sleeper2.sh         --      1   1  800mb 672:0 Q   --
```

- Jak widać, zadanie weszło do kolejki „batch” i otrzymało 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)
E = exitting	(w trakcie kończenia)
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_user_run`. Oznacza on: max. liczbę procesów, jaką użytkownik może puścić na raz. Prawdopodobnie administrator 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 też 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} sleeper3.sh
  exit
fi

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

Teraz podajmy powyższy skrypt do kolejki. (Nb. w razie konfliktu opcji, 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 na bieżąco wypisów czy grafiki,
a gdyby się kod wywrócił, 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`

- Ściągnijmy 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` . Każdą wytworzoną linię wpisujemy do `output/output_{nr}.dat` . Zapiszmy więc `jobScript.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 zadań (submission script). Powinien on:
  - przyjmować przedział indeksów numerujących segmenty (runy) danych
    ↳ (bezpiecznik na złą liczbę arg. wejścia)
  - ustalić parametry podawania zadania wspólne dla wszystkich z segmentów (runów)
    ↳ (i czytelnie je wyświetlić)
  - wykonać pętlę po segmentach (runach). W każdym kroku:
    - wstawić nr kroku do zmiennej przekazywanej do `jobScript.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: `submitJobs.sh`)

```
#!/bin/bash

if [ $# -ne 2 ] ; then
  echo Usage: ./submitJobs.sh  {Run From}  {Run To}
  exit
fi
queueName=batch
jobScript=./jobScript.sh
jobWorkDir=$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 $2) ; do
  logFile=./log/run${i}.log
  errFile=./log/run${i}.err
  args="Run=$i"

  command="export Run=$i; ./jobScript.sh 1>$logFile 2>$errFile"      ← (do testowania)

# 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 (`submitJobs.sh`).

    - Z powodów technicznych, zadaniem nie może być surowy kod wykonywalny Smash'a. Jako zadania potrzebujemy skryptu (`jobScript.sh`) , który dokona kilku ustawień i pod koniec włączy Smash'a, ponadto podając mu kilka opcji.

  - Zajrzyjmy wpierw do `jobScript.sh`. Przyjmujemy tu, że submission script dostarczy mu do środka zmienną `Run` z numerem runu (segmentu).

    Pojedyncze włączenie kodu `smash` potrzebuje:
    - udostępnienia ścieżki, na której jest biblioteka `libpythia8.so`, której potrzebuje kod `smash`
    - wejściowego pliku sterującego, definiującego fizykę symulacji (opcja `-i config.yaml` , może być ten sam dla każdego kroku pętli)
    - podania ścieżki, na której smash ma osadzać dane wynikowe (opcja `-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 generowane w osobnych procesach nie były identyczne. Bywa, że trzeba 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 wzorzec 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 `submitJobs.sh` . Widzimy tu niemal to samo, co na poprzedniej stronie.

Przykład SMASH c.d.

- Podajmy teraz 4 runy Smash'a do kolejki:

```
$ ./submitJobs.sh 1 4
```

- Możemy teraz podejrzeć kolejkę: `qstat -u {WaszLogin}`
- Jeśli chcemy zatrzymać run, piszemy: `qdel {nr runu}`
- Podejrzyjmy teraz log'i:

```
less log/smash_run{nr}.log
less log/smash_run{nr}.err
```

- Teraz sprawdźmy 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
```