Algorytmy równoległe 2015 (zad. 3)

Michał Liszcz

2015-12-12

Contents

1 Wstęp

Treść problemu:

przydział maszyn wirtualnych do zadań w chmurze obliczeniowej. Dane jest N zadań o znanym czasie wykonania t(1)..t(N) oraz ograniczenie czasowe D (deadline). Należy znaleźć taki przydział zadań do wirtualnych maszyn, aby koszt wykonania obliczenia był jak najmniejszy, oraz aby wszystkie zadania zakończyły się przed upływem terminu D. Jedna maszyna może w jednej chwili wykonywać tylko jedno zadanie. Koszt wirtualnej maszyny pobierany jest za każdą rozpoczętą godzinę jej działania, niezależnie od tego, czy jakieś zadanie jest na niej wykonywane. Wszystkie maszyny posiadają jednakową wydajność i koszt.

2 Algorytm sekwencyjny

Zaimplementowałem algorytm sekwencyjny zgodnie z metodą *branch-and-bound*.

Wykorzystaną strukturą danych jest graf. W każdym wierzchołku grafu znajduje się tablica przypisań zadań do maszyn. Drzewo buduję zaczynając od korzenia.

Korzeń zawsze zawiera informację o przypisaniu pierwszego zadania do pierwszej maszyny. Korzeń ma dwójkę potomków:

- pierwszy zawiera informację o przypisaniu drugiego zadania do pierwszej maszyny,
- drugi zawiera informację o przypisaniu drugiego zadania do drugiej maszyny.

Stosując powyższy schemat można zbudować drzewo, którego liście będą zawierać przypisanie ostatniego zadania do jednej z maszyn. Przechodząc od korzenia do liścia (lub na odwrót) można odtworzyć całe rozwiązanie. Konieczność przeglądania drzewa można wyeliminować, umieszczając w każdym węźle dodatkowo mapowania zawarte w jego rodzicu.

Wykorzystałem następujące struktury danych:

```
package object Model {

    case class Task(val id: Int, val duration: Double)

    case class Machine(val id: Int)

    case class Tree[T](val node: T, val children: Seq[Tree[T]])

    type Mapping = Map[Task, Machine]

    type MappingTree = Tree[Mapping]
}
```

Rozwiązanie rekurencyjne (z pominięciem szczegółów):

```
def step(task: Task,
         machine: Machine,
         parentMapping: Mapping,
         restTasks: List[Task],
         usedMachines: List[Machine],
         freeMachines: List[Machine]): MappingTree = {

    // consider using new machine if available
    val (newUsedMachines, newFreeMachines) = freeMachines match {
        case Nil => (usedMachines, Nil)
        case next :: rest => (next :: usedMachines, rest)
    }

    val mapping = parentMapping + (task -> machine)

    if (restTasks.size == 0) {
        // no more tasks - print solution
        val (maxTimespan, totalCost) = evaluateSolution(mapping)
    }

    new MappingTree(
            mapping,
            restTasks.headOption.map { nextTask =>
                // BRANCH!
                newUsedMachines
                    .filter { nextMachine =>
                        val nextMapping = mapping + (nextTask -> nextMachine)
                        val (time, _) = evaluateSolution(nextMapping)
                        // BOUND!
                        time < deadline
                     }
                    .map { nextMachine =>
                        step(nextTask,
                             nextMachine,
                             mapping,
                             restTasks.tail,
                             newUsedMachines,
                             newFreeMachines)
                    }
                } getOrElse(Nil))
}
```

Takiego rozwiązania niestety nie da się przedstawić w prosty sposób za pomocą rekurencji ogonowej.

W etapie konstruowania potomków (poprzez rekurencyjne wywołanie), sprawdzane jest ograniczenie na całkowity czas. Dla znacznej części rozwiązań można tym sposobem wcześnie odrzucić poddrzewa dużego rozmiaru.

Przykładowe drzewo dla trzech zadań i dwóch maszyn, bez żadnych ograniczeń:

```
Tree(
  Map(Task(0, 15.615810401889549) -> Machine(0)),
  List(
    Tree(
      Map(
        Task(0, 15.615810401889549) -> Machine(0),
        Task(1, 6.081826070068602) -> Machine(1)
      ),
      List(
        Tree(
          Map(
            Task(0, 15.615810401889549) -> Machine(0),
            Task(1, 6.081826070068602) -> Machine(1),
            Task(2, 10.91227882944709) -> Machine(1)
          ),
          List()
        ),
        Tree(
          Map(
            Task(0, 15.615810401889549) -> Machine(0),
            Task(1, 6.081826070068602) -> Machine(1),
            Task(2, 10.91227882944709) -> Machine(0)
          ),
          List()
        )
      )
    ),
    Tree(
      Map(
        Task(0, 15.615810401889549) -> Machine(0),
        Task(1, 6.081826070068602) -> Machine(0)
      ),
      List(
        Tree(
          Map(
            Task(0, 15.615810401889549) -> Machine(0),
            Task(1, 6.081826070068602) -> Machine(0),
            Task(2, 10.91227882944709) -> Machine(1)
          ),
          List()
        ),
        Tree(
          Map(
            Task(0, 15.615810401889549) -> Machine(0),
            Task(1, 6.081826070068602) -> Machine(0),
            Task(2, 10.91227882944709) -> Machine(0)
          ),
          List()
        )
      )
    )
  )
)
```

3 Algorytm równoległy

Ten rozdział opisuje podejście równoległe do przedstawionego wcześniej problemu.

3.1 Metodologia PCAM

3.1.1 Partitioning

Można rozważać następujący podział:

- pojedynczym zadaniem będzie generacja pojedynczego węzła w drzewie.

Wyznaczenie wartości jednego węzła jest jednak bardzo proste pod względem obliczeniowym, natomiast duża liczba węzłów wygeneruje dużą ilość komunikacji.

3.1.2 Communication

Należy ograniczyć czas potrzebny na komunikację względem czasu obliczeń. W tym celu trzeba zmniejszyć ilość komunikacji i zwiększyć rozmiar pojedynczego zadania. Można to osiągnąć poprzez sklejenie węzłów w poddrzewa.

3.1.3 Agglomeration

Należy złączyć ze sobą węzły, tworzą poddrzewa. Maksymalną wysokość pojedynczego poddrzewa można uzależnić:

- od rozmiaru problemu
- od położenia korzenia tego poddrzewa w całym drzewie (zależy od tego liczba dzieci - rozmiar)

Te wartości najlepiej będzie dobrać empirycznie.

3.1.4 Mapping

Proponuję rozwiązanie z pulą problemów:

- początkowo znajduje się tam problem polegający na wygenerowaniu pierwszego poddrzewa
- problem może zostać przypisany do procesora z użyciem:
  - wątku nadzorującego (*master*)
  - synchronizowanej struktury danych
- kiedy procesor skończy przetwarzać poddrzewo, dla każdego liścia generuje nowe zadanie wygenerowania odpowiedniego poddrzewa. Zadania procesor umieszcza w puli
- kolejne procesory pobierają zadania z puli w miarę swoich możliwości

3.2 Implementacja

Opisane powyżej rozwiązanie zaimplementowałem w języku Scala, wykorzystując bibliotekę Akka.

3.2.1 Opis algorytmu

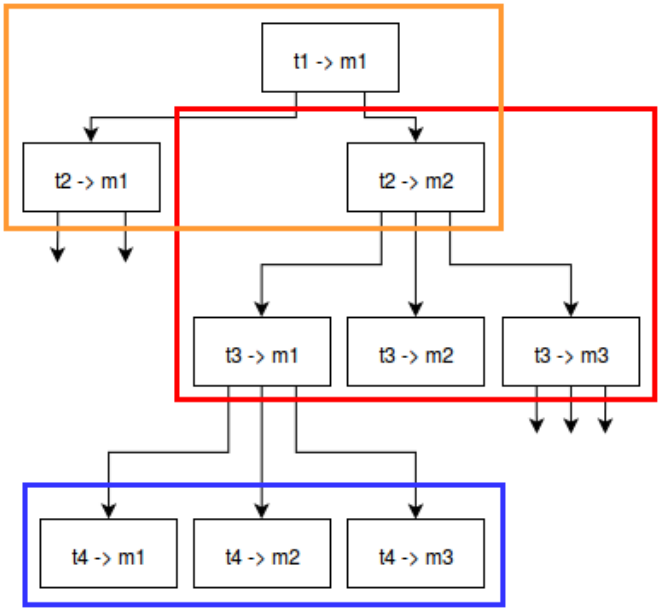

Figure 1: Drzewo przypisania zadań do maszyn. Czerwony prostokąt to przykładowa *praca*, wykonywana przez jeden procesor - poddrzewo o zadanej wysokości. Żółty protokąt to praca początkowa. Wykonanie pracy powoduje utworzenie nowego zestawu prac do wykonania lub zbioru wyników, jeżeli osiągnięte zostały liście drzewa (niebieski prostokąt).

Poniższy listing przedstawia komunikaty wymieniane między procesami w systemie:

```
package object Messages {

    case class WorkDescription(
        val task: Task,
        val machine: Machine,
        val parentMapping: Mapping,
        val restTasks: List[Task],
        val usedMachines: List[Machine],
        val freeMachines: List[Machine])

    case class WorkRequestMsg()

    case class WorkAssignmentMsg(val work: WorkDescription)

    type WorkResult = Tuple2[Seq[WorkDescription], Seq[Mapping]]

    case class WorkDoneMsg(val newWorkAndSolutions: WorkResult)

    case class WorkDoneAckMsg()

    case class ReportRequestMsg()

    case class ReportResponseMsg(val totalWork: Int,
                                 val discardedWork: Int)
}
```

W systemie wyróżnione są dwa typy procesów:

- `TreeMaster` - proces zajmujący się przydziałem zadań i zbieraniem wyników. Jest jeden taki proces.
- `TreeWorker` - proces liczący, przetwarzający porcję pracy. Jest wiele takich procesów.

Rozwiązanie zadanego problemu to skonstruowanie drzewa, którego liście będą zawierać szukane rozwiazania.

W podejściu równoległym zrezygnowałem z fizycznego budowania drzewa. Zamiast tego zdefiniowałem *pracę*. Najmniejsza porcja pracy to przypisanie jednego zadania do jednej maszyny - równoważnie konstrukcja pojedynczego węzła w drzewie.

Wynikiem wykonania takiej pracy może być:

- rozwiązanie - jeżeli nie ma więcej zadań do zakolejkowania,
- więcej pracy - jeżeli trzeba obliczyć kolejne węzły w drzewie (dzieci).

Początkowo do wykonania jest tylko jedna praca, polegająca na przetworzeniu korzenia drzewa. Pierwszy `TreeWorker` który zgłosi swoją gotowość do `TreeMaster` otrzyma to zadanie do wykonania. Zwróconym wynikiem będzie zestaw prac polegających na przetworzeniu potomków korzenia. Kolejne `TreeWorker` otrzymają te zadania.

Aby uniknąć nadmiernej komunikacji, jeden procesor może wykonać kilka etapów pracy, w sposób transparentny dla `TreeMaster`. Jest to równoważne przetworzeniu poddrzewa o zadanej głebokości.

Moje rozwiązanie zakłada stałą wysokość takiego poddrzewa. Przy jej dobraniu należy pamiętać że ma ona bezpośredni wpływ na część sekwencyjną algorytmu - początkowo jest tylko jedna praca do wykonania i dopóki nie zostanie skończona, inne procesory czekają bezczynnie.

Przyjęcie małej wysokości poddrzewa powoduje zmniejszenie czasu potrzebnego na wykonanie pojedynczej pracy i równocześnie zwiększenie ilości takich prac. Przekłada się to na mniejszą część sekwencyjną (pierwsza praca trwa krótko), ale zwiększa ilość komunikacji.

Można to poprawić, zmieniając dynamicznie rozmiar pracy.

Każdy proces który zakończył przetwarzanie pracy, zwraca do `TreeMaster` nową pracę i/lub znalezione rozwiązania. Po tym żąda od `TreeMaster` przydzielenia nowej pracy.

Jeżeli do wykonania nie ma żadnej pracy i żaden z `TreeWorker` aktualnie nie pracuje, algorytm jest zakończony. Z zebranych rozwiązań można wybrać najbardziej odpowiednie (o najmniejszym koszcie).

3.2.1.1 Raporty

W celu zmierzenia korzyści płynących z metody branch-and-bound, procesy `TreeWorker` zliczają liczbę odwiedzonych węzłów oraz liczbę wezłów gdzie warunki zadania zostały przekroczone (a poddrzewo odrzucone).

`TreeMaster` może zażądać od `TreeWorker` takiego raportu w celu wyliczenia końcowych statystyk oraz porównania obciążenia wszystkich `TreeWorker`.

4 Wyniki

Badałem dwa aspekty rozważanego problemu:

- wypły zastosowania podejścia branch-and-bound na ilość obliczeń,
- skalowalność algorytmu równoległego.

4.1 Branch-and-bound

Czas wykonania algorytmu zależy od ilości obliczeń, która wprost zależy od ilości węzłów które należy odwiedzić, przetwarzając drzewo.

n zadań można przypisać do M maszyn na c_n sposobów:

$$\begin{aligned} M_n &= \min(n, M) \\ c_n &= 1 \cdot 2 \cdot 3 \cdot ... \cdot M \cdot ... \cdot M = M_n! \cdot M^{n-M_n} \end{aligned} \tag{1}$$

Stąd, drzewo przypisania N zadań do M maszyn ma wysokość N, wszystkich możliwych rozwiązań jest c_N, a całkowita liczba węzłów w drzewie to:

$$S_N = \sum_{n=1,...,N} c_n \tag{2}$$

Węzłów spełniających warunki zadania jest na ogół znacznie mniej niż S_N. Akceptowalnych rozwiązań jest również mniej niż c_N.

Prostymi metrykami pozwalającymi oszacować korzyści z zastosowania metody branch-and-bound są stosunek odwiedzonych węzłów do wszystkich węzłów (K_N/S_N) oraz stosunek liczby znalezionych rozwiązań do wszystkich możliwych rozwiązań (x_n/c_n). Wielkości te są ze sobą skorelowane, w wynikach prezentuję jednak obie z nich.

Problem wybrany do analizy to przypisanie 12 zadań do 5 maszyn.

Wylosowałem czas wykonania każdego zadania zgodnie z rozkładem jednostanjym:

```
val tasks = Seq.fill(taskCount)(prng.nextDouble)
            .map(Math.abs).map(_ * 10.0)
            .zipWithIndex.map(_.swap).map(Task.tupled).toList
```

Całkowity rozmiar drzewa to $S_N = 11'718'753$ węzłów, natomiast ilość możliwych (włączając nieakceptowalne) rozwiązań to $c_N = 9'375'000$.

Badałem liczbę odwiedzonych węzłów w zależności od narzuconego ograniczenia czasowego(*deadline*). Wyniki przedstawia tabela poniżej.

deadline	odwiedzone węzły (K_N)	K_N/S_N	rozwiązania (x_n)	x_n/c_n
16.0	6'761	0.05%	0	0.00%
16.7	9'467	0.08%	1'480	0.01%
17.0	36'347	0.31%	16'380	0.17%
18.0	51'293	0.43%	27'234	0.29%
19.0	203'169	1.73%	132'730	1.41%
20.0	813'981	6.94%	542'694	5.78%
21.0	1'287'659	10.98%	903'346	9.63%
22.0	1'596'886	13.62%	1'118'886	11.93%

Widać że jeżeli na poszukiwane rozwiązanie narzucone są bardzo mocne ograniczenia, można je znaleźć, przeszukując jedynie niewielką część przestrzeni rozwiązań.

Analogiczny trend jest widoczny również dla innych konfiguracji problemu i czasu przetwarzanai poszczególnych zadań.

Otrzymane wyniki potwierdzają zasadność wykorzystania metody branch-and-bound w tego rodzaju problemach.

4.2 Skalowalność

Algorytm równoległy testowałem na maszynie wyposażonej w procesor Intel Core i5-4200u @ 1.6 GHz (2C/4T).

Rozmiar problemu wybrałem jak poprzednio: 12 zadań do 5 maszyn. Zachowany został również rozkład czasu jaki zamuje pojedyncze zadanie.

Wyniki przedstawia poniższa tabela. Wysokość poddrzewa przetwrzanego bez dodatkowej komunikacji ustaliłem na 5. Niewielka zmiana tej wartości nie przynosiła zauważalnych rezultatów.

procesy	czas wykonania [s]	odchylenie standardowe [s]
1	15.6095	0.4079260554
2	12.14475	0.4145844305
3	11.07425	0.1987835925
4	10.4565	0.583118913
5	10.64625	0.6384681537
6	11.85925	0.3236462833
7	11.19675	0.1186068435
8	10.717	0.5791896638

Widać że czas potrzebny na obliczenia maleje ze wzrostem liczby procesów liczących (z 15 sekund dla jednego do 10 sekund dla czterech procesów).

Dla więcej niż czterech procesorów, czas pozostaje stały lub nieznacznie rośnie. Wynika to najpewniej z użycia do testów procesora z czterema wątkami wykonania.

Dalsza częśc tego rozdziału przedstawia wykresy wykonane na podstawie danych z tabeli.

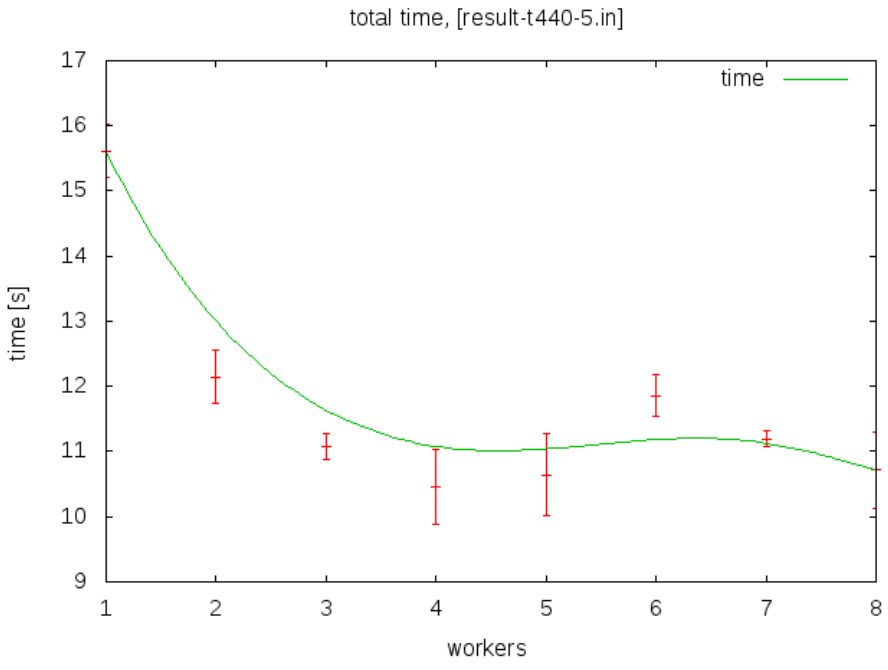

Figure 2: Czas wykonania programu. 12 zadań, 5 maszyn, deadline - 21.0.

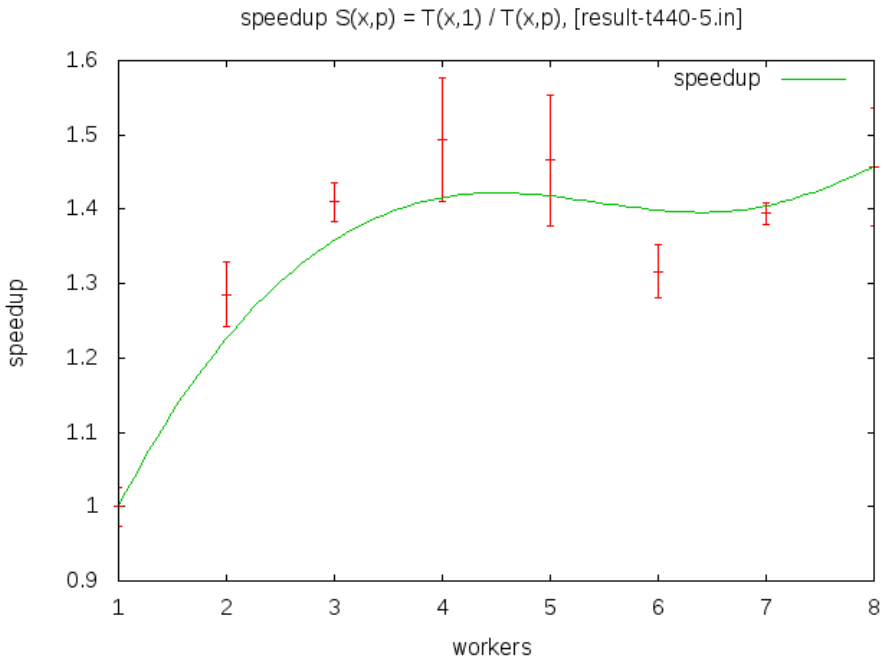

Figure 3: Przyspieszenie. 12 zadań, 5 maszyn, deadline - 21.0.

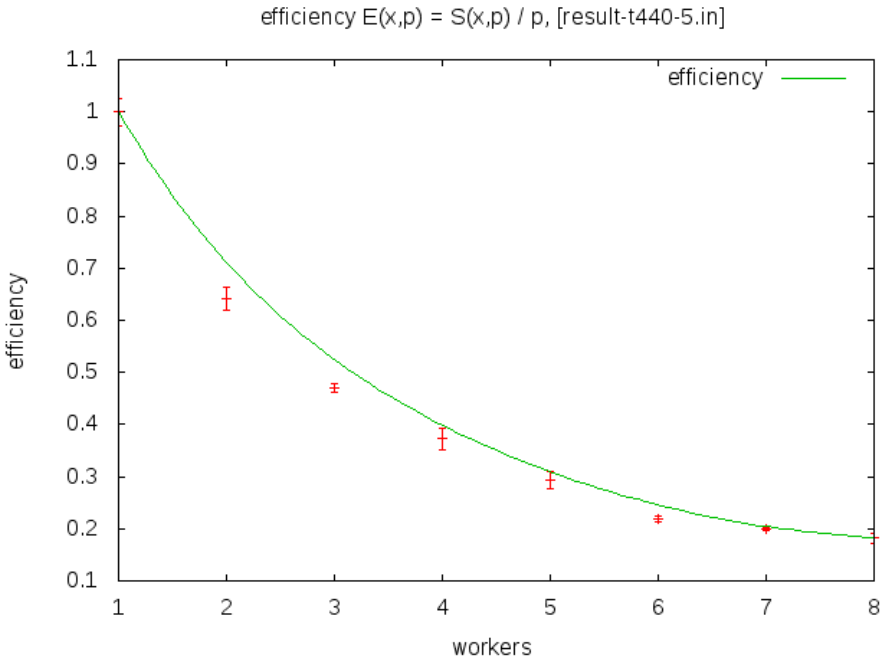

Figure 4: Efektywność. 12 zadań, 5 maszyn, deadline - 21.0.

Dla 1 do 4 procesów przyspieszenie można przybliżyć liniowo, efektywność spada ze względu na odbywającą się komunikację i istnienie części sekwencyjnej. Kształt krzywych jest zachowany również w przypadku innych rozmiarów problemu.

4.3 Obciążenie procesów

W tej klasie algorytmów ważny jest równomierny podział pracy miedzy procesy (i procesory).

W rozważanym przypadku podział pracy był idealnie równomierny. Łatwo to zweryfikować, zliczając węzły drzewa odwiedzone przez każdy z procesów. Taki wniosek jest prawdziwy dla dostatecznie dużych drzew.

```
WORKER REPORT: totalWork=320099 discardedWork=157656
WORKER REPORT: totalWork=311536 discardedWork=154323
WORKER REPORT: totalWork=331232 discardedWork=162868
WORKER REPORT: totalWork=324792 discardedWork=159050
```

`totalWork` na powyższym listingu to liczba odwiedzonych węzłów drzwa. `discardedWork` to liczba węzłów w których nastąpiło odcięcie całego poddrzewa.

References

[1] Foster, I., *Designing and Building Parallel Programs*, www.mcs.anl.gov/~itf/dbpp/.