Algorytmy równoległe 2015 (zad. 3)

Michał Liszcz

2015-12-02

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

3.1 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.

3.2 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.

3.3 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.

3.4 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

References

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