Zeszyty Naukowe WWSI, No 14, Vol. 10, 2016, s. 7-35

Porównanie wydajności algorytmu k-means zaimplementowanego w języku X10 i środowisku C++/MPI

Roman Wyrzykowski, Tomasz Karoń*

Politechnika Częstochowska, Instytut Informatyki Teoretycznej i Stosowanej

---

Streszczenie

W pracy opisano algorytm k-średnich oraz sposób jego implementacji w języku X10. Dokonano porównania tego rozwiązania z implementacją w języku C++11 z wykorzystaniem standardu MPI. Stwierdzono, że implementacja w języku X10 jest szybsza przy większej liczbie procesorów realizujących obliczenia niż implementacja w środowisku C++/MPI. Kod zapisany w języku X10 jest o 59% krótszy od kodu dla kombinacji C++/MPI.

Słowa kluczowe – algorytm k-średnich, język programowania X10, środowisko C++/MPI, porównanie.

1 Wprowadzenie

Zwiększanie wydajności obliczeniowej komputerów poprzez potęgowanie stopnia miniaturyzacji procesorów oraz przyspieszanie ich taktowania napotyka na trudności związane z odprowadzaniem ciepła z coraz szybciej przełączanych tranzystorów oraz opóźnieniami sygnałów transmitowanych w układach. Ze względu na powyższe ograniczenia poszukuje się nowych rozwiązań zwiększających wydajność [1].

Jednym z nich jest wykorzystywanie równoległej pracy wielu procesorów. Oznacza to konieczność rozbudowy tworzonego oprogramowania o mechanizmy

* E-mail: tkaron@icis.pcz.pl

Zgłoszono do druku 25 stycznia 2016 r.

podziału realizowanej pracy, pomiędzy współpracujące procesory oraz mechanizmy komunikacji pomiędzy nimi. Powoduje zwiększenie stopnia komplikacji takiego programu, a więc także wzrost trudności związanych z jego stworzeniem, debugowaniem i ulepszaniem. W efekcie spada wydajność programistów tworzących takie oprogramowanie.

Pewnym rozwiązaniem jest zastosowanie języków wysokiego poziomu zapewniających możliwość wydajnego tworzenia kodu współbieżnego. Przykładami takich języków są: Co-Array Fortran (CAF), Unified Parallel C (UPC) [2], Chapel [3], Titanium [4] oraz X10 [5]. Wszystkie wykorzystują model programowania o nazwie Partitioned Global Address Space (PGAS) lub jego rozszerzenie w postaci Asynchronous Partitioned Global Address Space (APGAS).

Model PGAS oferuje globalną przestrzeń pamięci utworzoną z pamięci wszystkich procesorów wykorzystywanych do realizacji obliczeń. Oznacza to, że każdy proces ma dostęp do dowolnego elementu tej przestrzeni, niezależnie od procesora, na którym jest uruchomiony. Jest to abstrakcja programistyczna podobna do pamięci współdzielonej. Model zakłada także istnienie powiązania pomiędzy podprzestrzenią pamięci globalnej, a związanym z nią procesorem. Dzięki temu języki programowania wykorzystujące model PGAS korzystają z komunikacji jednostronnej, nie wymagającej uruchamiania dodatkowego programu na zdalnym węźle. W ten sposób unika się nadmiaru komunikatów oraz oddziela transmisję danych od synchronizacji procesów [6].

Rozszerzeniem modelu PGAS jest model programistyczny Asynchronous Partitioned Global Address Space (APGAS), wprowadzający idee lokacji (ang. *place*) i aktywności (zadanie asynchroniczne), a także mechanizmy synchronizacyjne [**7**]. Przykładem języka programowania korzystającego z modelu APGAS jest język X10, rozwijany przez IBM w ramach projektu „Productive, Easy – to – use, Reliable Computing System" (PERCS [8]).

Tworząc ten język założono nie tylko wysoką wydajność tworzonego przy jego pomocy kodu, ale także jego uniwersalność oraz wysoką produktywność przy jego tworzeniu. Uniwersalność kodu polega na możliwości jego uruchomienia bez konieczności zmian. Wymagana jest co najwyżej kompilacja. Programy zapisane w tym języku uruchamiać można na szerokim spektrum sprzętu: od laptopów począwszy, poprzez klastry, na superkomputerach skończywszy [8].

Celem niniejszego artykułu jest dokonanie porównania wydajności kodu realizującego algorytm k-średnich, będącym jednym z bazowych algorytmów grupowania (klastrowania) danych. Algorytm zaimplementowano z wykorzystaniem języka X10 oraz środowiska C++/MPI. Implementacja w języku X10 realizuje obliczenia równoległe z wykorzystaniem modelu z przesyłaniem wiadomości [1]. W obu przypadkach

do realizacji przesyłania wiadomości wykorzystano bibliotekę OpenMPI, implementującą standard MPI-2 [9]. Jak zostanie zaprezentowane w dalszej części artykułu, w efekcie rozwiązań zastosowanych w bibliotece OpenMPI wykorzystywana jest także pamięć współbieżna. Implementacja w środowisku C++/MPI realizuje obliczenia równoległe zarówno z wykorzystaniem przesyłania wiadomości, jak i z wykorzystaniem pamięci współbieżnej [1]. Pozwoliło to na zastosowanie rozwiązań podobnych do implementacji w języku X10.

Jako dane wejściowe wykorzystano plik próbek mikrodanych do publicznego użytku, powstały w wyniku badań populacji USA przeprowadzonych przez Biuro Spisu Ludności Stanów Zjednoczonych Ameryki [10].

W sekcji 2 opisano algorytm k-średnich oraz przedstawiono kolejne kroki, w ramach których jest on realizowany. W sekcji 3 dokonano krótkiego przeglądu możliwości języka X10, opisując konstrukty językowe wykorzystane do zapisania algorytmu k-średnich.

Opis realizacji algorytmu k-średnich w języku X10 i w środowisku C++/MPI zawarto w sekcji 4. Objaśnienie środowiska uruchomieniowego wraz z informacją na temat jego przygotowania dla języka X10 znajduje się w sekcji 5. Opisano tam także wykorzystywane dane wejściowe oraz przebieg kompilacji obu kodów z uwzględnieniem optymalizacji. W kolejnej sekcji podano przebieg badań. Zawarto tam wyniki pomiarów czasów wykonywania obu kodów dla różnych kombinacji danych wejściowych. Przedstawiono także wykresy obrazujące zmiany przyśpieszenia w zależności od rozmiaru rozwiązywanego problemu. Ogólną dyskusję otrzymanych wyników oraz propozycję dalszych prac zawarto w sekcji 7.

2 Wprowadzenie do algorytmu k-średnich

Algorytm k-średnich, zwany też algorytmem centroidów lub algorytmem klastrowym, służy do znajdowania grup w danych [11]. Z punktu widzenia matematyki jest to poszukiwanie minimum wariancji grupowania zbioru danych w K grupach.

Zakładając, że mamy dane punkty $\{X_i | i = 1..n\} \subseteq R^W$, gdzie W to ilość współrzędnych punktów, poszukujemy K $(K < n)$ punktów $\{m_j\}_{j=1}^{K} \subseteq R^W$ będących centroidami K klastrów $C = \{C_1, C_2, \dots, C_K\}$ takich, że minimalizowana jest funkcja błędu [12]:

$$\sum_{j=1}^{K} \sum_{X_i \in C_j} d(X_i, m_j)^2 \quad (1)$$

Przez $d(X_i, m_j)$ rozumiemy funkcję odległości taką, że dla współrzędnych x,y,z zachodzą własności [11]:

1. $(d(x,y) \geq 0 \land d(x,y) = 0) \Leftrightarrow x = y$

2. $d(x, y) = d(y, x)$
3. $d(x, z) \leq d(x, y) + d(y, z)$

Najczęściej wykorzystywaną funkcją odległości jest odległość euklidesowa określana jako [11]

$$d_e(x, y) = \sqrt{\sum_i^n (x_i - y_i)^2} \quad (2)$$

Punkty $\{m_j\}_{j=1}^{K} \subseteq R^W$ nazywamy centroidami klastrów lub środkami ciężkości klastrów [13].

Opierając się na [13] realizację algorytmu zapiszemy w czterech następujących krokach:

1. Znalezienie centroidów startowych, czyli zbioru *K* punktów $\{m_j\}_{j=1}^{K} \subseteq R^W$, od których rozpoczynamy pracę algorytmu. Ogólnie rzecz biorąc, nie istnieje dobra metoda znajdowania tych punktów. W większości prac przyjmuje się metodę losową [11, 13, 14].
2. Obliczanie odległości polegające na wyznaczeniu dla zbioru punktów $\{X_i | i = 1..n\} \subseteq R^W$ funkcji odległości $d_e(x, y)$ dla każdego z *K* centroidów $\{m_j\}_{j=1}^{K} \subseteq R^W$. Następnie każdy punkt zostaje przypisany do grupy tworzonej przez najbliższy względem niego centroid [13]. Istnieją algorytmy, które pozwalają przyśpieszyć ten proces poprzez zmniejszenie ilości obliczeń funkcji odległości. Jednym z nich jest algorytm korzystający z twierdzenia o nierówności trójkąta opisany w pracy [15]. Innym algorytm, również wykorzystujący własności geometryczne położenia punktów w przestrzeni, opisano w pracy [16].
3. Przeliczenie centroidów polegające na wyznaczeniu nowych środków grup na podstawie punktów przypisanych do grup.
4. Sprawdzenie warunku zbieżności – kroki 2-3 są powtarzane aż osiągnięte zostanie pewne kryterium zbieżności. Może nim być brak zmniejszania sumarycznego błędu kwadratowego określanego jako:
$$SSE = \sum_{i=1}^{K} \sum_{p \in C_i} d(p, m_i)^2 \quad (3)$$
gdzie $p \in C_i$ oznacza każdy punkt w grupie C_i, a m_i jest centroidem tej grupy. Jednak znalezienie optymalnego grupowania dla takiego kryterium zbieżności jest problemem NP – trudnym [17]. Z tego powodu stosuje się różne inne kryteria zbieżności [18].

Koszt obliczeniowy algorytmu k-średnich będzie zależał od ilości powtórzeń kroków 2–4. Dla jednej iteracji koszt ten można wyznaczyć z formuły:

$$O(n \cdot K \cdot d) + 2 \cdot O(n \cdot d) = O(n \cdot K \cdot d) + O(n \cdot d) \quad (4)$$

gdzie *n* to ilość punktów, K to ilość grup, a *d* jest ilością współrzędnych punktów [12].

Metodą pozwalającą na zwiększenie wydajności algorytmu centroidów jest także redukcja rozmiaru danych z wykorzystaniem analizy głównych składowych (PCA). Oznacza to konieczność zastosowania dodatkowych algorytmów przygotowujących dane dla algorytmu k-średnich [19].

Algorytm klastrowy wymaga podania jako argumentu liczby grup *K*, na które podzielone zostaną dane. Niewłaściwa liczba grup nie pozwoli wykonać grupowania dobrej jakości [20]. Powoduje także zmniejszenie wydajności działania samego algorytmu poprzez zwiększenie liczby iteracji niezbędnych do osiągnięcia założonego kryterium zbieżności. Istnieją metody pozwalające oszacować prawidłową liczbę grup *K*. Wymagają one jednak informacji uzyskiwanej w trakcie realizacji algorytmu centroidów [21].

Do celów porównania wydajności kodu realizującego algorytm k-średnich założono, że wykonanych zostanie tylko jedenaście powtórzeń kroków 2–4. Nie zastosowano także zmniejszenia rozmiaru danych.

3 Wprowadzenie do języka programowania X10

X10 jest wysokowydajnym językiem programowania rozwijanym przez IBM. Język został zaprojektowany w taki sposób, aby możliwe było uruchamianie programów napisanych z jego wykorzystaniem zarówno na komputerach wyposażonych w pojedynczy procesor, jak i na dużych klastrach [22]. Wsparcie programowania współbieżnego uzyskano poprzez zastosowanie modelu programistycznego Asynchronous Partitioned Global Adress Space (APGAS) [5, 7]. Model ten stanowi rozszerzenie modelu programistycznego Partitioned Global Adress Space (PGAS) o koncepcje: lokacji (ang. *places*), aktywności (zadania asynchronicznego – *async*), a także mechanizmów synchronizujących [23].

Lokacje (ang. *places*) są jednostkami obliczeniowymi dysponującymi skończoną liczbą wątków sprzętowych i ograniczoną ilością jednakowo dostępnej pamięci współdzielonej. Mogą one uruchomić wiele wątków. Nie jest wymagane, aby były mapowane do procesorów mających te same zbiory rozkazów lub liczbę rdzeni. Samo mapowanie lokacji (ang. *places*) do węzłów określane jest przez użytkownika dopiero w momencie uruchomienia programu [5]. Obliczenia w X10 są wykonywane za pomocą obiektów uruchamianych w lekkich wątkach nazywanych aktywnościami (ang. *activities*). Szczegóły działania środowiska uruchomieniowego, w tym zasady pracy planisty sterującego przydziałem aktywności do wątków, opisano w pracy [24].

W języku X10 zasadniczym elementem jest aktywność asynchroniczna (*async*). Kod objęty wyrażeniem *async* uruchamiany jest w wskazanej lokacji i pozostaje tam przez cały czas, w którym działa. Istnieje wyrażenie *at*, pozwalające synchronicznie

zmienić lokację wykonywania aktywności. Wszystkie dane, z których korzysta aktywność, są kopiowane do nowej lokacji. Jest to więc potencjalnie kosztowna operacja [23].

Konstruktem synchronizującym wykonywanie aktywności asynchronicznych jest wyrażenie *finish*. Powoduje ono oczekiwanie na zakończenie wszystkich uruchomionych przez siebie aktywności asynchronicznych [23, 24].

Za pomocą wyżej wymienionych konstruktów można zapisać pięć wzorców wykorzystywanych w programowaniu współbieżnym [8]:

- *FINISH_ASYNC* – synchronizacja zakończenia dla pojedynczej aktywności, w tym aktywności uruchomionej w innej lokacji,
- *FINISH_LOCAL* – synchronizacja zakończenia aktywności uruchomionych w lokalnej lokacji,
- *FINISH_HERE* – synchronizacja nie obejmuje aktywności pracujących w zdalnych lokacjach,
- *FINISH_SPMD* – synchronizacja zakończenia zdalnych aktywności, bez aktywności zagnieżdżonych w zdalnych aktywnościach
- *FINISH_DENSE* – skalowalna implementacja zarządzania zakończeniem aktywności pozwalająca na zredukowanie ilości połączeń sieciowych.

Wymianę danych pomiędzy aktywnościami można zrealizować w języku X10 na kilka sposobów. Jak już wspomniano, przekazywanie danych z punktu do punktu realizuje konstrukt *at*. Przekazywanie danych w trybie wielu punktów można zrealizować za pomocą API o nazwie *x10.util.Team*. Wykorzystuje się tu dostępne usprawnienia sprzętowe lub stosuje standardowe techniki optymalizacyjne, takie jak drzewa rozgłoszeniowe lub bariery *butterfly*. Istnieje także metoda *asyncCopy* dostępna w API o nazwie *x10.regionarray.Array*, pozwalająca na kopiowanie zawartości tabeli pomiędzy lokacjami. Operacja ta odbywa się z wykorzystaniem mechanizmu zdalnego dostępu do pamięci RMA (ang. *Remote Memory Access*) [9, 24].

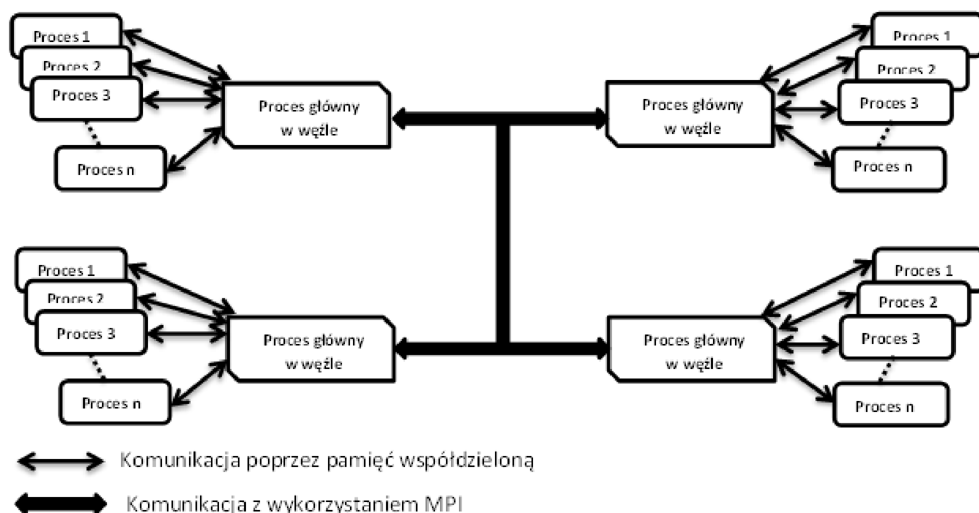
4 Implementacja współbieżnego algorytm k-średnich

4.1. Realizacja w środowisku C++/MPI

Algorytm centroidów można zapisać jako algorytm współbieżny z wykorzystaniem modelu przesyłania wiadomości. Jednym z takich systemów jest standard Message Passing Interface (MPI) [25]. Z punktu widzenia programisty należy napisać pojedynczy program, który uruchamiany jest w wielu kopiach. Kopie te wymieniają się danymi za pomocą mechanizmu przesyłania wiadomości. Program może zostać za-

pisany z wykorzystaniem języka C++ w standardzie ISO/IEC 14882-2011, potocznie nazywanym C++11 [26]. Pozwala to na wykorzystanie mechanizmów wielowątkowości oraz nowego modelu pamięci dostępnych w bibliotece standardowej tego języka [27].

Powyższe rozwiązania pozwalają dopasować rozwiązanie programistyczne współbieżnego algorytmu k-średnich do struktury klastra, na którym będzie ono uruchomione. W szczególności można wykorzystać fakt, że węzły wchodzące w skład klastra zawierają procesory wielordzeniowe. Zapisując program współbieżny w taki sposób, aby wymiana informacji pomiędzy procesami uruchomionymi na rdzeniach tego samego węzła zachodziła z wykorzystaniem pamięci współdzielonej, a wymiana pomiędzy węzłami – z wykorzystaniem standardu MPI, można uzyskać znaczne przyśpieszenie programu. Wynika to ze zmniejszenia czasu potrzebnego na przesyłanie informacji pomiędzy procesami [12]. Schemat komunikacji w programie pokazano na rysunku 1.



Rysunek 1. Schemat komunikacji w implementacji algorytmu k-means w środowisku C++/MPI

Przyjęto, że system MPI będzie uruchamiał pojedynczy proces na każdym węźle wskazanym w komendzie mpirun, oraz, że wszystkie węzły będą wyposażone w procesory o tych samych parametrach. Proces ten nazywany będzie dalej procesem głównym w węźle.

```
systemstatus::systemstatus(int argc, char **argv) {
   /*....*/
   local_number_of_cpu=thread::hardware_concurrency();
   sum=new tsum();
   /*....*/
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   MPI_Comm_size(MPI_COMM_WORLD,&numnodes);
   if (myid==root) {
     total_cpu+=local_number_of_cpu;
    for(int i=1; i<numnodes; i++) {
      MPI_Recv(&recv,1,MPI_INT,i,0,MPI_COMM_WORLD,&status);
      total_cpu+=recv;
   }
   MPI_Barrier(MPI_COMM_WORLD);
   } else {
    MPI_Send(&local_number_of_cpu,1,MPI_INT,0,0,MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
   }
}
```

Rysunek 2. Ustalenie sumarycznej liczby procesów dostępnych w programie – zmienna total_cpu

Program rozpoczyna pracę od zainicjowania systemu MPI, a następnie wykorzystuje metodę *thread::hardware_concurrency* do ustalenia liczby wątków (patrz rysunek 2), które będą wykonywane naprawdę współbieżnie w ramach jednego węzła [28]. W systemie wielordzeniowym jest to często liczba dostępnych rdzeni. Użycie tej funkcji pozwala wyeliminować zjawisko nadsubskrypcji, powodującej obniżenie wydajności programu związanej z kosztem przełączania kontekstu [27].

Kolejną czynnością jest ustalenie sumarycznej liczby procesów (wątków) dostępnych w systemie obliczeniowym, co odbywa się z wykorzystaniem systemu MPI. Informacja ta wykorzystywana jest do równomiernego rozdzielania danych wejściowych pomiędzy uruchomione procesy [1]. Kod realizujący powyższe zadania zaprezentowano na rysunku 2.

Dopiero w tym momencie rozpoczynana jest realizacja kroku 1 algorytmu k-średnich, tj. ustalenie centroidów startowych (patrz rysunek 3).

```
void systemstatus::throws() {
 /*....*/
 findfirstcentroid(k,filename,firstcentroid);
 /*....*/
 if (myid==root) {
  long int quantity_per_throw=ceil(quantity/total_cpu);
  /*....*/
  setupdata[0]=quantity_per_throw;
  /*....*/
  for(int i=1; i<numnodes; i++) {
   MPI_Send(setupdata,2,MPI_LONG,i,0,MPI_COMM_WORLD);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  /*....*/
  for(int i=0; i<local_number_of_cpu; i++) {
    /*....*/
   kmean kmean(k,start,end,filename,outfilename,names_columns,sum,nodeinfo,meas_path);
    /*....*/
    threads[i]=thread(kmean);
    threads[i].detach();
  }
 } else {
   /*....*/
   MPI_Recv(setupdata,2,MPI_LONG,0,0,MPI_COMM_WORLD,&status);
   MPI_Barrier(MPI_COMM_WORLD);
   long int quantity_per_throw=setupdata[0];
   /*....*/
   for(int i=0; i<local_number_of_cpu; i++) {
    /*....*/
   kmeans kmean(k,start,end,filename,outfilename,names_columns,sum,nodeinfo, meas_path);
    threads[i]=thread(kmean);
    threads[i].detach();
  }
 }
 /*....*/
}
```

Rysunek 3. Wyszukanie centroidów startowych i rozwidlenie procesów

Jak wskazano w sekcji drugiej, najczęściej wykorzystywana jest w tym celu metoda losowa [11, 13, 14]. W programie przyjęto rozwiązanie polegające na wskazaniu centroidów kolejno z pośród $K \cdot 10$ pierwszych wierszy danych wejściowych, gdzie K jest oznaczeniem liczby grup. Zadanie wyszukiwania centroidów startowych realizowane jest w każdym procesie głównym w węźle. Pozwala to uniknąć rozsyłania centroidów za pomocą przesyłania wiadomości [29, 1].

Następnie dokonywane jest rozwidlenie procesów w węźle do liczby wątków określonej z wykorzystaniem metody *thread::hardware_concurrency*. Kod realizujący powyższe zadania zaprezentowano na rysunku 3.

W tak uruchomionych wątkach realizowane są kroki 2 i 3 algorytmu klastrowego. W pierwszej kolejności następuje odczyt danych wejściowych z pliku wejściowego przez każdy wątek z osobna [30]. Obszary wczytywanych wierszy są tak ustalone, aby pokryły cały plik danych, przy zachowaniu warunku jednakowej ilości danych wejściowych dla każdego procesu.

Przed rozesłaniem danych używanych do przeliczenia centroidów, poszczególne wątki dokonują zsumowania danych pomiędzy sobą. Ustalenie momentu dokonania zsumowania danych odbywa się za pomocą zmiennej *addcounter*, korzystającej z klasy *std::atomic*<>, z zastosowaniem mechanizmu porządkowania operacji w ramach modelu porządkowania poprzez wzajemne wykluczanie [27]. Powyższe operacje oraz wymianę danych za pomocą funkcji *MPI_Allreduce* przedstawiono na rysunku 4.

```
void tsum::addgroupsum(vector<long double>* _sumgroup) {
  for(int i=0;i<_sumgroup->size();i++) {
    sum[i]+=_sumgroup->at(i);
    _sumgroup->at(i)=0;
  }
 addcounter->fetch_add(1,memory_order_release);
}

void tsum::sumreduce(void) {
  while(!(addcounter->load(memory_order_acquire)==local_number_of_cpu));
  MPI_Allreduce(sum.data(),sum.data(),sum.size(),MPI_ LONG_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  /*....*/
          }
```

Rysunek 4. Zsumowanie danych z wątków, wymiana danych pomiędzy procesami głównymi w węzłach oraz mechanizmy porządkowania operacji

Po wymianie danych za pomocą funkcji *MPI_Allreduce* systemu MPI następuje przeliczenie centroidów oraz realizacja kroku 4, tj. sprawdzenie warunku zbieżności. W programie jako warunek zbieżności przyjęto wartość minimalnego zmniejszenia sumarycznego błędu kwadratowego (3). Wartość ta podawana jest jako parametr uruchomieniowy programu.

Do celów badania wydajności przyjęto także dodatkowy warunek powodujący zakończenie programu po wykonaniu 11 iteracji kroków 2–3. Warto zwrócić uwagę, że wszystkie procesy wchodzą w pierwszą iterację w sposób asynchroniczny. Oznacza

to, że rozpoczyna się ona w różnych procesach w różnych chwilach czasu oraz trwa krócej niż pozostałe iteracje. Uwzględniając więc wnioski zawarte w pracy [29], przyjęto, że algorytm wykona 10 iteracji zaczynających się synchronicznie.

```
void kmeans::operator()() {
 /*....*/
 file.fileprocess(',',data,from,to-from);
 /*....*/
 while(true) {
  iteration++;
  if (myid==root) {
   calculate_kmeans(data);
   sum->addgroupsum(sumgroup.getdata());
   if (cpu==0) {
    sum->sumreduce();
    sum->resetcounter();
    for(int group=0; group<k; group++)
     SSE+=sum->getsse(group);
     if (((lastSSE-SSE)<dist)||(iteration==ITERATION_LIMIT)) {
      filesave.save(from,true);
      /*....*/
      sum->setthreadend();
      for(int i=1; i<localnumcpu; i++) sum->threadwaitoff(i);
      break;
     }
     for(list<int>::iterator col=columnnumber->begin(); col!=columnnumber->end(); col++)
         for(int group=0; group<k; group++)
              sum->setcentroid(*col,group,sum->getsum(*col,group)/sum->getcount(*col,group));
     for(int i=1; i<localnumcpu; i++) sum->threadwaitoff(i);
   }
   if (cpu!=0) {
    sum->checkthreadwait(cpu);
    if (sum->checkthreadend()) {
     filesave.save(from,true);
     break;
    }
   }
  } else {
  /*....else …*/
 }
 sum->end();
}
```

Rysunek 5. Zasadnicza część programu implementującego algorytm k-średnich w środowisku C++/MPI

Zasadniczą część programu implementującego algorytm k-średnich w środowisku C++/MPI przedstawiono na rysunku 5. Wykonywanie obliczeń realizuje metoda *calculate_kmeans*. Zadania związane z wymianą danych oraz synchronizacją procesów są realizowane przez metody *addgroupsum* i *sumreduce* obiektu *sum*, których szczegółowy kod zaprezentowano na rysunku 4. Obiekty *file* i *filesave* odpowiadają odpowiednio za odczyt danych wejściowych i zapis wyników.

Jak widać, za bezpośrednią realizację algorytmu k-means odpowiadają: metoda *calculate_kmeans*, obiekt *sum* oraz linie wewnątrz pętli *while.* Trzeba w tym miejscu zauważyć, że po dyrektywie *else* znajduje się kod analogiczny jak po instrukcji *if (myid==root)*, z tym że pominięto w nim linie odpowiedzialne za raportowanie postępów programu. Rozwiązanie takie pozwoliło zminimalizować liczbę instrukcji *if*. Ułatwiło także testowanie kodu.

W listingu na rysunku 5 pominięty kod po dyrektywie *else* oznaczono symbolem /* …else…*/.

Łącznie, bezpośrednią implementację realizacji algorytmu k-means zapisano w 319 liniach. Zakładając jednak końcową optymalizację, polegającą na doprowadzeniu do jednokrotnego zapisu kodu realizującego bezpośrednio algorytm k-means, kod wynikowy można by zapisać w 283 liniach.

4.2. Realizacja w języku X10

Algorytmu k-średnich zaimplementowany w języku X10 może być realizowany z wykorzystaniem różnych rozwiązań komunikacyjnych. Jest to uzależnione od decyzji podejmowanych w momencie kompilacji, co wykazano w sekcji 5.1. I odbywa się bez konieczności dokonywania zmian w kodzie zapisanym w języku X10. Jedną z możliwości jest wykorzystanie standardu MPI przez API o nazwie *x10.util.Team*. Samą wymianę danych można wykonać na dwa sposoby:

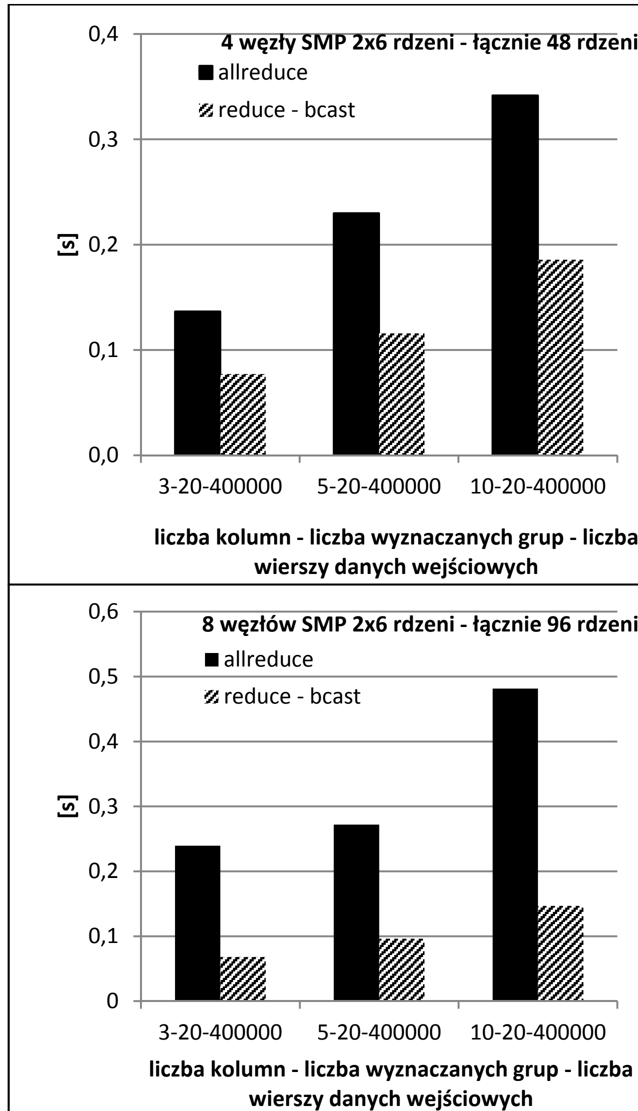
1. z wykorzystaniem metody *team.allreduce*,
2. z wykorzystaniem metod *team.reduce* i *team.bcast*.

Z punktu widzenia ostatecznego efektu wymiany danych, rozwiązania są równoważne.

Badania dowiodły jednak, że wydajniejsze jest zastosowanie metod *team.reduce* i *team.bcast*. Ma to związek z wykorzystywaniem przez metody *team.reduce* i *team.bcast* odpowiednio funkcji *MPI_Reduce* i *MPI_Bcast*, podobnie jak do realizacji metody *team.allreduce* stosowana jest funkcja *MPI_Allreduce* (patrz rysunek 6).

Wpływ na szybkość realizacji wymiany danych ma także fakt, że węzły połączone są ze sobą z wykorzystaniem sieci Infiniband (sekcja 5). Jak wskazano w pracy [31], biblioteka Open MPI, implementująca standard MPI-2, wykorzystuje udogodnienia

oferowane poprzez sieć Infiniband w postaci Remote Direct Memory Access (RDMA) [9] oraz Operation System Bypass.



Rysunek 6. Porównanie szybkości wykonywania kodu wykorzystującego operację *all-reduce* i parę operacji *reduce – bcast* na: 4 węzłach zawierających po dwa 6 rdzeniowe procesory w architekturze SMP oraz 8 węzłach zawierających po dwa 6 rdzeniowe procesory w architekturze SMP

Rozwiązania te pozwalają na bezpośredni dostęp do adaptera kanału hosta (HCA) przez proces z przestrzeni użytkownika z pominięciem systemu operacyjnego. Minimalizuje to czas oczekiwania oraz zmniejsza obciążenie CPU. Technika ta nosi nazwę kolejki Queue Pair (QP). Dodatkowo używa się techniki Shared Receive Queue (SRQ), wiążącej wiele kolejek QP. Pozwala to na współdzielenie zasobów przez wszystkie związane kolejki QP [31].

Niezależnie od powyższego, standard MPI-2 wprowadza mechanizm Remote Memory Access (RMA), pozwalający na realizację operacji w sposób jednostronny, to znaczy bez angażowania odbiorcy. RMA może być różnie realizowane: z wykorzystaniem pamięci współdzielonej, mechanizmu DMA oraz wsparcia sprzętowego [9]. W pracach [32, 33] wskazano, że stosuje się rozwiązania pozwalające na użycie mechanizmu RDMA celem dostępu do zdalnej przestrzeni adresowej. Stosuje się także optymalizację ze względu na wykorzystywanie przez węzły architektury SMP, dzięki czemu możliwe jest wykorzystanie pamięci współdzielonej i przyśpieszenie operacji grupowych standardu MPI prawie o 70%. Wskazano także, że stosuje się optymalizację poszczególnych operacji grupowych MPI, uwzględniając między innymi wielkość przesyłanych komunikatów [34].

Sumaryczny efekt powyższych rozwiązań spowodował, że para metod *team.reduce* i *team.bcast* jest wydajniejsza od metody *team.allreduce*. Zobrazowano to na rysunku 6, pokazując czasy wykonania dwóch wersji algorytmu k-średnich realizowanego na: 4 węzłach zawierających po dwa 6 rdzeniowe procesory w architekturze SMP (czyli łącznie 48 rdzeni) oraz 8 węzłach zawierających po dwa 6 rdzeniowe procesory w architekturze SMP (czyli łącznie 96 rdzeni). Widać, że wersja algorytmu k-means korzystająca z wywołania *team.allreduce* jest wolniejsza od wersji korzystającej z pary *team.reduce* i *team.bcast*. Co więcej, wraz z wzrostem liczby rdzeni następuje dalsze spowolnienie wykonywania *team.allreduce* i przyśpieszenie pary *team.reduce* i *team.bcast*.

Powyższy problem nie ma większego znaczenia dla implementacji w środowisku C++/MPI. Sposób komunikacji, w tym środowisku powoduje, że w wymianie danych uczestniczą tylko procesy główne w węźle. Liczba procesów biorących udział w wymianie danych za pomocą MPI jest więc 12 krotnie mniejsza niż dla implementacji w języku X10. Wraz z wzrostem liczby węzłów zmienia się o niewielką wartość.

Zapisując algorytm k-średnich z wykorzystaniem języka X10 i mając na uwadze model programistyczny APGAS, można stwierdzić, że program należy podzielić na sekwencyjne zadanie wstępne, przygotowujące program do pracy i współbieżne zadanie główne, realizujące algorytm centroidów (patrz rysunek 7).

Zadanie wstępne rozpoczyna pracę od pobrania argumentów uruchomieniowych oraz realizuje krok 1 algorytmu grupowania, tj. wyszukanie centroidów startowych.

Podobnie jak w implementacji w środowisku C++/MPI, wykorzystano metodę losową [11, 13, 14] realizowaną poprzez wybranie centroidów spośród $K \cdot 10$ wierszy danych wejściowych, gdzie K jest liczbą wyznaczanych grup.

```
public static def main(startval : Rail[String]) {
  /*....*/
  val Sys = new Systemstatus(startval);
  /*....*/
  val team = Team(new SparsePlaceGroup(new Rail[Place](Place.MAX_PLACES,(i : long) =>
Place.place(i))));
  val lines_per_places : Long = Sys.quantity/Place.ALL_PLACES;
  val Datafile = new Fileread(Sys.filename);
  /*....*/
  centrfind.findcentroid();
  finish for (p in Place.places()) {
   at (p) async {
    /*....*/  zadanie główne
   }
}
```

Rysunek 7. Kod w języku X10 dla zadania wstępnego

Krok 1 algorytmu centroidów realizowany jest w sekwencyjnym zadaniu wstępnym, ze względu na konieczność dostępu do pliku danych wejściowych (patrz rysunek 7). Gdyby wzorem rozwiązania zastosowanego w implementacji C++/MPI, zrealizowano go w współbieżnym zadaniu głównym, pojawiłby się wyścig w dostępie do danych wejściowych. W odróżnieniu bowiem od rozwiązania w C++/MPI, uruchomionych zadań głównych będzie co najmniej tyle, ile procesorów, na których program jest uruchomiony. W implementacji C++/MPI o dostęp do pliku danych wejściowych ubiegało się maksymalnie tyle procesów, na ilu węzłach program był uruchomiony, a więc wielokrotnie mniej. Zasadniczą cześć kodu realizującą powyższe zadania przedstawiono na rysunku 7.

Z punktu widzenia programisty pracującego z wykorzystaniem języka X10, zadanie główne tworzone jest poprzez uruchomienie aktywności realizujących algorytm centroidów w sposób asynchroniczny w wielu lokacjach (ang. *places*). Analogicznie do implementacji w C++/MPI, przyjęto, że wszystkie węzły będą wyposażone w procesory o tych samych parametrach. Pozwoliło to przydzielić każdej lokacji podproblem o dokładnie takiej samej liczbie wierszy danych.

Zadanie główne, którego kod zaprezentowano na rysunku 8, rozpoczyna pracę od wczytania danych z pliku wejściowego. Obszary wczytywanych wierszy są tak

ustalone, aby pokryły plik danych. Jest to rozwiązanie analogiczne do zastosowanego w implementacji C++/MPI [30]. Następnie realizowane są kroki 2 i 3 algorytmu klastrowego.

```
1. public static def main(startval : Rail[String]) {
2.  /*....*/ zadanie wstępne
3.  finish for (p in Place.places()) {
4.   at (p) async {
5.    var tempcentr : Rail[Double] = new Rail[Double](centrfind.size,0n);
6.    /*....*/
7.    val start_read_file : Long = lines_per_places*(p.id);
8.    val list = Datafile.fileprocess(start_read_file,lines_per_places);
9.    while(true) {
10.     iteration++;
11.     val kmeans = new Kmeans(list,centrfind);
12.     kmeans.calculate_kmeans(p);
13.     team.reduce(Place.place(0),kmeans.sum,0L,tempcentr,0L,kmeans.sum.size,Team.ADD);
14.     team.bcast(Place.place(0),tempcentr,0L,centrfind.groupsum,0L,kmeans.sum.size);
15.    /* team.allreduce(kmeans.sum,0L,centrfind.groupsum,0L,kmeans.sum.size,Team.ADD);   */
16.     centrfind.calculatecentroid();
17.     SSE=centrfind.getSSE();
18.     if (((lastSSE-SSE)<Sys.dist)||(ITERATION_LIMIT<iteration)) {
19.      val Savedata = new Filewrite(Sys.outfilename,kmeans,p.id);
20.      /*....*/
21.      Savedata.save(start_read_file);
22.      break;
23.     }
24.     lastSSE=SSE;
25.    }
26.   }
27.  }
28. }
```

Rysunek 8. Kod w języku X10 dla zadania głównego

Podobnie jak w implementacji C++ z wykorzystaniem MPI, krok 4, tj. sprawdzenie warunku zbieżności, realizowany jest poprzez sprawdzenie zmniejszenia sumarycznego błędu kwadratowego (3). Analogicznie, przyjęto dodatkowy warunek powodujący zakończenie programu po wykonaniu 11 iteracji kroków 2–4.

Wykonywanie obliczeń realizuje metoda *calculate_kmeans* oraz metody *findcentroid*, *calculatecentroid* i *getSEE* obiektu *centrfind*. Zadania związane z wymianą danych oraz synchronizacją procesów są realizowane przez parę *team.reduce* i *team.bcast*.

Na rysunku 8, w linii 15, pokazano alternatywny zapis wywołania metody *team.all-reduce*, równoważny pod względem komunikacji parze *team.reduce – team.bcast* (linie 13 i 14). Obiekty *Datafile* i *Savedata* odpowiadają odpowiednio za odczyt danych wejściowych i zapis wyników.

Jak widać, za bezpośrednią realizację algorytmu k-means odpowiadają: metoda *calculate_kmeans* oraz linie wewnątrz pętli *while*. Warto w tym miejscu zwrócić uwagę, że inaczej niż dla implementacji w C++/MPI, nie dokonywano rozróżnienia pomiędzy rodzajami węzłów. W efekcie kod bezpośrednio realizujący algorytm k-means zapisano tylko jednokrotnie. Łącznie, bezpośrednią implementację realizacji algorytmu k-means zapisano w 167 liniach.

5 Architektura i środowisko uruchomieniowe, dane wejściowe

5.1. Ogólna charakterystyka środowiska, dane wejściowe

Klaster obliczeniowy ZEUS znajduje się w Akademickim Centrum Komputerowym CYFRONET AGH [35]. Jest to grupa różnego rodzaju serwerów obliczeniowych udostępnianych poprzez system kolejkowy Torque. Do celów badań wykorzystano węzły klastra wyposażone w 2 procesory Intel® Xeon® X5650 [36] zawierające po 6 rdzeni i pracujące w architekturze SMP. Węzły były połączone ze sobą z wykorzystaniem sieci Infiniband [37]. Pliki zawierające dane wejściowe oraz wyniki pracy przechowywane były w systemie plików Lustre [38].

Klaster udostępnia różnego rodzaju pakiety obliczeniowe, kompilatory i biblioteki [39]. Do badań wykorzystano następujące pakiety: kompilator GNU GCC 4.8.2 [40], kompilator Java 1.7.05, pakiet ANT 1.9.0 [41] oraz bibliotekę OpenMPI w wersji 1.6.5 z wsparciem dla sieci Infiniband [42].

Kod zapisany z wykorzystaniem środowiska C++/MPI skompilowano z wykorzystaniem przełącznika –msse4.2 wymuszającego wykorzystanie technologii Streaming SIMD Extensions 4 (SSE 4) [43] oraz przełącznika –O3 wymuszającego pełną optymalizację przez kompilator [44]. Celem takiego postępowania było uzyskanie kodu wynikowego możliwie najlepiej zoptymalizowanego pod względem wydajności [45].

Jako plik danych wejściowych wykorzystano plik próbek mikrodanych do publicznego użytku powstały w wyniku badań populacji USA przeprowadzonych przez Biuro Spisu Ludności Stanów Zjednoczonych Ameryki [10]. Plik zawiera dane tekstowe zapisane w postaci tabelarycznej. Tabela składa się z 4432861 linii danych oraz 228 kolumn danych. Słowniczek wartości mogących wystąpić w poszczególnych polach dostępny jest w dokumentacji opisującej plik próbek mikrodanych [46].

Do celów badania wykorzystano pola z kolumn PWGTP1, PWGTP3, PWGTP5, PWGTP7, PWGTP8, PWGTP10, PWGTP12, PWGTP14, PWGTP16, PWGTP19. Zgodnie ze słowniczkiem wartości, pola te zawierają liczby całkowite ze zbioru $\langle -9999, 9999 \rangle$.

5.2. Konfigurowanie środowiska uruchomieniowego dla języka X10

Kompletną paczkę, ze źródłami dla kompilatora i bibliotek języka X10, można pobrać z witryny języka X10 [47]. Do badań wykorzystano wersję 2.4.2 języka X10. Kompilacja kodu źródłowego kompilatora i bibliotek języka X10 wymaga pakietu ANT oraz kompilatora Java.

Ze względów bezpieczeństwa, na klastrze ZEUS wyłączono możliwość logowania na węzłach z wykorzystaniem usługi SSH. Oznacza to, że kod wynikowy programu w języku X10 nie może korzystać z komunikacji pomiędzy lokacjami (ang. *places*) za pomocą gniazd. Zgodnie z pracą [24], za obsługę komunikacji w kodzie wynikowym odpowiada biblioteka X10RT. Istnieje kilka implementacji tej biblioteki [48, 5]. Do celów badań wykorzystano implementację korzystającą ze standardu MPI-2 [9] załączoną za pomocą opcji DX10RT_MPI=true. Celem uzyskania kodu wynikowego możliwie najlepiej zoptymalizowanego pod względem wydajności [45] skorzystano z przełącznika Doptimize=true [49].

Kod zapisany w języku X10 jest kompilowany najpierw do kodu pośredniego w innym języku programowania. Dopiero kod pośredni kompilowany jest do kodu wynikowego z użyciem kompilatora danego języka. Kod zapisany w języku X10 można skompilować do kodu pośredniego w języku C++ lub Java. Ścieżkę kompilacji z wykorzystaniem C++ nazywa się Native X10, a ścieżkę kompilacji z wykorzystaniem Java – Managed X10 [24].

Kod zapisany w języku X10 skompilowano z wykorzystaniem ścieżki Native X10. Załączono opcję –O wymuszającą optymalizację przez kompilator oraz opcję nakazującą wykorzystanie biblioteki X10RT implementującej komunikację z wykorzystaniem standardu MPI-2. Wykorzystano w tym celu kompilator x10c++ [8].

6 Porównanie wydajności algorytmu k-means zaimplementowanego w X10 i w środowisku C++/MPI

Obie wersje algorytmu K-means uruchomiono dla:

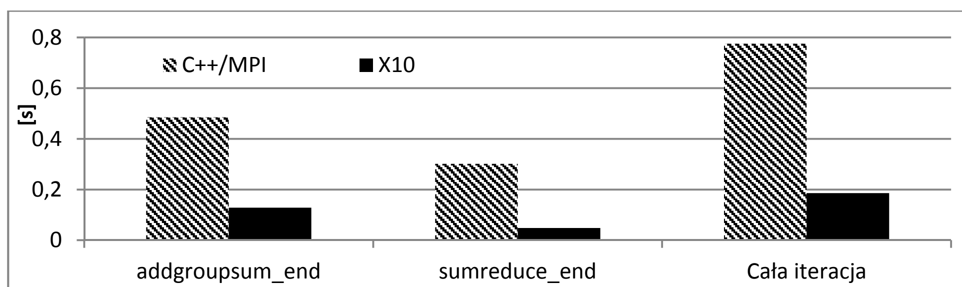
- 100000, 250000, 400000 wierszy danych,
- 3, 5, 10 kolumn danych,
- 10 i 20 grup (K).

Dało to 18 kombinacji danych wejściowych. Każda kombinacja danych była uruchomiona na 4 węzłach czyli 8 procesorach (48 rdzeniach), 2 węzłach czyli 4 procesorach (24 rdzeniach) i jednym procesorze czyli 6 rdzeniach. Otrzymano siatkę pomiarową zawierającą 54 punkty pomiarowe.

Ze względu na fakt wykonywania programu w środowisku wieloużytkownikowym, dla każdego punktu pomiarowego przeprowadzono 10 pomiarów czasów wykonania programu i jego części [29].

Podczas wykonywania programu zapisywano między innymi czasy wykonywania następujących części:

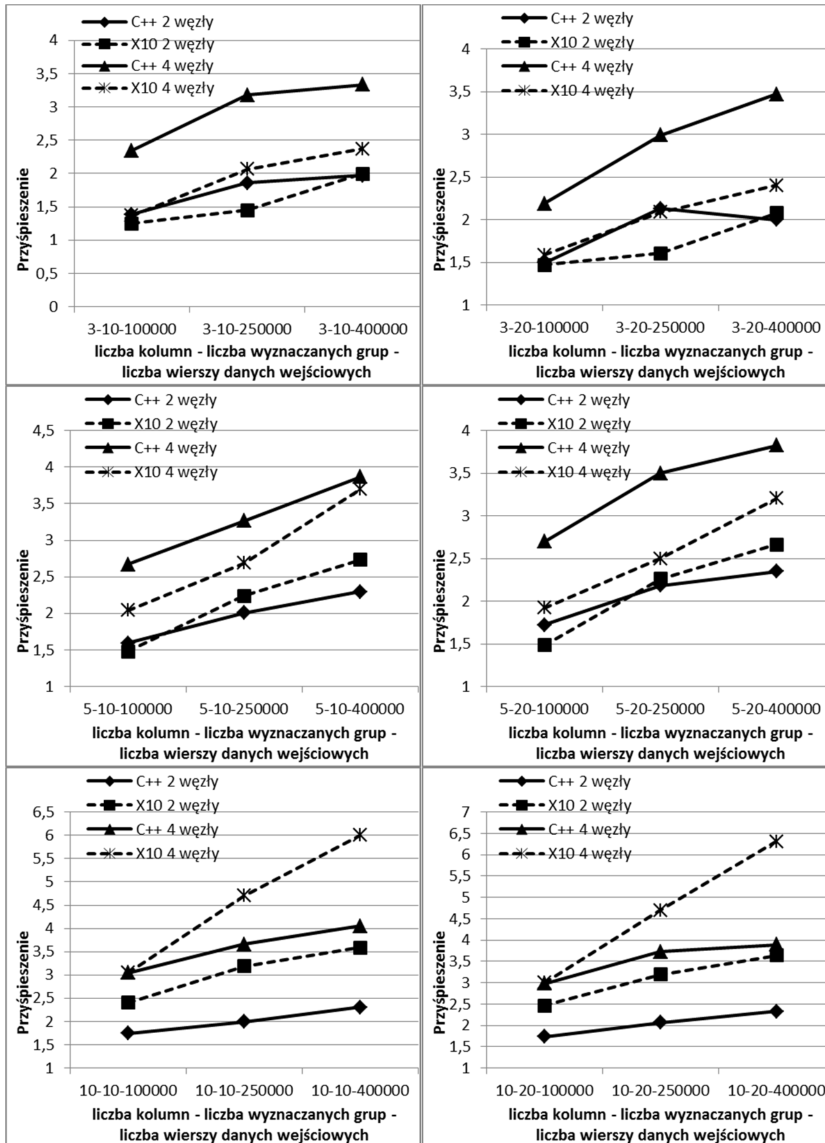
- operacji wczytywania danych z pliku wejściowego,
- pojedynczej iteracji algorytmu k-średnich (kroki 2 -4), a w tym (patrz rysunek 9):
  - obliczeń związanych z wyznaczaniem odległości punktów od centroidów oraz wyznaczenia sum dla nowych środków grup (addgroupsum_end),
  - wymiany danych pomiędzy węzłami (sumreduce_end),
  - wyznaczenia sumarycznego błędu kwadratowego i sprawdzenie warunku zbieżności (SSE_reach),
  - operacji zapisu danych wyjściowych,
  - całego programu.



Rysunek 9. Średnie czasy wykonania poszczególnych części kodu iteracji algorytmu k-średnich dla następujących parametrów: 4 węzły po 2 procesory każdy (łącznie 48 rdzeni), 10 kolumn, 20 grup, 400000 linii

Na rysunku 10 zamieszczono wykresy prezentujące zmiany przyśpieszenia dla ustalonej liczby kolumn i wyznaczanych grup przy zmianie liczby wierszy danych wejściowych. Dla każdej kombinacji powyższych wartości zaprezentowano zmiany przyspieszenia, dla obu wersji implementacji algorytmu k-średnich. Przyśpieszenie wyznaczono jako stosunek czasu realizacji kodu na 1 procesorze (6 rdzeni) oraz czasu realizacji kodu na: 2 węzłach (24 rdzenie) i 4 węzłach (48 rdzeni).

Maksymalna wartość przyśpieszenia, równa około 6.3, została uzyskana dla pomiaru wykonanego na 4 węzłach, czyli 8 procesorach (48 rdzeniach), dla kodu w X10 przy 10 kolumnach, 400`000 liniach i wyznaczaniu 20 grup.



Rysunek 10. Zmiany przyśpieszenia algorytmu centroidów przy zmianie liczby wierszy danych wejściowych

Analizując otrzymane wykresy, można zauważyć większe przyśpieszenie kodu zapisanego w środowisku C++/MPI dla małych rozmiarów problemu. W trakcie zwiększania rozmiaru problemu coraz większe przyśpieszenia są osiągane przez kod zapisany w języku X10. Wyjaśnienie tego faktu można odnaleźć na rysunku 9, prezentującym średnie czasy wykonania poszczególnych części kodu pojedynczej iteracji algorytmu k-średnich (kroki 2–4).

Widać, że o czasie trwania iteracji w C++/MPI decydują części kodu oznaczone jako addgroupsum_end oraz sumreduce_end. Wspomniane części kodu prezentują czasy realizacji wywołania następujących metod (rysunek 5):

- dla części oznaczonej jako addgroupsum_end są to metody *calculate_kmeans* i *addgroupsum* z obiektu *sum*,
- dla części oznaczonej jako sumreduce_end jest to metoda *sumreduce* z obiektu *sum*.

Metoda *addgroupsum* dla kodu w C++/MPI wykonuje dodawanie sum dla nowych środków grup w ramach pojedynczego węzła. Odbywa się to z wykorzystaniem pamięci współdzielonej i wymaga oczekiwania aż wszystkie procesy dotrą do tego miejsca.

Na rysunku 11 zawarto kod realizujący powyższe operacje, zapisany w języku C++ dla zmiennej typu *double*. Kod dla zmiennej typu *long double* był analogiczny, z tym, że w liniach 2–5 zamiast double użyto *long double*.

```
1.  int main(int argc, char **argv) {
2.     long N = 10000000;
3.     double val=DBL_MAX;
4.     double temp=0;
5.     map<int,double> result;
6.     for(int i=0;i<N;i++) {
7.        temp=val/2;
8.        val=val-temp;
9.        result[i]=val;
10.   }
11.}
```

Rysunek 11. Kod w C++ użyty do pomiaru czasu wykonania operacji arytmetycznych i zapisu wyniku dla zmiennych typu double

W kodzie X10, część oznaczona jako addgroupsum_end, reprezentuje tylko czas wywołania metody *calculate_kmeans* z obiektu *kmeans*, gdyż każda lokacja wyznacza tylko sumy częściowe (rysunek 8). Ich ostateczne dodawanie odbywa się później. Część kodu oznaczona jako sumreduce_end, w X10 reprezentuje wywołanie pary metod *team.reduce – team.bcast* (rysunek 8).

Na rysunku 12 zaprezentowano analogiczny kod dla języka X10.

```
1. public class Pomiar {
2.   public static def main(Rail[String]) {
3.     val N : long = 10000000;
4.     var value : Double = Double.MAX_VALUE;
5.     var temp : Double = 0;
6.     var result : HashMap[Long,Double] = new HashMap[Long,Double]();
7.     for(i  in 0 .. N) {
9.       temp=value/2.0;
10.      value=value-temp;
11.      result.put(i,value);
12.    }
13.  }
14. }
```

Rysunek 12. Kod w X10 użyty do pomiaru czasu wykonania operacji arytmetycznych i zapisu wyniku dla zmiennych typu double

Jak już wspomniano, część kodu oznaczona jako addgroupsum_end obejmuje obliczenia związane z wyznaczaniem odległości punktów od centroidów. W środowisku C++/MPI skorzystano z możliwości wykonania obliczeń dla typu *long double*, co pozwoliło otrzymać bardziej dokładne wyniki.

Język X10 dysponuje tylko typem *double*, którego użycie dało wyniki nieco mniej dokładne. Niedokładność ta objawiała się poprzez zaklasyfikowanie punktów leżących na granicach grup do innej grupy. Nie dokonano pomiaru czasu wykonywania elementu programu związanego z realizacją samych obliczeń, należy jednak sądzić, że użycie typu *long double* mogło wydłużyć czas wykonywania obliczeń i zapamiętywania wyników w kontenerach w środowisku C++/MPI [50].

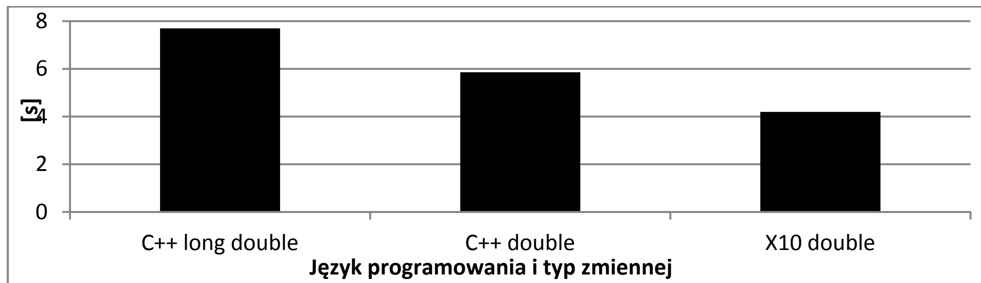
Celem sprawdzenia tej tezy, przeprowadzono dodatkowy pomiar czasu wykonania kodu zapisanego w języku C++ i X10, realizującego operacje dzielenia i odejmowania oraz zapamiętywania wyników w kontenerze z wykorzystaniem następujących typów:

- dla języka C++: *double* i *long double*, kontener typu map,
- dla jezyka X10: *double*, kontener typu *HashMap*.

Tabela 1. Czasy wykonywania kodu w C++ z wykorzystaniem MPI i X10 dla różnych liczb procesorów i węzłów (4 CPU – 2 węzły, 8CPU – 4 węzły)

Liczba:			**C++/MPI**			**X10**		
Kolumn	**Grup**	**Linii**	**1CPU**	**4CPU**	**8CPU**	**1CPU**	**4CPU**	**8CPU**
3	**10**	**100000**	0,22	0,16	0,10	0,08	0,06	0,06
		250000	0,63	0,34	0,20	0,13	0,09	0,06
		400000	1,01	0,51	0,30	0,19	0,09	0,08
	20	**100000**	0,22	0,15	0,10	0,08	0,05	0,05
		250000	0,64	0,30	0,22	0,13	0,08	0,06
		400000	1,01	0,51	0,30	0,19	0,09	0,08
5	**10**	**100000**	0,32	0,20	0,12	0,12	0,08	0,06
		250000	0,88	0,44	0,27	0,23	0,10	0,09
		400000	1,62	0,71	0,42	0,37	0,14	0,10
	20	**100000**	0,34	0,20	0,13	0,12	0,08	0,06
		250000	0,92	0,42	0,26	0,24	0,10	0,09
		400000	1,60	0,68	0,42	0,37	0,14	0,12
10	**10**	**100000**	0,61	0,35	0,20	0,30	0,13	0,10
		250000	1,68	0,85	0,46	0,69	0,22	0,15
		400000	3,08	1,33	0,76	1,17	0,33	0,20
	20	**100000**	0,60	0,35	0,20	0,30	0,12	0,10
		250000	1,72	0,84	0,46	0,70	0,22	0,15
		400000	3,01	1,29	0,78	1,17	0,32	0,19

Wyniki pomiaru zaprezentowano na rysunku 13.



Rysunek 13. Czasy wykonywania kodu z rysunków 11 i 12

Widać, że najdłużej wykonywany jest kod dla typu *long double* zapisany w języku C++, natomiast najszybszy jest kod zapisany w X10. Warto zauważyć, że kod zapisany w języku C++ i korzystający z zmiennej *double*, jest szybszy od kodu dla zmiennej *long double*, ale wolniejszy od kodu w języku X10.

W części kodu oznaczonej jako sumreduce_end następuje dodanie sum dla nowych środków grup. W kodzie C++/MPI wykonywane jest sumowanie wartości wyznaczonych dla poszczególnych węzłów, a w kodzie X10 następuje zsumowanie wartości wyznaczonych we wszystkich lokacjach. Kod w środowisku C++/MPI korzysta z funkcji *MPI_Allreduce*, a kod w języku X10 korzysta z pary metod *team.reduce* i *team.bcast*.

Jak wskazanono w sekcji 4.2, dla dużej liczby procesów (wątków) powoduje to istotne różnice w wydajności. Warto tu też zwrócić uwagę, że implementacja w X10 także korzysta z pamięci współdzielonej, jednak odbywa się to w efekcie rozwiązań zastosowanych w bibliotece OpenMPI implementującej standard MPI-2 dla sieci Infiniband, z uwzględnieniem architektury SMP [31, 32, 33, 34].

Wynikowe, średnie czasy zaprezentowano w tabeli 1 dla każdej kombinacji danych wejściowych przetwarzanych na różnej liczbie węzłów. Charakterystyczny jest fakt, że czasy dla środowiska C++/MPI są większe niż dla środowiska X10. Wyjaśnienie tego faktu można znaleźć w komentarzach do rysunku 8 oraz rysunków 10–12.

7 Wnioski i plan dalszych badań

W niniejszej pracy dokonano porównania dwu implementacji popularnego algorytmu grupowania danych. Jedna z implementacji stworzona została z wykorzystaniem języka C++ oraz standardu MPI. Wykorzystano w niej równocześnie dwa modele programowania współbieżnego: z pamięcią wspólną i model sieciowy. Druga implementacja wykorzystuje nowoczesny język programowania X10 stworzony przez IBM. Tworząc jej kod, skorzystano z modelu programowania sieciowego.

Analizując przyspieszenia obu implementacji, stwierdzono, że dla większej liczby procesorów szybszy jest program zapisany w języku X10 korzystający z rozsyłania danych przy pomocy metod *team.reduce* i *team.bcast*. Stwierdzono także znaczny wzrost czasu wykonywania programu w języku X10 korzystającego z metody *team.allreduce*. Kwestią zwracającą uwagę jest także długi czas wykonywania sumowania danych pośrednich z wykorzystaniem pamięci współdzielonej w środowisku C++/MPI.

Warto tu także zwrócić uwagę na wielkość kodu w obu implementacjach. Bezpośrednia implementacja algorytmu k-means w środowisku C++/MPI zawiera 283

linii kodu, a implementacja w X10 to 167 linii. Widać więc, że nakład pracy związany z tworzeniem kodu w X10 był mniejszy co przełożyło się na krótszy czas tworzenia kodu.

Dalsze badania powinny więc obejmować analizę wydajności obu implementacji po dokonaniu zmian zmierzających do jej poprawy. W C++/MPI zmiany powinny dotyczyć rozsyłania danych pomiędzy węzłami oraz sumowania danych pośrednich w poszczególnych węzłach z wykorzystaniem pamięci wspólnej. W implementacji X10 warto rozważyć użycie innych konstruktów języka. Warto także rozważyć implementację algorytmu, którego sposób działania nie wymuszałby w każdej iteracji wymiany danych, a więc także i synchronizacji uruchomionych procesów. Pozwoliło by to zmniejszyć wpływ czasu komunikacji na przyśpieszenie programu.

Przyszłe badania powinny także uwzględniać analizę wydajności tworzenia kodu. Już pobieżna analiza liczby linii kodu, bezpośrednio związanego z realizacją algorytmu k-means, wykazała 59% różnicę na korzyść języka X10. Dobrym punktem wyjścia do tych badań jest praca [51]. Dokonano w niej porównania wydajności kodowania 6 algorytmów w języku X10 oraz języku C z wykorzystaniem standardu MPI. Porównano rożne konstrukty języka X10 z analogicznymi rozwiązaniami w języku C i standardzie MPI.

Biorąc pod uwagę wzrastanie przyśpieszenia programu zaimplementowanego w języku X10 przy zwiększaniu liczby procesorów oraz uwzględniając fakt, że celem stworzenia języka X10 była między innymi możliwość uruchamiania kodu na dużych klastrach, należy rozważyć prowadzenie badań wydajności na większej liczbie procesorów.

Praca została wykonana z wykorzystaniem Infrastruktury PL-Grid.

Bibliografia

[1] Czech Z., *Wprowadzenie do obliczeń równoległych*, wyd. pierwsze,Warszawa 2010: Wydawnictwa Naukowe PWN

[2] Dotsenko Y., Mellor-Crummey J., Cantonnet F., El-Ghazawi T., Mohanti A., Yao Y., Chavarría-Miranda D., Coarfa C., *An evaluation of global address space languages: co-array fortran and unified parallel C*, w: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, Chicago 2005

[3] Callahan D., Zima H.P., Chamberlain B.L., *Parallel programmability and the chapel language*, "International Journal of High Performance Computing Applications" 2007, Vol. 21, No. 3

[4] Colella P., Gay D., Graham S., Hilfinger P., Krishnamurthy A., Liblit B., Pike

C.M., Semenzato G.L., Aiken A., *Titanium: A high-performance Java dialect*, ACM 1998 Workshop on Java for High-Performance Network Computing, 1997, http://pages.cs.wisc.edu/~liblit/titanium/titanium.pdf

[5] Herta B., Cunningham D., Grove D., Kambadur P.n, Saraswat V., Shinnar A., M. Takeuchi, M. Vaziri, O. Tardieu, *X10 and APGAS at Petascale*, w: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Orlando 2014

[6] Bonachea D., Chen W., Colella P., Datta K., Duell J., Graham S.L., Hargrove P., Hilfinger P., Husbands P., Iancu C., Kamil A., Nishtala R., Su J., Welcome M., Wen T., Yelick K., *Productivity and performance using partitioned global address space languages*, w: *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, Waterloo 2007

[7] Almasi G., Bikshandi G., Cascaval C., Grove D., Cunningham D., Tardieu O., Peshansky I., Kodali S., Saraswat V., *The Asynchronous Partitioned Global Address Space Model*, w: *Proceedings of The First Workshop on Advances in Message Passing*, Toronto 2010

[8] *X10: Performance and Productivity at Scale*, http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html

[9] *MPI: A Message – Passing Interface Standard version 2.2*, http://www.mpi-forum.org/docs/mpi22-report.pdf

[10] United States Census Bureau, *ACS Public Use Microdata Sample (PUMS) File*, http://www2.census.gov/acs2010_5yr/pums/csv_pus.zip

[11] Larose D.T., *Odkrywanie wiedzy z danych: wprowadzenie do eksploracji danych*, Warszawa 2013: Wydawnictwo Naukowe PWN

[12] Ranka S., Singh V., Alsabti K., *An efficient k-means clustering algorithm*, w: *Proceedings of the 1st Workshop on High Performance Data Mining*, Orlando 1998

[13] Modha D.S., Dhillon I.S., *A data – clustering algorithm on distributed memory multiprocessors*, w: *Large-Scale Parallel Data Mining*, Berlin Heidelberg 2000: Springer

[14] Fayyad U.M., Bradley P.S., *Refining Initial Points for K-means Clustering*, w: *Proceedings of the Fifteenth International Conference on Machine Learning*, San Francisco 1998: ICML

[15] Elkan Ch., *Using the triangle inequality to accelerate k-means*, w: *Twentieth International Conference on Machine Learning*, Washington 2003

[16] Moore A., Pelleg D., *Accelerating exact k-means algorithms with geometric reasoning*, w: *Proceedings of the fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego 1999

[17] Rebolledo D., Chan E., Campbell R.H., Farivar R., *A Parallel Implementation of K-Means Clustering on GPUs*, w: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and*

Applications, Las Vegas 2008
[18] Meila M., *The uniqueness of a good optimum for k-means*, w: *Proceedings of the 23rd International Conference on Machine Learning*, New York 2006
[19] Tajunisha S., *Performance analysis of k-means with different initialization methods for high diemnsional data*, "Interantional Journal of Artifical Intelligence & Applications" 2010, Vol. 1, No. 4
[20] Kamber M., Pei J., Han J., *Data Mining: Concepts and Techniques*, Burlington 2006: Morgan Kaufmann
[21] Dimov S.S., Nguyen C.D., Pham D.T., *Selection of K in K-means clustering, Proceedings of the Institution of Mechanical Engineers, Part C*, "Journal of Mechanical Engineering Science" 2005, Vol. 219, No. 1
[22] *Język programowania X10*, http://x10-lang.org/home/introduction.html
[23] *X10 Programming Language*, http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf
[24] Tardieu O., Cunningham D., Herta B., Peshansky I., Saraswat V., Grove D., *A performance model for X10 applications: what's going on under the hood?*, w: *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, San Jose 2011
[25] Otto S., Huss-Lederman S., Walker D., Dongarra J., Snir M., *MPI: The Complete Reference*, Cambridge 1996: MIT Press
[26] *JTC1/SC22/WG21 – The C++ Standards Committee*, http://www.open-std.org/JTC1/SC22/WG21/
[27] Williams A., *Język C++ i przetwarzanie współbieżne w akcji*, Gliwice 2013: Wydawnictwo HELION
[28] *C++ Reference*, http://www.cplusplus.com/reference/thread/thread/hardware_concurrency/
[29] Kwiatkowski J., *Parallel Applications Performance Evaluation Using the Concept of Granularity*, LNCS 2014, Vol. 8385
[30] Bisgin H., *Parallel clustering algorithms with application to climatology*, Informatics Institute, Istanbul 2008: Istambul Technical University
[31] Woodall T.S., Graham R.L., Maccabe A.B., Bridges P.G., Shipman G.M., *Infiniband scalability in Open MPI*, w: *20th International Parallel and Distributed Processing Symposium*, Rhodes Island 2006: IPDPS
[32] Gopalakrishnan S., Hyun-Wook J., Panda D.K., Huang W., *Scheduling of MPI-2 one sided operations over InfiniBand*, w: *Proceedings of the 19th IEEE Int. Parallel and Distributed Processing Symposium*, Denver 2005
[33] Jiuxing L., Hyun-Wook J., Panda D.K., Gropp W., Thakur R., Weihang J., *High performance MPI-2 one-sided communication over InfiniBand*, w: *IEEE Int. Symposium on Cluster Computing and the Grid*, Chicago 2004
[34] Nieplocha J., Panda D., Tipparaju V., *Fast collective operations using shared and remote memory access protocols on clusters*, w: *Proceedings of the Int. Symp. Parallel and Distributed Processing*, Nice 2003

[35] *Komputery Dużej Mocy w ACK CYFRONET AGH*, https://kdm.cyfronet.pl/portal/Zeus
[36] *Specyfikacja procesora Intel® Xeon® Processor X5650*, http://ark.intel.com/pl/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI#@specification
[37] *Infiniband Technology Specification*, http://www.ieee802.org/3/
[38] Karpusenko V., Yoo T., Vladimirov A., *File I/O on Intel Xeon Phi Coprocessors: RAM disks, VirtIO, NFS and Lustre*, http://hgpu.org, 2014
[39] *Oprogramowanie – Komputery Dużej Mocy w ACK CYFRONET AGH*, https://kdm.cyfronet.pl/portal/Oprogramowanie
[40] *GCC 4.8.2 manuals – GNU Project – Free Software Foundation (FSF)*, https://gcc.gnu.org/onlinedocs/4.8.2/
[41] *Apache Ant*, http://ant.apache.org/
[42] *Open MPI: Version 1.6.5*, Open MPI Software, http://www.open-mpi.org/software/ompi/v1.6/
[43] Kuah K., *Motion Estimation with Intel® Streaming SIMD Extensions 4 (Intel® SSE4)*, https://software.intel.com/en-us/articles/motion-estimation-with-intel-streaming-simd-extensions-4-intel-sse4/?iid=2121&wapkw=sse4
[44] *Using the GNU Compiler Collection (GCC), Optimize Options*, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[45] Anderson T.A., Liu H., Glew N., Petersen L., *Measuring the Haskell gap*, w: *Proceedings of the Int. 25th Symposium on Implementation and Application of Functional Languages*, Nijmegen 2013
[46] United States Census Bureau, *2006 – 2010 ACS PUMS DATA DICTIONARY*, http://www.census.gov/acs/www/Downloads/data_documentation/pums/DataDict/PUMS_Data_Dictionary_2006-2010.pdf
[47] Źródła języka X10, http://x10-lang.org/software/download-x10/release.html
[48] *Implementacje biblioteki X10RT*, http://x10-lang.org/documentation/practical-x10-programming/x10rt-implementations.html
[49] *Building X10 from source*, http://x10-lang.org/X10-development/building-X10-from-source.html
[50] Li X.S., Bailey D.H., Hida Y., *Algorithms for quad-double precision floating point arithmetic*, http://www.escholarship.org/uc/item/69q5t2mj
[51] Brezin J., Swart C.B., Halverson Ch.A., Richards J.T., *A decade of progress in parallel programming productivity*, "Communications of the ACM" 2014, Vol. 57, No. 11

Performance comparison of the k-means algorithm implemented in the X10 programming language and the C++/MPI environment

Abstract

In this work the k-means algorithm and the way of its implementation in the X10 programming language are described. The achieved results are compared with the implementation of the same algorithm in the C++11 programming language using the MPI standard. It was confirmed that the implementation in the X10 programming language is faster on a large number of processors than the implementation in the C++/MPI environment. Additionally, the X10 code is about 59% shorter than the code for the C++/MPI combination.

Keywords – k-means algorithm, X10 programming language, C++/MPI environment, comparison