Opracowanie w modelu PGAS wybranych, równoległych algorytmów grafowych i ich implementacja przy użyciu języka Java

Magdalena Ryczkowska

Wydział Matematyki i Informatyki
Uniwersytet Mikołaja Kopernika
Chopina 12/18, 87-100 Toruń
gdama@mat.umk.pl

Opiekun naukowy:
prof. dr hab. Piotr Bała
Interdyscyplinarne Centrum Modelowania
Matematycznego i Komputerowego
Uniwersytet Warszawski
bala@icm.edu.pl

Cel oraz wkład pracy

Cel oraz wkład pracy:

- opracowanie nowych, równoległych algorytmów grafowych dostosowanych do modelu PGAS
- zastosowane optymalizacje np. nakładanie komunikacji i obliczeń
- wydajna implementacja w języku Java dla środowiska równoległego i rozproszonego (dostosowanie do specyfiki języka Java)
- weryfikacja skalowalności i wydajności opracowanych algorytmów na różnych architekturach sprzętowych
- porównanie z dostępnymi rozwiązaniami w języku C bazującymi na MPI

Cel oraz wkład pracy

Cel oraz wkład pracy:

- opracowanie nowych, równoległych algorytmów grafowych dostosowanych do modelu PGAS
- zastosowane optymalizacje np. nakładanie komunikacji i obliczeń
- wydajna implementacja w języku Java dla środowiska równoległego i rozproszonego (dostosowanie do specyfiki języka Java)
- weryfikacja skalowalności i wydajności opracowanych algorytmów na różnych architekturach sprzętowych
- porównanie z dostępnymi rozwiązaniami w języku C bazującymi na MPI

Cel oraz wkład pracy

Cel oraz wkład pracy:

- opracowanie nowych, równoległych algorytmów grafowych dostosowanych do modelu PGAS
- zastosowane optymalizacje np. nakładanie komunikacji i obliczeń
- wydajna implementacja w języku Java dla środowiska równoległego i rozproszonego (dostosowanie do specyfiki języka Java)
- weryfikacja skalowalności i wydajności opracowanych algorytmów na różnych architekturach sprzętowych
- porównanie z dostępnymi rozwiązaniami w języku C bazującymi na MPI

Cel oraz wkład pracy

Cel oraz wkład pracy:

- opracowanie nowych, równoległych algorytmów grafowych dostosowanych do modelu PGAS
- zastosowane optymalizacje np. nakładanie komunikacji i obliczeń
- wydajna implementacja w języku Java dla środowiska równoległego i rozproszonego (dostosowanie do specyfiki języka Java)
- weryfikacja skalowalności i wydajności opracowanych algorytmów na różnych architekturach sprzętowych
- porównanie z dostępnymi rozwiązaniami w języku C bazującymi na MPI

Cel oraz wkład pracy

Cel oraz wkład pracy:

- opracowanie nowych, równoległych algorytmów grafowych dostosowanych do modelu PGAS
- zastosowane optymalizacje np. nakładanie komunikacji i obliczeń
- wydajna implementacja w języku Java dla środowiska równoległego i rozproszonego (dostosowanie do specyfiki języka Java)
- weryfikacja skalowalności i wydajności opracowanych algorytmów na różnych architekturach sprzętowych
- porównanie z dostępnymi rozwiązaniami w języku C bazującymi na MPI

Cel oraz wkład pracy

Cel oraz wkład pracy:

- opracowanie nowych, równoległych algorytmów grafowych dostosowanych do modelu PGAS
- zastosowane optymalizacje np. nakładanie komunikacji i obliczeń
- wydajna implementacja w języku Java dla środowiska równoległego i rozproszonego (dostosowanie do specyfiki języka Java)
- weryfikacja skalowalności i wydajności opracowanych algorytmów na różnych architekturach sprzętowych
- porównanie z dostępnymi rozwiązaniami w języku C bazującymi na MPI

Plan prezentacji

1. Wprowadzenie
2. Model PGAS
   - Główne założenia
   - Biblioteka PCJ
3. Graph500
   - Specyfikacja
4. Graph500 w PGAS i PCJ
   - Kernel 1
   - Kernel 2
   - Wyniki
5. Dalsze prace

Plan prezentacji

1. Wprowadzenie
2. Model PGAS
   - Główne założenia
   - Biblioteka PCJ
3. Graph500
   - Specyfikacja
4. Graph500 w PGAS i PCJ
   - Kernel 1
   - Kernel 2
   - Wyniki
5. Dalsze prace

Plan prezentacji

1. Wprowadzenie
2. Model PGAS
   - Główne założenia
   - Biblioteka PCJ
3. Graph500
   - Specyfikacja
4. Graph500 w PGAS i PCJ
   - Kernel 1
   - Kernel 2
   - Wyniki
5. Dalsze prace

Plan prezentacji

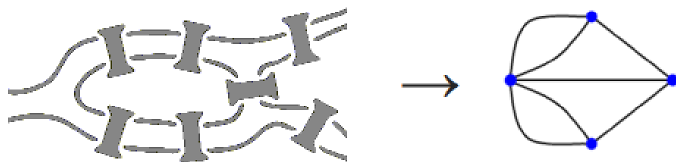
1. Wprowadzenie
2. Model PGAS
   - Główne założenia
   - Biblioteka PCJ
3. Graph500
   - Specyfikacja
4. Graph500 w PGAS i PCJ
   - Kernel 1
   - Kernel 2
   - Wyniki
5. Dalsze prace

Plan prezentacji

1. Wprowadzenie
2. Model PGAS
   - Główne założenia
   - Biblioteka PCJ
3. Graph500
   - Specyfikacja
4. Graph500 w PGAS i PCJ
   - Kernel 1
   - Kernel 2
   - Wyniki
5. Dalsze prace

Pierwsza praca z teorii grafów

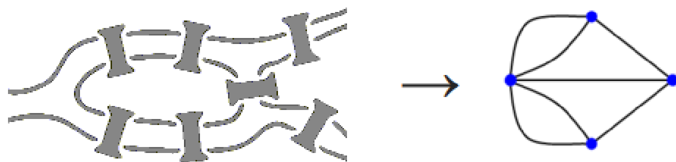
Pierwsze rozważania na temat grafów sięgają XVIII wieku i dotyczą tzw. problemu mostów królewieckich.



Sformułowanie zagadnienia przez Eulera i opublikowanie pod tytułem: *Solutio problematis ad geometriam situs pertinenti* uznawane jest za pierwszą pracę z teorii grafów.

Pierwsza praca z teorii grafów

Pierwsze rozważania na temat grafów sięgają XVIII wieku i dotyczą tzw. problemu mostów królewieckich.



Sformułowanie zagadnienia przez Eulera i opublikowanie pod tytułem: *Solutio problematis ad geometriam situs pertinenti* uznawane jest za pierwszą pracę z teorii grafów.

https://pl.wikipedia.org/wiki/Zagadnienie_mostów_królewieckich
http://www.math.edu.pl/mosty-krolewieckie

Rozwój tematyki związanej z grafami

Od tego czasu obserwujemy znaczny rozwój tematyki związanej z grafami.
Wiele problemów obliczeniowych może być łatwo przedstawionych w języku grafów, które w naturalny sposób obrazują wszelkiego rodzaju relacje.
Grafy wykorzystywane są w wielu różnych dziedzinach np. w socjologii, biologii ...

Rozwój tematyki związanej z grafami

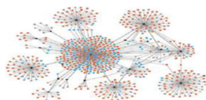
Od tego czasu obserwujemy znaczny rozwój tematyki związanej z grafami. Wiele problemów obliczeniowych może być łatwo przedstawionych w języku grafów, które w naturalny sposób obrazują wszelkiego rodzaju relacje.
Grafy wykorzystywane są w wielu różnych dziedzinach np. w socjologii, biologii ...

Rozwój tematyki związanej z grafami

Od tego czasu obserwujemy znaczny rozwój tematyki związanej z grafami. Wiele problemów obliczeniowych może być łatwo przedstawionych w języku grafów, które w naturalny sposób obrazują wszelkiego rodzaju relacje. Grafy wykorzystywane są w wielu różnych dziedzinach np. w socjologii, biologii ...

Rozwój tematyki związanej z grafami

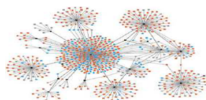
Od tego czasu obserwujemy znaczny rozwój tematyki związanej z grafami. Wiele problemów obliczeniowych może być łatwo przedstawionych w języku grafów, które w naturalny sposób obrazują wszelkiego rodzaju relacje. Grafy wykorzystywane są w wielu różnych dziedzinach np. w socjologii, biologii ...



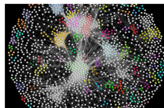
sieć społecznościowa

Rozwój tematyki związanej z grafami

Od tego czasu obserwujemy znaczny rozwój tematyki związanej z grafami. Wiele problemów obliczeniowych może być łatwo przedstawionych w języku grafów, które w naturalny sposób obrazują wszelkiego rodzaju relacje. Grafy wykorzystywane są w wielu różnych dziedzinach np. w socjologii, biologii ...



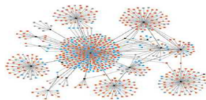
sieć społecznościowa



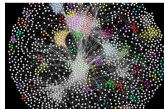
interakcje białek

Rozwój tematyki związanej z grafami

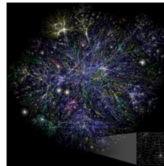
Od tego czasu obserwujemy znaczny rozwój tematyki związanej z grafami. Wiele problemów obliczeniowych może być łatwo przedstawionych w języku grafów, które w naturalny sposób obrazują wszelkiego rodzaju relacje. Grafy wykorzystywane są w wielu różnych dziedzinach np. w socjologii, biologii ...



sieć społecznościowa



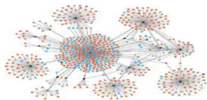
interakcje białek



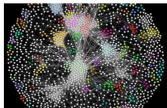
mapa Internetu

Rozwój tematyki związanej z grafami

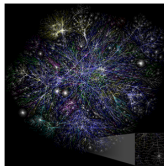
Od tego czasu obserwujemy znaczny rozwój tematyki związanej z grafami. Wiele problemów obliczeniowych może być łatwo przedstawionych w języku grafów, które w naturalny sposób obrazują wszelkiego rodzaju relacje. Grafy wykorzystywane są w wielu różnych dziedzinach np. w socjologii, biologii ...



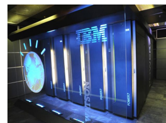
sieć społecznościowa



interakcje białek



mapa Internetu



IBM Watson

http://humannaturelab.net/resources/images/
http://commons.wikimedia.org/wiki/File:Internet_map_4096.png
http://hms.harvard.edu/news/researchers-build-largest-protein-interaction-map-date-10-27-11
http://www.techrepublic.com/article/ibm-watson-a-shining-example-of-how-to-take-big-data-to-the-next-level/

Wykorzystanie komputera dla obliczeń grafowych

Badanie właściwości grafów często wymaga dużych mocy obliczeniowych.
Przykładem wykorzystania komputerów w teorii grafów jest eksperyment przeprowadzony przez matematyków Kenneth'a Appel'a oraz Wolfgang'a Haken'a.
W 1976 roku udowodnili oni twierdzenie o czterech barwach dla grafów planarnych wykorzystując program napisany w języku Fortran.

https://pl.wikipedia.org/wiki/Twierdzenie_o_czterech_barwach

Wykorzystanie komputera dla obliczeń grafowych

Badanie właściwości grafów często wymaga dużych mocy obliczeniowych. Przykładem wykorzystania komputerów w teorii grafów jest eksperyment przeprowadzony przez matematyków Kenneth'a Appel'a oraz Wolfgang'a Haken'a.

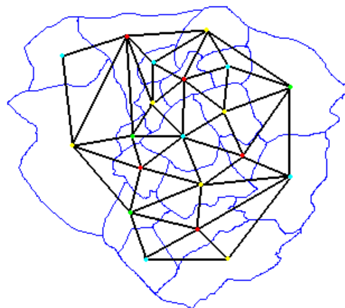
W 1976 roku udowodnili oni twierdzenie o czterech barwach dla grafów planarnych wykorzystując program napisany w języku Fortran.

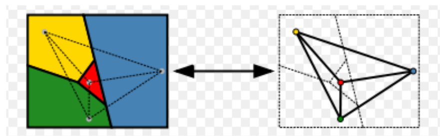
https://pl.wikipedia.org/wiki/Twierdzenie_o_czterech_barwach

Wykorzystanie komputera dla obliczeń grafowych

Badanie właściwości grafów często wymaga dużych mocy obliczeniowych. Przykładem wykorzystania komputerów w teorii grafów jest eksperyment przeprowadzony przez matematyków Kenneth'a Appel'a oraz Wolfgang'a Haken'a.

W 1976 roku udowodnili oni twierdzenie o czterech barwach dla grafów planarnych wykorzystując program napisany w języku Fortran.

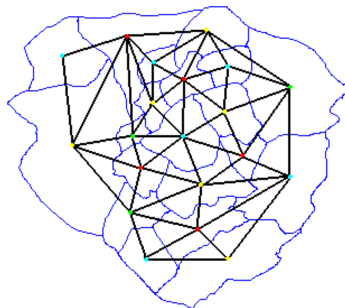


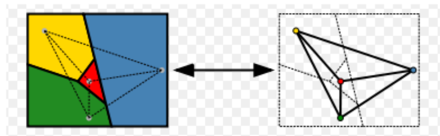


https://pl.wikipedia.org/wiki/Twierdzenie_o_czterech_barwach

Wykorzystanie komputera dla obliczeń grafowych

Badanie właściwości grafów często wymaga dużych mocy obliczeniowych. Przykładem wykorzystania komputerów w teorii grafów jest eksperyment przeprowadzony przez matematyków Kenneth'a Appel'a oraz Wolfgang'a Haken'a.
W 1976 roku udowodnili oni twierdzenie o czterech barwach dla grafów planarnych wykorzystując program napisany w języku Fortran.





https://pl.wikipedia.org/wiki/Twierdzenie_o_czterech_barwach

Trudności w obliczeniach na grafach

Wśród problemów jakie można wyróżnić przy obliczeniach na grafach są:

- znaczne rozmiary grafów, zapewnienie wystarczającej ilości pamięci potrzebnej do przechowywania i przetwarzania grafów (miliardy wierzchołków) np. graf Internetu, Facebook
- różnorodność w strukturze grafu np. grafy regularne, losowe czy sieci typu Scale Free

Trudności w obliczeniach na grafach

Wśród problemów jakie można wyróżnić przy obliczeniach na grafach są:

- znaczne rozmiary grafów, zapewnienie wystarczającej ilości pamięci potrzebnej do przechowywania i przetwarzania grafów (miliardy wierzchołków) np. graf Internetu, Facebook
- różnorodność w strukturze grafu np. grafy regularne, losowe czy sieci typu Scale Free

Trudności w obliczeniach na grafach

Wśród problemów jakie można wyróżnić przy obliczeniach na grafach są:

- znaczne rozmiary grafów, zapewnienie wystarczającej ilości pamięci potrzebnej do przechowywania i przetwarzania grafów (miliardy wierzchołków) np. graf Internetu, Facebook
- różnorodność w strukturze grafu np. grafy regularne, losowe czy sieci typu Scale Free

Trudności w obliczeniach na grafach

Obliczenia na grafach przekraczają możliwości pojedynczego komputera.

Wielkość analizowanych grafów coraz częściej wymaga zastosowania algorytmów i komputerów równoległych.

Trudności w obliczeniach na grafach

Obliczenia na grafach przekraczają możliwości pojedynczego komputera.

Wielkość analizowanych grafów coraz częściej wymaga zastosowania algorytmów i komputerów równoległych.

Trudności w obliczeniach na grafach

Przy równoległych obliczeniach ważnym problemem jest również:

- znaczna ilość komunikacji oraz synchronizacji co znacząco utrudnia efektywne zrównoleglenie
- skomplikowany schemat równoległości

Trudności w obliczeniach na grafach

Przy równoległych obliczeniach ważnym problemem jest również:

- znaczna ilość komunikacji oraz synchronizacji co znacząco utrudnia efektywne zrównoleglenie
- skomplikowany schemat równoległości

Trudności w obliczeniach na grafach

Przy równoległych obliczeniach ważnym problemem jest również:

- znaczna ilość komunikacji oraz synchronizacji co znacząco utrudnia efektywne zrównoleglenie
- skomplikowany schemat równoległości

Trudności w obliczeniach na grafach

Z punktu widzenia komputera obliczenia na grafach są:

- różne od tradycyjnych intensywnych obliczeń na liczbach zmiennoprzecinkowych mierzonych na przykład za pomocą benchmarku LINPACK
- systemy oparte o MapReduce nie zawsze są odpowiednie do przetwarzania danych w postaci ogromnych grafów

Trudności w obliczeniach na grafach

Z punktu widzenia komputera obliczenia na grafach są:

- różne od tradycyjnych intensywnych obliczeń na liczbach zmiennoprzecinkowych mierzonych na przykład za pomocą benchmarku LINPACK
- systemy oparte o MapReduce nie zawsze są odpowiednie do przetwarzania danych w postaci ogromnych grafów

Trudności w obliczeniach na grafach

Z punktu widzenia komputera obliczenia na grafach są:

- różne od tradycyjnych intensywnych obliczeń na liczbach zmiennoprzecinkowych mierzonych na przykład za pomocą benchmarku LINPACK
- systemy oparte o MapReduce nie zawsze są odpowiednie do przetwarzania danych w postaci ogromnych grafów

Motywacja badań

Stworzono wiele narzędzi dedykowanych obliczeniom na grafach.

Większość narzędzi do przetwarzania grafów bazuje na tradycyjnym języku programowania jakim jest C/C++.

Coraz więcej aplikacji tworzonych jest w Javie, dlatego wzrasta również zainteresowanie na przetwarzanie dużych grafów w języku Java.

Obecnie nie ma aplikacji w Javie do analizy dużych grafów.

Motywacja badań

Stworzono wiele narzędzi dedykowanych obliczeniom na grafach.

Większość narzędzi do przetwarzania grafów bazuje na tradycyjnym języku programowania jakim jest C/C++.

Coraz więcej aplikacji tworzonych jest w Javie, dlatego wzrasta również zainteresowanie na przetwarzanie dużych grafów w języku Java.

Obecnie nie ma aplikacji w Javie do analizy dużych grafów.

Motywacja badań

Stworzono wiele narzędzi dedykowanych obliczeniom na grafach.

Większość narzędzi do przetwarzania grafów bazuje na tradycyjnym języku programowania jakim jest C/C++.

Coraz więcej aplikacji tworzonych jest w Javie, dlatego wzrasta również zainteresowanie na przetwarzanie dużych grafów w języku Java.

Obecnie nie ma aplikacji w Javie do analizy dużych grafów.

Motywacja badań

Stworzono wiele narzędzi dedykowanych obliczeniom na grafach.

Większość narzędzi do przetwarzania grafów bazuje na tradycyjnym języku programowania jakim jest C/C++.

Coraz więcej aplikacji tworzonych jest w Javie, dlatego wzrasta również zainteresowanie na przetwarzanie dużych grafów w języku Java.

Obecnie nie ma aplikacji w Javie do analizy dużych grafów.

Tradycyjne modele HPC

Wsród tradycyjnych modeli programowania HPC możemy wyróżnić:

- model pamięci współdzielonej (wspólnej) np. OpenMP (Open Multi-Processing)
- model przesyłania komunikatów MPI (Message-Passing Interface) (procesy, każdy z oddzielną pamięcią wymieniają wiadomości)

Tradycyjne modele HPC

Wsród tradycyjnych modeli programowania HPC możemy wyróżnić:

- model pamięci współdzielonej (wspólnej) np. OpenMP (Open Multi-Processing)
- model przesyłania komunikatów MPI (Message-Passing Interface) (procesy, każdy z oddzielną pamięcią wymieniają wiadomości)

Tradycyjne modele HPC

Wsród tradycyjnych modeli programowania HPC możemy wyróżnić:

- model pamięci współdzielonej (wspólnej) np. OpenMP (Open Multi-Processing)
- model przesyłania komunikatów MPI (Message-Passing Interface) (procesy, każdy z oddzielną pamięcią wymieniają wiadomości)

Tradycyjne modele HPC - model pamięci współdzielonej

Model pamięci współdzielonej np. OpenMP

- pamięć wspólna
- komunikacja odbywa się poprzez zapisywanie i odczyt z pamięci (wiele procesów może pisać i czytać z pamięci współdzielonej)
- łatwość programowania

Tradycyjne modele HPC - model pamięci współdzielonej

Model pamięci współdzielonej np. OpenMP

- pamięć wspólna
- komunikacja odbywa się poprzez zapisywanie i odczyt z pamięci (wiele procesów może pisać i czytać z pamięci współdzielonej)
- łatwość programowania

Tradycyjne modele HPC - model pamięci współdzielonej

Model pamięci współdzielonej np. OpenMP

- pamięć wspólna
- komunikacja odbywa się poprzez zapisywanie i odczyt z pamięci (wiele procesów może pisać i czytać z pamięci współdzielonej)
- łatwość programowania

Tradycyjne modele HPC - model pamięci współdzielonej

Model pamięci współdzielonej np. OpenMP

- pamięć wspólna
- komunikacja odbywa się poprzez zapisywanie i odczyt z pamięci (wiele procesów może pisać i czytać z pamięci współdzielonej)
- łatwość programowania

Tradycyjne modele HPC - model przesyłania komunikatów

Model przesyłania komunikatów MPI
(procesy, każdy z oddzielną pamięcią wymieniają wiadomości)

- jawna komunikacja
- osobne (fizycznie) obszary pamięci dla procesów
- bardziej skalowalne i wydajniejsze

Tradycyjne modele HPC - model przesyłania komunikatów

Model przesyłania komunikatów MPI
(procesy, każdy z oddzielną pamięcią wymieniają wiadomości)

- jawna komunikacja
- osobne (fizycznie) obszary pamięci dla procesów
- bardziej skalowalne i wydajniejsze

Tradycyjne modele HPC - model przesyłania komunikatów

Model przesyłania komunikatów MPI
(procesy, każdy z oddzielną pamięcią wymieniają wiadomości)

- jawna komunikacja
- osobne (fizycznie) obszary pamięci dla procesów
- bardziej skalowalne i wydajniejsze

Tradycyjne modele HPC - model przesyłania komunikatów

Model przesyłania komunikatów MPI
(procesy, każdy z oddzielną pamięcią wymieniają wiadomości)

- jawna komunikacja
- osobne (fizycznie) obszary pamięci dla procesów
- bardziej skalowalne i wydajniejsze

Model PGAS - dlaczego został stworzony?

Rozwój architektur sprzętowych wiąże się z problemami takimi jak: NUMA (Non-Uniform Memory Access), NUCC (Non-Uniform Cluster Computing).

Rozpoczęto prace nad nowymi metodami programowania równoległego i rozproszonego.

Model PGAS (Partitioned Global Address Space)

Model PGAS

Główne założenia:

- pojedyncza przestrzeń adresowa
- dostarczane są dodatkowe mechanizmy, aby rozróżnić lokalny i zdalny dostęp do danych
- szczegóły komunikacji ukryte przed użytkownikiem
- komunikacja jednostronna

Model PGAS

Główne założenia:

- pojedyncza przestrzeń adresowa
- dostarczane są dodatkowe mechanizmy, aby rozróżnić lokalny i zdalny dostęp do danych
- szczegóły komunikacji ukryte przed użytkownikiem
- komunikacja jednostronna

Model PGAS

Główne założenia:

- pojedyncza przestrzeń adresowa
- dostarczane są dodatkowe mechanizmy, aby rozróżnić lokalny i zdalny dostęp do danych
- szczegóły komunikacji ukryte przed użytkownikiem
- komunikacja jednostronna

Model PGAS

Główne założenia:

- pojedyncza przestrzeń adresowa
- dostarczane są dodatkowe mechanizmy, aby rozróżnić lokalny i zdalny dostęp do danych
- szczegóły komunikacji ukryte przed użytkownikiem
- komunikacja jednostronna

Model PGAS

Główne założenia:

- pojedyncza przestrzeń adresowa
- dostarczane są dodatkowe mechanizmy, aby rozróżnić lokalny i zdalny dostęp do danych
- szczegóły komunikacji ukryte przed użytkownikiem
- komunikacja jednostronna

Model PGAS

Model PGAS można sklasyfikować jako model pośredni między MPI, a modelem pamięci współdzielonej

- z modelu pamięci wspólnej zakłada, że procesy działają na pojedynczej pamięci która jest współdzielona pomiędzy procesami
- z modelu przesyłania komunikatów czerpie ideę kosztu związanego z komunikacją miedzy procesami

Model PGAS

Model PGAS można sklasyfikować jako model pośredni między MPI, a modelem pamięci współdzielonej

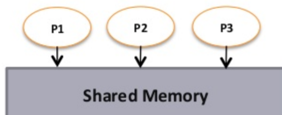
- z modelu pamięci wspólnej zakłada, że procesy działają na pojedynczej pamięci która jest współdzielona pomiędzy procesami
- z modelu przesyłania komunikatów czerpie ideę kosztu związanego z komunikacją miedzy procesami

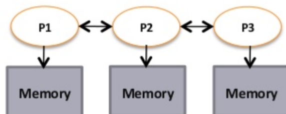
Model PGAS

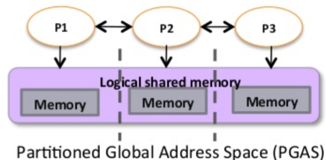
Model PGAS można sklasyfikować jako model pośredni między MPI, a modelem pamięci współdzielonej

- z modelu pamięci wspólnej zakłada, że procesy działają na pojedynczej pamięci która jest współdzielona pomiędzy procesami
- z modelu przesyłania komunikatów czerpie ideę kosztu związanego z komunikacją miedzy procesami

Schematy modeli







http://www.slideshare.net/insideHPC/programming-models-for-exascale-systems-59869854

Języki PGAS

Zgodnie z pracą *Partitioned Global Address Space Languages* języki PGAS podzielone zostały na cztery grupy:

- oryginalne języki PGAS (późne lata 1990) np. CoArray Fortran (1998) (D), Titanium (D), UPC (D)
- języki PGAS HPCS (High Productivity Computing System) (2004) np. Chapel (L), X10 (L), Fortress (L)
- retrospektywne języki PGAS (wczesne lata 1990) np. HPF (D), ZPL (L), Global Arrays (MW)
- współczesne (po 2005) np. XCalableMP (E)

L: new language
D: language dialect
E: pragma extension
MW: middleware

Języki PGAS

Zgodnie z pracą *Partitioned Global Address Space Languages* języki PGAS podzielone zostały na cztery grupy:

- oryginalne języki PGAS (późne lata 1990) np. CoArray Fortran (1998) (D), Titanium (D), UPC (D)
- języki PGAS HPCS (High Productivity Computing System) (2004) np. Chapel (L), X10 (L), Fortress (L)
- retrospektywne języki PGAS (wczesne lata 1990) np. HPF (D), ZPL (L), Global Arrays (MW)
- współczesne (po 2005) np. XCalableMP (E)

L: new language
D: language dialect
E: pragma extension
MW: middleware

Języki PGAS

Zgodnie z pracą *Partitioned Global Address Space Languages* języki PGAS podzielone zostały na cztery grupy:

- oryginalne języki PGAS (późne lata 1990) np. CoArray Fortran (1998) (D), Titanium (D), UPC (D)
- języki PGAS HPCS (High Productivity Computing System) (2004) np. Chapel (L), X10 (L), Fortress (L)
- retrospektywne języki PGAS (wczesne lata 1990) np. HPF (D), ZPL (L), Global Arrays (MW)
- współczesne (po 2005) np. XCalableMP (E)

L: new language
D: language dialect
E: pragma extension
MW: middleware

Języki PGAS

Zgodnie z pracą *Partitioned Global Address Space Languages* języki PGAS podzielone zostały na cztery grupy:

- oryginalne języki PGAS (późne lata 1990) np. CoArray Fortran (1998) (D), Titanium (D), UPC (D)
- języki PGAS HPCS (High Productivity Computing System) (2004) np. Chapel (L), X10 (L), Fortress (L)
- retrospektywne języki PGAS (wczesne lata 1990) np. HPF (D), ZPL (L), Global Arrays (MW)
- współczesne (po 2005) np. XCalableMP (E)

L: new language
D: language dialect
E: pragma extension
MW: middleware

Języki PGAS

Zgodnie z pracą *Partitioned Global Address Space Languages* języki PGAS podzielone zostały na cztery grupy:

- oryginalne języki PGAS (późne lata 1990) np. CoArray Fortran (1998) (D), Titanium (D), UPC (D)
- języki PGAS HPCS (High Productivity Computing System) (2004) np. Chapel (L), X10 (L), Fortress (L)
- retrospektywne języki PGAS (wczesne lata 1990) np. HPF (D), ZPL (L), Global Arrays (MW)
- współczesne (po 2005) np. XCalableMP (E)

L: new language
D: language dialect
E: pragma extension
MW: middleware

PCJ

Jako narzędzie do implementacji została użyta biblioteka PCJ (Parallel Computations in Java)

PCJ to biblioteka do obliczeń równoległych i rozproszonych w języku Java bazująca na modelu PGAS.

http://pcj.icm.edu.pl/

PCJ

Jako narzędzie do implementacji została użyta biblioteka PCJ (Parallel Computations in Java)

PCJ to biblioteka do obliczeń równoległych i rozproszonych w języku Java bazująca na modelu PGAS.

http://pcj.icm.edu.pl/

PCJ - dlaczego PCJ?

Dlaczego PCJ?

- biblioteka oparta o język Java
- samodzielna biblioteka w postaci jar
- niezależna od dodatkowych bibliotek (nie używa zewnętrznych bibliotek ani technologii JNI)
- jest oparta na natywnych mechanizmach Javy
  - nie wymaga specjalnego kompilatora czy innych rozszerzeń
  - używana beż żadnych modyfikacji języka Java

PCJ - dlaczego PCJ?

Dlaczego PCJ?

- biblioteka oparta o język Java
- samodzielna biblioteka w postaci jar
- niezależna od dodatkowych bibliotek (nie używa zewnętrznych bibliotek ani technologii JNI)
- jest oparta na natywnych mechanizmach Javy
  - nie wymaga specjalnego kompilatora czy innych rozszerzeń
  - używana beż żadnych modyfikacji języka Java

PCJ - dlaczego PCJ?

Dlaczego PCJ?

- biblioteka oparta o język Java
- samodzielna biblioteka w postaci jar
- niezależna od dodatkowych bibliotek (nie używa zewnętrznych bibliotek ani technologii JNI)
- jest oparta na natywnych mechanizmach Javy
  - nie wymaga specjalnego kompilatora czy innych rozszerzeń
  - używana beż żadnych modyfikacji języka Java

PCJ - dlaczego PCJ?

Dlaczego PCJ?

- biblioteka oparta o język Java
- samodzielna biblioteka w postaci jar
- niezależna od dodatkowych bibliotek (nie używa zewnętrznych bibliotek ani technologii JNI)
- jest oparta na natywnych mechanizmach Javy
  - nie wymaga specjalnego kompilatora czy innych rozszerzeń
  - używana beż żadnych modyfikacji języka Java

PCJ - dlaczego PCJ?

Dlaczego PCJ?

- biblioteka oparta o język Java
- samodzielna biblioteka w postaci jar
- niezależna od dodatkowych bibliotek (nie używa zewnętrznych bibliotek ani technologii JNI)
- jest oparta na natywnych mechanizmach Javy
  - nie wymaga specjalnego kompilatora czy innych rozszerzeń
  - używana beż żadnych modyfikacji języka Java

PCJ - dlaczego PCJ?

Dlaczego PCJ?

- PCJ jest odpowiedzialny za odpowiednie ustawienia i pozwala użytkownikowi na rozpoczęcie obliczeń na wielu węzłach przy wielu wątkach na węźle
- otwarte oprogramowanie (open source) o licencji BSD (kod źródłowy dostępny na GitHubie)
- początkowo napisana przy użyciu Java SE7, a obecnie bazuje na Java SE8

PCJ - dlaczego PCJ?

Dlaczego PCJ?

- PCJ jest odpowiedzialny za odpowiednie ustawienia i pozwala użytkownikowi na rozpoczęcie obliczeń na wielu węzłach przy wielu wątkach na węźle
- otwarte oprogramowanie (open source) o licencji BSD (kod źródłowy dostępny na GitHubie)
- początkowo napisana przy użyciu Java SE7, a obecnie bazuje na Java SE8

PCJ - dlaczego PCJ?

Dlaczego PCJ?

- PCJ jest odpowiedzialny za odpowiednie ustawienia i pozwala użytkownikowi na rozpoczęcie obliczeń na wielu węzłach przy wielu wątkach na węźle
- otwarte oprogramowanie (open source) o licencji BSD (kod źródłowy dostępny na GitHubie)
- początkowo napisana przy użyciu Java SE7, a obecnie bazuje na Java SE8

PCJ - dlaczego PCJ?

Dlaczego PCJ?

- PCJ jest odpowiedzialny za odpowiednie ustawienia i pozwala użytkownikowi na rozpoczęcie obliczeń na wielu węzłach przy wielu wątkach na węźle
- otwarte oprogramowanie (open source) o licencji BSD (kod źródłowy dostępny na GitHubie)
- początkowo napisana przy użyciu Java SE7, a obecnie bazuje na Java SE8

PCJ - komunikacja

Typowa aplikacja PCJ nie ogranicza się do pojedynczej JVM. Komunikacja jest realizowana przez:

- Java Concurrency
- komunikacja sieciowa poprzez gniazda

PCJ - główne cechy

Główne cechy biblioteki PCJ:

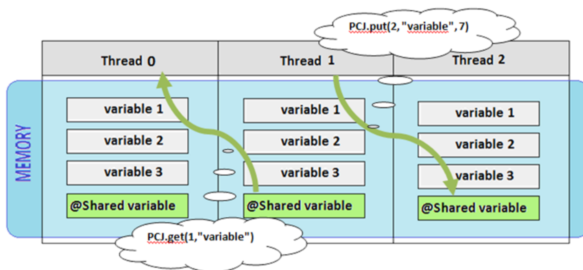
- program składa się z jednakowych wątków, które prowadzą obliczenia na heterogenicznych danych (SPMD)

```
public interface StartPoint {
    public void main();
}
```

PCJ - główne cechy

Główne cechy biblioteki PCJ:

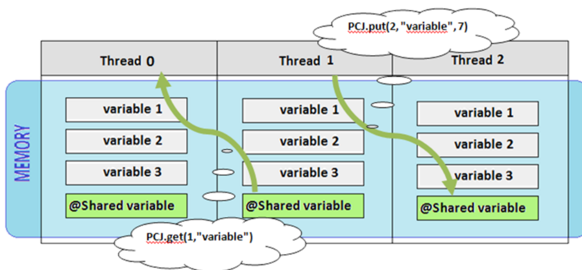
- rozproszony system pamięci jest widziany jako globalna przestrzeń adresowa



PCJ - główne cechy

Główne cechy biblioteki PCJ:

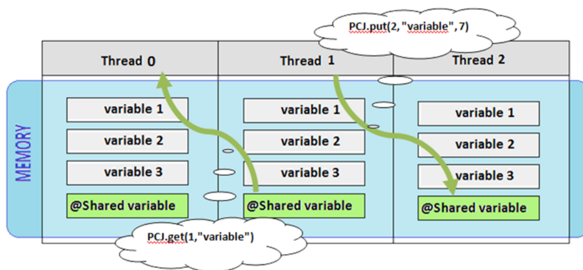
- ukrycie szczegółów komunikacyjnych np. obsługa wątków, programowanie sieciowe



PCJ - główne cechy

Główne cechy biblioteki PCJ:

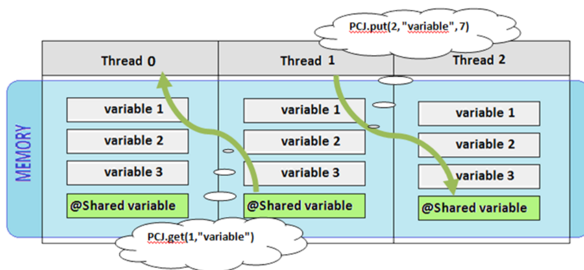
- każdy wątek wykonuje swoje własne obliczenia i ma dostęp do swojej lokalnej pamięci



PCJ - główne cechy

Główne cechy biblioteki PCJ:

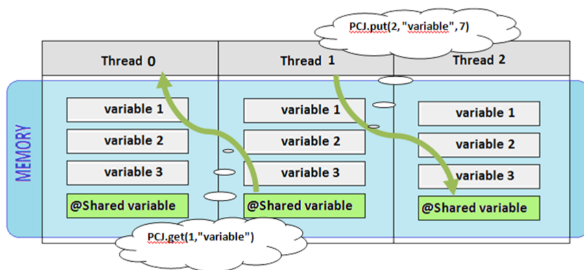
- domyślnie zmienne przechowywane są w pamięci lokalnej dla wątków wykonania programu



PCJ - główne cechy

Główne cechy biblioteki PCJ:

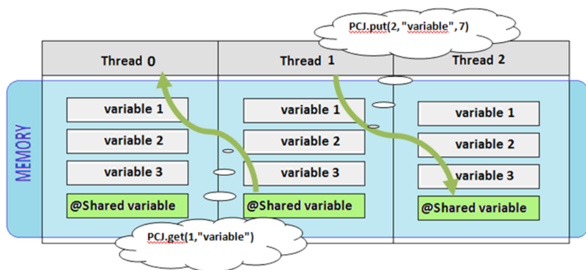
- zmienne oznaczone adnotacją @Shared stają się dostępne dla innych wątków, mogą być współdzielone między wątkami



PCJ - główne cechy

Główne cechy biblioteki PCJ:

- bazuje na komunikacji jednostronnej



PCJ - API

PCJ dostarcza metod potrzebnych do tworzenia równoległych i rozproszonych aplikacji:

- jednostronna komunikacja za pomocą put oraz get

```
PCJ.put(int taskId, String variableName, Object variableData, int indices...)

PCJ.get(int taskId, String variableName, int indices...)
```

PCJ - API

- PCJ.broadcast(String variable, Object newValue)
- PCJ.barrier()
- int PCJ.waitFor(String variable)
- tworzenie grup wątków

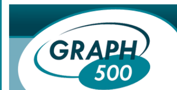
Graph500

Benchamrk Graph500
został stworzony w celu przeprowadzenia oceny skalownalności klastrów komputerów w kontekście intensywnej wymiany danych.

Benchmark wymusza znaczną ilośc komunikacji i synchronizacji oraz znaczne zapotrzebowanie i ociążenie pamięci do przechowywania oraz procesowania grafu.

Top 10
(June 2016)

Rank	Machine
1	K computer - RIKEN Advanced Institute for Computational Science (AICS) (82944 nodes, 663552 cores)
2	Sunway TaihuLight - National Supercomputing Center in Wuxi (40768 nodes, 10599680 cores)
3	DOE/NNSA/LLNL Sequoia - Lawrence Livermore National Laboratory (98304 nodes, 1572864 cores)



Graph500 jest konkurencyjnym do Top500 sposobem badania mocy obliczeniowej komputerow. Zamiast mierzyć moc potrzebną do obliczeń zmiennoprzecinkowych bada wydajność maszyny przy przetwarzaniu grafów.

http://www.graph500.org/

Graph500

Benchamrk Graph500
został stworzony w celu przeprowadzenia oceny skalownalności klastrów komputerów w kontekście intensywnej wymiany danych.

Benchmark wymusza znaczną ilośc komunikacji i synchronizacji oraz znaczne zapotrzebowanie i ociążenie pamięci do przechowywania oraz procesowania grafu.

Top 10
(June 2016)

Rank	Machine
1	K computer - RIKEN Advanced Institute for Computational Science (AICS) (82944 nodes, 663552 cores)
2	Sunway TaihuLight - National Supercomputing Center in Wuxi (40768 nodes, 10599680 cores)
3	DOE/NNSA/LLNL Sequoia - Lawrence Livermore National Laboratory (98304 nodes, 1572864 cores)

Graph500 jest konkurencyjnym do Top500 sposobem badania mocy obliczeniowej komputerow. Zamiast mierzyć moc potrzebną do obliczeń zmiennoprzecinkowych bada wydajność maszyny przy przetwarzaniu grafów.

http://www.graph500.org/

Graph500

Benchamrk Graph500
został stworzony w celu przeprowadzenia oceny skalownalności klastrów komputerów w kontekście intensywnej wymiany danych.

Benchmark wymusza znaczną ilośc komunikacji i synchronizacji oraz znaczne zapotrzebowanie i ociążenie pamięci do przechowywania oraz procesowania grafu.

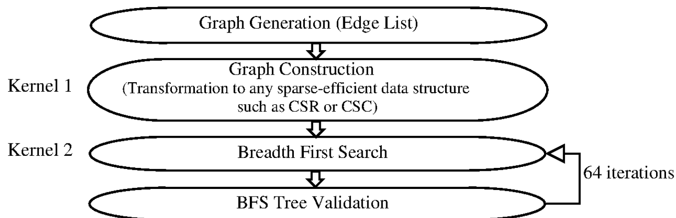
Top 10
(June 2016)

Rank	Machine
1	K computer - RIKEN Advanced Institute for Computational Science (AICS) (82944 nodes, 663552 cores)
2	Sunway TaihuLight - National Supercomputing Center in Wuxi (40768 nodes, 10599680 cores)
3	DOE/NNSA/LLNL Sequoia - Lawrence Livermore National Laboratory (98304 nodes, 1572864 cores)

Graph500 jest konkurencyjnym do Top500 sposobem badania mocy obliczeniowej komputerow. Zamiast mierzyć moc potrzebną do obliczeń zmiennoprzecinkowych bada wydajność maszyny przy przetwarzaniu grafów.

http://www.graph500.org/

Graph500 - schemat

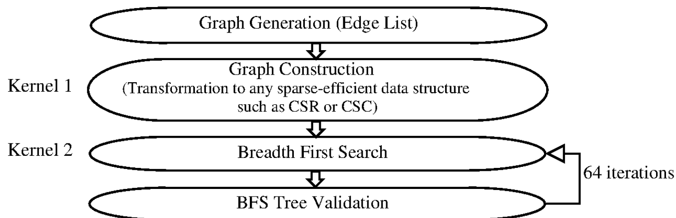


W ocenie wydajności pod uwagę brane są kernele 1 oraz 2.

Gdy obydwa kernele kończą pracę następuje faza walidacji w celu sprawdzenia czy wynik BFS jest poprawny.
Jest 64 iteracji kernala 2 razem z testem walidacji.

Żadne dalsze modyfikacje pomiędzy obliczeniami kernelów nie są dozwolone.

Graph500 - schemat

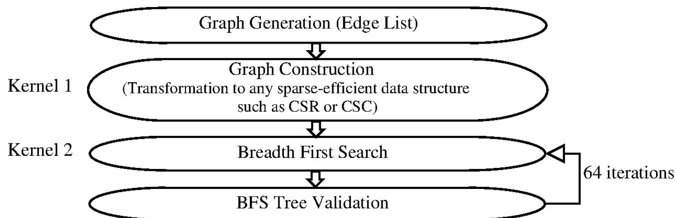


W ocenie wydajności pod uwagę brane są kernele 1 oraz 2.

Gdy obydwa kernele kończą pracę następuje faza walidacji w celu sprawdzenia czy wynik BFS jest poprawny.
Jest 64 iteracji kernala 2 razem z testem walidacji.

Żadne dalsze modyfikacje pomiędzy obliczeniami kernelów nie są dozwolone.

Graph500 - schemat

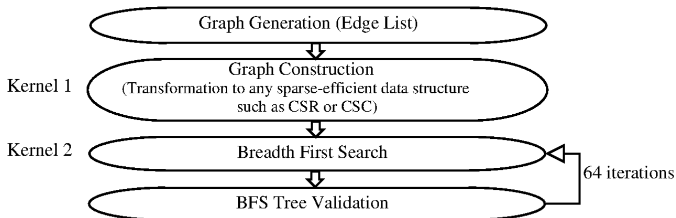


W ocenie wydajności pod uwagę brane są kernele 1 oraz 2.

Gdy obydwa kernele kończą pracę następuje faza walidacji w celu sprawdzenia czy wynik BFS jest poprawny.
Jest 64 iteracji kernala 2 razem z testem walidacji.

Żadne dalsze modyfikacje pomiędzy obliczeniami kernelów nie są dozwolone.

Graph500 - schemat

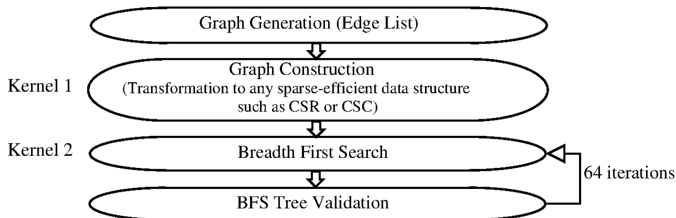


W ocenie wydajności pod uwagę brane są kernele 1 oraz 2.

Gdy obydwa kernele kończą pracę następuje faza walidacji w celu sprawdzenia czy wynik BFS jest poprawny.
Jest 64 iteracji kernala 2 razem z testem walidacji.

Żadne dalsze modyfikacje pomiędzy obliczeniami kernelów nie są dozwolone.

Graph500 - schemat



W ocenie wydajności pod uwagę brane są kernele 1 oraz 2.

Gdy obydwa kernele kończą pracę następuje faza walidacji w celu sprawdzenia czy wynik BFS jest poprawny.
Jest 64 iteracji kernala 2 razem z testem walidacji.

Żadne dalsze modyfikacje pomiędzy obliczeniami kernelów nie są dozwolone.

Graph500 - Generator

Benchmark Graph500 zawiera generator grafów, który konstuuje graf w postaci listy krawędzi.

Każda krawędź jest nieskierowana z punktami końcowymi krawędzi określonymi jako <wierzchołek startowy, wierzchołek końcowy>. Wierzchołki mają identyfikatory liczone od 0.

Generator pozwala zapisać wygenerowany graf do pliku binarnego - każda krawędź zajmuje 16 bajtów danych (8 bajtów na wierzchołek - bez znaku)

Graph500 - Generator

Benchmark Graph500 zawiera generator grafów, który konstuuje graf w postaci listy krawędzi.

Każda krawędź jest nieskierowana z punktami końcowymi krawędzi określonymi jako <wierzchołek startowy, wierzchołek końcowy>. Wierzchołki mają identyfikatory liczone od 0.

Generator pozwala zapisać wygenerowany graf do pliku binarnego - każda krawędź zajmuje 16 bajtów danych (8 bajtów na wierzchołek - bez znaku)

Graph500 - Generator

Benchmark Graph500 zawiera generator grafów, który konstuuje graf w postaci listy krawędzi.

Każda krawędź jest nieskierowana z punktami końcowymi krawędzi określonymi jako <wierzchołek startowy, wierzchołek końcowy>. Wierzchołki mają identyfikatory liczone od 0.

Generator pozwala zapisać wygenerowany graf do pliku binarnego - każda krawędź zajmuje 16 bajtów danych (8 bajtów na wierzchołek - bez znaku)

Graph500 - Generator

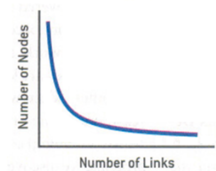
Benchmark Graph500 zawiera generator grafów, który konstuuje graf w postaci listy krawędzi.

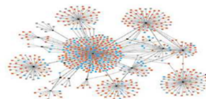
Każda krawędź jest nieskierowana z punktami końcowymi krawędzi określonymi jako <wierzchołek startowy, wierzchołek końcowy>. Wierzchołki mają identyfikatory liczone od 0.

Generator pozwala zapisać wygenerowany graf do pliku binarnego - każda krawędź zajmuje 16 bajtów danych (8 bajtów na wierzchołek - bez znaku)

Graph500 - Generator

Generator wykorzystuje model grafów Kroneckera - rzadkie sieci z małą średnicą, typu Scale Free.
Są to sieci które dobrze symulują grafy świata rzeczywistego.



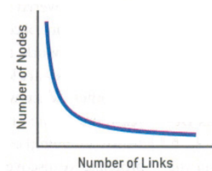


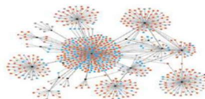
Prawdopodobieństwo wystąpienia połączenia pomiędzy nowym węzłem a każdym węzłem należącym do sieci bardzo silnie zależy od posiadanej przez te węzły liczby krawędzi k i wynosi $P(k) \sim k^{-c}$.
Wykładnik potęgi zależy od rodzaju rozpatrywanej sieci np. dla WWW około 2.2.
Najwięcej nowych węzłów łączy się z węzłami, które już posiadają najwięcej sąsiadów.

http://www.slideshare.net/networksuw/random-graph-models-9968852
http://humannaturelab.net/resources/images/

Graph500 - Generator

Generator wykorzystuje model grafów Kroneckera - rzadkie sieci z małą średnicą, typu Scale Free.
Są to sieci które dobrze symulują grafy świata rzeczywistego.



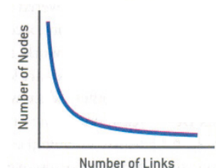


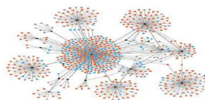
Prawdopodobieństwo wystąpienia połączenia pomiędzy nowym węzłem a każdym węzłem należącym do sieci bardzo silnie zależy od posiadanej przez te węzły liczby krawędzi k i wynosi $P(k) \sim k^{-c}$.
Wykładnik potęgi zależy od rodzaju rozpatrywanej sieci np. dla WWW około 2.2.
Najwięcej nowych węzłów łączy się z węzłami, które już posiadają najwięcej sąsiadów.

http://www.slideshare.net/networksuw/random-graph-models-9968852
http://humannaturelab.net/resources/images/

Graph500 - Generator

Generator wykorzystuje model grafów Kroneckera - rzadkie sieci z małą średnicą, typu Scale Free.
Są to sieci które dobrze symulują grafy świata rzeczywistego.



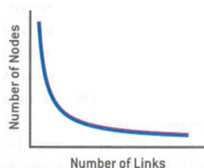


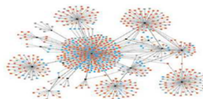
Prawdopodobieństwo wystąpienia połączenia pomiędzy nowym węzłem a każdym węzłem należącym do sieci bardzo silnie zależy od posiadanej przez te węzły liczby krawędzi k i wynosi $P(k) \sim k^{-c}$.
Wykładnik potęgi zależy od rodzaju rozpatrywanej sieci np. dla WWW około 2.2.
Najwięcej nowych węzłów łączy się z węzłami, które już posiadają najwięcej sąsiadów.

http://www.slideshare.net/networksuw/random-graph-models-9968852
http://humannaturelab.net/resources/images/

Graph500 - Generator

Generator wykorzystuje model grafów Kroneckera - rzadkie sieci z małą średnicą, typu Scale Free.
Są to sieci które dobrze symulują grafy świata rzeczywistego.





Prawdopodobieństwo wystąpienia połączenia pomiędzy nowym węzłem a każdym węzłem należącym do sieci bardzo silnie zależy od posiadanej przez te węzły liczby krawędzi k i wynosi $P(k) \sim k^{-c}$.
Wykładnik potęgi zależy od rodzaju rozpatrywanej sieci np. dla WWW około 2.2.
Najwięcej nowych węzłów łączy się z węzłami, które już posiadają najwięcej sąsiadów.

http://www.slideshare.net/networksuw/random-graph-models-9968852
http://humannaturelab.net/resources/images/

Graph500 - Kernel 1

Pierwszy kernel:

- odpowiada za konstrukcję z listy krawędzi, nieskierowanego grafu w dowolnym formacie używanym przez kolejny kernel.

W obliczeniach dane są jedynie lista krawędzi oraz wielkość tej listy. Pozostałe informacje takie jak liczba wierzchołków muszą być obliczone w trakcie działania programu.

Pierwszy kernel polega na budowie z listy krawędzi dowolnej reprezentacji grafu nieskierowanego, która lepiej nadaje się do przechowywania grafów rzadkich. Specyfikacja benchmarku Graph500 nie narzuca konkretnego formatu.

Graph500 - Kernel 1

Pierwszy kernel:

- odpowiada za konstrukcję z listy krawędzi, nieskierowanego grafu w dowolnym formacie używanym przez kolejny kernel.

W obliczeniach dane są jedynie lista krawędzi oraz wielkość tej listy. Pozostałe informacje takie jak liczba wierzchołków muszą być obliczone w trakcie działania programu.

Pierwszy kernel polega na budowie z listy krawędzi dowolnej reprezentacji grafu nieskierowanego, która lepiej nadaje się do przechowywania grafów rzadkich. Specyfikacja benchmarku Graph500 nie narzuca konkretnego formatu.

Graph500 - Kernel 1

Pierwszy kernel:

- odpowiada za konstrukcję z listy krawędzi, nieskierowanego grafu w dowolnym formacie używanym przez kolejny kernel.

W obliczeniach dane są jedynie lista krawędzi oraz wielkość tej listy. Pozostałe informacje takie jak liczba wierzchołków muszą być obliczone w trakcie działania programu.

Pierwszy kernel polega na budowie z listy krawędzi dowolnej reprezentacji grafu nieskierowanego, która lepiej nadaje się do przechowywania grafów rzadkich. Specyfikacja benchmarku Graph500 nie narzuca konkretnego formatu.

Graph500 - Kernel 2

Drugi kernel:

- odpowiada za przeszukiwanie grafu wszerz.

Obliczenia wykonywane są 64 razy, jedno po drugim.

Każdy przebieg rozpoczyna się od innego, wybranego losowo werzchołka startowego (źródła).

Źródłowy wierzchołek jest losowo wybrany z wierzchołków grafu.

Aby uniknąć trywialnych przeszukiwań, tylko wierzchołki ze stopniem co najmniej 1 brane są pod uwagę (nie licząc pętli).

Graph500 - Kernel 2

Drugi kernel:

- odpowiada za przeszukiwanie grafu wszerz.

Obliczenia wykonywane są 64 razy, jedno po drugim.

Każdy przebieg rozpoczyna się od innego, wybranego losowo werzchołka startowego (źródła).

Źródłowy wierzchołek jest losowo wybrany z wierzchołków grafu.

Aby uniknąć trywialnych przeszukiwań, tylko wierzchołki ze stopniem co najmniej 1 brane są pod uwagę (nie licząc pętli).

Graph500 - Kernel 2

Drugi kernel:

- odpowiada za przeszukiwanie grafu wszerz.

Obliczenia wykonywane są 64 razy, jedno po drugim.

Każdy przebieg rozpoczyna się od innego, wybranego losowo werzchołka startowego (źródła).

Źródłowy wierzchołek jest losowo wybrany z wierzchołków grafu.

Aby uniknąć trywialnych przeszukiwań, tylko wierzchołki ze stopniem co najmniej 1 brane są pod uwagę (nie licząc pętli).

Graph500 - Kernel 2

Drugi kernel:

- odpowiada za przeszukiwanie grafu wszerz.

Obliczenia wykonywane są 64 razy, jedno po drugim.

Każdy przebieg rozpoczyna się od innego, wybranego losowo werzchołka startowego (źródła).

Źródłowy wierzchołek jest losowo wybrany z wierzchołków grafu.

Aby uniknąć trywialnych przeszukiwań, tylko wierzchołki ze stopniem co najmniej 1 brane są pod uwagę (nie licząc pętli).

Graph500 - Kernel 2

Drugi kernel:

- odpowiada za przeszukiwanie grafu wszerz.

Obliczenia wykonywane są 64 razy, jedno po drugim.

Każdy przebieg rozpoczyna się od innego, wybranego losowo werzchołka startowego (źródła).

Źródłowy wierzchołek jest losowo wybrany z wierzchołków grafu.

Aby uniknąć trywialnych przeszukiwań, tylko wierzchołki ze stopniem co najmniej 1 brane są pod uwagę (nie licząc pętli).

Graph500 - Walidacja Kernela 2

Aby zweryfikować poprawność wyniku przeprowadzone są następujące testy:

- drzewo BFS nie zawiera cykli
- każda krawędź łączy wierzchołki, których odległość od źródła różni się o 1
- każda krawędź w wejściowej liście ma wierzchołki o odległościach które różnią się o najwyżej 1 albo oba nie są w drzewie BFS
- drzewo BFS rozpina całą spójną składową
- wierzchołek i jego rodzic są połączone krawędzią w początkowym grafie

Graph500 - Walidacja Kernela 2

Aby zweryfikować poprawność wyniku przeprowadzone są następujące testy:

- drzewo BFS nie zawiera cykli
- każda krawędź łączy wierzchołki, których odległość od źródła różni się o 1
- każda krawędź w wejściowej liście ma wierzchołki o odległościach które różnią się o najwyżej 1 albo oba nie są w drzewie BFS
- drzewo BFS rozpina całą spójną składową
- wierzchołek i jego rodzic są połączone krawędzią w początkowym grafie

Graph500 - Walidacja Kernela 2

Aby zweryfikować poprawność wyniku przeprowadzone są następujące testy:

- drzewo BFS nie zawiera cykli
- każda krawędź łączy wierzchołki, których odległość od źródła różni się o 1
- każda krawędź w wejściowej liście ma wierzchołki o odległościach które różnią się o najwyżej 1 albo oba nie są w drzewie BFS
- drzewo BFS rozpina całą spójną składową
- wierzchołek i jego rodzic są połączone krawędzią w początkowym grafie

Graph500 - Walidacja Kernela 2

Aby zweryfikować poprawność wyniku przeprowadzone są następujące testy:

- drzewo BFS nie zawiera cykli
- każda krawędź łączy wierzchołki, których odległość od źródła różni się o 1
- każda krawędź w wejściowej liście ma wierzchołki o odległościach które różnią się o najwyżej 1 albo oba nie są w drzewie BFS
- drzewo BFS rozpina całą spójną składową
- wierzchołek i jego rodzic są połączone krawędzią w początkowym grafie

Graph500 - Walidacja Kernela 2

Aby zweryfikować poprawność wyniku przeprowadzone są następujące testy:

- drzewo BFS nie zawiera cykli
- każda krawędź łączy wierzchołki, których odległość od źródła różni się o 1
- każda krawędź w wejściowej liście ma wierzchołki o odległościach które różnią się o najwyżej 1 albo oba nie są w drzewie BFS
- drzewo BFS rozpina całą spójną składową
- wierzchołek i jego rodzic są połączone krawędzią w początkowym grafie

Graph500 - Walidacja Kernela 2

Aby zweryfikować poprawność wyniku przeprowadzone są następujące testy:

- drzewo BFS nie zawiera cykli
- każda krawędź łączy wierzchołki, których odległość od źródła różni się o 1
- każda krawędź w wejściowej liście ma wierzchołki o odległościach które różnią się o najwyżej 1 albo oba nie są w drzewie BFS
- drzewo BFS rozpina całą spójną składową
- wierzchołek i jego rodzic są połączone krawędzią w początkowym grafie

Graph500 - implementacje

Benchmark Graph500 dostarcza kilka implementacji:

- GNU Octave
- sekwencyjne
- OpenMP
- Cray XMT
- kilka implementacji wykorzystujących MPI

Graph500 - wielkość grafu

Wielkość grafu opisana jest za pomoca następujących zmiennych:

- SCALE - logarytm o podstawie 2 z ilości wierzchołków
- edgefactor - stosunek liczby krawędzi grafu do liczby wierzchołków

Zmienne SCALE oraz edgefactor determinują wielkość grafu:

- liczba wierzchołków $N = 2^{SCALE}$
- liczba krawędzi $M = edgefactor \cdot N$

Graph500 - wielkość grafu

Wielkość grafu opisana jest za pomoca następujących zmiennych:

- SCALE - logarytm o podstawie 2 z ilości wierzchołków
- edgefactor - stosunek liczby krawędzi grafu do liczby wierzchołków

Zmienne SCALE oraz edgefactor determinują wielkość grafu:

- liczba wierzchołków $N = 2^{SCALE}$
- liczba krawędzi $M = edgefactor \cdot N$

Graph500 - Generator - wielkości plików

Wielkości wygenerowanych plików:

SCALE	Rozmiar pliku w GB
24	4
25	8
26	16
27	32
28	64
29	128
30	256

Graph500 w PGAS i PCJ - Kernel 1

Kernel 1:

Dane: graf G w postaci listy krawędzi (graf nieskierowany)

Wynik: graf G w postaci CSR (Compressed Sparse Row)

Graph500 w PGAS i PCJ - Kernel 1 - CSR

CSR (Compressed Sparse Row) jest efektywną reprezentacją grafów rzadkich, gdzie graf jest przechowywany w trzech jednowymiarowych tablicach:

- pierwsza tablica przechowuje przesunięcie (offset) które wskazuje na wierzchołek początkowy krawędzi (tablica wierzchołków)
- druga tablica przechowuje numery kolumn wszystkich niezerowych elementów dla macierzy sąsiedztwa czytając wiersze w kolejności od góry do dołu - punkty końcowe krawędzi (tablica krawędzi)
- trzecia tablica przechowuje wartości numeryczne

Graph500 w PGAS i PCJ - Kernel 1 - CSR

CSR (Compressed Sparse Row) jest efektywną reprezentacją grafów rzadkich, gdzie graf jest przechowywany w trzech jednowymiarowych tablicach:

- pierwsza tablica przechowuje przesunięcie (offset) które wskazuje na wierzchołek początkowy krawędzi (tablica wierzchołków)
- druga tablica przechowuje numery kolumn wszystkich niezerowych elementów dla macierzy sąsiedztwa czytając wiersze w kolejności od góry do dołu - punkty końcowe krawędzi (tablica krawędzi)
- trzecia tablica przechowuje wartości numeryczne

Graph500 w PGAS i PCJ - Kernel 1 - CSR

CSR (Compressed Sparse Row) jest efektywną reprezentacją grafów rzadkich, gdzie graf jest przechowywany w trzech jednowymiarowych tablicach:

- pierwsza tablica przechowuje przesunięcie (offset) które wskazuje na wierzchołek początkowy krawędzi (tablica wierzchołków)
- druga tablica przechowuje numery kolumn wszystkich niezerowych elementów dla macierzy sąsiedztwa czytając wiersze w kolejności od góry do dołu - punkty końcowe krawędzi (tablica krawędzi)
- trzecia tablica przechowuje wartości numeryczne

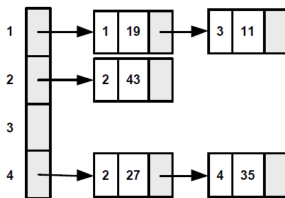
Graph500 w PGAS i PCJ - Kernel 1 - CSR

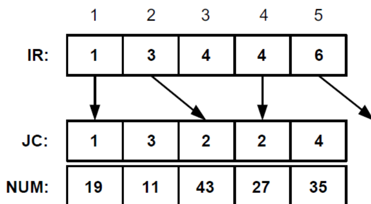
CSR (Compressed Sparse Row) jest efektywną reprezentacją grafów rzadkich, gdzie graf jest przechowywany w trzech jednowymiarowych tablicach:

- pierwsza tablica przechowuje przesunięcie (offset) które wskazuje na wierzchołek początkowy krawędzi (tablica wierzchołków)
- druga tablica przechowuje numery kolumn wszystkich niezerowych elementów dla macierzy sąsiedztwa czytając wiersze w kolejności od góry do dołu - punkty końcowe krawędzi (tablica krawędzi)
- trzecia tablica przechowuje wartości numeryczne

Graph500 w PGAS i PCJ - Kernel 1 - CSR

$$A = \begin{pmatrix} 19 & 0 & 11 & 0 \\ 0 & 43 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 27 & 0 & 35 \end{pmatrix}$$





Przykład zaczerpnięty z książki *Graph Algorithms in the Language of Linear Algebra*

Graph500 w PGAS i PCJ - Kernel 1 - CSR

CSR jest prawie identyczna z reprezentacją w postaci listy krawędzi.

W praktyce CSR:

- ma mniej narzutu pamięciowego

  Strzałki w reprezentacji listy sąsiedztwa są wskaźnikami na miejsca w pamięci (w CSR już nie).

- dużo lepszą efektywność buforowania

  Zamiast przechowywać tablicę list, składa się z tablic które przechowują całe wiersze w ciągłej porcji pamięci. Wierzchołki sąsiednie do v_i są zaraz obok wierzchołków sąsiednich wierzchołka $v_i + 1$. Tablica wierzchołków przechowuje początkowy punkt każdego ciągłego sąsiedniego bloku wierzchołka.

Graph500 w PGAS i PCJ - Kernel 1 - CSR

CSR jest prawie identyczna z reprezentacją w postaci listy krawędzi.

W praktyce CSR:

- ma mniej narzutu pamięciowego

  Strzałki w reprezentacji listy sąsiedztwa są wskaźnikami na miejsca w pamięci (w CSR już nie).

- dużo lepszą efektywność buforowania

  Zamiast przechowywać tablicę list, składa się z tablic które przechowują całe wiersze w ciągłej porcji pamięci. Wierzchołki sąsiednie do v_i są zaraz obok wierzchołków sąsiednich wierzchołka $v_i + 1$. Tablica wierzchołków przechowuje początkowy punkt każdego ciągłego sąsiedniego bloku wierzchołka.

Graph500 w PGAS i PCJ - Kernel 1 - CSR

CSR jest prawie identyczna z reprezentacją w postaci listy krawędzi.

W praktyce CSR:

- ma mniej narzutu pamięciowego

  Strzałki w reprezentacji listy sąsiedztwa są wskaźnikami na miejsca w pamięci (w CSR już nie).

- dużo lepszą efektywność buforowania

  Zamiast przechowywać tablicę list, składa się z tablic które przechowują całe wiersze w ciągłej porcji pamięci. Wierzchołki sąsiednie do v_i są zaraz obok wierzchołków sąsiednich wierzchołka $v_i + 1$. Tablica wierzchołków przechowuje początkowy punkt każdego ciągłego sąsiedniego bloku wierzchołka.

Graph500 w PGAS i PCJ - Kernel 1 - CSR

CSR jest prawie identyczna z reprezentacją w postaci listy krawędzi.

W praktyce CSR:

- ma mniej narzutu pamięciowego

  Strzałki w reprezentacji listy sąsiedztwa są wskaźnikami na miejsca w pamięci (w CSR już nie).

- dużo lepszą efektywność buforowania

  Zamiast przechowywać tablicę list, składa się z tablic które przechowują całe wiersze w ciągłej porcji pamięci. Wierzchołki sąsiednie do v_i są zaraz obok wierzchołków sąsiednich wierzchołka $v_i + 1$. Tablica wierzchołków przechowuje początkowy punkt każdego ciągłego sąsiedniego bloku wierzchołka.

Graph500 w PGAS i PCJ - Kernel 1 - CSR

CSR jest prawie identyczna z reprezentacją w postaci listy krawędzi.

W praktyce CSR:

- ma mniej narzutu pamięciowego

  Strzałki w reprezentacji listy sąsiedztwa są wskaźnikami na miejsca w pamięci (w CSR już nie).

- dużo lepszą efektywność buforowania

  Zamiast przechowywać tablicę list, składa się z tablic które przechowują całe wiersze w ciągłej porcji pamięci. Wierzchołki sąsiednie do v_i są zaraz obok wierzchołków sąsiednich wierzchołka $v_i + 1$. Tablica wierzchołków przechowuje początkowy punkt każdego ciągłego sąsiedniego bloku wierzchołka.

Graph500 w PGAS i PCJ - Kernel 1 - CSR

W eksperymencie przeprowadzonym w 1998 roku reprezentacja oparta o tablice okazała się 10 razy szybsza do przeszukania niż reprezentacja listy sąsiedztwa (spowodowane wysokim kosztem śledzenia wskaźników w połączonych strukturach danych).

CSR ma też wady, które polegają na nieefektywnych operacjach dodawania lub usuwania krawędzi.

CSR nadaje się do przechowywania grafów statycznych.

J.R. Black, C.U. Martel and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Algorithm Engineering*, 37-48, 1998.

Graph500 w PGAS i PCJ - Kernel 1 - CSR

W eksperymencie przeprowadzonym w 1998 roku reprezentacja oparta o tablice okazała się 10 razy szybsza do przeszukania niż reprezentacja listy sąsiedztwa (spowodowane wysokim kosztem śledzenia wskaźników w połączonych strukturach danych).

CSR ma też wady, które polegają na nieefektywnych operacjach dodawania lub usuwania krawędzi.

CSR nadaje się do przechowywania grafów statycznych.

J.R. Black, C.U. Martel and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Algorithm Engineering*, 37-48, 1998.

Graph500 w PGAS i PCJ - Kernel 1 - CSR

W eksperymencie przeprowadzonym w 1998 roku reprezentacja oparta o tablice okazała się 10 razy szybsza do przeszukania niż reprezentacja listy sąsiedztwa (spowodowane wysokim kosztem śledzenia wskaźników w połączonych strukturach danych).

CSR ma też wady, które polegają na nieefektywnych operacjach dodawania lub usuwania krawędzi.

CSR nadaje się do przechowywania grafów statycznych.

J.R. Black, C.U. Martel and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Algorithm Engineering*, 37-48, 1998.

Graph500 w PGAS i PCJ - Kernel 1 - CSR

W eksperymencie przeprowadzonym w 1998 roku reprezentacja oparta o tablice okazała się 10 razy szybsza do przeszukania niż reprezentacja listy sąsiedztwa (spowodowane wysokim kosztem śledzenia wskaźników w połączonych strukturach danych).

CSR ma też wady, które polegają na nieefektywnych operacjach dodawania lub usuwania krawędzi.

CSR nadaje się do przechowywania grafów statycznych.

J.R. Black, C.U. Martel and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Algorithm Engineering*, 37-48, 1998.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- graf w postaci CSR (przechowywany w dwóch wektorach)
  Ponieważ graf jest statyczny i nie zmienia się podczas obliczeń, bedziemy tworzyć graf w postaci CSR.
- rozmieszczenie wierzchołków i krawędzi grafu w systemie rozproszonym jest realizowane przez 1-wymiarowy podział danych (1D partitioning)

1-wymiarowy podział danych można łatwo zwizualizować przez 1-wymiarowy podział macierzy sąsiedztwa grafu.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- graf w postaci CSR (przechowywany w dwóch wektorach)
  Ponieważ graf jest statyczny i nie zmienia się podczas obliczeń, bedziemy tworzyć graf w postaci CSR.
- rozmieszczenie wierzchołków i krawędzi grafu w systemie rozproszonym jest realizowane przez 1-wymiarowy podział danych (1D partitioning)

1-wymiarowy podział danych można łatwo zwizualizować przez 1-wymiarowy podział macierzy sąsiedztwa grafu.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- graf w postaci CSR (przechowywany w dwóch wektorach)
  Ponieważ graf jest statyczny i nie zmienia się podczas obliczeń, bedziemy tworzyć graf w postaci CSR.
- rozmieszczenie wierzchołków i krawędzi grafu w systemie rozproszonym jest realizowane przez 1-wymiarowy podział danych (1D partitioning)

1-wymiarowy podział danych można łatwo zwizualizować przez 1-wymiarowy podział macierzy sąsiedztwa grafu.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- graf w postaci CSR (przechowywany w dwóch wektorach)
  Ponieważ graf jest statyczny i nie zmienia się podczas obliczeń, bedziemy tworzyć graf w postaci CSR.
- rozmieszczenie wierzchołków i krawędzi grafu w systemie rozproszonym jest realizowane przez 1-wymiarowy podział danych (1D partitioning)

1-wymiarowy podział danych można łatwo zwizualizować przez 1-wymiarowy podział macierzy sąsiedztwa grafu.

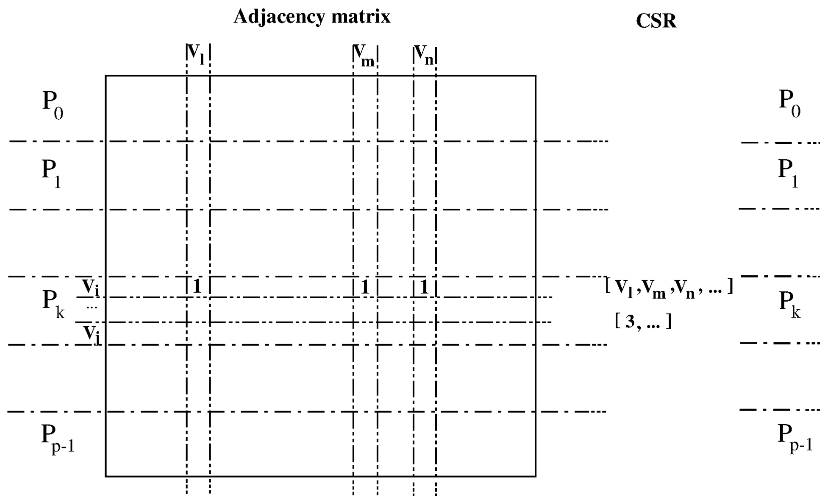
Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- graf w postaci CSR (przechowywany w dwóch wektorach)
  Ponieważ graf jest statyczny i nie zmienia się podczas obliczeń, bedziemy tworzyć graf w postaci CSR.
- rozmieszczenie wierzchołków i krawędzi grafu w systemie rozproszonym jest realizowane przez 1-wymiarowy podział danych (1D partitioning)

1-wymiarowy podział danych można łatwo zwizualizować przez 1-wymiarowy podział macierzy sąsiedztwa grafu.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm



Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- dystrybucja wierzchołków pomiędzy wątki jest realizowana blokowo - wątek 0 posiada wierzchołki $< 0, X >$, wątek 1 posiada wierzchołki $< X + 1, Y >$ itd.
- wszystkie wierzchołki i krawędzie początkowego grafu są rozdzielone - każdy wątek posiada N/p wierzchołków wraz z sąsiadującymi krawędziami

N - liczba wierzchołków w grafie
p - liczba wątków

W przypadku grafów nieskierowanych wszystkie krawędzie są przechowywane dwa razy.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- dystrybucja wierzchołków pomiędzy wątki jest realizowana blokowo - wątek 0 posiada wierzchołki $< 0, X >$, wątek 1 posiada wierzchołki $< X + 1, Y >$ itd.
- wszystkie wierzchołki i krawędzie początkowego grafu są rozdzielone - każdy wątek posiada N/p wierzchołków wraz z sąsiadującymi krawędziami

N - liczba wierzchołków w grafie
p - liczba wątków

W przypadku grafów nieskierowanych wszystkie krawędzie są przechowywane dwa razy.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- dystrybucja wierzchołków pomiędzy wątki jest realizowana blokowo - wątek 0 posiada wierzchołki $< 0, X >$, wątek 1 posiada wierzchołki $< X + 1, Y >$ itd.
- wszystkie wierzchołki i krawędzie początkowego grafu są rozdzielone - każdy wątek posiada N/p wierzchołków wraz z sąsiadującymi krawędziami

N - liczba wierzchołków w grafie
p - liczba wątków

W przypadku grafów nieskierowanych wszystkie krawędzie są przechowywane dwa razy.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- dystrybucja wierzchołków pomiędzy wątki jest realizowana blokowo - wątek 0 posiada wierzchołki $< 0, X >$, wątek 1 posiada wierzchołki $< X + 1, Y >$ itd.
- wszystkie wierzchołki i krawędzie początkowego grafu są rozdzielone - każdy wątek posiada N/p wierzchołków wraz z sąsiadującymi krawędziami

N - liczba wierzchołków w grafie
p - liczba wątków

W przypadku grafów nieskierowanych wszystkie krawędzie są przechowywane dwa razy.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Implementacja w PCJ:

- dystrybucja wierzchołków pomiędzy wątki jest realizowana blokowo - wątek 0 posiada wierzchołki $< 0, X >$, wątek 1 posiada wierzchołki $< X + 1, Y >$ itd.
- wszystkie wierzchołki i krawędzie początkowego grafu są rozdzielone - każdy wątek posiada N/p wierzchołków wraz z sąsiadującymi krawędziami

N - liczba wierzchołków w grafie
p - liczba wątków

W przypadku grafów nieskierowanych wszystkie krawędzie są przechowywane dwa razy.

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Proces konstrukcji grafu z listy krawędzi do postaci CSR można podzielić na dwie części:

- podział danych
- obliczenie CSR - wszystkie wątki tworzą reprezentacje CSR części grafu, biorąc pod uwagę tylko posiadane wierzchołki i ich incydentne krawędzie

Graph500 w PGAS i PCJ - Kernel 1 - Algorytm

Proces konstrukcji grafu z listy krawędzi do postaci CSR można podzielić na dwie części:

- podział danych
- obliczenie CSR - wszystkie wątki tworzą reprezentacje CSR części grafu, biorąc pod uwagę tylko posiadane wierzchołki i ich incydentne krawędzie

Graph500 w PGAS i PCJ - Kernel 1 - Pseudokod

Algorithm

Creation of CSR graph representation from a list of edge tuples.

Input: Undirected graph G(V, E) in the form of list of edge tuples L
Output: CSR representation of the graph G

```
1: for all processors in parallel do
2:     localMax ← FIND-MAX(Lp) ▷ Lp is a chunk of L processed by processor p
3: if processor 0
4:     globalMax ← REDUCE(localMax)
5:     BROADCAST(globalMax)
6: for all processors in parallel do
7:     FIND-RANGE(globalMax) ▷ range of vertices the processor owns
8:     COMPUTE-CSR(L) ▷ CSR computation for owned vertices
```

Graph500 w PGAS i PCJ - Kernel 1 - Pseudokod

Algorithm

Creation of CSR graph representation from a list of edge tuples.

Input: Undirected graph G(V, E) in the form of list of edge tuples L
Output: CSR representation of the graph G
1: **for** all processors **in parallel do**
2: localMax ← FIND-MAX(Lp) ▷ Lp is a chunk of L processed by processor p
3: **if** processor 0
4: globalMax ← REDUCE(localMax)
5: BROADCAST(globalMax)
6: **for** all processors **in parallel do**
7: FIND-RANGE(globalMax) ▷ range of vertices the processor owns
8: COMPUTE-CSR(L) ▷ CSR computation for owned vertices

Pierwszym krokiem algorytmu jest znalezienie maksymalnego identyfikatora wierzchołka w grafie listy krawędzi.

Graph500 w PGAS i PCJ - Kernel 1 - Pseudokod

Algorithm

Creation of CSR graph representation from a list of edge tuples.

Input: Undirected graph G(V, E) in the form of list of edge tuples L
Output: CSR representation of the graph G
1: **for** all processors **in parallel do**
2: localMax ← FIND-MAX(Lp) ▷ Lp is a chunk of L processed by processor p
3: **if** processor 0
4: globalMax ← REDUCE(localMax)
5: BROADCAST(globalMax)
6: **for** all processors **in parallel do**
7: FIND-RANGE(globalMax) ▷ range of vertices the processor owns
8: COMPUTE-CSR(L) ▷ CSR computation for owned vertices

Każdy wątek przeszukuje inną część listy krawędzi i znajduje jego maksimum.

Graph500 w PGAS i PCJ - Kernel 1 - Pseudokod

Algorithm

Creation of CSR graph representation from a list of edge tuples.

Input: Undirected graph G(V, E) in the form of list of edge tuples L
Output: CSR representation of the graph G
1: **for** all processors **in parallel do**
2: localMax ← FIND-MAX(Lp) ▷ Lp is a chunk of L processed by processor p
3: **if** processor 0
4: globalMax ← REDUCE(localMax)
5: BROADCAST(globalMax)
6: **for** all processors **in parallel do**
7: FIND-RANGE(globalMax) ▷ range of vertices the processor owns
8: COMPUTE-CSR(L) ▷ CSR computation for owned vertices

Następnie robiona jest redukcja i jest znajdowany globalny maksymalny wierzchołek.

Graph500 w PGAS i PCJ - Kernel 1 - Pseudokod

Algorithm

Creation of CSR graph representation from a list of edge tuples.

Input: Undirected graph G(V, E) in the form of list of edge tuples L
Output: CSR representation of the graph G
1: **for** all processors **in parallel do**
2: localMax ← FIND-MAX(Lp) ▷ Lp is a chunk of L processed by processor p
3: **if** processor 0
4: globalMax ← REDUCE(localMax)
5: BROADCAST(globalMax)
6: **for** all processors **in parallel do**
7: FIND-RANGE(globalMax) ▷ range of vertices the processor owns
8: COMPUTE-CSR(L) ▷ CSR computation for owned vertices

Wątek oblicza zakres posiadanych wierzchołków bazując na 1-wymiarowym podziale danych.

Graph500 w PGAS i PCJ - Kernel 1 - Pseudokod

Algorithm

Creation of CSR graph representation from a list of edge tuples.

Input: Undirected graph G(V, E) in the form of list of edge tuples L
Output: CSR representation of the graph G

1: **for** all processors **in parallel do**
2: localMax $\leftarrow$ FIND-MAX(Lp) $\triangleright$ Lp is a chunk of L processed by processor p
3: **if** processor 0
4: globalMax $\leftarrow$ REDUCE(localMax)
5: BROADCAST(globalMax)
6: **for** all processors **in parallel do**
7: FIND-RANGE(globalMax) $\triangleright$ range of vertices the processor owns
8: COMPUTE-CSR(L) $\triangleright$ CSR computation for owned vertices

Wierzchołki grafu są podzielone w taki sposób, że każdy wierzchołek ma dokładnie jednego właściciela.

Graph500 w PGAS i PCJ - Kernel 1 - Pseudokod

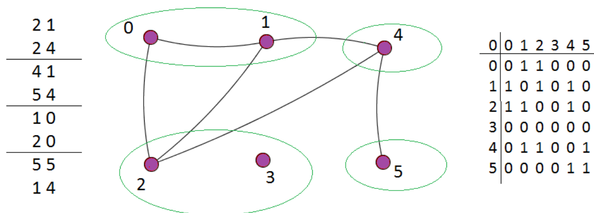
Algorithm

Creation of CSR graph representation from a list of edge tuples.

Input: Undirected graph G(V, E) in the form of list of edge tuples L
Output: CSR representation of the graph G
1: **for** all processors **in parallel do**
2: localMax $\leftarrow$ FIND-MAX(Lp) ▷ Lp is a chunk of L processed by processor p
3: **if** processor 0
4: globalMax $\leftarrow$ REDUCE(localMax)
5: BROADCAST(globalMax)
6: **for** all processors **in parallel do**
7: FIND-RANGE(globalMax) ▷ range of vertices the processor owns
8: COMPUTE-CSR(L) ▷ CSR computation for owned vertices

Każdy wątek wybiera tylko krawędzie sąsiadujące do posiadanych wierzchołków.

Graph500 w PGAS i PCJ - Kernel 1 - Przykład



Wierzchołki:	Proces:	
0 1	0	[2, 6] [1, 2, 2, 4, 0, 4]
2 3	1	[3, 3] [1, 4, 0]
4	2	[4] [2, 1, 5, 1]
5	3	[1] [4]

Graph500 w PGAS i PCJ - Kernel 2

Kernel 2:

Dane: graf G(V,E) w postaci CSR, podzielony pomiędzy wątkami, źródło $v \in V$

Wynik: tablica poprzedników - drzewo BFS

Graph500 w PGAS i PCJ - Kernel 2

BFS rozpoczyna od wyróżnionego wierzchołka pobranego z Genaratora benchmarku Graph500.

Źródłowy wierzchołek jest losowo wybrany z wierzchołków grafu.

Aby uniknąć trywialnych przeszukiwań, tylko wierzchołki ze stopniem co najmniej 1 nie licząc pętli są brane pod uwagę.

Graph500 w PGAS i PCJ - Kernel 2

BFS rozpoczyna od wyróżnionego wierzchołka pobranego z Genaratora benchmarku Graph500.

Źródłowy wierzchołek jest losowo wybrany z wierzchołków grafu.

Aby uniknąć trywialnych przeszukiwań, tylko wierzchołki ze stopniem co najmniej 1 nie licząc pętli są brane pod uwagę.

Graph500 w PGAS i PCJ - Kernel 2

BFS rozpoczyna od wyróżnionego wierzchołka pobranego z Genaratora benchmarku Graph500.

Źródłowy wierzchołek jest losowo wybrany z wierzchołków grafu.

Aby uniknąć trywialnych przeszukiwań, tylko wierzchołki ze stopniem co najmniej 1 nie licząc pętli są brane pod uwagę.

Graph500 w PGAS i PCJ - Kernel 2

Po obliczeniach Kernela 1, początkowy graf jest trzymany w rozproszonym formacie CSR.

Każdy wątek trzyma swój własny podzbiór wierzchołków razem z sąsiednimi krawędziami.

Implementacja równoległego algorytmu BFS jest oparta na idei synchronizacji wątków po każdym poziomie.

Graph500 w PGAS i PCJ - Kernel 2

Po obliczeniach Kernela 1, początkowy graf jest trzymany w rozproszonym formacie CSR.
Każdy wątek trzyma swój własny podzbiór wierzchołków razem z sąsiednimi krawędziami.

Implementacja równoległego algorytmu BFS jest oparta na idei synchronizacji wątków po każdym poziomie.

Graph500 w PGAS i PCJ - Kernel 2

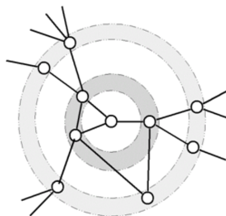
Po obliczeniach Kernela 1, początkowy graf jest trzymany w rozproszonym formacie CSR.
Każdy wątek trzyma swój własny podzbiór wierzchołków razem z sąsiednimi krawędziami.

Implementacja równoległego algorytmu BFS jest oparta na idei synchronizacji wątków po każdym poziomie.

Graph500 w PGAS i PCJ - Kernel 2

Każdy wierzchołek w odległości k od źródła jest odwiedzony wcześniej niż wierzchołek na poziomie $k+1$.

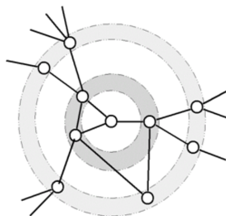
Odległość pomiędzy dwoma wierzchołkami jest najkrótszą ścieżką łączącą te wierzchołki.



Nieskierowany graf ze źródłowym wierzchołkiem jako punktem centralnym razem z wierzchołkami na poziomie 1 oraz 2.

Graph500 w PGAS i PCJ - Kernel 2

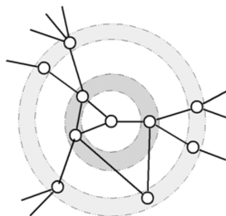
Każdy wierzchołek w odległości k od źródła jest odwiedzony wcześniej niż wierzchołek na poziomie $k+1$.
Odległość pomiędzy dwoma wierzchołkami jest najkrótszą ścieżką łączącą te wierzchołki.



Nieskierowany graf ze źródłowym wierzchołkiem jako punktem centralnym razem z wierzchołkami na poziomie 1 oraz 2.

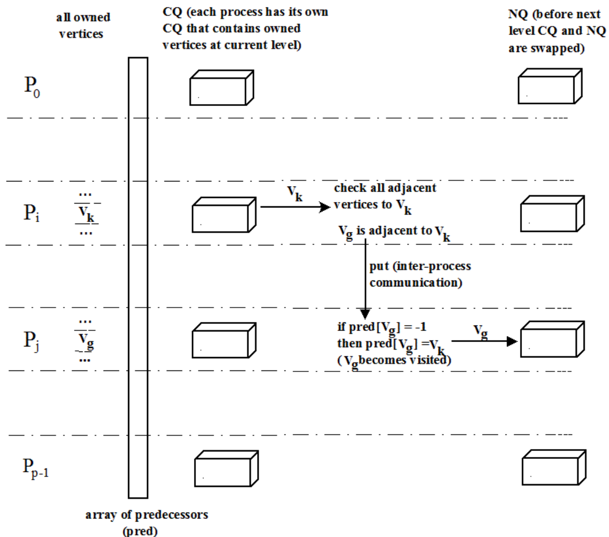
Graph500 w PGAS i PCJ - Kernel 2

Każdy wierzchołek w odległości k od źródła jest odwiedzony wcześniej niż wierzchołek na poziomie $k+1$.
Odległość pomiędzy dwoma wierzchołkami jest najkrótszą ścieżką łączącą te wierzchołki.



Nieskierowany graf ze źródłowym wierzchołkiem jako punktem centralnym razem z wierzchołkami na poziomie 1 oraz 2.

Graph500 w PGAS i PCJ - Kernel 2



Graph500 w PGAS i PCJ - Kernel 2

Aby przyspieszyć działanie zastosowano kilka optymalizacji:

- bitmapa, zawierająca informacje o odwiedzonych wierzchołkach, aby zredukować ilość potrzebnej pamięci, a co najważniejsze ograniczyć komunikację pomiędzy wątkami
  Każdy wątek przechwuje wektor bitów $bitmap[v]$ - bit ustawiony jest na 1 jeśli wierzchołek jest odwiedzony
  Po każdym poziomie informacje w bitmapie są wymieniane pomiędzy wątkami
- użycie tablicowych buforów danych (ważne ze względu na specyfikę biblioteki PCJ, która wspiera wysyłanie tablic po indeksach), co pozwala na zminimalizowanie narzutu związanego z rozpoczęciem komunikacji (zamiast przesyłać pojedynczy wierzchołek, każdy wątek grupuje kilka wiadomości)
- nakładanie komunikacji i obliczeń, poprzez wykonywanie niezależnych procedur np. VISIT-VERTICES w trakcie oczekiwania na zakończenie komunikacji - wątek przetwarza dane, które już otrzymał (PCJ.waitFor("variable",0) - zwraca liczbę otrzymanych danych)

Graph500 w PGAS i PCJ - Kernel 2

Aby przyspieszyć działanie zastosowano kilka optymalizacji:

- bitmapa, zawierająca informacje o odwiedzonych wierzchołkach, aby zredukować ilość potrzebnej pamięci, a co najważniejsze ograniczyć komunikację pomiędzy wątkami
  Każdy wątek przechwuje wektor bitów *bitmap*$[v]$ - bit ustawiony jest na 1 jeśli wierzchołek jest odwiedzony
  Po każdym poziomie informacje w bitmapie są wymieniane pomiędzy wątkami
- użycie tablicowych buforów danych (ważne ze względu na specyfikę biblioteki PCJ, która wspiera wysyłanie tablic po indeksach), co pozwala na zminimalizowanie narzutu związanego z rozpoczęciem komunikacji (zamiast przesyłać pojedynczy wierzchołek, każdy wątek grupuje kilka wiadomości)
- nakładanie komunikacji i obliczeń, poprzez wykonywanie niezależnych procedur np. VISIT-VERTICES w trakcie oczekiwania na zakończenie komunikacji - wątek przetwarza dane, które już otrzymał (PCJ.waitFor("variable",0) - zwraca liczbę otrzymanych danych)

Graph500 w PGAS i PCJ - Kernel 2

Aby przyspieszyć działanie zastosowano kilka optymalizacji:

- bitmapa, zawierająca informacje o odwiedzonych wierzchołkach, aby zredukować ilość potrzebnej pamięci, a co najważniejsze ograniczyć komunikację pomiędzy wątkami
  Każdy wątek przechwuje wektor bitów *bitmap*$[v]$ - bit ustawiony jest na 1 jeśli wierzchołek jest odwiedzony
  Po każdym poziomie informacje w bitmapie są wymieniane pomiędzy wątkami
- użycie tablicowych buforów danych (ważne ze względu na specyfikę biblioteki PCJ, która wspiera wysyłanie tablic po indeksach), co pozwala na zminimalizowanie narzutu związanego z rozpoczęciem komunikacji (zamiast przesyłać pojedynczy wierzchołek, każdy wątek grupuje kilka wiadomości)
- nakładanie komunikacji i obliczeń, poprzez wykonywanie niezależnych procedur np. VISIT-VERTICES w trakcie oczekiwania na zakończenie komunikacji - wątek przetwarza dane, które już otrzymał (PCJ.waitFor("variable",0) - zwraca liczbę otrzymanych danych)

Graph500 w PGAS i PCJ - Kernel 2

Aby przyspieszyć działanie zastosowano kilka optymalizacji:

- bitmapa, zawierająca informacje o odwiedzonych wierzchołkach, aby zredukować ilość potrzebnej pamięci, a co najważniejsze ograniczyć komunikację pomiędzy wątkami
  Każdy wątek przechwuje wektor bitów $bitmap[v]$ - bit ustawiony jest na 1 jeśli wierzchołek jest odwiedzony
  Po każdym poziomie informacje w bitmapie są wymieniane pomiędzy wątkami
- użycie tablicowych buforów danych (ważne ze względu na specyfikę biblioteki PCJ, która wspiera wysyłanie tablic po indeksach), co pozwala na zminimalizowanie narzutu związanego z rozpoczęciem komunikacji (zamiast przesyłać pojedynczy wierzchołek, każdy wątek grupuje kilka wiadomości)
- nakładanie komunikacji i obliczeń, poprzez wykonywanie niezależnych procedur np. VISIT-VERTICES w trakcie oczekiwania na zakończenie komunikacji - wątek przetwarza dane, które już otrzymał (PCJ.waitFor("variable",0) - zwraca liczbę otrzymanych danych)

Graph500 w PGAS i PCJ - Kernel 2

Aby przyspieszyć działanie zastosowano kilka optymalizacji:

- bitmapa, zawierająca informacje o odwiedzonych wierzchołkach, aby zredukować ilość potrzebnej pamięci, a co najważniejsze ograniczyć komunikację pomiędzy wątkami
  Każdy wątek przechwuje wektor bitów $bitmap[v]$ - bit ustawiony jest na 1 jeśli wierzchołek jest odwiedzony
  Po każdym poziomie informacje w bitmapie są wymieniane pomiędzy wątkami
- użycie tablicowych buforów danych (ważne ze względu na specyfikę biblioteki PCJ, która wspiera wysyłanie tablic po indeksach), co pozwala na zminimalizowanie narzutu związanego z rozpoczęciem komunikacji (zamiast przesyłać pojedynczy wierzchołek, każdy wątek grupuje kilka wiadomości)
- nakładanie komunikacji i obliczeń, poprzez wykonywanie niezależnych procedur np. VISIT-VERTICES w trakcie oczekiwania na zakończenie komunikacji - wątek przetwarza dane, które już otrzymał (PCJ.waitFor("variable",0) - zwraca liczbę otrzymanych danych)

Graph500 w PGAS i PCJ - Kernel 2

Aby przyspieszyć działanie zastosowano kilka optymalizacji:

- bitmapa, zawierająca informacje o odwiedzonych wierzchołkach, aby zredukować ilość potrzebnej pamięci, a co najważniejsze ograniczyć komunikację pomiędzy wątkami
  Każdy wątek przechwuje wektor bitów *bitmap*$[v]$ - bit ustawiony jest na 1 jeśli wierzchołek jest odwiedzony
  Po każdym poziomie informacje w bitmapie są wymieniane pomiędzy wątkami
- użycie tablicowych buforów danych (ważne ze względu na specyfikę biblioteki PCJ, która wspiera wysyłanie tablic po indeksach), co pozwala na zminimalizowanie narzutu związanego z rozpoczęciem komunikacji (zamiast przesyłać pojedynczy wierzchołek, każdy wątek grupuje kilka wiadomości)
- nakładanie komunikacji i obliczeń, poprzez wykonywanie niezależnych procedur np. VISIT-VERTICES w trakcie oczekiwania na zakończenie komunikacji - wątek przetwarza dane, które już otrzymał (PCJ.waitFor("variable",0) - zwraca liczbę otrzymanych danych)

Graph500 w PGAS i PCJ - Kernel 2

Algorithm

BFS pseudocode.

Input: Graph G(V, E) in the form of CSR distributed by blocks between processors, source vertex $s \in V$
Output: Array of predecessors (pred) in the BFS outcome tree rooted at s

```
BFS(s)
 1: INIT-DATA()
 2: currentLvlIndex ← 0
 3: if CHECK-IF-I-OWN-THE-VERTEX(s)
 4:    MARK-SOURCE-AS-VISITED(s)
 5: while (true)
 6:    for each v1 in currentLvl
 7:      for each v2 adjacent to v1
 8:        v2Owner ← FIND-TASK-FOR-VERTEX(v2)
 9:        if bitmap[v2Owner].get(FIND-LOCAL-NBR(v2)) == false)
10:          if CHECK-IF-I-OWN-THE-VERTEX(v2)
11:            VISIT(v2, v1)
12:          else
13:            ACCUMULATE-BUFFER-SEND(v2Owner, v2, v1)
14:    for each process p
15:      BUFFER-SEND(p)                              ▷ Send buffer to task p
16:    for each process p                           ▷ Send number of parts
17:      PCJ.put(p,"partsSent",toSendPredPartCounter[p])
```

Graph500 w PGAS i PCJ - Kernel 2

```
18:   WAIT-AND-VISIT()
19:   if PCJ.myId() ! = 0
20:     PCJ.put(0,"verticesInNextLvl", nextLvlIndex, PCJ.myId())
21:   else
22:     computeNextLvl = REDUCE(verticesInNextLvl)
23:     PCJ.broadcast("computeNextLvl", computeNextLvl)
24:   PCJ.waitFor("computeNextLvl")
25:   if computeNextLvl == true
26:     INIT-FOR-NEXT-LOOP()
27:     ALL-TO-ALL(bitmap[PCJ.myId()])
28:     PCJ.waitFor("bitmap")
29:   else
30:     break
```

Graph500 w PGAS i PCJ - Kernel 2

```
MARK-SOURCE-AS-VISITED(s)
1: currentLvl[currentLvlIndex++] ← s
2: sLocal ← FIND-LOCAL-NBR(s)
3: pred[sLocal] ← -1
4: bitmap[PCJ.myId()].set(sLocal)                    ▷ Mark source vertex as visited

VISIT(v2, v1)
1: v2Local ← FIND-LOCAL-NBR(v2)
2: pred[v2Local] ← v1
3: nextLvl[nextLvlIndex++] ← v2
4: bitmap[PCJ.myId()].set(v2Local)                   ▷ Mark vertex v2 as visited

ACCUMULATE-BUFFER-SEND(v2Owner, v2, v1)
1: ADD-TO-BUFFER(toSendPredBuffer[v2Owner], v2)
2: ADD-TO-BUFFER(toSendPredBuffer[v2Owner], v1)
3: if BUFFER-IS-FULL(toSendPredBuffer[v2Owner])
4:     PCJ.put(v2Owner,"rcvedPred",toSendPredBuffer[v2Owner], ..)
5:     toSendPredPartCounter[v2Owner]++              ▷ Count sent data parts
```

Graph500 w PGAS i PCJ - Kernel 2

```
BUFFER-SEND(p)
1: if BUFFER-IS-NOT-EMPTY(toSendPredBuffer[p])
2:     PCJ.put(p,"rcvedPred",toSendPredBuffer[p],..)        ▷ Send data
3:     toSendPredPartCounter[p]++                 ▷ Count sent out data parts
```

```
WAIT-AND-VISIT()
 1: partsSentCounter = PCJ.waitFor("partsSent", 0)
 2: while partsSentCounter < PCJ.threadCount()
 3:     VISIT-VERTICES(rcvedPred)
 4:     partsSentCounter = PCJ.waitFor("partsSent", 0)
 5: chunksWaitingForCounter ← SUM(partsSent)
 6: rcvedPredCounter = PCJ.waitFor("rcvedPred", 0)
 7: while rcvedPredCounter < chunksWaitingForCounter
 8:     VISIT-VERTICES(rcvedPred)                ▷ Visit received vertices
 9:     rcvedPredCounter = PCJ.waitFor("rcvedPred", 0)
10: VISIT-VERTICES(rcvedPred)           ▷ All parts came, so visit vertices
```

Graph500 w PGAS i PCJ - Dane testowe

Dane testowe: Generator Grafów Kroneckera benchmarku Graph 500 (grafy w postaci listy krawędzi)

Testy przeprowadzono na grafach o rozmiarach:

- $edgefactor = 16$
- $SCALE \in \{25, 26, 27, 28, 29, 30\}$

SCALE - logarytm o podstawie 2 z ilości wierzchołków
edgefactor - stosunek liczby krawędzi grafu do liczny wierzchołków

- liczba wierzchołków $N = 2^{SCALE}$
- liczba krawędzi $M = edgefactor \cdot N$

Graph500 w PGAS i PCJ - Środowisko testowe

klaster Hydra - obliczenia wykonywane były na wezłach:

- Istanbul - 96 wezłów z dwoma 6-rdzeniowymi procesorami AMD Opteron(tm) Processor 2435 2.6 GHz oraz z 32 GB pamięci RAM oraz łączem Infiniband DDR + 1Gb Ethernet
- Interlagos - 16 wezłów z czterema 16-rdzeniowymi procesorami AMD Opteron(TM) Processor 6272 2.2 GHz oraz z 512 GB pamięci RAM oraz łączem 10Gb Ethernet
- Magnycours - 30 węzłów z czterema 12-rdzeniowymi procesorami AMD Opteron(TM) Processor 6174 2.2 GHz oraz z 256 GB pamięci oraz łączem 10Gb Ethernet

Graph500 w PGAS i PCJ - Środowisko testowe

klaster Hydra - obliczenia wykonywane były na wezłach:

- Istanbul - 96 wezłów z dwoma 6-rdzeniowymi procesorami AMD Opteron(tm) Processor 2435 2.6 GHz oraz z 32 GB pamięci RAM oraz łączem Infiniband DDR + 1Gb Ethernet
- Interlagos - 16 wezłów z czterema 16-rdzeniowymi procesorami AMD Opteron(TM) Processor 6272 2.2 GHz oraz z 512 GB pamięci RAM oraz łączem 10Gb Ethernet
- Magnycours - 30 węzłów z czterema 12-rdzeniowymi procesorami AMD Opteron(TM) Processor 6174 2.2 GHz oraz z 256 GB pamięci oraz łączem 10Gb Ethernet

Graph500 w PGAS i PCJ - Środowisko testowe

klaster Hydra - obliczenia wykonywane były na wezłach:

- Istanbul - 96 wezłów z dwoma 6-rdzeniowymi procesorami AMD Opteron(tm) Processor 2435 2.6 GHz oraz z 32 GB pamięci RAM oraz łączem Infiniband DDR + 1Gb Ethernet
- Interlagos - 16 wezłów z czterema 16-rdzeniowymi procesorami AMD Opteron(TM) Processor 6272 2.2 GHz oraz z 512 GB pamięci RAM oraz łączem 10Gb Ethernet
- Magnycours - 30 węzłów z czterema 12-rdzeniowymi procesorami AMD Opteron(TM) Processor 6174 2.2 GHz oraz z 256 GB pamięci oraz łączem 10Gb Ethernet

Graph500 w PGAS i PCJ - Środowisko testowe

klaster Hydra - obliczenia wykonywane były na wezłach:

- Istanbul - 96 wezłów z dwoma 6-rdzeniowymi procesorami AMD Opteron(tm) Processor 2435 2.6 GHz oraz z 32 GB pamięci RAM oraz łączem Infiniband DDR + 1Gb Ethernet
- Interlagos - 16 wezłów z czterema 16-rdzeniowymi procesorami AMD Opteron(TM) Processor 6272 2.2 GHz oraz z 512 GB pamięci RAM oraz łączem 10Gb Ethernet
- Magnycours - 30 węzłów z czterema 12-rdzeniowymi procesorami AMD Opteron(TM) Processor 6174 2.2 GHz oraz z 256 GB pamięci oraz łączem 10Gb Ethernet

Graph500 w PGAS i PCJ - Środowisko testowe

klaster Topola:

- węzły haswell z dwoma 14-rdzeniowymi procesorami Intel(R) Xeon(R) CPU E5-2697 v3 o taktowaniu 2.1GHz z AVX2, do 3.0GHz oraz łączem Infiniband FDR + 1Gb Ethernet, 60 węzłów z pamięcią RAM 128GB oraz 163 węzły zpamięcią RAM 64GB

klaster Okeanos

- 1084 wezłów obliczeniowych z dwoma 24-rdzeniowymi procesorami Intel Xeon CPU E5-2690 v3 2.6 GHz oraz z 128 GB pamięci RAM. Każdy węzeł posiada dedykowane porty komunikacji Cray Aries, a wszystkie węzły połączone są wysokowydajną siecią o topologii Dragonfly (testowany w ramach programu Early Science)

Graph500 w PGAS i PCJ - Środowisko testowe

klaster Topola:

- węzły haswell z dwoma 14-rdzeniowymi procesorami Intel(R) Xeon(R) CPU E5-2697 v3 o taktowaniu 2.1GHz z AVX2, do 3.0GHz oraz łączem Infiniband FDR + 1Gb Ethernet, 60 węzłów z pamięcią RAM 128GB oraz 163 węzły zpamięcią RAM 64GB

klaster Okeanos

- 1084 wezłów obliczeniowych z dwoma 24-rdzeniowymi procesorami Intel Xeon CPU E5-2690 v3 2.6 GHz oraz z 128 GB pamięci RAM. Każdy węzeł posiada dedykowane porty komunikacji Cray Aries, a wszystkie węzły połączone są wysokowydajną siecią o topologii Dragonfly (testowany w ramach programu Early Science)

Graph500 w PGAS i PCJ - Środowisko testowe

klaster Topola:

- węzły haswell z dwoma 14-rdzeniowymi procesorami Intel(R) Xeon(R) CPU E5-2697 v3 o taktowaniu 2.1GHz z AVX2, do 3.0GHz oraz łączem Infiniband FDR + 1Gb Ethernet, 60 węzłów z pamięcią RAM 128GB oraz 163 węzły zpamięcią RAM 64GB

klaster Okeanos

- 1084 wezłów obliczeniowych z dwoma 24-rdzeniowymi procesorami Intel Xeon CPU E5-2690 v3 2.6 GHz oraz z 128 GB pamięci RAM. Każdy węzeł posiada dedykowane porty komunikacji Cray Aries, a wszystkie węzły połączone są wysokowydajną siecią o topologii Dragonfly (testowany w ramach programu Early Science)

Graph500 w PGAS i PCJ - Środowisko testowe

klaster Topola:

- węzły haswell z dwoma 14-rdzeniowymi procesorami Intel(R) Xeon(R) CPU E5-2697 v3 o taktowaniu 2.1GHz z AVX2, do 3.0GHz oraz łączem Infiniband FDR + 1Gb Ethernet, 60 węzłów z pamięcią RAM 128GB oraz 163 węzły zpamięcią RAM 64GB

klaster Okeanos

- 1084 wezłów obliczeniowych z dwoma 24-rdzeniowymi procesorami Intel Xeon CPU E5-2690 v3 2.6 GHz oraz z 128 GB pamięci RAM. Każdy węzeł posiada dedykowane porty komunikacji Cray Aries, a wszystkie węzły połączone są wysokowydajną siecią o topologii Dragonfly (testowany w ramach programu Early Science)

Graph500 w PGAS i PCJ - Środowisko testowe

Do testów użyto 64-bitową Wirtualną Maszynę Javy Oracle (Java 8) oraz OpenMPI w wersji 1.6.5.

Testy przeprowadzone były przy różnych ilościach procesów na węzeł (ozn. pn - per node) ze względu na znaczne wymagania pamięciowe obliczeń.

Wydajność algorytmu oprócz czasu wykonania była mierzona przy użyciu wskaźnika TEPS (Traversed Edges per Second).

Graph500 w PGAS i PCJ - Środowisko testowe

Do testów użyto 64-bitową Wirtualną Maszynę Javy Oracle (Java 8) oraz OpenMPI w wersji 1.6.5.

Testy przeprowadzone były przy różnych ilościach procesów na węzeł (ozn. pn - per node) ze względu na znaczne wymagania pamięciowe obliczeń.

Wydajność algorytmu oprócz czasu wykonania była mierzona przy użyciu wskaźnika TEPS (Traversed Edges per Second).

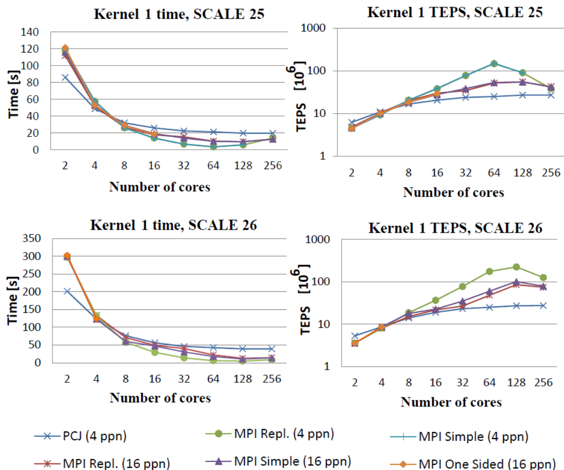
Graph500 w PGAS i PCJ - Środowisko testowe

Do testów użyto 64-bitową Wirtualną Maszynę Javy Oracle (Java 8) oraz OpenMPI w wersji 1.6.5.

Testy przeprowadzone były przy różnych ilościach procesów na węzeł (ozn. pn - per node) ze względu na znaczne wymagania pamięciowe obliczeń.

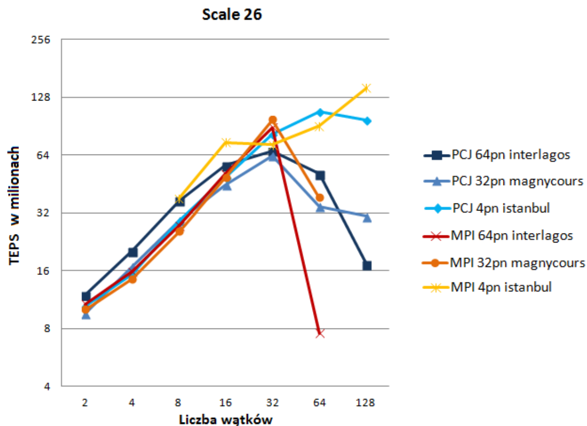
Wydajność algorytmu oprócz czasu wykonania była mierzona przy użyciu wskaźnika TEPS (Traversed Edges per Second).

Graph500 w PGAS i PCJ - Kernel 1

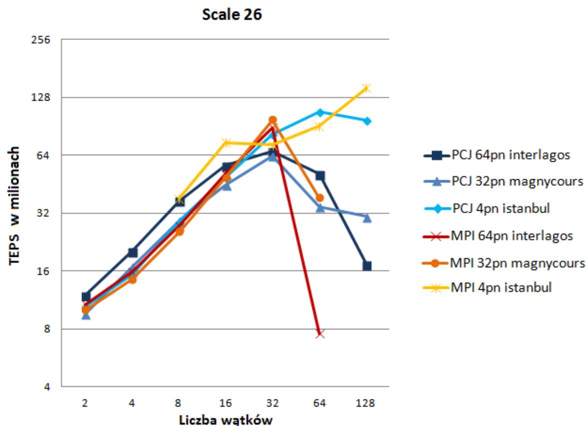


Całkowity czas wykonania Kernela 1 razem z wydajnością TEPS dla grafów SCALE 25 oraz 26 (Topola).

Graph500 w PGAS i PCJ - Kernel 2

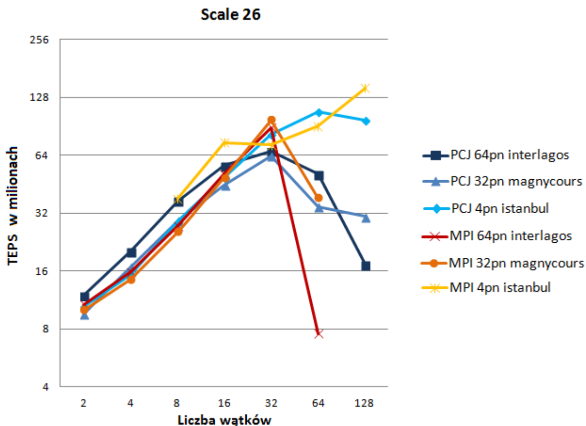


Graph500 w PGAS i PCJ - Kernel 2



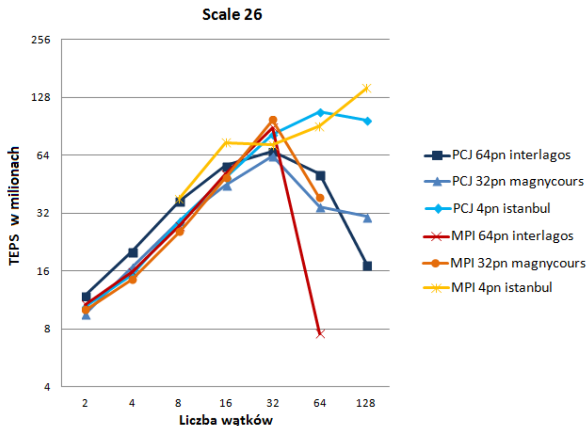
Implementacja w Javie przy użyciu PCJ ma podobną wydajność i skalowalność co implementacja MPI w C.

Graph500 w PGAS i PCJ - Kernel 2



Przyspieszenie jest liniowe do 32 wątków i jest niezależne od ilości wątków na pojedynczym węźle.

Graph500 w PGAS i PCJ - Kernel 2



Wyniki dla większej liczby węzłów, pogarszają się co wynika z przewagi czasu poświęconego na komunikację w stosunku do czasu obliczeń (najlepsze rezultaty uzyskano dla węzłów z połączeniem Infiniband) (dużo lepsze wyniki uzyskano dla klastra Okeanos).

Graph500 w PGAS i PCJ - Kernel 2

Przy badaniu wydajności algorytm podzielono na cztery części:

- część 1 - inicjalizacja struktur danych, przetworzenie wszystkich wierzchołków w obecnym poziomie oraz wysłanie danych do innych wątków
- część 2 - oczekiwanie na dane i przeprocesowanie odebranych danych
- część 3 - redukcja i rozgłoszenie oraz oczekiwanie na wiadomość dotyczącą obliczenia kolejnego poziomu
- część 4 - powtórna inicjalizacja danych oraz komunikacja all-to-all - wymiana bitmapy

Graph500 w PGAS i PCJ - Kernel 2

Przy badaniu wydajności algorytm podzielono na cztery części:

- część 1 - inicjalizacja struktur danych, przetworzenie wszystkich wierzchołków w obecnym poziomie oraz wysłanie danych do innych wątków
- część 2 - oczekiwanie na dane i przeprocesowanie odebranych danych
- część 3 - redukcja i rozgłoszenie oraz oczekiwanie na wiadomość dotyczącą obliczenia kolejnego poziomu
- część 4 - powtórna inicjalizacja danych oraz komunikacja all-to-all - wymiana bitmapy

Graph500 w PGAS i PCJ - Kernel 2

Przy badaniu wydajności algorytm podzielono na cztery części:

- część 1 - inicjalizacja struktur danych, przetworzenie wszystkich wierzchołków w obecnym poziomie oraz wysłanie danych do innych wątków
- część 2 - oczekiwanie na dane i przeprocesowanie odebranych danych
- część 3 - redukcja i rozgłoszenie oraz oczekiwanie na wiadomość dotyczącą obliczenia kolejnego poziomu
- część 4 - powtórna inicjalizacja danych oraz komunikacja all-to-all - wymiana bitmapy

Graph500 w PGAS i PCJ - Kernel 2

Przy badaniu wydajności algorytm podzielono na cztery części:

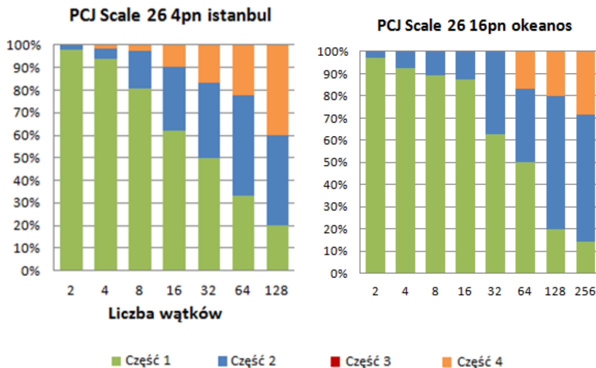
- część 1 - inicjalizacja struktur danych, przetworzenie wszystkich wierzchołków w obecnym poziomie oraz wysłanie danych do innych wątków
- część 2 - oczekiwanie na dane i przeprocesowanie odebranych danych
- część 3 - redukcja i rozgłoszenie oraz oczekiwanie na wiadomość dotyczącą obliczenia kolejnego poziomu
- część 4 - powtórna inicjalizacja danych oraz komunikacja all-to-all - wymiana bitmapy

Graph500 w PGAS i PCJ - Kernel 2

Przy badaniu wydajności algorytm podzielono na cztery części:

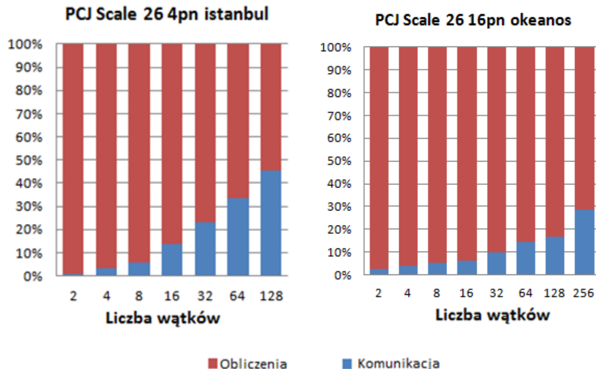
- część 1 - inicjalizacja struktur danych, przetworzenie wszystkich wierzchołków w obecnym poziomie oraz wysłanie danych do innych wątków
- część 2 - oczekiwanie na dane i przeprocesowanie odebranych danych
- część 3 - redukcja i rozgłoszenie oraz oczekiwanie na wiadomość dotyczącą obliczenia kolejnego poziomu
- część 4 - powtórna inicjalizacja danych oraz komunikacja all-to-all - wymiana bitmapy

Graph500 w PGAS i PCJ - Kernel 2



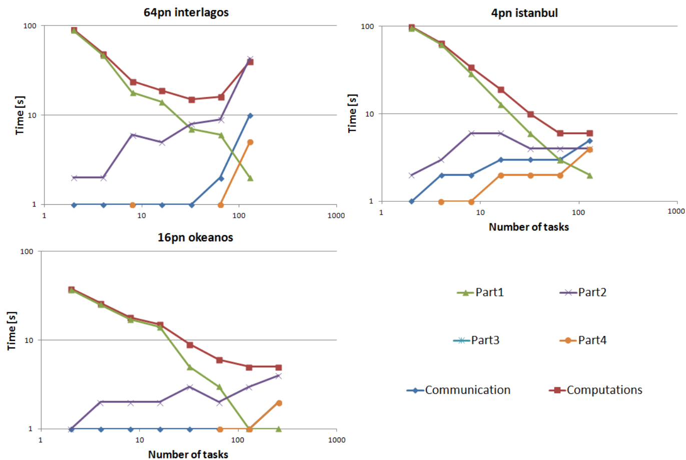
Procentowy czas wykonania algorytmu BFS dla grafu Scale 26 dla PCJ na węzłach istanbul i okeanos.

Graph500 w PGAS i PCJ - Kernel 2

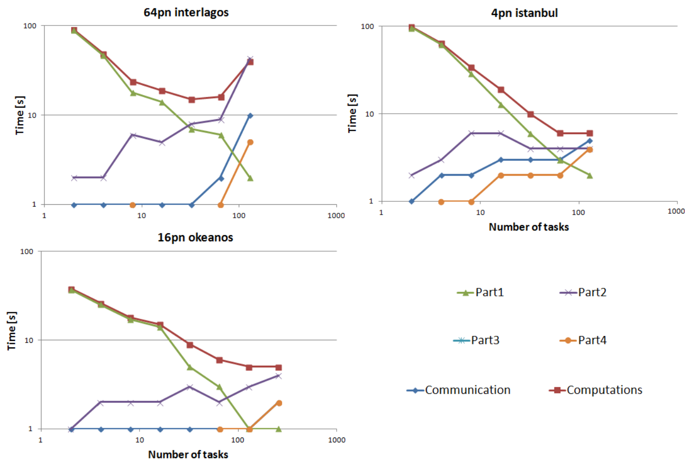


Procentowy czas wykonania algorytmu BFS dla grafu Scale 26 dla PCJ na węzłach istanbul i okeanos.

Graph500 w PGAS i PCJ - Kernel 2 - profil wydajności dla grafu SCALE 26

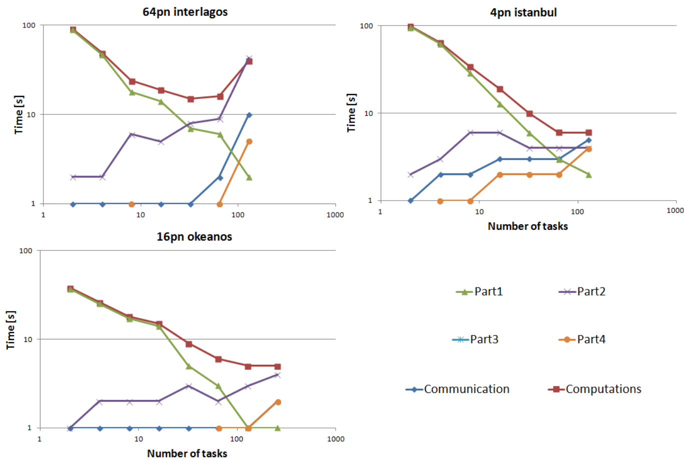


Graph500 w PGAS i PCJ - Kernel 2 - profil wydajności dla grafu SCALE 26



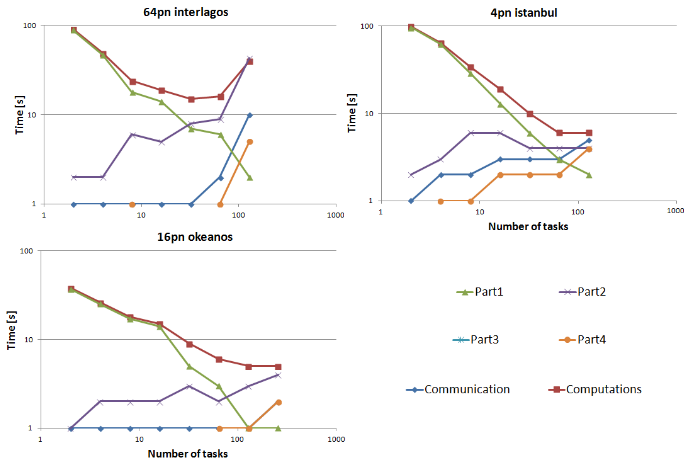
Przy mniejszej liczbie wątków najdłuższą częścią algorytmu jest część 1.

Graph500 w PGAS i PCJ - Kernel 2 - profil wydajności dla grafu SCALE 26



Część 2 jest kluczowa przy większej liczbie wątków - dla węzłów z szybkim łączem czas spędzony w tej części algorytmu jest mniejszy, co pozwala na lepszą skalowalność.

Graph500 w PGAS i PCJ - Kernel 2 - profil wydajności dla grafu SCALE 26



Części 3 oraz 4 wnoszą mniej do całkowitego czasu wykonania.

Dalsze prace

Dalsze prace badawcze będą obejmowały następujące zagadnienia:

- porównanie z Graph500 na Cray XC40
- porównanie z rozwiązaniami opartymi o MapReduce np. Giraph
- zwiększenie skali obliczeń, pod względem wielkości testowanych grafów (SCALE 30-32) oraz wykorzystywanych zasobów obliczeniowych - większej liczby węzłów
- dalsza optymalizacja algorytmów i ich implementacji w Javie
  - komunikacja all-to-all na żądanie (zamiast wymiany po każdym poziomie)
  - kompresja bitmapy np. poprzez javaewah

Dalsze prace

Dalsze prace badawcze będą obejmowały następujące zagadnienia:

- porównanie z Graph500 na Cray XC40
- porównanie z rozwiązaniami opartymi o MapReduce np. Giraph
- zwiększenie skali obliczeń, pod względem wielkości testowanych grafów (SCALE 30-32) oraz wykorzystywanych zasobów obliczeniowych - większej liczby węzłów
- dalsza optymalizacja algorytmów i ich implementacji w Javie
  - komunikacja all-to-all na żądanie (zamiast wymiany po każdym poziomie)
  - kompresja bitmapy np. poprzez javaewah

Dalsze prace

Dalsze prace badawcze będą obejmowały następujące zagadnienia:

- porównanie z Graph500 na Cray XC40
- porównanie z rozwiązaniami opartymi o MapReduce np. Giraph
- zwiększenie skali obliczeń, pod względem wielkości testowanych grafów (SCALE 30-32) oraz wykorzystywanych zasobów obliczeniowych - większej liczby węzłów
- dalsza optymalizacja algorytmów i ich implementacji w Javie
  - komunikacja all-to-all na żądanie (zamiast wymiany po każdym poziomie)
  - kompresja bitmapy np. poprzez javaewah

Dalsze prace

Dalsze prace badawcze będą obejmowały następujące zagadnienia:

- porównanie z Graph500 na Cray XC40
- porównanie z rozwiązaniami opartymi o MapReduce np. Giraph
- zwiększenie skali obliczeń, pod względem wielkości testowanych grafów (SCALE 30-32) oraz wykorzystywanych zasobów obliczeniowych - większej liczby węzłów
- dalsza optymalizacja algorytmów i ich implementacji w Javie
  - komunikacja all-to-all na żądanie (zamiast wymiany po każdym poziomie)
  - kompresja bitmapy np. poprzez javaewah

Dalsze prace

Dalsze prace badawcze będą obejmowały następujące zagadnienia:

- porównanie z Graph500 na Cray XC40
- porównanie z rozwiązaniami opartymi o MapReduce np. Giraph
- zwiększenie skali obliczeń, pod względem wielkości testowanych grafów (SCALE 30-32) oraz wykorzystywanych zasobów obliczeniowych - większej liczby węzłów
- dalsza optymalizacja algorytmów i ich implementacji w Javie
  - komunikacja all-to-all na żądanie (zamiast wymiany po każdym poziomie)
  - kompresja bitmapy np. poprzez javaewah

Dalsze prace

Dalsze prace badawcze będą obejmowały następujące zagadnienia:

- porównanie z Graph500 na Cray XC40
- porównanie z rozwiązaniami opartymi o MapReduce np. Giraph
- zwiększenie skali obliczeń, pod względem wielkości testowanych grafów (SCALE 30-32) oraz wykorzystywanych zasobów obliczeniowych - większej liczby węzłów
- dalsza optymalizacja algorytmów i ich implementacji w Javie
  - komunikacja all-to-all na żądanie (zamiast wymiany po każdym poziomie)
  - kompresja bitmapy np. poprzez javaewah

Dalsze prace

Dalsze prace badawcze będą obejmowały następujące zagadnienia:

- porównanie z Graph500 na Cray XC40
- porównanie z rozwiązaniami opartymi o MapReduce np. Giraph
- zwiększenie skali obliczeń, pod względem wielkości testowanych grafów (SCALE 30-32) oraz wykorzystywanych zasobów obliczeniowych - większej liczby węzłów
- dalsza optymalizacja algorytmów i ich implementacji w Javie
  - komunikacja all-to-all na żądanie (zamiast wymiany po każdym poziomie)
  - kompresja bitmapy np. poprzez javaewah

Opublikowane prace

Lista publikacji:

- M. Ryczkowska, M. Nowicki, P. Bała, Level-synchronous BFS algorithm implemented in Java using PCJ Library, *2016 International Conference on Computational Science and Computational Intelligence (CSCI), Parallel and Distributed Computing and Computational Science* (praca przyjęta)
- M. Ryczkowska, M. Nowicki, P. Bała, The Performance Evaluation of the Java Implementation of Graph500, In: R. Wyrzykowski et al. (Eds.) *Parallel Processing and Applied Mathematics* Springer 2016 pp. 221-230
- M. Nowicki, M. Ryczkowska, Ł. Górski, M. Szynkiewicz, P. Bała, PCJ - a Java library for heterogenous parallel computing In: X. Zhuang (Ed.) *Recent Advances in Information Science (Recent Advances in Computer Engineering Series vol 36)* WSEAS Press 2016 pp. 66-72
- M. Ryczkowska, Evaluating PCJ library for graph problems - Graph500 in PCJ In: W. W. Smari and V. Zeljkovic (Eds.) *2014 International Conference on High Performance Computing and Simulation (HPCS)* IEEE 2014 pp. 1005-1007