27.01.2014

Paweł Szynkiewicz
Jakub Tyburski

Sprawozdanie z projektu PORR (część I)

Algorytm genetyczny z kodowaniem binarnym

Prowadzący: dr Adam Woźniak

1. Algorytm genetyczny

Algorytmy genetyczne to proste heurystyki populacyjne. Pomysł technik ewolucyjnych pochodzi z symulacji procesu ewolucji żywych organizmów. Stan algorytmu opisuje zbiór odpowiednio zakodowanych łańcuchów (np. binarnych), będących analogiem kodów genetycznych (chromosomów). Do każdego łańcucha jest przypisana miara przystosowania (w zadaniach optymalizacji funkcja celu). Multizbiór łańcuchów (populacja) jest losowo wybierany z przestrzeni poszukiwań i poprzez mechanizmy wyboru, krzyżowania i mutacji ewoluuje w kierunku minimum globalnego funkcji celu.

Prosty algorytm genetyczny GA (Genetic Algorithm) [1] jest skonstruowany z następujących operacji:

Inicjalizacja - polega na wylosowaniu początkowej populacji P. Zmienne zadania optymalizacji muszą być zakodowane w postaci skończonego ciągu znaków z pewnego skończonego alfabetu, np. w postaci łańcucha binarnego: 01101 - 1, 11000 - osobnik 2.

Selekcja - kopiowanie z poprzedniej populacji do nowej może odbywać się według różnych reguł. Warunkiem jest jednak, aby prawdopodobieństwo wyboru osobników o większej mierze przystosowania było większe (przykład - ,,koło ruletki").

Krzyżowanie - zazwyczaj przebiega w dwóch etapach. Najpierw kojarzymy losowo w pary ciągi z populacji rodziców. Następnie, w sposób losowy, z jednakowym prawdopodobieństwem, wybieramy miejsce przecięcia k spośród l-1 początkowych pozycji w ciągu kodowym (np. dla osobników 01101 i 11100 oraz k=3 otrzymujemy osobniki 01100 i 11101.

Mutacja - polega na losowej zmianie jednego bitu ciągu z populacji (odwrócenie bitu).

Krzyżowanie i mutacja są nazywane operacjami genetycznymi. Populacja potomna generowana jest w następujący sposób:

1. **Ewaluacja** – Dla każdego osobnika obliczana jest wartość funkcji przystosowania osobnika.

2. **Selekcja** – Wybór osobników do populacji potomnej, na podstawie ich przystosowania.
3. **Krzyżowanie** – Osobniki wybrane w selekcji są krzyżowane z pewnym prawdopodobieństwem (*prawdopodobieństwo krzyżowania*).
4. **Mutacja** - Każdy allel osobnika z populacji potomnej może zmutować z pewnym prawdopodobieństwem (*prawdopodobieństwo mutacji*)

1.1. Kodowanie

Ważną część algorytmu genetycznego stanowi kodowanie informacji o osobniku do postaci ciągu binarnego. Ponieważ celem algorytmu jest minimalizacja *n*-wymiarowej funkcji rzeczywistoliczbowej, pojedynczy osobnik w algorytmie reprezentuje pewien punkt w przestrzeni $\mathbb{R}^n$. Konieczne jest więc zakodowanie wektora $(x_{n-1}, x_{n-2}, \ldots, x_0)$ liczb rzeczywistych do ciągu binarnego $(b_{m-1}, b_{m-2}, \ldots, b_0)$, $b_i \in \{0,1\}$. Kolejne zmienne rzeczywiste x_j są kodowane do postaci podciągów binarnych, podciągi są następnie konkatenowane do jednego ciągu binarnego. W algorytmie realizowane jest podwójne kodowanie zmiennoprzecinkowych liczb rzeczywistych do postaci binarnej.

- Kodowanie stało-przecinkowe (fix-point) E_{fp}
  Zakłada się, że wartość liczby rzeczywistej będziemy kodować na *K* bitach. Dodatkowo liczba rzeczywista x_i musi należeć do pewnego przedziału $lb \le x_i \le ub$. Do odkodowania liczby rzeczywistej, zakodowanej przez E_{fp} stosuje się wzór:

$$x := lb + E_{fp}(x) \cdot \frac{ub - lb}{2^K - 1}$$

  Kodowanie stało-przecinkowe E_{fp} zapewnia że wszystkie wartości z przedziału (l, u) mają odwzorowanie $1-1$ w przestrzeni *K*-bitowych kodów binarnych.

- Kodowanie Graya E_G
  Kodowanie Graya przekształca standardowy kod binarny na kod Gray. Kod Graya charakteryzuje się tym, że dwa kolejne słowa kodowe różnią się tylko stanem jednego bitu. Jest również kodem cyklicznym. Dzięki tej właściwości niewielkie różnice chromosomów wiążą się z niewielkimi różnicami w fenotypie osobników (co nie byłoby prawdą w przypadku zwykłego kodu binarnego).

Liczba rzeczywista kodowana jest najpierw stało-przecinkowo, a następnie do postaci kodu Graya. Kodowanie binarne w programie to : $E := E_G \circ E_{fp}$

1.2. Selekcja

Selekcja to sposób wyboru osobników do populacji potomnej. Istnieje wiele metod selekcji ogólnie stosowanych w algorytmach genetycznych. W programie zaimplementowano trzy metody selekcji.

- **Proporcjonalna**
  Metoda nazywana metodą ruletki. Osobnik wybierany jest do nowej populacji z prawdopodobieństwem proporcjonalnym do wielkości jego przystosowania.

- **Rankingowa**
  W programie zaimplementowano uproszczoną wersję metody rankingowej. Do populacji potomnej wybierane jest *k* najlepiej przystosowanych osobników i są oni powielani do osiągnięcia pożądanej wielkości populacji potomnej.

- **Turniejowa**
  Metoda turniejowa polega na przeprowadzeniu *n* turniejów na *k* osobnikach z populacji. *n* to rozmiar populacji potomnej, a *k* rozmiar turnieju. Uczestnicy każdego turnieju są wybierani losowo z populacji macierzystej. Zwycięzca każdego z turniejów, czyli najlepiej przystosowany osobnik zostaje wybrany do populacji potomnej. W programie zaimplementowano zmodyfikowaną wersję metody turniejowej. W kolejnych *l-tym* turnieju brany jest pod uwagę najlepszy osobnik wyłoniony w *l* - 1 przeprowadzonych turniejach. Metoda powoduje, że populacja ma małą różnorodności genetyczną (efekt niepożądany w alg. genetycznym), jednak dla większych wymiarów funkcji testowych, daje najlepsze wyniki.

2. Funkcje testowe

Badania prowadzono dla dwóch funkcji testowych, dla różnych wymiarów zadania (*n* = 2, 10, 20, 50, 100).

2.1. Funkcja Griewanka

Określona wzorem:

$$GR(x_i) = \frac{1}{40}\sum_{i=1}^{n}(x_i^2) + 1 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{i}\right)$$

Przy ograniczeniach zmiennych:

$$-40 \le x_i \le 40$$

Przyjmuje minimum globalne w punkcie:

$$f_{min} = 0, \quad dla \quad x_i = 0$$

2.1. Funkcja Ackleya

Określona wzorem:

$$AC(x_i) = -20\ \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e$$

Przy ograniczeniach zmiennych:

$$-30 \le x_i \le 30$$

Przyjmuje minimum globalne w punkcie:

$$f_{min} = 0,\ \ dla\ \ x_i = 0$$

3. Program

W ramach projektu powstał program do minimalizacji wielowymiarowych funkcji rzeczywistych za pomocą algorytmu genetycznego wykorzystującego kodowanie binarne. Program został napisany w języku C w standardzie GNU99 i skompilowany za pomocą kompilatora GCC 4.8.2 w środowisku linux. Wersje równoległe programu wykorzystuję bibliotekę *OpenMP* 3.1, oraz interfejs *MPI*.

3.1. Parametry

Program uruchamiany jest z następującymi parametrami:

1. Parametry zadania:
   - Wybór funkcji do minimalizacji
     - funkcja Griewanka
     - funkcja Ackleya
   - Wymiar funkcji (1..100)
2. Parametry algorytmu genetycznego
   - Całkowitoliczbowe
     - Ilość generacji
     - Rozmiar populacji
     - Parametr metody selekcji
   - Rzeczywistoliczbowe
     - Prawdopodobieństwo krzyżowania
     - Prawdopodobieństwo mutacji

        - Parametr metody selekcji
    - Metoda selekcji
        - Proporcjonalna (ruletka)
        - Rankingowa
        - Turniejowa
3. Parametry wykonania
    - Verbose – komunikaty
    - Seed – ziarno dla losowania liczb pseudolosowych

3.2. Struktura

Główną częścią programu jest pętla powtarzana, aż do wykonania zadanej ilości generacji. W pętli realizowane są kolejne kroki algorytmu genetycznego.

```
For generacja = 0 to MAX_GENERACJA do
        selekcja                    /* wybierz osobniki do nowej populacji */
        generacja                   /* generuj nową populację na podstawie selekcji */
        krzyżowanie                 /* krzyżuj losowo wybrane osobniki w populacji potomnej*/
        mutacja                     /* mutuj losowe geny */
        ewaluacja                   /* oblicz przystosowanie osobników z nowej populacji */
        zapamiętaj najlepszy        /* zapamiętaj najlepiej przystosowanego osobnika */
endfor
```

3.3. Wywołanie

OpenMP

Program wywoływany poleceniem ./[omp]release. W celu zadania wartości parametrom biblioteki OpenMP należy poprzedzić wywołanie programu komendą env,

```
env PARAMETR_OMP="<wartość parametru>" ./[omp]release
```

MPI

Program wywoływany poleceniem ./[mpi]release. Aby uruchomić wywołanie programu na *n* procesach wykonujących się na rdzeniach należy poprzedzić wywołanie programu komendą mpiexec

```
mpiexec -n<ilość procesów> ./[mpi]release
```

```
GENALG – algorytm genetyczny do minimalizacji funkcji Griewanka i
Ackleya

użycie: GENALG [argumenty]

Argumenty:
NAZWA       OPIS                                  DOMYŚLNA
-h          Wyświetla tą wiadomość
-v          verbose – dodatkowe komunikaty
-d[N]       Wymiar funkcji                           2
-e[N]       Seed dla randomizacji                    0
-g[N]       Maksymalna ilość generacji              50
-k[N]       Parametr metody selekcji [INT]           2
-r[N]       Parametr metody selekcji [FLOAT]       0.5
-p[N]       Rozmiar populacji                       20
-m[p]       Prawd. mutacji allela                 0.01
-x[p]       Prawd. krzyżowania dwóch osobników     0.2
-s[R|T|B]   Metoda selekcji osobników               B
-f[GR|AC]   Funkcja do minimalizacji               GA
-i[1|2|3]   Wersja MPI                              1
-t[N]       Interwał migracji genotypów            20
-o[N]       Rozmiar migracji genotypów              2
```

Wypis programu po uruchomieniu w konsoli z flagą -h (help)

4. Realizacja równoległa

Do zrównoleglenia algorytmu zastosowano dyrektywy OpenMP. Zrównoleglono najważniejsze pętle występujące w programie. W pętlach wykonywane są operacje na wszystkich osobnikach w obecnej populacji.

- Ewaluacja populacji – obliczanie przystosowania osobników
  - dekodowanie genotypu
  - obliczanie wartości funkcji testowej
- Mutacja
- Krzyżowanie
- Selekcja
  - Metoda proporcjonalna
  - Metoda turniejowa
    (Nie zrównoleglono metody rankingowej. Zrównoleglenie metody wymagałoby zupełnie innego podejścia implementacyjengo)

Zastosowano różne możliwości rozdzielania pętli:

- schedule (static)
- schedule (dynamic)

Zrównoleglano działanie pętli *for* w funkcjach wykonujących operacje na populacji (plik population.c). Poniżej przedstawiony jest ogólny schemat wykorzystania dyrektyw *OpenMP* w programie. Metoda zrównoleglenia różni się w zależności od tego, czy w pętli generowane są liczby pseudolosowe, i czy aktualizowana jest wartość pewnych dzielonych zmiennych skalarnych:

```
#pragma omp parallel
{
        // Jeżeli w pętli for będą generowane liczby pseudolosowe,
        // do dla każdego wątku należy wyliczyć stan generatora pseudolosowego
        // i zapisać go w zmiennej prywatnej każdego wątku.

        #pragma omp for
        for i ← osobniki w populacji {
                // wykonaj funkcję na pojedynczym osobniku

                #pragma omp critical
                {
                        // aktualizuj wartość skalarnej zmiennej dzielonej (np. przystosowanie
                        // najlepszego osobnika)
                }
        } /* end for */

        #pragma omp master
        {
                // jeżeli w pętli generowano liczby pseudolosowe, to przypisz globalnemu
                // stanowi generatora losowego, wartość jaką osiągną prywatny stan generatora
                // wątku głównego po wykonaniu pętli for.
        }
}
```

5. Realizacja rozproszona

Wersja rozproszona programu wykorzystuje interfejs protokołu *MPI*. Obliczenia są prowadzone przez *n* (=2, 4) różnych procesów. Procesy mogą się wykonywać na różnych komputerach, lub na różnych rdzeniach jednego procesora. Zaproponowano dwa podejścia rozproszenia obliczeń. Są to dwie różne realizacje algorytmu wielopopulacyjnego z modelem wyspowym.

Równoległe wykonywanie szeregowego algorytmu bez komunikacji – (*VMPI1*)

Program uruchamiany jest na *n* procesach dla takich samych parametrów wykonania (poza ziarnem dla generatora liczb losowych). Obliczenia są prowadzone niezależnie. Proces o *ID* = 0 pełni rolę koordynatora. Po zakończeniu swoich obliczeń, koordynator wywołuje funkcję interfejsu *MPI* – MPI_Gather za pomocą której pobiera ostateczne wyniki obliczeń pozostałych procesów. Ostatecznie zwracany jest najlepszy otrzymany wynik.

Migracja najlepszych osobników – (*VMPI2*)

Wersja programu wykonująca się na oddzielnych procesach została wzbogacona o komunikację między procesami. W trakcie kolnych iteracji algorytmu genetycznego, każdy proces wysyła oraz otrzymuje pewną liczbę osobników pochodzących z innego procesu (innej populacji). Do komunikacji wykorzystywane są funkcje interfejsu *MPI* – MPI_Recv, MPI_Send.

Parametry algorytmu wielopopulacyjnego są następujące:

- **Topologia połączeń –** zastosowano topologię pierścienia jednokierunkowego. Każdy proces wysyła ustaloną liczbę osobników z lokalnej populacji od swojego poprzednika. Proces o *ID = n* wysyła osobników do procesu o *ID = 0*.
- **Metoda wyboru osobników migrujących –** wybierane są osobniki najlepiej o największej wartości funkcji przystosowania.
- **Liczebność migracji –** liczba osobników, jaką każdy proces wysyła i otrzymuje w jednej fazie migracji. Parametr ten może zostać zadany przez użytkownika jako argument wywołania programu.
- **Częstość migracji –** co ile iteracji algorytmu genetycznego należy przeprowadzić migrację. Podobnie jak w przypadku liczebności populacji, parametr może zostać dobrany przez użytkownika.

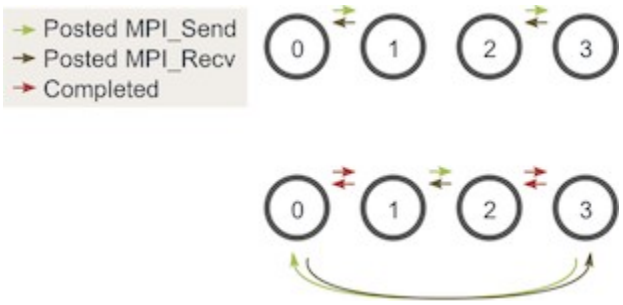
Osobniki otrzymane przez proces w fazie migracji zastępują najgorsze osobniki w populacji lokalnej. Pod koniec wykonania programu, tak jak w podejściu ***VMPI1,*** wyniki otrzymane przez procesy są zbierane i porównywane przez koordynatora.

Zapobieganie zakleszczeniu

Zła implementacja komunikacji przy użytej topologii pierścienia jednokierunkowego, może prowadzić do zakleszczeń wykonania programu. Funkcja MPI_Send blokuje dalsze wykonanie procesu do chwili, aż do chwili gdy bufor służący do wysyłania komunikatów pozwoli na umieszczenia kolejnej wiadomości. Jeżeli bufor jest całkowicie zapełniony, wykonanie funkcji MPI_Send zostanie wstrzymane do chwili zwolenianie się miejsca na buforze, czyli do czasu gdy proces odbierający wywoła funkcję MPI_Recv.

Rysunek 1: Wywołanie funkcji MPI_Send

Rysunek (*Rys. 1*) prezentuje sytuację, w której wszystkie procesy wywołują funkcję MPI_Send w tej samej chwili wykonania. Jeżeli dojdzie do przepełnienia wszystkich buforów komunikacyjnych wszystkie procesy zawieszą się na funkcji MPI_Send w oczekiwaniu na odebranie komunikatu przez odbiorcę (MPI_Recv). Nastąpi zakleszczenie programu. Rozmiar bufora zależy od lokalnej implementacji biblioteki *MPI* oraz od sieci, w której prowadzona jest komunikacja. Przyjęcie założeń co do rozmiaru byłoby błędem, ponieważ w ogólności, program mógłby działać błędnie.



Rysunek 2: Wywołanie funkcji MPI_Send i MPI_Recv naprzemiennie

Na rysunku (*Rys. 2*) pokazano rozwiązanie problemu zakleszczenia w pierścieniu jednokierunkowym. Zakłada ono, że w pierwszym kroku komunikacji procesy o parzystym *ID* wywołują funkcję , a o nieparzystym funkcję . W drugim kroku ma miejsce sytuacja odwrotna. Ponieważ w obu fazach ma miejsce co najmniej jedno nadanie i odebranie komunikatu, jeżeli obie funkcje wykonają się poprawnie, nigdy nie dojdzie do przepełnienia wszystkich buforów komunikacji, a co za tym idzie do zakleszczenia programu [2].

6. Testy algorytmu genetycznego

Wykonano testy algorytmu genetycznego dla funkcji Griewanka i Ackleya. Rozważano różne wymiary zadania (n = 2, 10, 20, 50, 100).
Wszystkie testy wykonano na 4-rdzeniowym komputerze o procesorze Intel Core 2 Quadro, z pamięcią 4 GB RAM, pod systemem Linux Ubuntu 11.10

Ze względu na występowanie w algorytmie losowania (przy każdym uruchomieniu jest losowana inna populacja) większość testów została uruchomiony 5 razy. W tabelach z wynikami podane są

wyniki: średni z pięciu uruchomień, najlepszy i najgorszy.

Badano dokładność rozwiązania (wyznaczone minimum globalne) oraz wydajność algorytmu.

W tabelach i na rysunkach prezentowane są wyniki badań. Symbole w tabelach oznaczają:

n – wymiar zadania,
P – liczność populacji,
G(min) – najlepszy wynik z 5 uruchomień (wartość funkcji testowej w minimum),
G(max) – najgorszy wynik z 5 uruchomień (wartość funkcji testowej w minimum),
G(ev) – uśredniony wynik z 5 uruchomień,
T(ev) – uśredniony czas obliczeń,

5.1. Algorytm sekwencyjny

Celem testów algorytmu sekwencyjnego było:

- Zbadanie dokładności rozwiązania dla różnych wymiarów zadania.
- Zbadanie wydajności algorytmu dla różnych wymiarów zadania.
- Zbadanie wpływu rozmiaru populacji na rozwiązanie.
- Porównanie rozwiązań i wydajności algorytmu dla różnych metod selekcji.

Dla wszystkich testów wykonano 100 iteracji algorytmu genetycznego. Przyjęto prawdopodobieństwo krzyżowania równe 0.25 i prawdopodobieństwo mutacji 0.003.

Funkcja Griewanka					
n	*populacja*	*G (min)*	*G (max)*	*G (ev)*	*T (ev)*
2	500	0.0	1.05	0.21	0.46
10	1000	0.0	0.0	0.0	0.58
20	2000	0.0	0.0	0.0	1.63
50	5000	0.13	1.05	0.51	6.59
100	10000	5.66	8.71	7.30	29.37

Funkcja Ackleya					
n	*populacja*	*G (min)*	*G (max)*	*G (ev)*	*T (ev)*
2	500	0.0	6.56	2.02	0.04
10	2000	0.02	2.01	0.41	0.6
20	3000	0.01	0.02	0.01	1.72
50	5000	2.79	3.93	3.3	6.95
100	10000	6.16	10.82	8.21	28.05

Tabela 1: Wyniki algorytmu (wyznaczone rozwiązania i czasy działania) dla różnych wymiarów zadania, selekcja turniejowa.

Rysunek 3: Czas wykonania funkcji testowych dla różnych wymiarów zadania

Funkcja Griewanka				
P	*G (min)*	*G (max)*	*G (ev)*	*T (ev)*
5000	10.58	20.4	12.96	13.12
10000	5.17	8.65	7.02	26.07
20000	2.27	3.84	3.41	52.39

Funkcja Ackleya				
P	*G (min)*	*G (max)*	*G (ev)*	*T (ev)*
5000	8.8	11.35	9.84	14.06
10000	6.52	9.48	8.11	28.05
20000	5.7	6.23	5.88	56.27

Tabela 2: Wpływ liczności populacji na jakość rozwiązania i czas obliczeń (n=100, selekcja

turniejowa).

Rysunek 4: Czas wykonania funkcji testowych dla różnych rozmiarów populacji

Funkcja Griewanka				
selekcja	*G (min)*	*G (max)*	*G (ev)*	*T (ev)*
Proporcjonalna	644.33	735.47	701.62	31.64
Rankingowa	124.96	155.90	143.26	26.59
Turniejowa	4.16	9.88	6.28	26.15

Funkcja Ackleya				
selekcja	*G (min)*	*G (max)*	*G (ev)*	*T (ev)*
Proporcjonalna	20.9	20.7	20.94	33.79
Rankingowa	17.46	17.65	17.59	29.04
Turniejowa	7.39	8.55	7.71	28.34

Tabela 3: Wpływ metody selekcji na jakość rozwiązania i czas obliczeń (n=100, P=10000).

Wnioski

Najlepsze wyniki otrzymano korzystając z turniejowej metody selekcji. Metoda ta ma też najszybszy czas wykonania. Jakość wyników spadała wraz ze wzrostem wymiarowości funkcji testowych. Wyniki próbowano poprawiać, zwiększając populację, co przyniosło względnie dobre efekty. Wpływ wielkości populacji na jakość wyników jest widoczny w *Tabeli 2*. Można jednak zauważyć, że zwiększanie populacji, przy wzroście wymiarowości zadania, znacznie spowalnia czas działania programu (*Tabela 1*).
Na podstawie otrzymanych wyników, w celu otrzymania jak najbardziej zadowalającego rozwiązania, zaleca się korzystanie z turniejowej metody selekcji. Należy dobrać wielkość populacji stosownie do wymiaru zadania. Warto mieć jednak na uwadze fakt, że metoda turniejowa; mimo iż zapewnia najszybszą zbieżność algorytmu; może okazać się niewydolna, przy minimalizacji funkcji z bardziej wydatnymi minimami lokalnymi. Wynika to z małej różnorodności genetycznej, którą metoda wprowadza do populacji.

5.2. Algorytm równoległy - OpenMP

Celem testów algorytmu równoległego było:

- Zbadanie wpływu zrównoleglenia algorytmu na wydajność w zależności od wymiaru i parametrów zadania.
- Zbadanie wpływu metody rozdziału pętli „for” na szybkość rozwiązania.

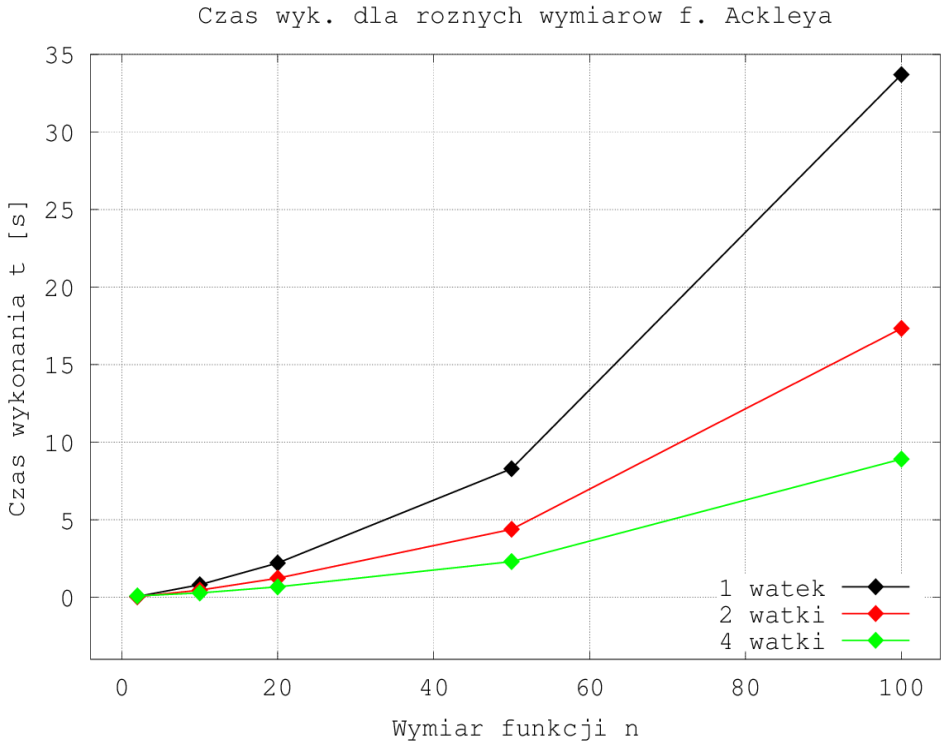
Funkcja Griewanka							
n	*P*	*Sekwencyjny*		*Równoległy (2 wątki)*		*Równoległy (4 wątki)*	
		T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5	500	0.05	0.0	0.04	0.0	0.05	0.0
10	2000	0.8	4.95	0.45	5.18	0.28	4.57
20	3000	2.17	32.63	1.18	30.71	0.65	30.54
50	5000	8.39	223.29	4.16	244.29	2.18	217.01
100	10000	31.52	696.13	16.2	702.25	8.36	722.71

Funkcja Ackleya							
n	*P*	*Sekwencyjny*		*Równoległy (2 wątki)*		*Równoległy (4 wątki)*	
		T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5	500	0.05	0.02	0.04	0.01	0.09	0.03
10	2000	0.81	3.58	0.46	4.16	0.28	3.99
20	3000	2.21	10.74	1.23	11	0.67	11.23
50	5000	8.29	20.26	4.38	20.38	2.3	20.33
100	10000	33.7	21.7	17.33	20.94	8.91	20.94

Tabela 4: Porównanie wydajności algorytmu sekwencyjnego i równoległego (różne wymiary

zadania i liczności populacji, selekcja proporcjonalna).

Rysunek 5: Czas wykonania funkcji testowych dla różnych wymiarów zadania, porównanie zrównoleglenia – funkcja Griewanka

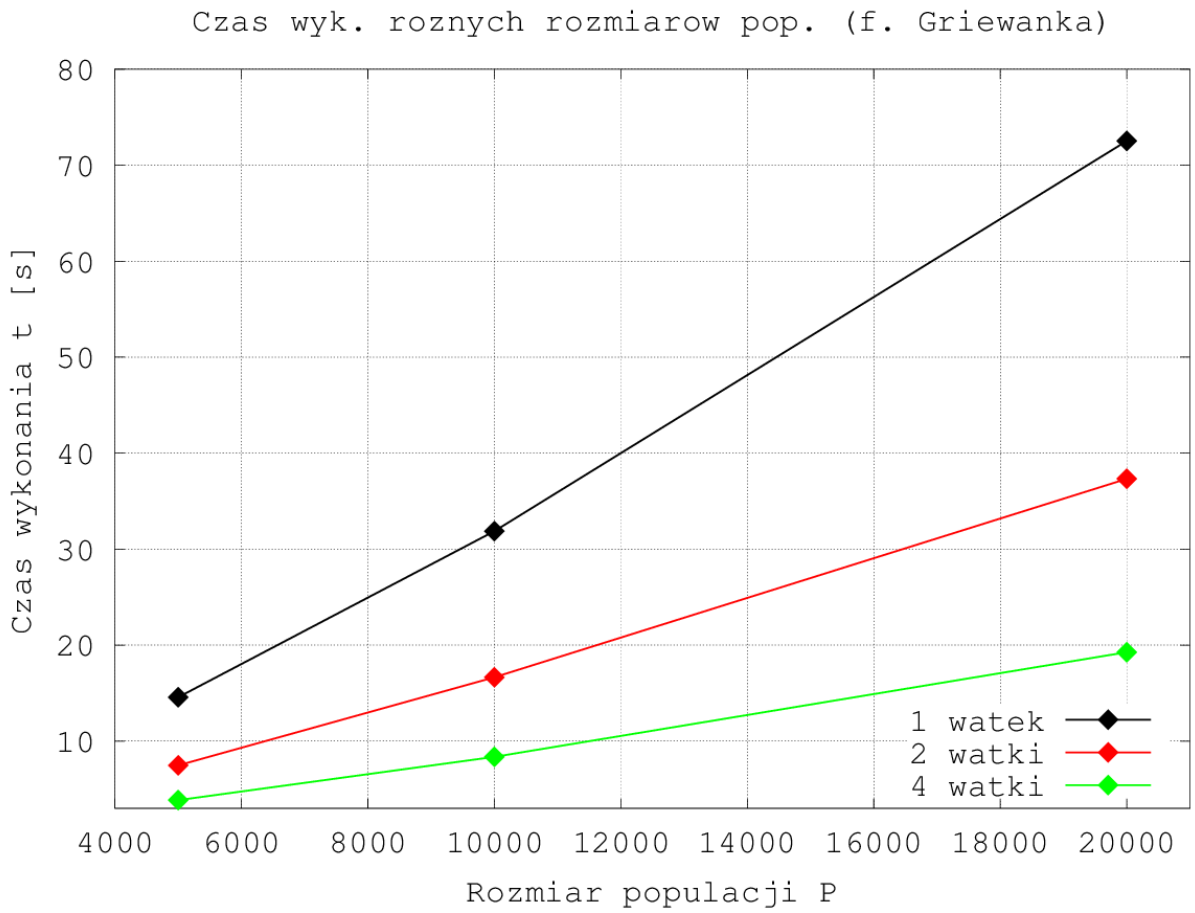


Rysunek 6: Czas wykonania funkcji testowych dla różnych wymiarów zadania, porównanie zrównoleglenia – funkcja Ackleya

Funkcja Griewanka						
P	*Sekwencyjny*		*Równoległy (2 wątki)*		*Równoległy (4 wątki)*	
	T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5000	14.58	723.8	7.47	711.08	3.85	693
10000	31.87	707.02	16.65	688.5	8.36	704.54
20000	72.53	685.46	37.33	684.17	19.27	683.24

Funkcja Ackleya						
P	*Sekwencyjny*		*Równoległy (2 wątki)*		*Równoległy (4 wątki)*	
	T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5000	15.6	20.95	8.03	20.95	4.12	20.95
10000	33.81	20.95	17.3	20.94	9.8	20.92
20000	76.97	20.94	39.5	20.93	20.39	20.92

Tabela 5: Zależność przyspieszenia obliczeń od liczności populacji (n=100, selekcja proporcjonalna).



Rysunek 7: Czas wykonania funkcji testowych dla różnych rozmiarów populacji, porównanie

zrównoleglenia – funkcja Griewanka

Rysunek 8: Czas wykonania funkcji testowych dla różnych rozmiarów populacji, porównanie zrównoleglenia – funkcja Ackleya

Funkcja Griewanka	
Schedule	*T (ev)*
static	6.96
dynamic	7.07
static, kwant = 1000	8.14
dynamic, kwant = 1000	8.08

Tabela 6: Wpływ metody rozdziału pętli na przyspieszenie obliczeń w wersji równoległej (n=100, P=10000, selekcja proporcjonalna, 4 wątki)

Wnioski
Zrównoleglenie programu za pomocą dyrektyw biblioteki *OpenMP* przyniosło zadowalające efekty. Otrzymane przyspieszenie jest bliskie *p*-krotnemu, gdzie *p* to liczna rdzeni (oraz tworzonych wątków) na maszynie testowej i wynosi średnio 3.79. Przyspieszenie działania programu zrównoleglonego zaczyna być widoczne już od *n* większego do 10 i rozmiaru populacji *P* ponad 2000. Dla problemu mniejszej skali tworzenie wątków nie jest opłacalne (*Tabela 4*).
Dla wszystkich wielkości populacji w *Tabeli 5* obserwowane jest podobne przyspieszenie wersji równoległe, dla obu funkcji testowych.
Ingerowanie w sposób podziału pracy pętli for (*Tabela 6*) okazało się nieopłacalne. Najlepsze wyniki daje standardowy tryb *static*. Wynika to z faktu, że wszystkie iteracje pętli powinny mieć podobny czas wykonania i kompilator jest w stanie dobrze rozdzielić zadania już w czasie kompilacji.

Uzyskane przyspieszenie obliczeń *PS* wyznaczone z wzoru:

$$PS = \frac{czas\,obliczeń\,(na\ jednym\ wątku)}{czas\,obliczeń\,(na\ czterech\ wątkach)} \simeq 3.79 \leq 4$$

Uwaga
Ponieważ algorytm genetyczny jest algorytm probabilistycznym, jednym z ważniejszych narzędzi koniecznych do jego poprawnego działania, jest generator liczb pseudolosowych. W równoległej wersji programu należało zaistniała potrzeba skorzystania z bezpieczniej wątkowo funkcji generującej liczby pseudolosowe. Popularna funkcja biblioteki standardowej języka C – *rand()* nie spełnia tych wymagań. Stan generatora tej funkcji jest zmienną globalną, niechronioną żadnymi narzędziami synchronizacji. Użycie tej funkcji wraz z dyrektywami *OpenMP* skutkuje bardzo wolnym działaniu programu. Dlatego w programie użyto funkcji z rodziny dran48, które stan generatora dostają w argumencie wywołania.

5.3. Algorytm rozproszony – MPI

Celem testów algorytmu rozproszonego było:

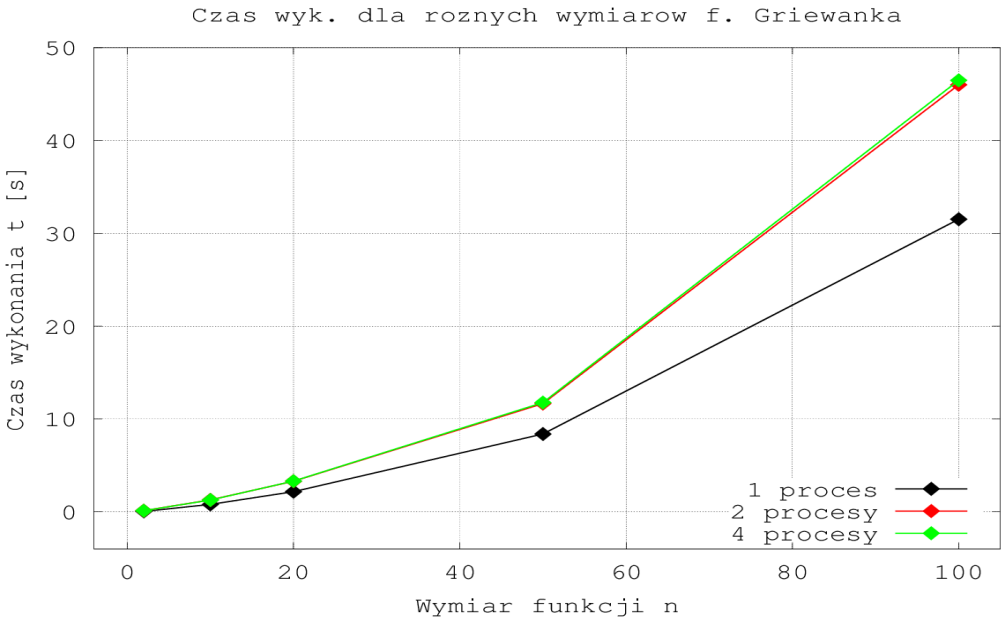
- Zbadanie wpływu rozproszenia algorytmu na wydajność i jakość w zależności od wymiaru i parametrów zadania.
- Zbadanie jak rodzaj algorytmu wielopopulacyjnego z modelem wyspowym wpływa na szybkość i jakość rozwiązania rozwiązania.

Środowisko rozproszone symulowano na jednej maszynie wielordzeniowej. Każdy proces wykonywał się na oddzielnym rdzeniu.

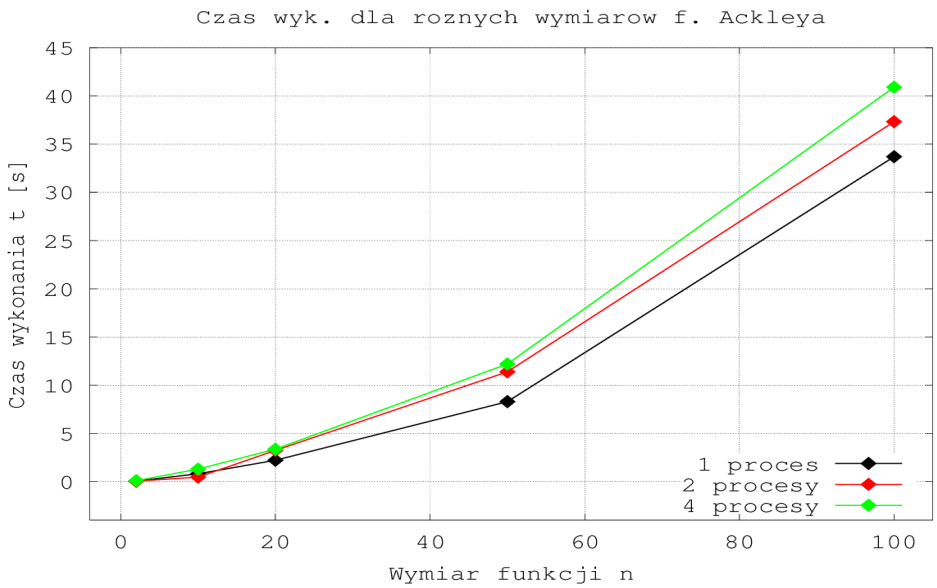
Funkcja Griewanka							
n	*P*	*Sekwencyjny*		*Rozproszony (2 proc.)*		*Rozproszony (4 proc.)*	
		T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5	500	0.05	0.0	0.13	0.0	0.13	0.0
10	2000	0.8	4.95	1.29	2.27	1.27	2.1
20	3000	2.17	32.63	3.3	33.94	3.31	24.78
50	5000	8.39	223.29	11.66	183.99	11.77	171.2
100	10000	31.52	696.13	46.0	689.72	46.48	652.12

Funkcja Ackleya							
n	*P*	*Sekwencyjny*		*Rozproszony (2 proc.)*		*Rozproszony (4 proc.)*	
		T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5	500	0.05	0.02	0.04	0.72	0.09	0.04
10	2000	0.81	3.58	0.46	4.05	1.29	3.86
20	3000	2.21	10.74	3.23	11.11	3.36	10.35
50	5000	8.29	20.26	11.38	20.48	12.2	20.28
100	10000	33.7	21.7	37.33	20.89	40.91	20.87

Tabela 7: Porównanie wydajności algorytmu sekwencyjnego i równoległego bez migracji (różne wymiary zadania i liczności populacji, selekcja proporcjonalna). Rozmiar populacji, w każdym z procesów jest równy rozmiarowi populacji w wersji sekwencyjnej.



Rysunek 9: Czas wykonania funkcji testowych dla różnych wymiarów zadania, algorytm rozproszony bez migracji. Rozmiar populacji, w każdym z procesów jest równy rozmiarowi populacji w wersji sekwencyjnej. – funkcja Griewanka



Rysunek 10: Czas wykonania funkcji testowych dla różnych wymiarów zadania, algorytm rozproszony bez migracji. Rozmiar populacji, w każdym z procesów jest równy rozmiarowi populacji w wersji sekwencyjnej. – funkcja Ackleya

Funkcja Griewanka									
n	*P1*	*Sekwencyjny*		*P2*	*Rozproszony (2 proc.)*		*P4*	*Rozproszony (4 proc.)*	
		T (ev)	*G (ev)*		*T (ev)*	*G (ev)*		*T (ev)*	*G (ev)*
5	500	0.05	0.0	250	0.05	0.4	125	0.03	0.7
10	2000	0.8	4.95	1000	0.49	4.86	500	0.21	5.83
20	3000	2.17	32.63	1500	1.34	32.82	750	0.76	34.31
50	5000	8.39	223.29	2500	5.02	242.37	1250	2.29	247.76
100	10000	31.52	696.13	5000	19.63	702.44	2500	8.98	667.52

Funkcja Ackleya									
n	*P1*	*Sekwencyjny*		*P2*	*Rozproszony (2 proc.)*		*P4*	*Rozproszony (4 proc.)*	
		T (ev)	*G (ev)*		*T (ev)*	*G (ev)*		*T (ev)*	*G (ev)*
5	500	0.05	0.02	250	0.06	0.81	125	0.02	1.1
10	2000	0.81	3.58	1000	0.51	4.26	500	0.23	4.95
20	3000	2.21	10.74	1500	1.37	12.8	750	0.61	11.74
50	5000	8.29	20.26	2500	5.19	20.11	1250	2.39	20.32
100	10000	33.7	21.7	5000	20.64	20.95	2500	9.54	20.93

Tabela 8: Porównanie wydajności algorytmu sekwencyjnego i rozproszonego bez migracji (różne wymiary zadania i liczności populacji, selekcja proporcjonalna). Rozmiar zsumowanych populacji wszystkich procesów jest równy rozmiarowi populacji w wersji sekwencyjnej.

Rysunek 11: Czas wykonania funkcji testowych dla różnych wymiarów zadania, algorytm rozproszony bez migracji. Rozmiar zsumowanych populacji wszystkich procesów jest równy rozmiarowi populacji w wersji sekwencyjnej. – funkcja Griewanka

Rysunek 12: Czas wykonania funkcji testowych dla różnych wymiarów zadania, algorytm rozproszony bez migracji. Rozmiar zsumowanych populacji wszystkich procesów jest równy rozmiarowi populacji w wersji sekwencyjnej. – funkcja Ackleya

Funkcja Griewanka						
P	*Sekwencyjny*		*Rozproszony (2 proc.)*		*Rozproszony (4 proc.)*	
	T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5000	14.58	723.8	19.64	730.97	19.76	705.08
10000	31.87	707.02	45.77	679.61	46.36	640.66
20000	72.53	685.46	117.44	589.74	119.63	611.08

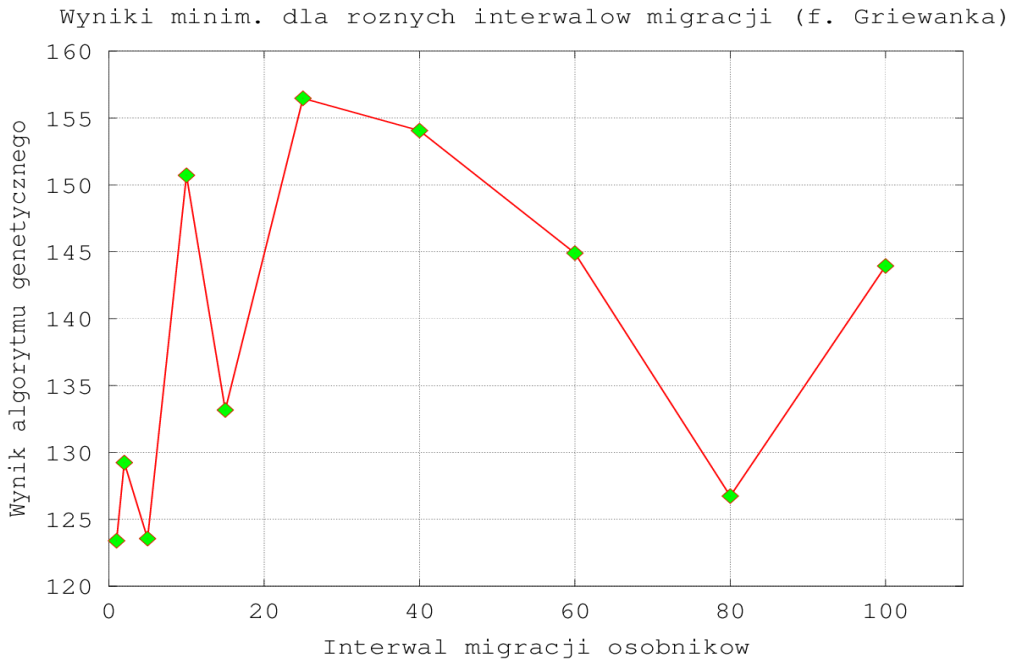
Funkcja Ackleya						
P	*Sekwencyjny*		*Rozproszony (2 proc.)*		*Rozproszony (4 proc.)*	
	T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5000	15.6	20.95	20.56	20.99	20.8	20.88
10000	33.81	20.95	47.62	20.86	48.31	20.84
20000	76.97	20.94	122.2	20.79	123.54	20.81

Tabela 9: Czas obliczeń i jakość wyniku w zależności od rozmiaru populacji, algorytm rozproszony z migracją (*n*=100, selekcja proporcjonalna, interwał migracji = 15, rozmiar migracji = 10).

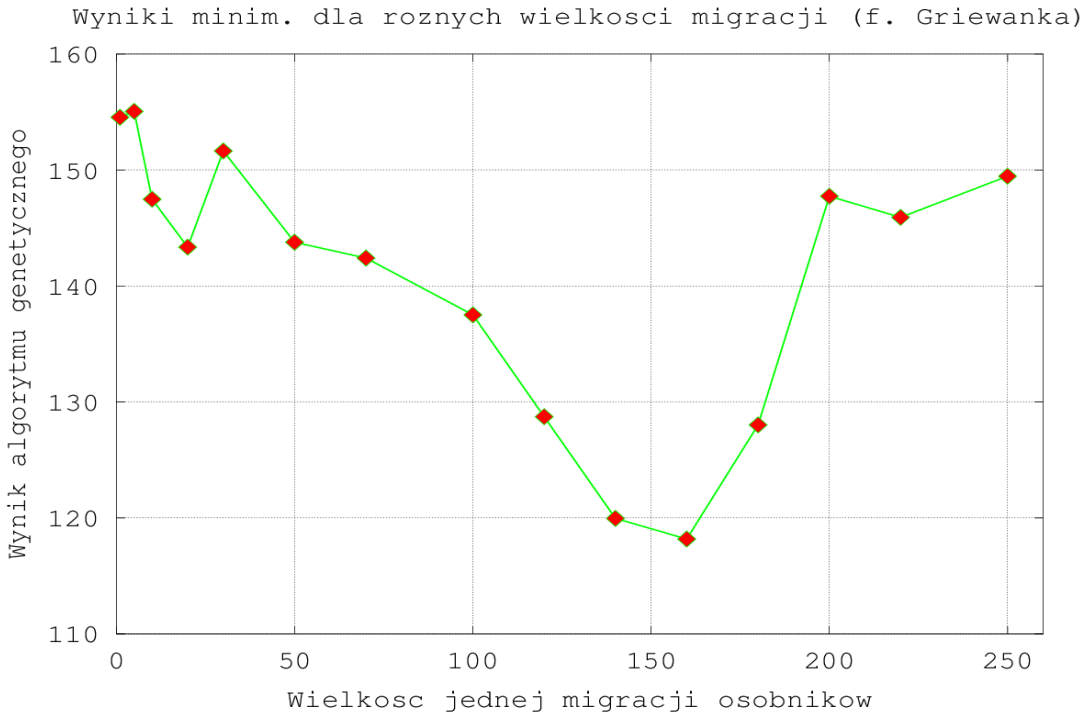
Funkcja Griewanka						
P	*Sekwencyjny*		*Rozproszony (2 proc.)*		*Rozproszony (4 proc.)*	
	T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5000	14.58	191.01	16.14	175.06	16.26	175.95
10000	33.87	181.21	37.61	175.37	39.9	164.29
20000	65.53	177.46	66.09	172.09	65.59	162.56

Funkcja Ackleya						
P	*Sekwencyjny*		*Rozproszony (2 proc.)*		*Rozproszony (4 proc.)*	
	T (ev)	*G (ev)*	*T (ev)*	*G (ev)*	*T (ev)*	*G (ev)*
5000	15.6	18.21	17.25	17.9	17.39	17.01
10000	33.81	18.12	39.75	17.62	42.66	17.13
20000	70.97	17.91	82.55	17.79	85.13	16.91

Tabela 10: Czas obliczeń i jakość wyniku w zależności od rozmiaru populacji, algorytm rozproszony z migracją (*n*=100, selekcja rankingowa interwał migracji = 15, rozmiar migracji = 10).



Rysunek 13: Jakość wyniku w zależności od okresu (interwału) co jaki ma miejsce migracja osobników . – funkcja Griewanka (populacja = 2000, liczba iteracji = 200, metoda selekcji rankingowa)



Rysunek 14: Jakość wyniku w zależności od ilości osobników przesyłanych w fazie migracji. – funkcja Griewanka (populacja = 2000, liczba iteracji = 200, metoda selekcji rankingowa)

Funkcja Griewanka				
Selekcja	*Sekwencyjny*		*Rozproszony (4 proc.)*	
	T (ev)	*G (ev)*	*T (ev)*	*G (ev)*
Proporcjonalna	13.99	696.80	14.07	636.28
Rankingowa	12.81	167.62	12.85	145.44
Turniejowa	12.83	2.82	13.38	0.63

Tabela 11: Porównanie wyników algorytmu sekwencyjnego i rozproszonego z migracją, dla różnych metod selekcji osobników – funkcja Griewanka (populacja = 2000, liczba iteracji = 200, interwał migracji = 10, rozmiar migracji = 50)

Wnioski

Testy wykazały, że czas wykonania algorytmu rozproszonego bez migracji jest większy niż czas wykonania algorytmu sekwencyjnego (*Tabela 7*). Jest to zgodne z oczekiwaniami, ponieważ algorytm rozproszony bez migracji, jest równoważny jednoczesnemu wykonaniu algorytmu sekwencyjnego na *n* procesach. Testy obu algorytmów przeprowadzono na tej samej maszynie wielordzeniowej, w przypadku testów algorytmu wykorzystującego *MPI,* symulowano środowisko rozproszone wykorzystując wiele rdzeni. Różniąca czasów wykonania rosła wraz z wielkością populacji. Z tabeli (*Tabela 7*) wynika jednak, że algorytm rozproszony otrzymywał lepsze wyniki.

W tabeli (*Tabela 8*) zamieszczono wyniki testów w algorytmu wielopopulacyjengo i porównano je z wynikami algorytmu sekwencyjnego działającego na populacji o rozmiarze równym sumie wszystkich populacji w algorytmie rozproszonym. Wyniki otrzymane przez algorytmy, dla takich samych wymiarów zadania, są porównywalne. Zmniejszenie populacji dla pojedynczego procesu, znacznie wpłynęło na czas działania algorytmu rozproszonego. Wykonuje się średnio *n*-razy szybciej od algorytmu sekwencyjnego, gdzie *n* to liczba równocześnie wykonujących się procesów.

Wprowadzenie migracji osobników ma na celu poprawienie wyników otrzymywanych przez algorytm genetyczny. Tabele (*Tabela 9, 10*) pokazują na ile lepsze okazały się średnie wyniki otrzymywane przez algorytm rozproszony z migracją, od wyników zwykłego algorytmu sekwencyjnego. Poprawa jest widoczna głównie dla funkcji Griewanka i selekcji proporcjonalnej. Pozostałe wyniki stanowią niewielką poprawę, a czas wykonania algorytmu rozproszonego jest zauważalnie większy.

Wzbogacenie algorytmu wielopopulacyjnego o migrację osobników skutkuje poprawą jakości rozwiązań [3]. Należy jednak odpowiednio dobrać parametry migracji. Rozpatrujemy dwa parametry migracji osobników: interwał migracji oraz wielkość pojedynczej migracji. Na wykresie (*Rys. 13*) zaznaczono wyniki otrzymywane przy różnych interwałach migracji. Ciężko jest określić zależność jakości wyniku od wielkości interwału [3]. Rozbieżne wyniki widoczne na wykresie związane są raczej z niedeterministyczną naturą algorytmu genetycznego. Inaczej jest w przypadku rozmiaru migracji. Parametr ten znacznie wpływa na jakość otrzymywanych wyników. Nie jest jednak jasne w jaki sposób należy dobierać ten parametr. Zbyt mały jak i zbyt duży rozmiar migracji może prowadzić do pogorszenia się wyników. Wykres (*Rys. 14*) pokazuje wyniki otrzymywane przy różnych rozmiarach migracji. Ogólnie stwierdzono, że rozmiar migracji nie powinien przekraczać 10% populacji. Zbyt duży rozmiar migracji spowalnia także

działanie programu, ponieważ wprowadza narzuty związane z przesyłaniem osobników do procesów.

Podsumowując, algorytm rozproszony daje lepsze wyniki od algorytmu sekwencyjnego. Zwłaszcza wprowadzenie migracji zdaje się być obiecującą metodą poprawienia jakości otrzymywanych rozwiązań (*Tabela 11*). Dobranie odpowiednich parametrów migracji wymaga jednak przeprowadzenia wyczerpujących testów.

Bibliografia

[1] Michalewicz Z., Algorytmy genetyczne + struktury danych = programy ewolucyjne, WNT, 1996.

[2] Kendall W., A Comprehensive MPI Tutorial Resource, http://mpitutorial.com/

[3] http://algorytmy-genetyczne.eprace.edu.pl