Testowanie przypuszczenia Beal'a z wykorzystaniem klasycznych procesorów

Testing Beal conjecture using classical processors

Monika Kwiatkowska[1] i Łukasz Świerczewski[2]

Treść: Praca obejmuje testowanie przypuszczenia Beal'a z wykorzystaniem klasycznych procesorów. Dodatkowo w wybranych funkcjach oprogramowania wykorzystano standard OpenMP, co umożliwiło zrównoleglenie obliczeń. Do obliczeń wykorzystano jednostki obliczeniowe wchodzące w skład komputerów IBM Blue Gene/P, IBM Blue Gene/Q oraz IBM Power 775. Testy wykonano także na superkomputerze HP BladeSystem/Actina, Hydra dostępnym w ICM UW - użyto tam węzła obliczeniowego posiadającego dwa procesory Intel Xeon X5660. Porównano wydajność własnych rozwiązań napisanych w języku C z możliwościami oprogramowania napisanego w języku Python przez Peter'a Novig'a.

Słowa kluczowe: przypuszczenie Beal'a, Blue Gene/P, Blue Gene/Q, Power 775

Abstarct: This paper includes the testing of Beal's conjecture using classical processors. Additionally some features of OpenMP standard were used in software what allowed to parallel the calculation. Calculations were based on computational units included in the computers IBM Blue Gene/L, IBM Blue Gene/Q, and the IBM Power 775 tests have been performed on the supercomputer HP BladeSystem / Actina, Hydra available in the ICM UW - computing nodes with Intel Xeon processors X5660 were used there. The performance of own solutions written in C was compared with the capabilities of software written in Python by Peter Novig.

Keywords: Beal conjecture, Blue Gene/P, Blue Gene/Q, Power 775

Acknowledgment

Interdisciplinary Centre for Mathematical and Computational Modeling (ICM), Warsaw University, Poland is acknowledged for providing the computer facilities under the Grant No. G55-11.

Wprowadzenie

Przypuszczenie Beal'a jest nieudowodnionym twierdzeniem matematycznym z teorii liczb. Mówi ono, że jeśli:

$x^m + y^n = z^r$

gdzie x, y, z, m, n oraz r są dodatnimi liczbami całkowitymi, oraz m, n, $r > 2$, to x, y, z mają wspólny dzielnik będący liczbą pierwszą. Z powyższego wynika, że nie znajdziemy rozwiązania powyższego równania dla wartości x, y, z, które są parami względnie pierwsze.
Przypuszczenie zostało sformułowane w roku 1993 przez Andrew Beal'a podczas jego prac nad uogólnieniami twierdzenia Fermata. Ufundował on w roku 1997 nagrodę w wysokości $5000 za dostarczenie dowodu lub kontrprzykładu dla swojej teorii. Na przestrzeni lat nagroda była kilkukrotnie podnoszona i w tej chwili (rok 2015) wynosi $1000000.

Rozwiązanie Peter'a Norvig'a

Peter Norvig na swojej stronie [4] zaprezentował rozwiązanie w języku Python analizujące przypuszczenie Beal'a. Jeden z dwóch kodów źródłowych (ten bardziej zoptymalizowany) zaprezentowano na Listingu 1.

```
1  def beal(max_base, max_power):
2      bases, powers, table, pow = initial_data(max_base, max_power)
3      for x in bases:
4          powx = pow[x]
5          for y in bases:
6              if y > x or gcd(x,y) > 1: continue
7              powy = pow[y]
8              for m in powers:
9                  xm = powx[m]
10                 for n in powers:
11                     sum = xm + powy[n]
12                     r = table.get(sum)
13                     if r: report(x, m, y, n, nth_root(sum, r), r)
```

Listing. 1. Rozwiązanie w języku Python zaproponowane przez Peter'a Norvig'a.
Listing 1. The solution in Python programming language proposed by Peter Norvig.

W Tab. 1. zaprezentowano główne wyniki jakie uzyskał Peter Norvig [4] realizując kod przedstawiony na Listingu 1. Podczas obliczeń wykorzystano interpreter języka Python w wersji 1.5 oraz procesor o częstotliwości taktowania 400 MHz.

1. Uniwersytet Marii Curie-Skłodowskiej w Lublinie, Wydział Matematyki, Fizyki i Informatyki
2. Państwowa Wyższa Szkołą Informatyki i Przedsiębiorczości w Łomży, Instytut Automatyki i Robotyki

Tab. 1. Czasy realizacji rozwiązania Peter'a Norvig'a przedstawione w [4]. Czasy mierzone w godzinach.
Tab. 1. Execution times of Peter Norvig solution presented in [4]. Times measured in hours.

	max_ power=7	max_ power=10	max_ power=30	max_ power=100	max_ power=1000
max_base=100	-	-	-	0,6	19,0
max_base = 1000	-	-	-	6,2	?
max_ base=10000	-	-	52,0	993,0	?
max_ base=100000	-	109,0	?	?	?
max_ base=250000	1323,0	?	?	?	?

Własne wyniki czasowe dla tego samego kodu realizowanego jednak na procesorze Intel Xeon X5660 z wykorzystaniem interpretera Python w wersji 2.6.6 zaprezentowano w Tab. 2.

Tab. 2. Czasy realizacji rozwiązania Peter'a Norvig'a z wykorzystaniem procesora Intel Xeon X5660 i Pythona w wersji 2.6.6. Czasy mierzone w godzinach.
Tab. 2. Execution times of Peter Norvig solution with the use of Intel Xeon X5600 processor and 2.6.6. version of Python. Times measured in hours.

	max_ power=7	max_ power=10	max_ power=30	max_ power=100	max_ power=1000
max_ base=100	-	-	-	0,003	0,527
max_ base = 1000	-	-	-	0,968	?
max_ base=10000	-	-	42,257	> 168,000	?
max_ base=100000	-	> 168,00	?	?	?
max_ base=250000	> 168,00	?	?	?	?

Jak widać uzyskane przyśpieszenie jest w dużym stopniu zależne od parametrów z jakimi był uruchamiany program. Na procesorze 400 MHz i Pythonie 1.5 dla parametrów max_power=100 oraz max_base=100 czas wykonywania wyniósł 0.6 godziny. Po zastosowaniu nowszej wersji Pythona i procesora Intel Xeon X5660 czas ten spadł do 12.86 sekundy (0.003 godziny). Daje to więc przyśpieszenie równe aż równo 200 razy. Jeżeli jednak uwzględnimy większy przedział dla podstaw (max_base=1000) to wartość przyśpieszenia spadnie do 6.404.

Wersja pierwsza własnego rozwiązania

Zaimplementowano funkcję o poniższym nagłówku:

```
unsigned int beal_conjecture_test(
        unsigned long long int min_base,
        unsigned long long int max_base,
        unsigned long long int min_power,
        unsigned long long int max_power,
        beal_test_result_t** results);
```

Gdzie *beal_test_result_t* jest strukturą, w której będą ewentualnie przechowywane odnalezione kontrprzykłady. Struktura ta zdefiniowana jest następująco:

```
typedef struct beal_test_result {
        unsigned long long int x;
        unsigned long long int y;
        unsigned long long int z;
        unsigned long long int m;
        unsigned long long int n;
        unsigned long long int r;
} beal_test_result_t;
```

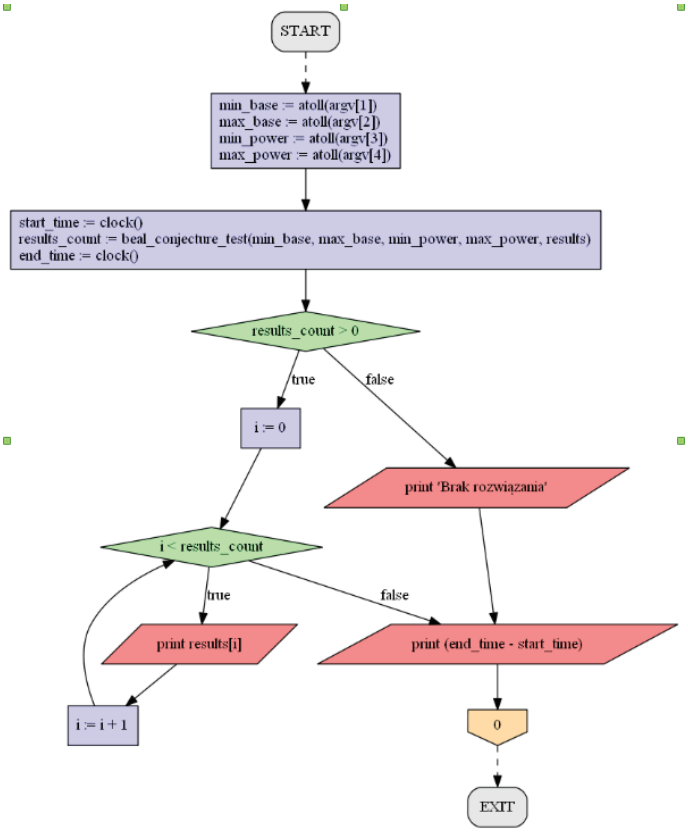
Grupuje ona wartości wszystkich podstaw i potęg równania $x^m + y^n = z^r$.

Funkcja *beal_conjecture_test* wykonuje zasadniczą część analizy Przypuszczenia Beala. Dla podanych przedziałów wartości podstaw (*min_base, max_base*) i potęg *(min_power, max_power*) generowane są wszystkie możliwe kombinacje wartości x, y, z, m, n, r z równania $x^m + y^n = z^r$. Jeśli po podstawieniu (z uwzględnieniem, że x, y, z muszą być parami względnie pierwsze) wartości równanie jest spełnione, oznacza to, że został znaleziony kontrprzykład. Wartości wszystkich zmiennych równania zapisywane są wtedy w strukturze *beal_test_result_t* i umieszczane w tablicy takich struktur. Po zapisaniu danych znalezionego rozwiązania, funkcja kontynuuje poszukiwania kolejnych rozwiązań. Po zbadaniu wszystkich kombinacji, tablica struktur jest zwracana przez jeden z parametrów funkcji, a wartość będąca wynikiem funkcji określa ilość znalezionych kontrprzykładów.
W pierwszej kolejności rozpatrywane są wszystkie kombinacje wartości podstaw x i y (dwie najbardziej zewnętrzne pętle *for*). Dla tych par, które są względnie pierwsze ($gcd(x,y)==1$), sprawdzane są następnie wszystkie możliwe wartości podstawy z. Jeśli z jest względnie pierwsze zarówno z x jak i z y, dla trójki x, y, z, sprawdzane są wszystkie możliwe kombinacje potęg z zadanego przedziału (trzy najbardziej wewnętrzne pętle *for*).
Program operuje na 64-bitowych dodatnich liczbach całkowitych. Wszelkie obliczenia są wykonywane modulo 2^{64}, co stwarza ryzyko fałszywych trafień. Takie przypadki powinny być wyeliminowane już ręcznie (poza programem).

Tab. 3. Czasy realizacji pierwszego rozwiązania napisanego w języku ANSI C na procesorze Intel Xeon X5660 (kompilacja kompilatorem GCC 4.4.7 z optymalizacją trzeciego stopnia O3). Min_base = 2, Min_ power = 3. Czasy mierzone w godzinach.
Tab. 3. Execution times of the first solution written in ANSI C programming language on Intel Xeon X5660 processor (compilation with GCC 4.4.7 compiler, with 3rd level optimisation O3). Min_base = 2, Min_power = 3. Times measured in hours.

	max_ power=7	max_ power=10	max_ power=30	max_ power=100	max_ power=1000
max_base=100	0,000	0,000	0,000	0,032	38,223
max_base = 1000	0,016	0,030	0,654	34,780	> 168,000
max_ base=10000	19,356	33,189	> 168,000		
max_ base=100000	> 168,000				
max_ base=250000	> 168,000				

Podstawowy schemat blokowy przedstawiający działanie programu przedstawiono na Ryc. 1.



Ryc. 1. Uproszczony schemat blokowy przedstawiający działanie programu.

Fig. 1. Simplified block diagram showing how the programme works.

Wersja druga własnego rozwiązania

W celu przyspieszenia wykonywania obliczeń, zoptymalizowana funkcja wykorzystuje dodatkową pamięć (tablicę), w której zapisywane są wszystkie możliwe wyniki potęgowania dla podstaw z przedziału [*min_base*, *max_base*] i potęg z przedziału [*min_power*, *max_power*]. Pierwszy indeks utworzonej tablicy odpowiada wartościom podstawym, spośród tych analizowanych, a drugi indeks – wartościom wykładnika. Dla przykładu, w komórce o indeksach [0][0], znajdzie się wartość $min\_base^{min\_power}$, w komórce o indeksach [0][1] – $min\_base^{(min\_power+1)}$, [1][0] – $(min\_base+1)^{min\_power}$, itd. Dzięki takiemu podejściu, rozmiar wymaganej tablicy zależy jedynie od szerokości przedziałów wartości podstaw i potęg.

Tab. 4. Czasy realizacji drugiego rozwiązania napisanego w języku ANSI C na procesorze Intel Xeon X5660 (kompilacja kompilatorem GCC 4.4.7 z optymalizacją trzeciego stopnia O3). Czasy mierzone w godzinach.

Tab. 4. Execution times of the second solution written in ANSI C programming language on Intel Xeon X5660 processor (compilation with GCC 4.4.7 compiler, with third level optimisation O3). Times measured in hours.

	max_ power=7	max_ power=10	max_ power=30	max_ power=100	max_ power=1000
max_ base=100	0,000	0,000	0,000	0,026	25,798
max_ base = 1000	0,016	0,030	0,662	27,136	> 168,000
max_ base=10000	19,851	33,693	> 168,000		
max_ base=100000	> 168,000				
max_ base=250000	> 168,000				

Wersja trzecia własnego rozwiązania

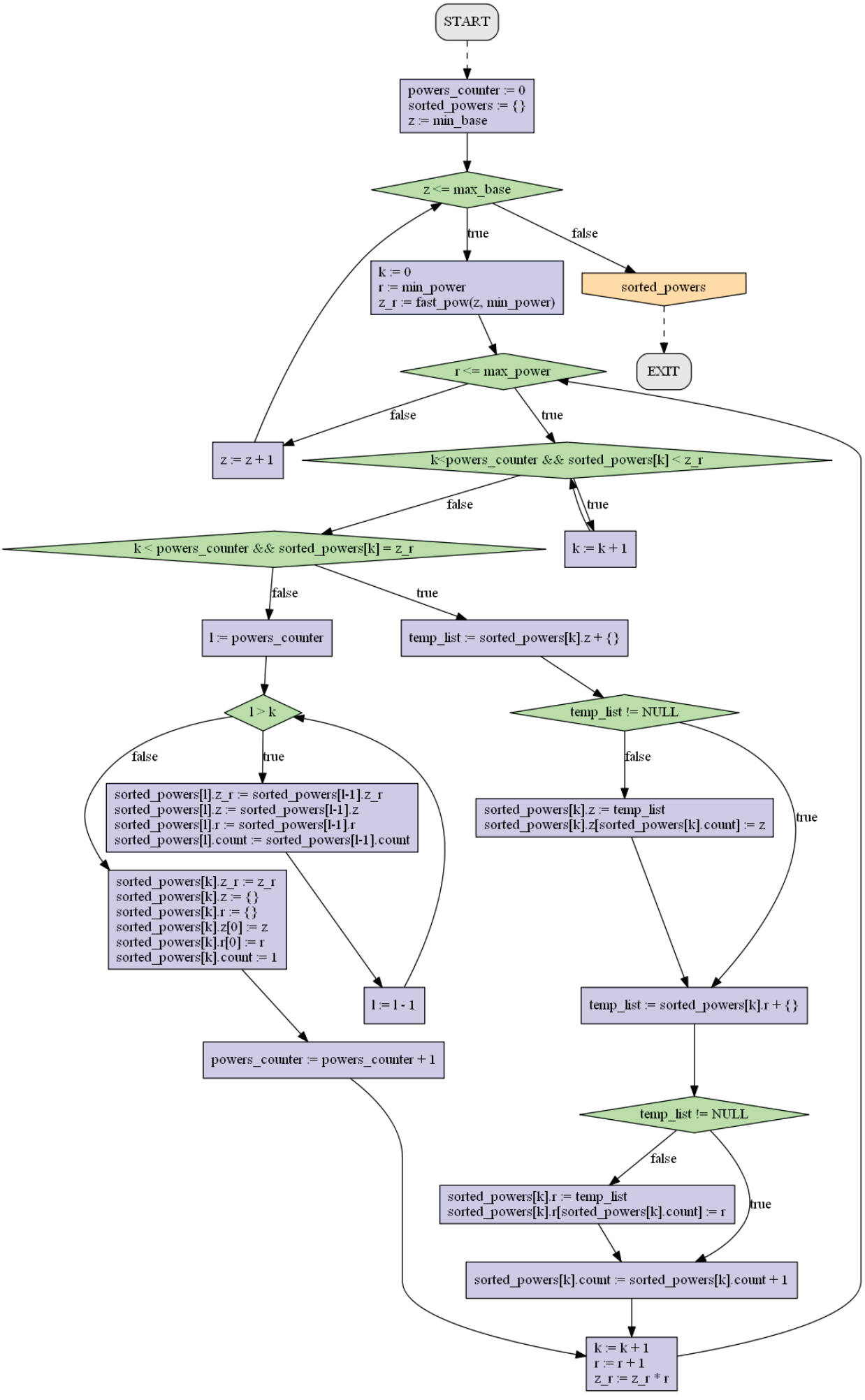
W trzecim algorytmie zastosowanie znajduje również druga tablica (*sorted_powers*), przechowująca struktury *powers_list_t*. Dla każdego możliwego wyniku potęgowania (końcowej wartości), gdzie podstawa jest z przedziału [*min_base, max_base*], a potęga z przedziału [*min_power, max_power*], w tablicy znajduje się dokładnie jeden rekord. Oprócz wyniku potęgowania, przechowywane są również podstawa i wykładnik (potęga). W przypadku, gdy więcej niż jedna kombinacja podstawy i wykładnika daje taki sam wynik (np. $10^4 = 100^2$), w strukturze na danej pozycji zapisane będą wszystkie wartości podstaw i wykładników dających taki wynik.

W tablicy *sorted_powers* będziemy szukać wyników potęgowania po wartości. Aby można było robić to efektywnie, tablica musi być posortowana. Tablica jest sortowana „w locie" podczas tworzenia. Wykorzystujemy fakt, że kolejne potęgi zadanej podstawy tworzą ciąg rosnący, dzięki czemu dodawanie elementów z zachowaniem sortowania jest szybsze, niż gdybyśmy sortowali tablicę po wypełnieniu.

```
for (xi=0; xi<=max_base_index; ++xi)
{
    x = min_base + xi;

    for (yi=xi+1; yi<=max_base_index; ++yi)
    {
        y = min_base + yi;

        if (gcd(x, y) == 1ULL)
        {
            for (mi=0; mi<=max_power_index; ++mi)
            {
                x_m = powers[xi][mi];

                for (ni=0; ni<=max_power_index; ++ni)
                {
                    y_n = powers[yi][ni];
                    i = binary_search(sorted_powers, 0, powers_counter-1, x_m + y_n);

                    if (i != -1)
                    {
                        for (j=0; j<sorted_powers[i].count; ++j)
                        {
                            z = sorted_powers[i].z[j];

                            if (z != x && z != y && gcd(z, x) == 1ULL && gcd(z, y) == 1ULL)
                            {
                                ++results_count;

                                temp_results = (beal_test_result_t*)
                                    realloc(*results, results_count * sizeof(beal_test_result_t));

                                if (temp_results)
                                {
                                    *results = temp_results;
                                    (*results)[results_count-1].x = x;
                                    (*results)[results_count-1].y = y;
                                    (*results)[results_count-1].z = z;
                                    (*results)[results_count-1].m = min_power+mi;
                                    (*results)[results_count-1].n = min_power+ni;
                                    (*results)[results_count-1].r = sorted_powers[i].r[j];
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Listing 2. Część główna kodu maksymalnie zoptymalizowanej funkcji beal_conjecture_test odpowiedzialna za poszukiwanie kontrprzykładu. Źródło: Opracowanie własne.

Listing 2. The part of maximally optimised function's code beal_conjecture_test responsible for creating the sorted_powers table.



build_sorted_powers(min_base, max_base, min_power, max_power, <out>powers_counter)

Ryc. 2. Schemat blokowy procedury build_sorted_powers.
Fig. 2. The block diagram of build_sorted_powers procedure.

```
for (xi=0; xi<=max_base_index; ++xi) {
    x = min_base + xi;
    for (yi=xi+1; yi<=max_base_index; ++yi) {
        y = min_base + yi;
        if (gcd(x, y) == 1ULL) {
            for (mi=0; mi<=max_power_index; ++mi) {
                x_m = powers[xi][mi];
                for (ni=0; ni<=max_power_index; ++ni) {
                    y_n = powers[yi][ni];
                    i = binary_search(sorted_powers, 0, powers_counter-1, x_m + y_n);
                    if (i != -1) {
                        for (j=0; j<sorted_powers[i].count; ++j) {
                            z = sorted_powers[i].z[j];
                            if (z != x && z != y && gcd(z, x) == 1ULL && gcd(z, y) == 1ULL) {
                                ++results_count;
                                temp_results = (beal_test_result_t*)
                                    realloc(*results, results_count * sizeof(beal_test_result_t));
                                if (temp_results) {
                                    *results = temp_results;
                                    (*results)[results_count-1].x = x;
                                    (*results)[results_count-1].y = y;
                                    (*results)[results_count-1].z = z;
                                    (*results)[results_count-1].m = min_power+mi;
                                    (*results)[results_count-1].n = min_power+ni;
                                    (*results)[results_count-1].r = sorted_powers[i].r[j];
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Listing 3. Część główna kodu maksymalnie zoptymalizowanej funkcji beal_conjecture_test odpowiedzialna za poszukiwanie kontrprzykładu.
Listing 3. The main part of maximally optimised function's code beal_conjecture_test responsible for counterexample searching.

Tab. 5. Czasy realizacji trzeciego rozwiązania napisanego w języku ANSI C na procesorze Intel Xeon X5660 (kompilacja kompilatorem GCC 4.4.7 z optymalizacją trzeciego stopnia O3). Czasy mierzone w godzinach.
Tab. 5. Execution times of the third solution written in ANSI C programming language on Intel Xeon X5660 processor (compilation with GCC 4.4.7 compiler, with third level optimisation O3). Times measured in hours.

	max_ power=7	max_ power=10	max_ power=30	max_ power=100	max_ power=1000
max_ base=100	0,000	0,000	0,000	0,000	0,050
max_ base = 1000	0,000	0,000	0,003	0,042	5,921
max_ base=10000	0,018	0,046	0,561	5,780	> 168,000
max_ base=100000	1,906	5,348	79,925	> 168,000	
max_ base=250000	17,543	43,025	> 168,000		

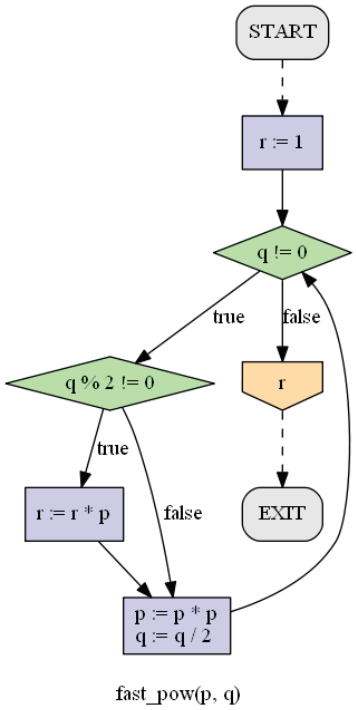
Wykorzystane algorytmy elementarne

W funkcji głównej *beal_conjecture_test* byliśmy zmuszeni do zastosowania kilku elementarnych algorytmów. W tym miejscu warto jest poświęcić chwilę na ich analizę.

Szybkie potęgowanie

```
unsigned long long int fast_pow(unsigned long long int p, unsigned long long int q):
```

Funkcja implementuje algorytm szybkiego obliczania potęgi o wykładniku naturalnym. Potęgowanie odbywa się poprzez serię operacji mnożenia. Dzięki wykorzystaniu właściwości binarnej reprezentacji wykładnika, liczba potrzebnych operacji mnożenia jest rzędu $log_2 q$ (metoda naiwna potrzebuje q operacji mnożenia).
Wykorzystujemy fakt, że podnoszenie do potęgi, która sama jest potęgą dwójki, można wykonać przy pomocy serii operacji podniesienia do kwadratu (podnosimy do kwadratu wynik poprzedniej operacji). Jedynki w binarnej reprezentacji liczby naturalnej określają, jakie potęgi dwójki wchodzą w jej skład. Każde potęgowanie możemy rozbić na iloczyn potęg, z których każda ma wykładnik będący potęgą dwójki.
Schemat blokowy algorytmu szybkiego potęgowania został przedstawiony na Ryc. 3.



Ryc. 3. Schemat blokowy algorytmu szybkiego potęgowania.
Fig. 3. The block diagram of fast raising to a power algorithm.

Największy wspólny dzielnik

```
unsigned long long int gcd(unsigned long long int a, unsigned long long int b);
```

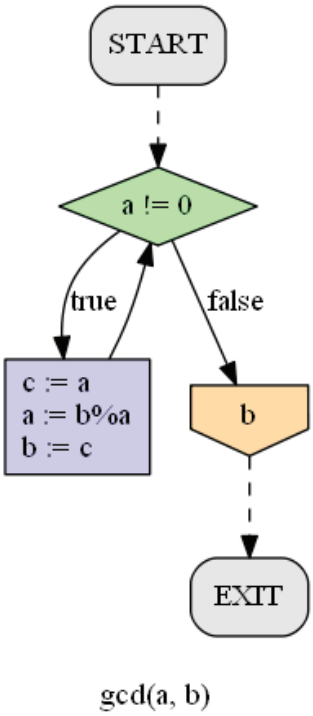
Funkcja oblicza największy wspólnego dzielnik (*nwd*) dwóch liczb naturalnych, wykorzystując algorytm Euklidesa. Algorytm oparty jest na następujących zależnościach:

$nwd(b, 0) = b$,
$nwd(b, a) = nwd(a, b \bmod a)$.

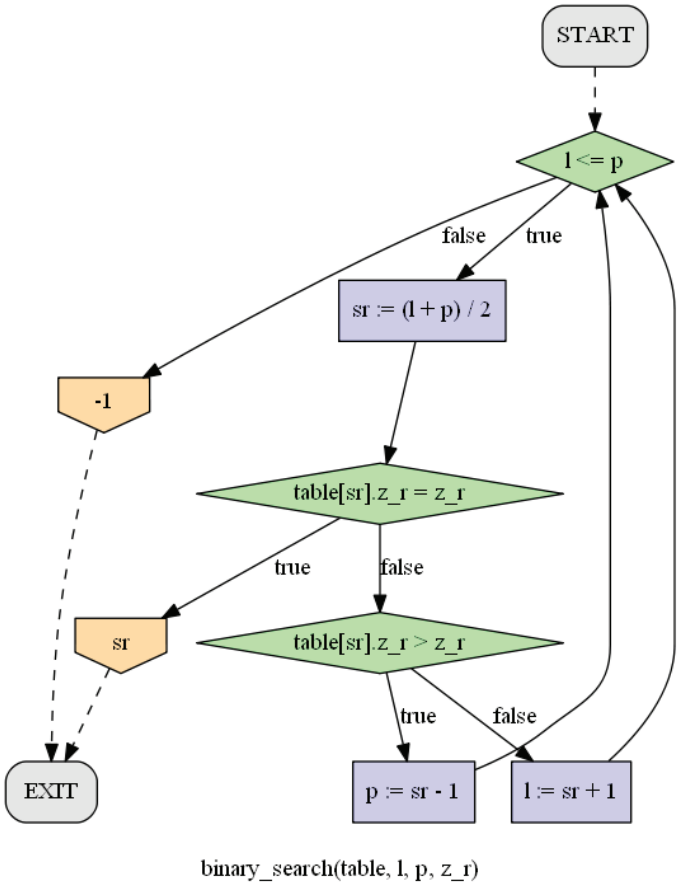
W naszej implementacji, wartość zmiennych a i b w kolejnych iteracjach wyliczane są jako:

$a_{t+1} = b_t \bmod a_t$,
$b_{t+1} = a_t$.

Zatrzymujemy się w momencie, gdy nowa wartość a jest równa 0. Wynikiem, czyli największym wspólnym dzielnikiem, jest wartość b.
Schemat blokowy algorytmu NWD został przedstawiony na Ryc. 4.



Ryc. 4. Schemat blokowy algorytmu NWD.
Fig. 4. The block diagram of NWD algorithm.



Ryc. 5. Schemat blokowy algorytmu przeszukiwania tablicy metodą podziałów.
Fig. 5. The block diagram of table searching with divisions method.

Przeszukiwanie tablicy metodą podziałów

```
int binary_search(powers_list_t* table, int l, int
p, unsigned long long int z_r);
```

Funkcja sprawdza, czy podany element znajduje się w przekazanej tablicy. Tablica musi być posortowana, aby algorytm działał poprawnie. Na początku sprawdzany jest środkowy element tablicy. Jeśli jest taki sam jak szukany, algorytm się kończy. Jeśli środkowy element jest większy od szukanego, powtarzamy procedurę dla lewej połówki tablicy. Jeśli jest mniejszy – powtarzamy procedurę dla prawej połówki tablicy.
Jeśli uda się znaleźć szukany element, funkcja zwraca jego indeks w tablicy. Jeśli elementu nie ma, funkcja zwraca wartość -1.
Schemat blokowy algorytmu przeszukiwania tablicy metodą podziałów został przedstawiony na Ryc. 5.

Zrównoleglenie kodu w OpenMP

Program można bardzo prosto zrównoleglić z wykorzystaniem środowiska OpenMP [5] [6]. Do trzeciej wersji kodu wystarczy dodać dwie pragmy – jedną odpowiedzialną za zrównoleglenie najbardziej zewnętrznej pętli for i drugiej odpowiedzialnej za wyodrębnienie sekcji krytycznej.

Pierwsza dyrektywa *#pragma* powinna wyglądać następująco:

#pragma omp parallel for private(*xi, x, yi, y, mi, x_m, ni, y_n, i, j, z*) *num_threads*(*number_of_threads*)

Druga dyrektywa *#pragma* to zwykłe ujęcie bloku instrukcji warunkowej:

if (*z != x && z != y && gcd*(*z, x*) == *1ULL && gcd*(*z, y*) == *1ULL*)

w

#pragma omp critical

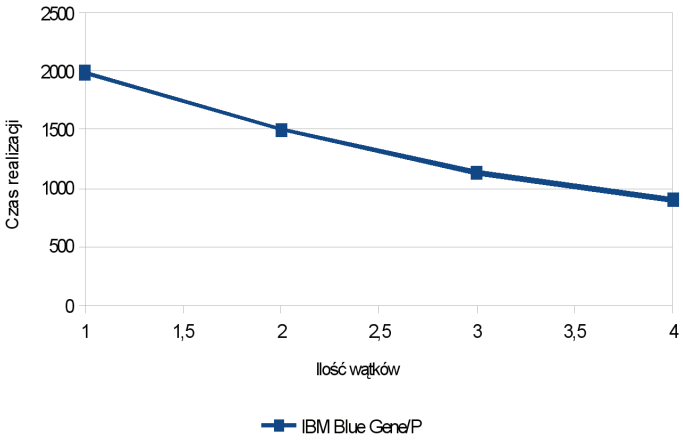
Wyniki czasowe zrównoleglenia uzyskane na czterech różnych platformach sprzętowych (IBM Blue Gene/P, IBM

Blue Gene/Q, IBM Power 775 oraz 2x Intel Xeon X5660) przedstawiono w Tab. 6.

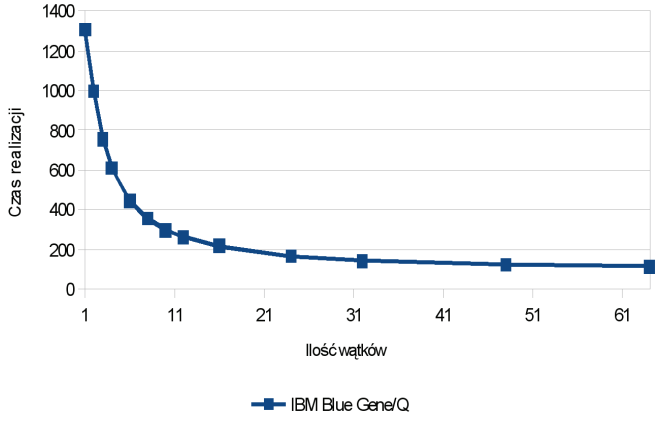
Tab. 6. Czasy realizacji rozwiązania zrównoleglonego w środowisku OpenMP wykonywanego z parametrami max_base = 500, max_power = 500. Czasy mierzone w sekundach.
Tab. 6. Execution times of parallel solution in OpenMP environment with parameters: max_base=500, max power = 500. Times measured in seconds.

Ilość wątków	IBM Blue Gene/P	IBM Blue Gene/Q	IBM Power 775	2x Intel Xeon X5660
1	1991	1305	1734	1612
2	1505	996	1309	1229
3	1133	755	986	924
4	901	609	784	729
6	-	445	562	533
8	-	356	443	421
10	-	298	364	349
12	-	262	314	302
16	-	218	253	-
24	-	166	181	-
32	-	143	147	-
48	-	125	-	-
64	-	115	-	-

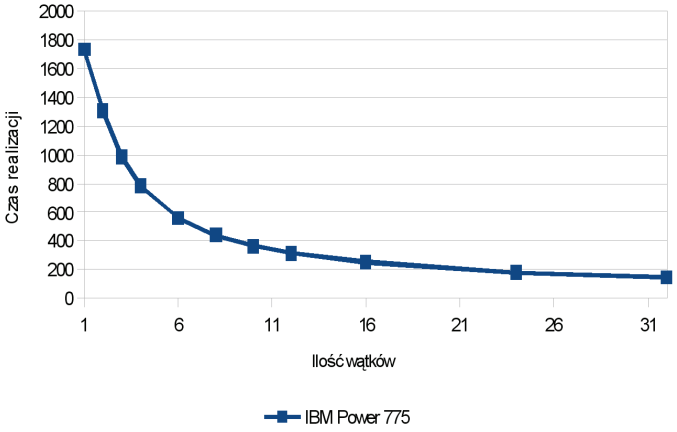
Wykresy przedstawiające spadek czasu realizacji programu dzięki zrównolegleniu dla badanych platform sprzętowych zaprezentowano na Ryc. 6 (IBM Blue Gene/P), Ryc. 7 (IBM Blue Gene/Q), Ryc. 8 (IBM Power 775) oraz Ryc. 9 (2x Intel Xeon X5660).



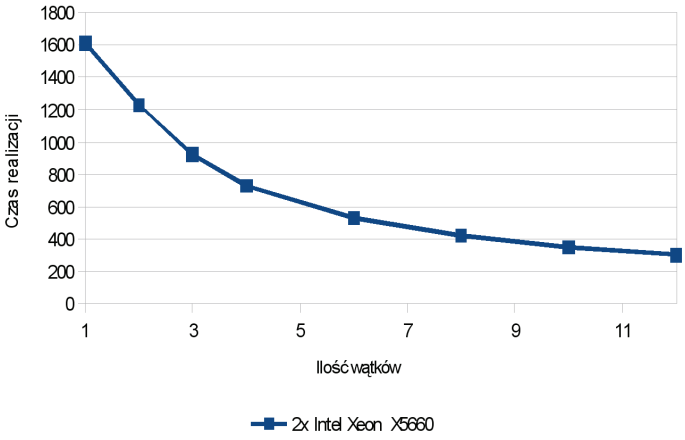
Ryc. 6. Redukcja czasu wykonywania programu na Blue Gene/P dzięki zrównolegleniu w środowisku OpenMP. Czasy mierzone w sekundach.
Fig. 6. Time reduction of program performing on Blue Gene/P thanks to parallelization in OpenMP environment. Times measured in seconds.



Ryc. 7. Redukcja czasu wykonywania programu na Blue Gene/Q dzięki zrównolegleniu w środowisku OpenMP. Czasy mierzone w sekundach.
Fig. 7. Time reduction of program performing on Blue Gene/Q thanks to parallelization in OpenMP environment. Times measured in seconds.



Ryc. 8. Redukcja czasu wykonywania programu na IBM Power 775 dzięki zrównolegleniu w środowisku OpenMP. Czasy mierzone w sekundach.
Fig. 8. Time reduction of program performing on IBM Power 775 thanks to parallelization in OpenMP environment. Times measured in seconds.



Ryc. 9. Redukcja czasu wykonywania programu na 2x Intel Xeon X5660 dzięki zrównolegleniu w środowisku OpenMP. Czasy mierzone w sekundach.
Fig. 9. Time reduction of program performing on 2x Intel Xeon X5660 thanks to parallelization in OpenMP environment. Times measured in seconds.

Zestawienie maksymalnych przyśpieszeń jakie udało się uzyskać na badanych czterech platformach sprzętowych przedstawiono w Tab. 6.

Tab. 6. Uzyskane maksymalne przyśpieszenie dzięki zastosowaniu rozwiązania zrównoleglonego w środowisku OpenMP na różnych platformach sprzętowych.
Tab. 6. Maximal acceleration obtained thanks to the use of parallel solution in OpenMP environment on various hardware platforms.

	IBM Blue Gene/P	IBM Blue Gene/Q	IBM Power 775	2x Intel Xeon X5660
Maksymalne uzyskane przyśpieszenie:	2.209	11.347	11.795	5,337

Porównanie wydajności najszybszego z zaimplementowanych rozwiązań z implementacją CUDA GPU zrealizowaną przez Jeet'a Chauhan'a

Porównano wydajność własnych rozwiązań z implementacją zaprezentowaną przez Jeet'a Chauhan'a [8]. Jak się okazało oprogramowanie Chauhan'a, pomimo że wykorzystywało GPU rozwiązywało problem w sposób wysoce niezadowalający pod względem wydajnościowym. Nie realne w tym przypadku okazały się testy dla *max_base = 500* oraz *max_powers = 500* – zajęłyby relatywnie dużo czasu. Dlatego też w tym przypadku dla celów porównawczych zmniejszono badany przedział do *max_base = 100* oraz *max_powers = 100*.
Wyniki zaprezentowano w Tab. 8.

Tab. 8. Porównanie wydajności programu GPGPU Jeet'a Chauhan'a z oprogramowaniem własnym na kilku różnych platformach sprzętowych. Źródło: Opracowanie własne.

	Procesor CPU	RAM	Procesor GPU	Kompilator	Czas realizacji
Jeet Chauhan Implementation	Intel i7 920 @ 2.80 GHz	24 GB DDR3	nVidia Tesla C2050	Cuda compilation tools, release 3.2, V0.2.1221	953,000 s
Jeet Chauhan Implementation	Intel Xeon E5-2670 @ 2.60GHz	512 GB DDR3	nVidia Tesla K10	Cuda compilation tools, release 4.2, V0.2.1221	1116,000 s
Jeet Chauhan Implementation	Intel Xeon E5-2670 @ 2.60GHz	512 GB DDR3	nVidia Tesla K10	Cuda compilation tools, release 5.0, V0.2.1221	1160,000 s
Jeet Chauhan Implementation	Intel Xeon E5-2670 @ 2.60GHz	512 GB DDR3	nVidia Tesla K10	Cuda compilation tools, release 6.0, V6.0.1	1160,000 s
Jeet Chauhan Implementation	Intel Xeon E5-2670 @ 2.60GHz	512 GB DDR3	nVidia Tesla K20	Cuda compilation tools, release 4.2, V0.2.1221	585,000 s
Jeet Chauhan Implementation	Intel Xeon E5-2670 @ 2.60GHz	512 GB DDR3	nVidia Tesla K20	Cuda compilation tools, release 5.0, V0.2.1221	585,000 s
Jeet Chauhan Implementation	Intel Xeon E5-2670 @ 2.60GHz	512 GB DDR3	nVidia Tesla K20	Cuda compilation tools, release 6.0, V6.0.1	588,000 s
Łukasz Świerczewski Implementation	Intel i7 920 @ 2.80 GHz – 1 wątek	24 GB DDR3	Nie dotyczy	gcc version 4.1.2 20080704 (Red Hat 4.1.2-52)	1,790 s
Łukasz Świerczewski Implementation	Intel i7 920 @ 2.80 GHz – 8 wątków	24 GB DDR3	Nie dotyczy	gcc version 4.1.2 20080704 (Red Hat 4.1.2-52)	0,472 s
Łukasz Świerczewski Implementation	Intel Xeon E5-2670 @ 2.60GHz – 1 wątek	512 GB RAM	Nie dotyczy	gcc version 4.4.7 20120313 (Red Hat 4.4.7-11) (GCC)	1,100 s
Łukasz Świerczewski Implementation	Intel Xeon E5-2670 @ 2.60GHz – 16 wątków	512 GB RAM	Nie dotyczy	gcc version 4.4.7 20120313 (Red Hat 4.4.7-11) (GCC)	0,298 s

Wnioski i perspektywy dalszych badań

Pomimo wielu prób i doświadczeń przeprowadzonych w ramach pisania tego artykułu nie udało się odnaleźć kontrprzykładu dla przypuszczenia Beal'a.
Podczas obliczeń ograniczono się jedynie do klasycznych procesorów CPU. Można jednak dodatkowo wykorzystać akceleratory graficzne co byłoby bardzo ciekawym podejściem do problemu. Dzięki takiemu rozwiązaniu najprawdopodobniej możliwe byłoby uzyskanie znacznie lepszych wyników końcowych. Oczywiście istnieją już zaprogramowane implementacje w CUDA [7] – należy do nich m.in. rozwiązanie zaprezentowane w pracy magisterskiej Jeet'a Chauhan'a [8]. Istnieje także implementacja w języku Brook+ [9]. Pomysły te mają jednak swoje ograniczenia i nie nadają się do zastosowania na wielką skalę. Przykładowo w pracy Jeet'a Chauhan'a [8] rozważono algorytm działający na GPU ale jedynie dla bardzo niewielkich liczb. Przeprowadzone analizy na potrzeby tej pracy wykazały także, że oprogramowanie Jeet'a Chauhan'a jest niesamowicie wolne. Połączeniem możliwości jakie daje środowisko MPI [10] z wykorzystaniem akceleratorów graficznych zajął się także Naveen Kumar Reddy Nandipati [11]. W tym miejscu należy także wspomnieć o standardzie OpenCL [12], który jest jednolitym środowiskiem umożliwiającym programowanie kart graficznych (zarówno AMD jak i nVidia) oraz np. procesorów IBM Cell [13]. Do obliczeń można także wykorzystać platformę BOINC [14] [15] (Berkeley Open Infrastructure for Network Computing), która umożliwiłaby połączenie mocy obliczeniowych tysięcy komputerów połączonych za pomocą sieci Internet. Inne aspekty badawcze, które można wykorzystać w dalszej parcy zaprezentowali polscy badacze [16] [17].

Podziękowania

Obliczenia wykonano w Interdyscyplinarnym Centrum Modelowania Matematycznego i Komputerowego (ICM) Uniwersytetu Warszawskiego w ramach grantu obliczeniowego nr G55-11.

Literatura

1. Mauldin, R. Daniel. "A generalization of Fermat's Last Theorem: the Beal conjecture and prize problem." *Notices of the American Mathematical Society* 44.11 (1997), 1436-1437.
2. Beal Prize - American Mathematical Society, URL: http://www.ams.org/profession/prizes-awards/ams-supported/beal-prize, Ostatni dostęp: 18.04.2014
3. Beal Conjecture, URL: http://www.bealconjecture.com/, Ostatni dostęp: 18.04.2014
4. Beal's Conjecture: A Search for Counterexamples, URL: http://norvig.com/beal.html, Ostatni dostęp: 18.04.2014
5. Dagum, Leonardo, and Ramesh Menon. "OpenMP: an

industry standard API for shared-memory programming." *Computational Science & Engineering*, IEEE 5.1 (1998), 46-55.
6. Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas. Using OpenMP: *portable shared memory parallel programming*. Vol. 10. MIT press, 2008.
7. Sanders, Jason, and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.
8. Chauhan, Jeet. *Nvidia GeForce 8400 GPGPU implementation of a CUDA C parallel algorithm to search for counterexamples to Beal's Conjecture*. Diss. San Diego State University, 2010.
9. Shah, Nirav. *AMID Firestream 9170 GPGPU implementation of a Brook+ parallel algorithm to search for counterexamples to Beal's Conjecture*. Diss. San Diego State University, 2010.
10. Gropp, William, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.
11. Nandipati, Naveen Kumar Reddy. "Heterogeneous NPACI-ROCKS/MPI/CUDA distributed multi-GPGPU application for seeking counterexamples to Beal's Conjecture; ROCKS/CUDA integration component.", 2011.
12. Stone, John E., David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." *Computing in science & engineering* 12.3 (2010): 66.
13. Chen, Thomas, et al. "Cell broadband engine architecture and its first implementation—a performance view." *IBM Journal of Research and Development* 51.5 (2007), 559-572.
14. Anderson, David P. "Boinc: A system for public-resource computing and storage." *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE, 2004.
15. Estrada, Trilce, Michela Taufer, and David P. Anderson. "Performance prediction and analysis of BOINC projects: An empirical study with EmBOINC." *Journal of Grid Computing* 7.4 (2009), 537-554.
16. H. Zarzycki H, J. Czerniak, *Development and Code Management of Large Software Systems*, PSZW, nr 27, s. 327-339, Bydgoszcz 2010.
17. H. Zarzycki, Application of the finite difference CN method to value derivatives, PSZW, nr 42, s. 267-277, Bydgoszcz 2011.