Filip Hallo

Metody dyskretne w złożonych systemach obliczeniowych

Sprawozdanie z projektu 2 – optymalizacje i zrównoleglanie projektu 1.

# Część pierwsza – optymalizacje wersji sekwencyjnej

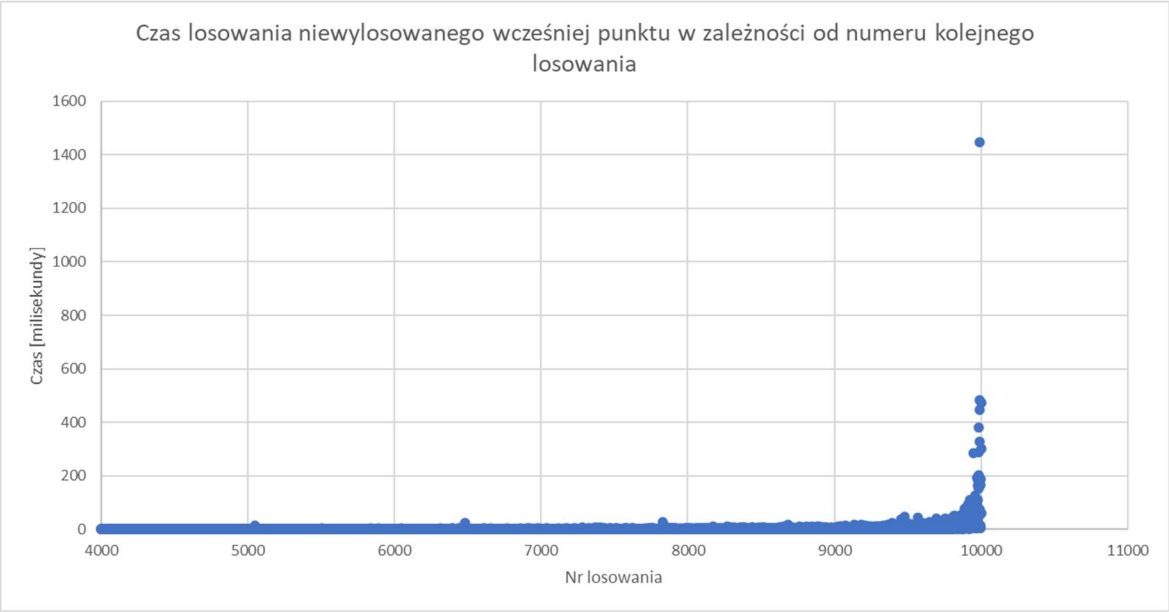
W celu badania efektywności zmian kodu w celu poprawy wydajności przyjęto punkt odniesienia– sąsiedztwo Moore'a, siatka 100 x 100, ustalone zarodki, 10 iteracji Monte Carlo. Ilekroć przedstawiono wyniki nieopatrzone komentarzem do danych do symulacji, odnoszą się one do tych właśnie parametrów wejściowych.

Na podstawie sporządzonego wcześniej profilu aplikacji oraz znajomości zagadnień i mnogości implementacji poszczególnych funkcjonalności wiadomo, że losowanie współrzędnych w metodzie Monte Carlo odbywało się w bardzo nieefektywny sposób. Współrzędne były losowane jako dwa inty, następnie wywoływana była funkcja sprawdzająca, czy wcześniej taki punkt był wylosowany. Wewnątrz tej funkcji współrzędne były konwertowane na string.

```
bool MonteCarlo::check_coordinates(int random_row, int random_col, std::vector < std::string > &coordinates_done){
        std::string coordinates = std::to_string(random_row) + "," + std::to_string(random_col);
        if (std::count(coordinates_done.begin(), coordinates_done.end(), coordinates) == 0){
            coordinates_done.push_back(coordinates);
            return false;
        }
        else
            return true;
}
```

*Rysunek 1 - przykład nieefektywnej implementacji funkcjonalności na potrzeby Monte Carlo*

Takie podejście posiada dwie zasadnicze wady: wraz ze zwiększaniem się liczby wylosowanych i obsłużonych komórek maleje prawdopodobieństwo wylosowania współrzędnych, które jeszcze nie zostały obsłużone. Skutkuje to znaczącym wzrostem czasu losowania nieobsłużonej kombinacji współrzędnych pod koniec losowania, kiedy większość z komórek siatki została już wylosowana i obsłużona. Drugim problemem jest konwersja i obsługa obiektu typu string, co jest czasochłonne.




*Rysunek 2 - czas potrzebny na wylosowanie n-tej pary współrzędnych*

W związku z powyższym zaimplementowano inne podejście opierające się na wygenerowaniu wektora zawierającego tuple intów, będące uporządkowanymi współrzędnymi siatki. Taki wektor poddano „przemieszaniu” przy użyciu metody std::shuffle. Następnie przetwarzano po kolei elementy tak zmodyfikowanego wektora, rozpakowując tuple do dwóch intów i przetwarzając je jako współrzędne. Takie podejście pozwoliło wyeliminować oba problemy nakreślone powyżej. Odnotowano znaczne skrócenie czasu symulacji, o ponad 98 punktów procentowych. Wyniki przedstawiono w tabeli 1.

*Tabela 1 - rezultat wprowadzonych implementacji*

| Modyfikacje MC | Czas symulacji [us] | Czas symulacji [ms] | % czasu bazowego |
|---|---|---|---|
| Brak | 19254057 | 19254.057 | 100% |
| Zmiana sposobu "losowania" komórek | 254163 | 254.163 | 1.3% |

W metodzie automatów komórkowych występowały dwie funkcje - is_empty() i is_done(). Funkcja is_empy() odpowiada za sprawdzenie, czy siatka jest pusta. Jeśli zwróci prawdę, to nie wykonujemy symulacji, jeśli fałsz, wykonujemy. Wywoływanie tej funkcji przy każdej iteracji pętli while po siatce jest pozbawione sensu; wynik funkcji jest potrzebny tylko raz - w celu sprawdzenia, czy należy przeprowadzić rozrost metodą CA. W dodatku wewnątrz metody analizie poddawana jest cała przestrzeń (zliczane są komórki ogółem i komórki posiadające wartość 0). Jeżeli ilość komórek z zerami jest równa ilości komórek ogółem, metoda zwraca prawdę.

```
void Simulation2D::run_moore() {
    std::cout<<"Running Moore"<<std::endl;
    while (!is_done() && !is_empty()){
        apply_bc();
        for(int i=1; i<rows-1; i++){
            for(int j = 1; j < cols-1; j ++){
                std::map<int, int> counter;

                counter[grid_t[i-1][j-1]]++;
                counter[grid_t[i-1][j]]++;
                counter[grid_t[i-1][j+1]]++;
                counter[grid_t[i][j-1]]++;
                counter[grid_t[i][j+1]]++;
                counter[grid_t[i+1][j-1]]++;
                counter[grid_t[i+1][j]]++;
                counter[grid_t[i+1][j+1]]++;

                grid_t1[i][j] = get_most_common_value(counter);
            }
        }
        swap_arrays();
    }
}
```

*Rysunek 3 - nieefektywna implementacja metody automatów komórkowych*

Usunięto funkcję is_empy() funkcję poza warunek pętli while, przez co wykona się ona tylko raz, na początku symulacji.

Funkcja is_done() sprawdza, czy w siatce występuje choć jedna komórka z wartością zerową. Jeśli tak, metoda zwraca fałsz i symulacja wykonuje się kolejny raz, aż do wypełnienia całej przestrzeni. W ramach optymalizacji zbadano, czy zrezygnowanie z wywołania funkcji i przeniesienie jej funkcjonalności do wnętrza pętli while przyniesie poprawę wydajności. Takie podejście wymusiło nieco inną implementację, co przełożyło się na spadek wydajności, gdyż wykonywane jest więcej rozkazów.

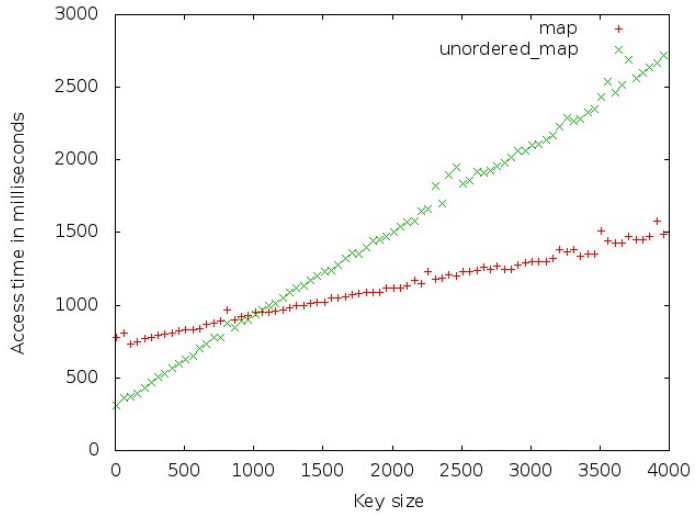
```
void Simulation2D::run_moore() {
    std::cout<<"Running Moore"<<std::endl;
    while (true){
        int zero_counter=0;
        apply_bc();
        //#pragma omp parallel for num_threads(2)
        for(int i=1; i<rows-1; i++){
            for(int j = 1; j < cols-1; ++j){
                std::map<int, int> counter;
                //std::map<int, int> counter2;


                counter[grid_t[i-1][j-1]]++;
                counter[grid_t[i-1][j]]++;
                counter[grid_t[i-1][j+1]]++;
                counter[grid_t[i][j-1]]++;
                counter[grid_t[i][j+1]]++;
                counter[grid_t[i+1][j-1]]++;
                counter[grid_t[i+1][j]]++;
                counter[grid_t[i+1][j+1]]++;
                grid_t1[i][j] = get_most_common_value(counter); // Druga opcja

                if (grid_t1[i][j] == 0)
                {
                    zero_counter++;
                }
            }

        }
        swap_arrays();
        if (zero_counter == 0)
        {
            break;
        }
    }
}
```

*Rysunek 4 - zmieniona implementacja metody automatów komórkowych*

W wyniku wprowadzonych zmian zaobserwowano wysłużenie czasu wykonania symulacji o 5 punktów procentowych względem wariantu podstawowego. Czas symulacji metodą CA wyniósł 226 milisekund w porównaniu do 215 przed zmianą.

Kolejną optymalizacją było zastosowanie memcpy przy podmianie siatek. Zamiast przepisywać po kolei komórki z siatki 1 na siatkę 2. Uzyskano skrócenie czasu symulacji o ponad 10% względem wersji bazowej. Niestety w wyniku przepisywania pamięci pierwotnie zaprojektowany destruktor nie był w stanie spełnić swojej funkcji, co powodowało liczne błędy, zatem zrezygnowano z tego podejścia.

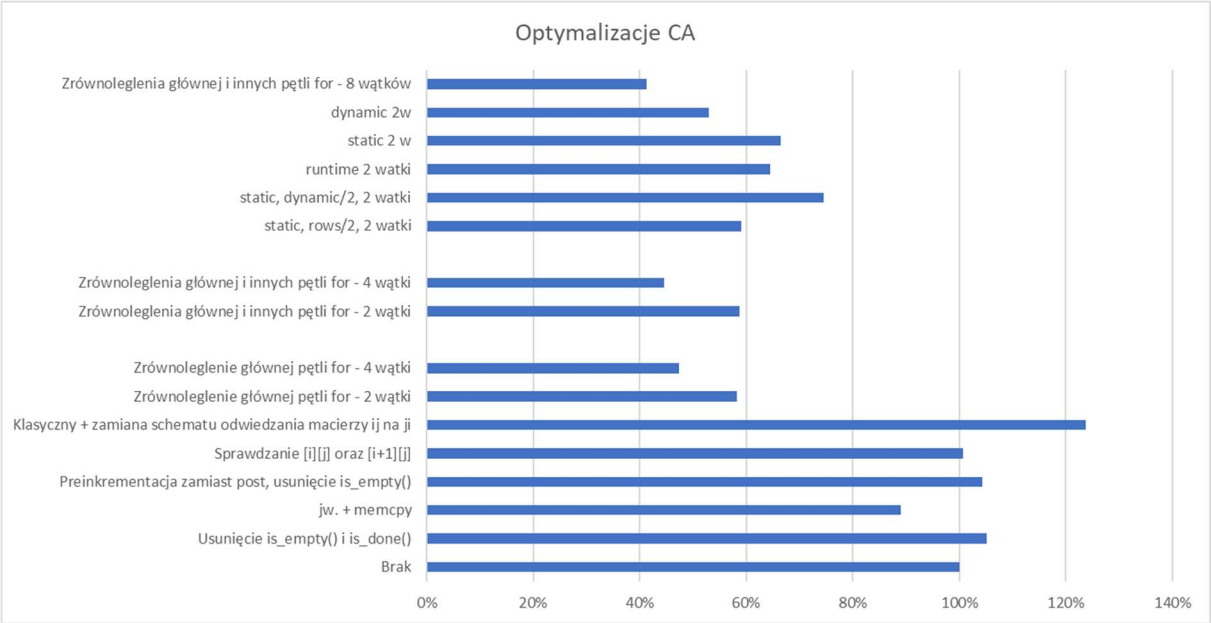
Sąsiedztwo ustalane jest dla każdej komórki przy użyciu std::map. Obiegowa opinia wskazuje, że std::unordered_map jest szybsza. W związku z tym zamieniono te struktury, co wydłużyło obliczenia o 2 punkty procentowe względem czasu bazowego. Jak przedstawiono na rysunku 6 użycie unordered map jest opłacalne jedynie od pewnego rozmiaru klucza, co uzasadnia spadek wydajności.




*Rysunek 5 - czas dostępu do std::map i std::unordered_map dla różnych rozmiarów klucza*

Źródło: https://blog.dubbelboer.com/2012/12/04/lru-cache.html

W ramach pozostałwych optymalizacji zmieniono sposób zapisu bryły 3D do pliku. Niestety każda zmiana sposobu zapisu owocowała wydłużeniem czasu obliczeń, stąd zdecydowano się na porzucenie tych zmian. Ponadto analizowano również wpływ zastosowanie preinkrementacji zamiast post, usunięcia metody is_empty() z wewętrznej pętli, co wydłużyło obliczenia, dlatego również nie zdecydowano się na zastosowanie zmian. Loop unrolling, czyli sprawdzanie [i][j] oraz [i+1][j] w jednej pętli, zamiana sposobu odwiedzania macierzy ij ji również przyczyniły się do spadku wydajności. Na tej podstawie zdecydowano, że wersja bez poprawek jest wejściem do procesu zrównoleglenia.



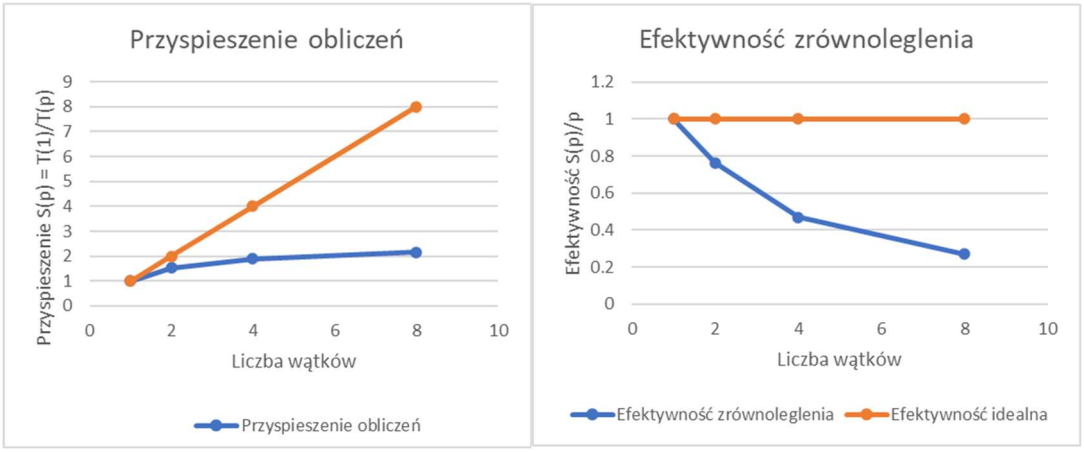

*Rysunek 6 - zbiorcze zestawienie wpływu zmian w kodzie na czas wykonania. Na osi poziomej % czasu bazowego*

# Część druga – zrównoleglenie

Pierwszy wariant zakładał zrównoleglenie pętli przy użyciu OpenMP. Zbadano jak zrównoleglenie pętli for (głównej – odpowiedzialnej za przechodzenie po przestrzeni i badanie sąsiedztwa i pobocznych – wykonujących zamianę przestrzeni z kroku czasowego t+1 na t, realizujących warunki brzegowe) wpłynie na przyspieszenie obliczeń. Obliczono też efektywność zrównoleglenia dla różnej ilości wątków. Takie podejście umożliwiło obiektywną ocenę efektów, w tym skalowalności rozwiązania.

*Tabela 2 - wyniki dla zrównoleglenia przy użyciu OpenMP*

| Modyfikacje CA | Czas symulacji [us] | Czas symulacji [ms] | % czasu bazowego |
|---|---|---|---|
| Brak | 214954 | 215 | 100% |
| Zrównoleglenie głównej pętli for - 2 wątki | 125177 | 125 | 58.2% |
| Zrównoleglenie głównej pętli for - 4 wątki | 101726 | 102 | 47.3% |
| Zrównoleglenia głównej i innych pętli for - 2 wątki | 126185 | 126 | 58.7% |
| Zrównoleglenia głównej i innych pętli for - 4 wątki | 95961 | 96 | 44.6% |




*Rysunek 7 – wyniki dla zrównoleglenia wszystkich pętli a) przyspieszenie obliczeń, b) efektywność zrównoleglenia*

Drugi wariant obejmował zrównoleglenie przy użyciu biblioteki thread. Zaimplementowano metodę make_coordinates_vector(), która zwraca wektor tupli współrzędnych, będący argumentem kolejno omówionej metody.

```
std::vector<std::tuple<int, int>> Simulation2D::make_coordinates_vector()
{
    std::vector<std::tuple<int, int>> coordinates;
    for (int i = 1; i < rows - 1; i++)
    {
        for (int j = 1; j < cols - 1; j++)
        {
            coordinates.push_back(std::make_tuple(i, j));
        }
    }
    return coordinates;
}
```

*Rysunek 8 - metoda zwracająca wektor tupli współrzędnych*

W celu równomiernego rozłożenia zadań na wątki zaimplementowano metodę split przyjmującą wektor, będący wyjściem powyższej metody oraz jednego inta – Nsplit, który określa ilość części, na które ma zostać podzielony wektor wejściowy. Następnie metoda split zwraca wektor wektorów tupli intów, na bazie którego tworzone są zadania dla wątków.

```
std::vector<std::vector<std::tuple<int, int>>> Simulation2D::split(const std::vector<std::tuple<int, int>> &v, int Nsplit)
{
    int n = v.size();
    int size_max = n / Nsplit + (n % Nsplit != 0);
    std::vector<std::vector<std::tuple<int, int>>> split;
    for (int ibegin = 0; ibegin < n; ibegin += size_max)
    {
        int iend = ibegin + size_max;
        if (iend > n)
            iend = n;
        split.emplace_back(std::vector<std::tuple<int, int>>(v.begin() + ibegin, v.begin() + iend));
    }
    return split;
}
```

*Rysunek 9 - metoda dzieląca przestrzeń równomiernie na Nsplit części*

Ostatecznie dopasowano metodę wykonującą symulację do wielowątkowych zadań, jej obecną formę zaprezentowano na rysunku 8.

```
void Simulation2D::run_moore()
{
    std::cout << "Running Moore" << std::endl;
    while (!is_done())
    {
        apply_bc();
        int numberOfThreads = 2;

        std::vector<std::thread> threadsVec;
        std::vector<std::vector<std::tuple<int, int>>> batchesForThreads = split(make_coordinates_vector(), numberOfThreads);

        for (int th_no = 0; th_no < numberOfThreads; th_no++)
        {
            threadsVec.push_back(std::thread(&Simulation2D::makeStepOnGridTh, this, batchesForThreads.at(th_no)));
        }

        for (auto &t : threadsVec)
            t.join();

        swap_arrays();
    }
}
```

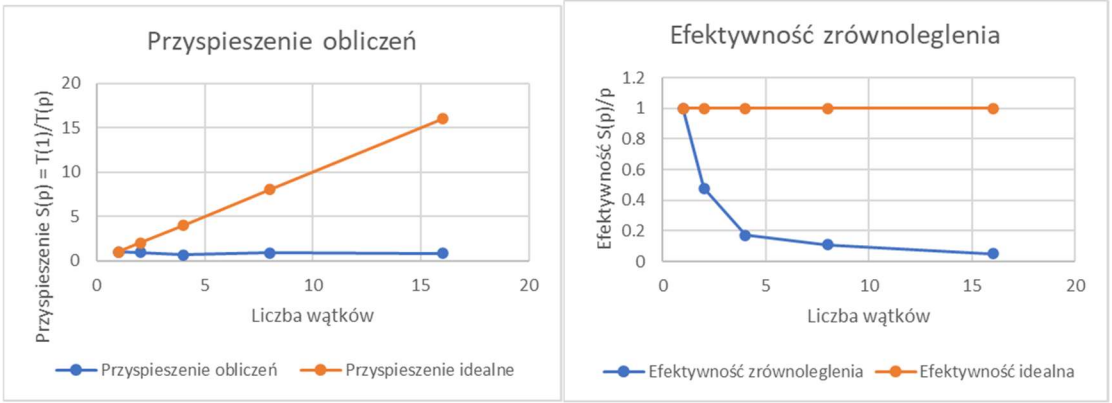
*Rysunek 10 - wielowątkowa implementacja CA przy użyciu std::threads*

W wyniku powyższych kroków otrzymano spójny i z perspektywy użytkownika łatwo skalowalny program realizujący to samo zadanie wielowątkowo. Na jego przykładzie przebadano czas symulacji, a wyniki zamieszczono w tabeli 3.

*Tabela 3 - wyniki wydajnościowe dla różnej ilości wątków dla siatki 100x100*

| Liczba wątków | Czas symulacji [us] | Czas symulacji [ms] | Przyspieszenie obli | Efektywność zrównoleglenia | Efektywnośc idealna |
|---|---|---|---|---|---|
| 1 | 309620 | 310 | 1 | 1 | 1 |
| 2 | 295774 | 296 | 0.955280667 | 0.477640333 | 1 |
| 4 | 211806 | 212 | 0.684083716 | 0.171020929 | 1 |
| 8 | 269546 | 270 | 0.870570377 | 0.108821297 | 1 |
| 16 | 252025 | 252 | 0.813981655 | 0.050873853 | 1 |

Na podstawie powyższej tabeli, w celu lepszego zobrazowania zagadnienia sporządzono następujące wykresy:




*Rysunek 11 - skalowalność rozwiązania - a) przyspieszenie obliczeń, b) efektywność zrównoleglenia*

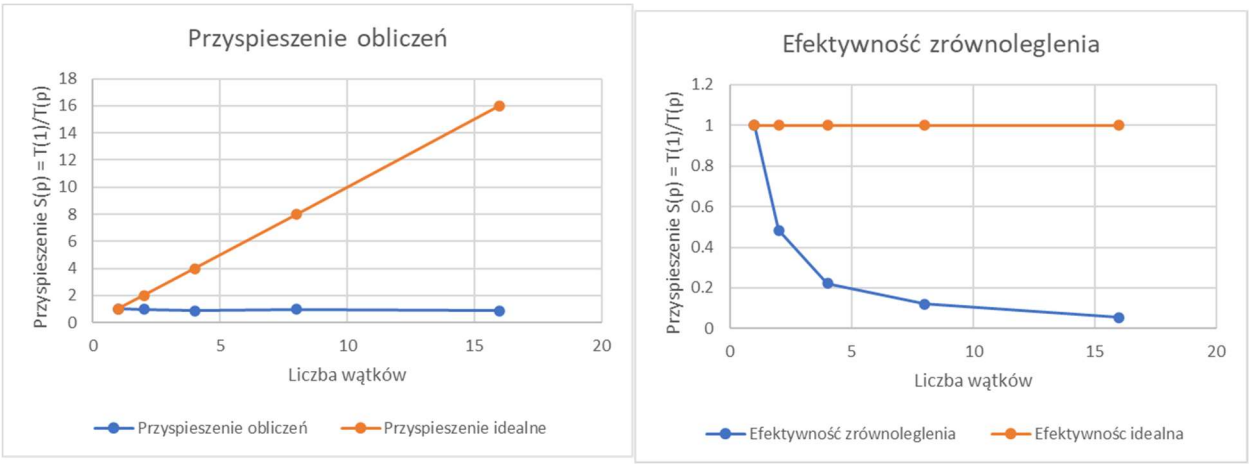
W wyniku ich analizy stwierdzono, iż zaproponowane rozwiązanie cechuje się gorszą skalowalnością niż poprzednie, korzystające z OpenMP.

Podejrzewano, że słaba skalowalność może wynikać z niewielkiego rozmiaru siatki. W takim przypadku narzuty związane ze zrównolegleniem (np. podział pracy, utworzenie wątków) będą stanowiły większy odsetek czasu symulacji. W celu zweryfikowania hipotezy przebadano działanie programu dla siatki 2D 1000x1000. Uzyskane wyniki obalają hipotezę o małej siatce jako przyczynie słabej skalowalności - podsumowanie przedstawiono w tabeli 3. Przedstawione wyniki nie odbiegają znacznie od wyników dla mniejszej siatki 2D o rozmiarze 100x100.

*Tabela 4 - wyniki wydajnościowe dla różnej ilości wątków dla siatko 1000x1000*

| Liczba wątków | Czas symulacji [us] | Czas symulacji [ms] | Przyspieszenie obli | Efektywność zrównoleglenia | Efektywnośc idealna |
|---|---|---|---|---|---|
| 1 | 200972609 | 200973 | 1 | 1 | 1 |
| 2 | 194449225 | 194449 | 0.96754093 | 0.483770465 | 1 |
| 4 | 178045723 | 178046 | 0.885920344 | 0.221480086 | 1 |
| 8 | 195587656 | 195588 | 0.973205538 | 0.121650692 | 1 |
| 16 | 175710082 | 175710 | 0.874298656 | 0.054643666 | 1 |

Na podstawie powyższej tabeli, w celu lepszego zobrazowania zagadnienia sporządzono następujące wykresy:




*Rysunek 12 - skalowalność rozwiązania a) przyspieszenie obliczeń, b) efektywność zrównoleglenia*

W metodzie Monte Carlo zrównoleglenie jest nieco bardziej wymagające. Ze względu na operowanie na tej samej przestrzeni wątki nie powinny sobie wchodzić w drogę. Mając na uwadze powyższe zdecydowano się na podobną implementację podziału zadań, biorąc jedynie pod uwagę pewien margines bezpieczeństwa. Zgodnie z tą ideą każdy wątek pracuje na takim obszarze siatki, aby jego działanie nie miało wpływu na inny wątek. Na koniec, po wykonaniu pracy równoległej, jeden wątek wykonuje operacje na pozostawionym uprzednio marginesie bezpieczeństwa, przez co uzyskano w pełni deterministyczne działanie symulacji.

# Podsumowanie

Przedstawione poprawki nie zawsze wiązały się z poprawą wydajności, jednak zdobyta w ten sposób wiedza o konsekwencjach ich zastosowania jest cenną podstawą do wprowadzenia bardziej celnych poprawek w przyszłości.

Zrównoleglenie automatów komórkowych jest zadaniem mniej wymagającym niż zrównoleglenie Monte Carlo.

Oba rozwiązania cechowały się nienajlepszą skalowalnością, jednakże wykorzystanie OpenMP pozwoliło na lepszą skalowalność niż std::thread.