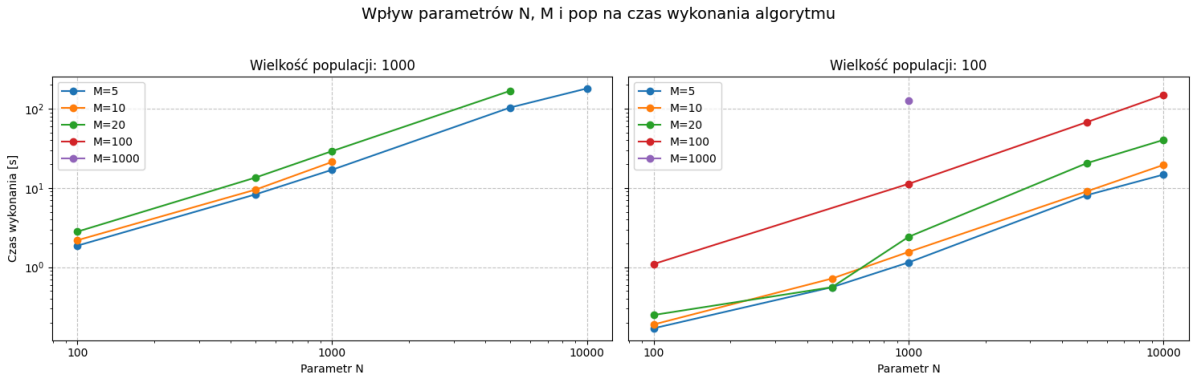
Eksperymenty

Sekwencyjny



Złożoność czasowa algorytmu bazowego: Θ(G · P · n · m)
Złożonosc pamięciowa algorytmu bazowego: Θ(n·m + P·n)

Wyniki profilowania:

main	21626 (99,95%)	7 (0,03%)
std::vector<std::vector<int,std::allocator<int> >,std::allocator<std::vector<int,std::allocator<int> > > >::push_back	5736 (26,51%)	3 (0,01%)
makespan	5656 (26,14%)	2656 (12,28%)
order_crossover	4120 (19,04%)	13 (0,06%)
tournament_selection	2359 (10,90%)	5 (0,02%)
std::vector<int,std::allocator<int> >::vector<int,std::allocator<int> >	1938 (8,96%)	3 (0,01%)

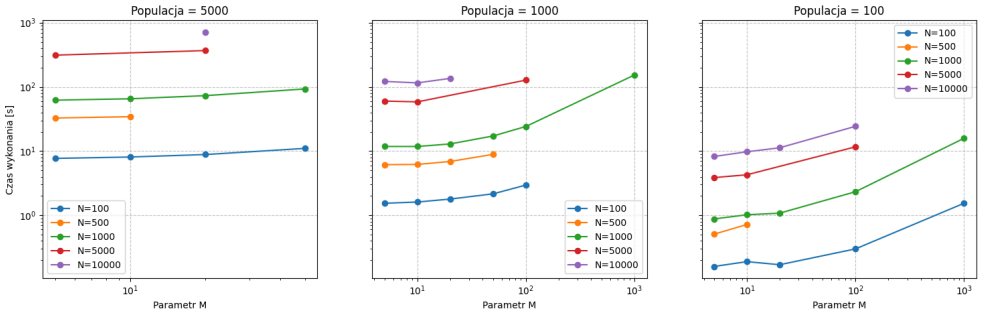
1. Początkowo algorytm wykonywał się 1 min 11 s, a funkcja order_crossover zajmowa 50% udziału procesora
2. W order_crossover instrukcja przeszukująca czy dane zadanie już zostało użyte w potomku miała złożoność czasowa O(n^2), bo przeszukiwanie wektora było zagnieżdżone w pętli o długości n. Teraz korzystamy z tablicy “used” gdzie przechowujemy informacje czy dane zadanie było już użyte w tworzonym potomku. Każdorazowe dodania zadania do potomka aktualizuje tablice. Dzięki temu sprawdzenie przynależności zadania do potomka jest w czasie stałym i zredukowanym do O(n). Całkowity czas wykonania algorytmu spadł do 40,872s.
3. Funkcja makespan w pierwotnej wersji przy każdym wywołaniu tworzony był nowy wektor machine_ready, co powodowało dużą liczbę alokacji pamięci. Teraz wprowadziliśmy zewnętrzny bufor przekazywany przez referencję, który jest wielokrotnie wykorzystywany i jedynie zerowany przed kolejnym użyciem. Czas wykonania algorytmu skrócił się do 28,056s
4. Kolejna zmiana dotyczy ponownie funkcji order_crossover. Problem polegał na tym, że przy każdym wywołaniu tworzył się nowy wektor child i tablica used. Teraz tworzymy je raz w main() przed pętlą GA i resetujemy je w order_crossover, gdzie są parametrami referencyjnymi. Czas 26,997s
5. W makespan zmieniliśmy zerowanie machine_ready z std::fill na pętle for. Czas: **25,666s.**

Równoległy

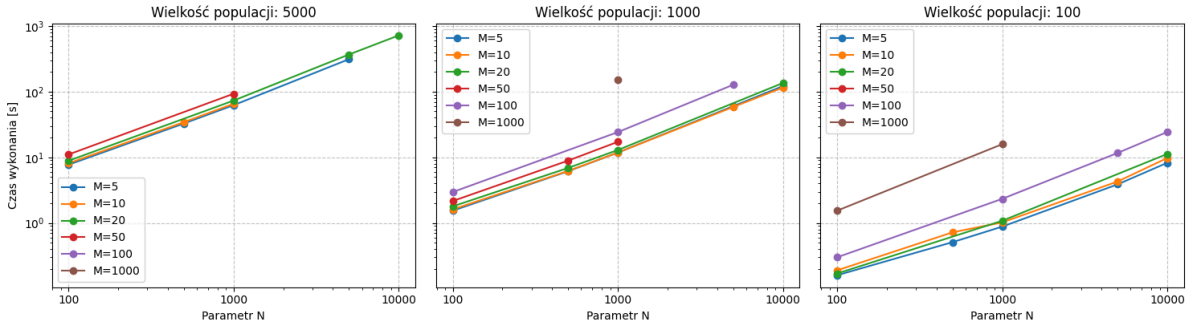
Profilowanie:

Nazwa funkcji	Udział łączny procesora [jednostka, %]	Udział własny procesora [jednostka
Algorytm-GA-CPU-pararell (PID: 11296)	25560 (100,00%)	0 (0,00%)
[Kod systemowy] ntdll.dll!0x00007ffeb262c53c	25550 (99,96%)	4 (0,02%)
mainCRTStartup	25546 (99,95%)	0 (0,00%)
__scrt_common_main	25546 (99,95%)	0 (0,00%)
__scrt_common_main_seh	25546 (99,95%)	0 (0,00%)
invoke_main	25542 (99,93%)	0 (0,00%)
main	25542 (99,93%)	15 (0,06%)
makespan	8002 (31,31%)	2302 (9,01%)
order_crossover	7400 (28,95%)	7 (0,03%)
std::vector<std::vector<int,std::allocator<int> >,std::allocator<std::vector<int,std::allocator<int> > > >::push_back	6276 (24,55%)	2 (0,01%)

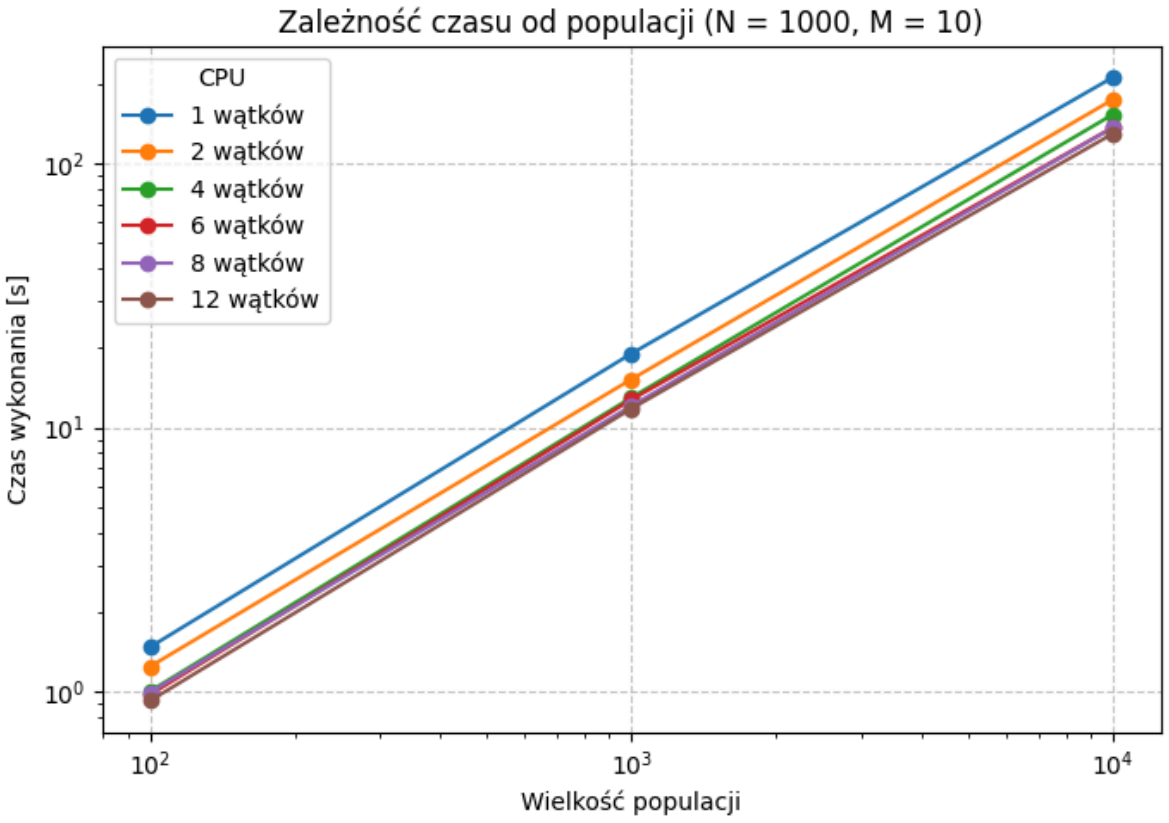
1. Selekcja turniejowa zwracała całego osobnika, co powodowało wielokrotne kopiowanie wektorów. Zmieniono tak, żeby zwracała indeks najlepszego osobnika. Cały rodzic jest pobierany bezpośrednio z populacji przez referencję. Po tej zmianie czas zredukował się z 32,72s do 29,311s.
2. Zamiast sortować całą populację w celu wyboru najlepszych osobników, nie sortujemy ich w pełni a wybieramy tylko 2 najlepszych.
3. Optymalizacja makespan: zamiast obliczać pełny harmonogram jak już wiadomo że rozwiązanie jest gorsze to przerywa. **Czas 28,126s.**

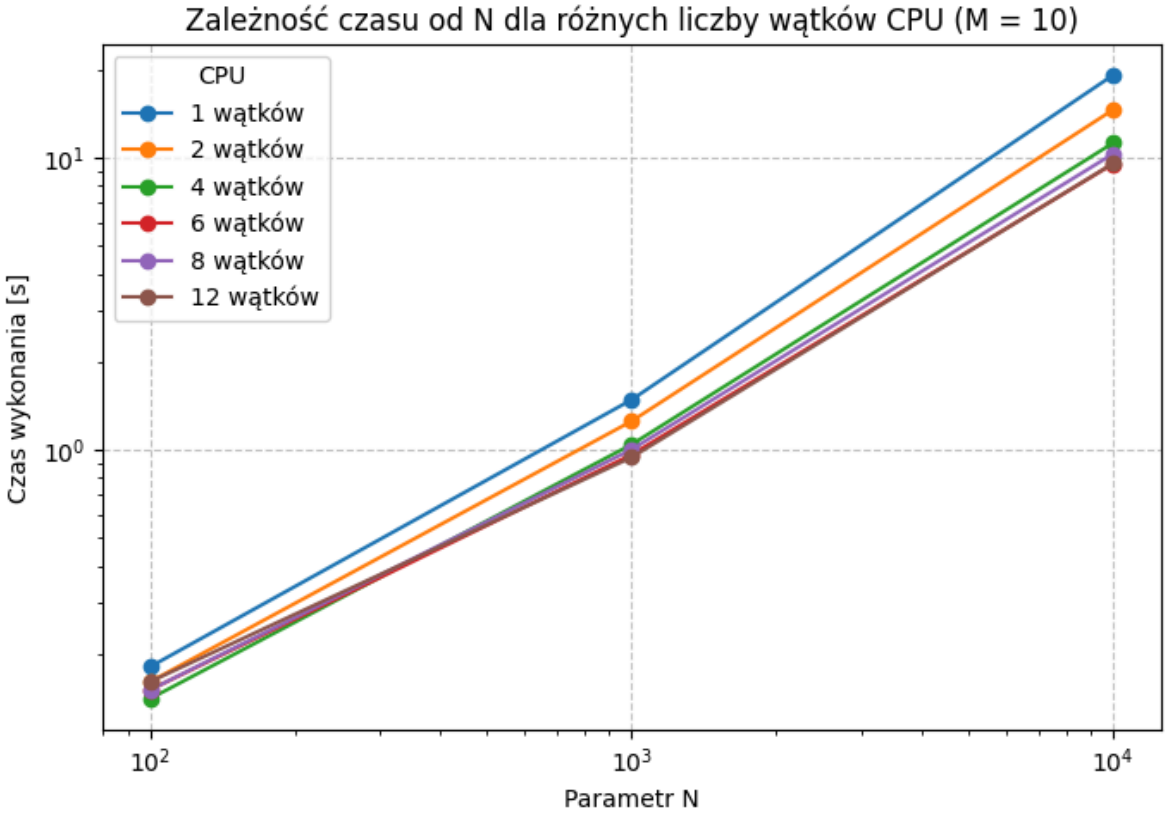


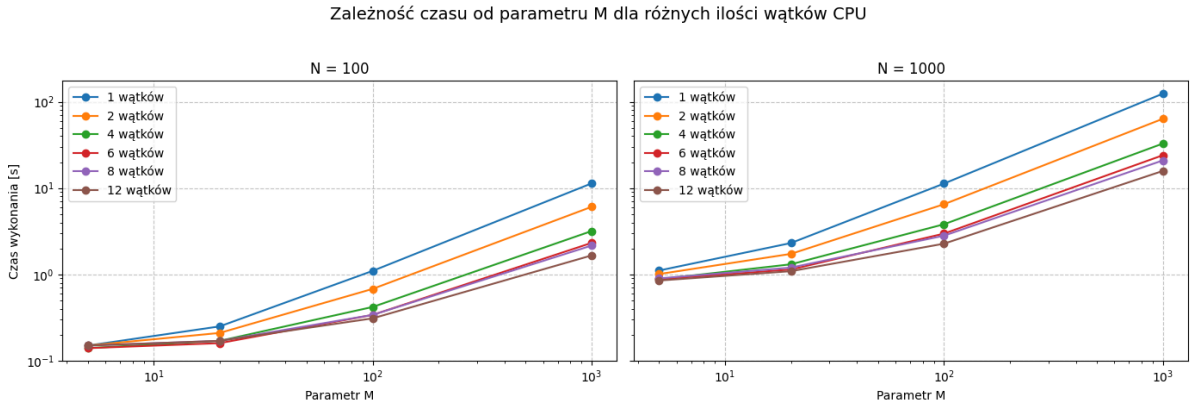
Wpływ parametrów N, M i pop na czas wykonania algorytmu



Przyspieszenie dla różnych wielkości problemu





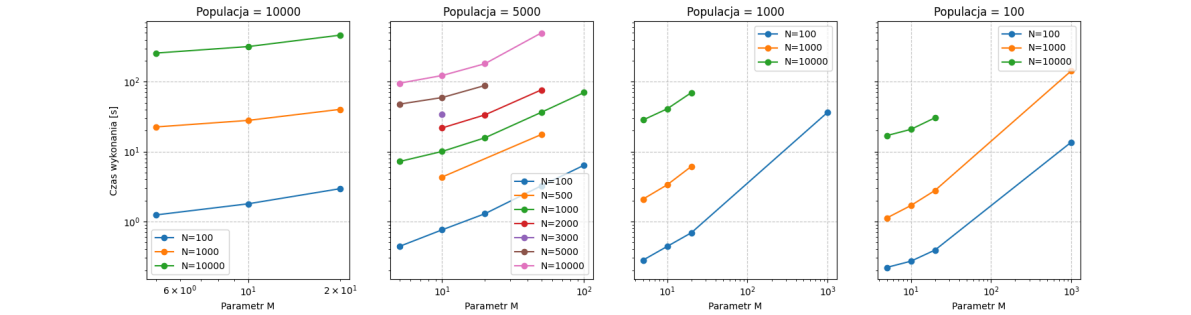


W przypadku algorytmu równoległego, na procesorze 6 rdzeni 12 wątków, przyspieszenie wynosiło od 147% do 790%, więc efektywność przyspieszenia wynosiła od 12% do 66%. Przyspieszenie było bardziej efektywne dla większego rozmiaru rozwiązywanego problemu, w szczególności przy zwiększonym M-liczbie maszyn w problemie FSSP.
Niska efektywność przyspieszenia była spowodowana przez niskie użycie wielu wątków dla mniejszych problemów, przy danych gdzie M=1000, Użycie CPU w trakcie działania algorytmu wynosiło 90-100% a efektywność przyspieszenia 66%.

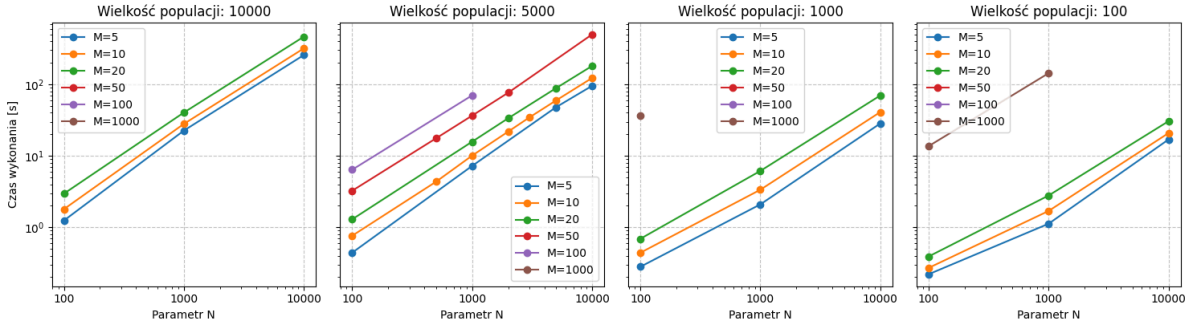
Rozproszony

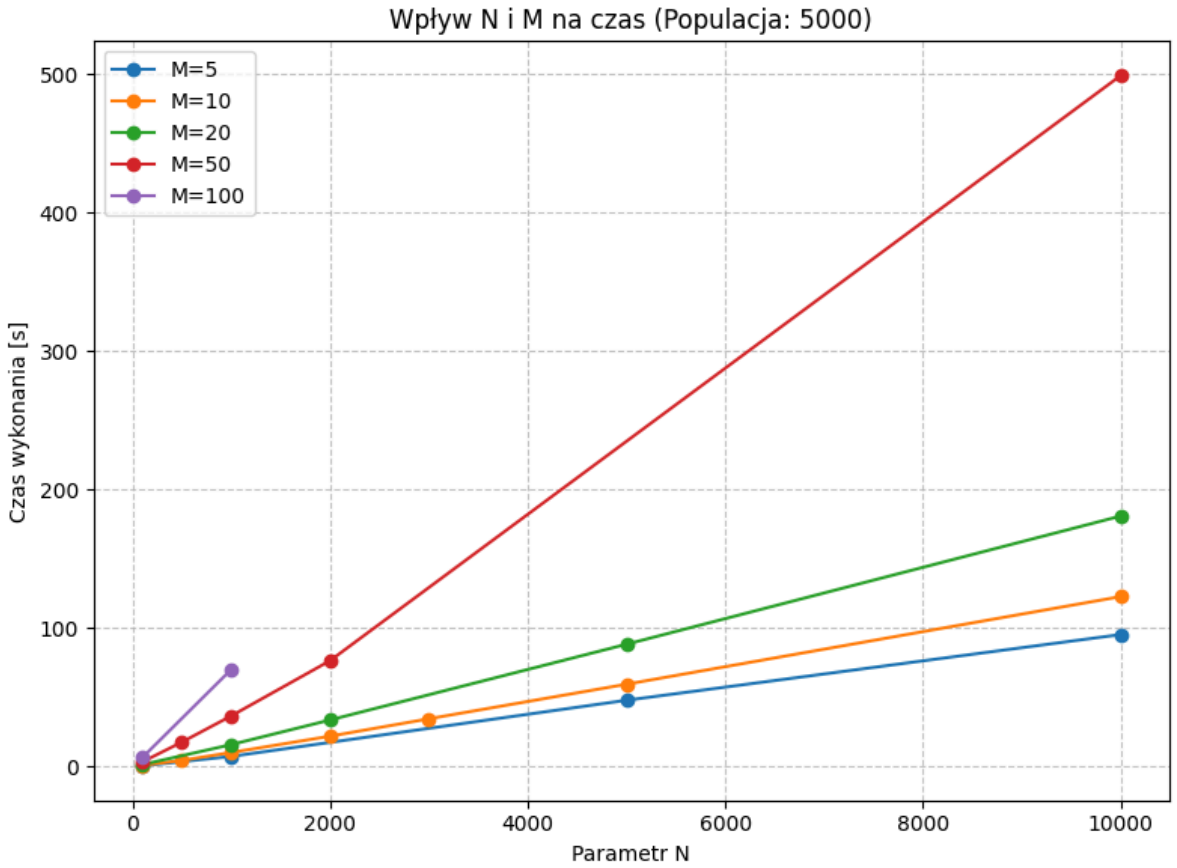
Algorytm rozproszony działał podobnie jak algorytm równoległy na którym jest bazowany, narzut związany z komunikacją nie wpłynął znacznie na wyniki eksperymentów.
Przy małych problemach zaobserwowano około 8% dłuższy czas działania algorytmu wysyłającego 100x więcej danych (mig-int 1 vs mig-int 100) w ciągu 10 000 generacji algorytmu genetycznego

GPU



Wpływ parametrów N, M i pop na czas wykonania algorytmu





Wersja algorytmu na GPU osiąga około 400% przyspieszenia przy populacji 1000 osobników i 1700% przyspieszenia na populacji 5000 osobników.