SZEREGOWANIE WIELOKRYTERIALNE W ZASTOSOWANIACH SIECIOWYCH

Jarosław RUDY, Dominik ŻELAZNY

Streszczenie: W dobie usług internetowych i obliczeń w chmurze, natura usług sieciowych podlega ewolucji. Zamiast krótkich żądań transferu statycznych zasobów wielkości kilku megabajtów, usługi sieciowe coraz częściej wymagają przesłania znacznej ilości danych i znacznej mocy obliczeniowej, co sprawia, że czas potrzebny na wypracowanie i dostarczenie odpowiedzi obejmuje minuty lub nawet godziny. Zależność ta zachodzi również w przypadku farm serwerów czy też klastrów komputerowych, które używają pojedynczego rozdzielacza zadań (*task dispatcher*) lub równoważnika obciążeń (*load balancer*) do kontroli dostępu do serwerów końcowych (*back-end severs*). Jednakże, przy tak dużym wzroście czasu wypracowania odpowiedzi oraz wymagań klientów dotyczących jakości usług (*Quality of Service* – QoS), jest możliwe użycie bardziej złożonych metod rozdzielania żądań klientów oraz różnych kryteriów usługi. W pracy użyto do tego celu zmodyfikowanego algorytmu *Pareto Inspired Memetic Algorithm* (PIMA) do wielokryterialnego szeregowania przychodzących żądań i porównano go z innymi podejściami do implementacji równoważenia obciążeń, takimi jak *Round Robin* (RR), *Shortest Job First* (SJF) czy *Earliest Deadline First* (EDF).

Słowa kluczowe: wielokryterialne szeregowanie sieciowe, równoważenie obciążeń, algorytm memetyczny, Pareto optymalność, jakość usługi (QoS)

1. Wprowadzenie

W erze usług internetowych i obliczeń w chmurze coraz więcej zadań, pierwotnie wykonywanych na lokalnych maszynach, może być zlecanych do wykonania zdalnie. Użytkownicy takich usług nie potrzebują już dysponować specjalistycznym sprzętem lub oprogramowaniem, wymaganym do wykonania pewnych zadań samodzielnie. Program kliencki oraz podłączenie do sieci to wszystko czego potrzeba, by skorzystać z potrzebnych usług i wykonać wymagane zadania lub obliczenia na zdalnym serwerze. Co więcej, pojedyncza usługa jest często implementowana przez zbiór identycznych serwerów końcowych. W celu obsłużenia wielu żądań, poszczególne zadania są przypisywane do konkretnych serwerów przy użyciu algorytmów szeregowania. W przypadku klasycznych żądań (np. odwołania do strony HTTP) czas odpowiedzi jest kluczowym czynnikiem, więc do wybrania maszyny końcowej (*back-end machine*) wykorzystywane są szybkie algorytmy (karuzelowy RR, na przykład). Te podejścia realizują równoważenie obciążeń poprzez równą dystrybucję żądań pomiędzy dostępnymi serwerami oraz spełniają wymogi jakości usługi, rozumianych w kategoriach warunków szeregowania sieciowego, takich jak czas odpowiedzi czy przepustowość.

W tej pracy podjęto inne podejście. Rozpatruje się specjalną klasę żądań i usług, gdzie czas potrzebny na ich zrealizowanie jest wystarczająco długi, by bardziej złożone i czasochłonne metody, jak algorytmy genetyczne lub memetyczne, mogły być użyte do szeregowania nadchodzących żądań. Rozważamy też jakość usługi QoS na wyższym poziomie: zarówno użytkownik usługi jak i jej dostawca mogą sprecyzować kilka

kryteriów (jak czas odpowiedzi dla użytkownika, czas wykonania dla dostawcy oraz opóźnienia wykonania dla obu stron), które chcą oni optymalizować. W związku z tym użyliśmy algorytmu memetycznego dla wielokryterialnego szeregowania sieciowego i porównaliśmy wyniki z kilkoma szybkimi oraz prostymi algorytmami konstrukcyjnymi. Pozostała część pracy podzielona jest jak następuje. W sekcji 2 przedstawiono krótki przegląd problemów szeregowania sieciowego i równoważenia obciążenia w jego klasycznym sensie. Potem, w sekcji 3, skupia się na naszym podejściu wielokryterialnym do szeregowania sieciowego na przykładzie czasochłonnych zadań. W sekcji 4 opisano algorytm memetyczny użyty w szeregowaniu sieciowym oraz przedstawia się jego pochodzenie. Sekcja 5 zawiera wyniki badań porównawczych algorytmu i kilku innych zaimplementowanych algorytmów konstrukcyjnych. Na końcu, w sekcji 6 zaprezentowane są wnioski oraz podsumowanie pracy.

2. Klasyczne szeregowania sieciowe

W ujęciu klasycznym, szeregowanie sieciowe pojedynczej usługi internetowej jest najczęściej implementowane jako zbiór końcowych serwerów, które mogą być zarówno identyczne jak i różnić się w kategoriach wydajności (np. czasu potrzebnego na udzielenie odpowiedzi dla danego żądania), typowo w formie farmy serwerów. Zazwyczaj, żądania są relatywnie krótkie i proste, oraz mogą być obsługiwane przez dowolny z serwerów końcowych. W tym przypadku, żądania są rozdzielane przez system w momencie ich nadejścia i szybkie algorytmy, jak *Least Busy Machine* (LBM) lub karuzelowy *Round Robin* (RR), używane są do rozdzielenia pracy równomiernie pomiędzy serwery, tym samym implementując podstawowe równoważenie obciążeń. Pozwala to na zredukowanie liczby przeciążonych lub bezczynnych maszyn (w rezultacie przekłada się też na zmniejszenie konsumpcji energii), zmniejszenie średniego czasu wymaganego do przygotowania odpowiedzi oraz zrównoważenie transferu w samej sieci komputerowej. To z kolei jest postrzegane przez klienta jako dotrzymanie pewnego poziomu jakości usług (QoS). Niestety, pożądany poziom QoS rzadko jest specyfikowany przez klienta i najczęściej pozostaje niskopoziomowym parametrem połączenia sieci komputerowej. Jednak, inne czynniki jakości, jak QoE (*Quality of Experience*, czyli jakość doświadczenia) czy jakość usługi (*Quality of Service*), istnieją i mogą być użyte do pomiaru jakości na wyższym poziomie.

Dyskutując o problemach rozdzielania żądań (zadań) i równoważenia obciążeń, ważne jest by uwzględnić miejsce w systemie, gdzie powyższe czynności mają miejsce. Cardellini i inni [2] rozróżniają cztery różne typy rozdzielania żądań:

1. Po stronie klienta (*client-based*). W tym podejściu aplikacje klienckie same wybierają serwer, do którego chcą wysłać żądanie. Obejmuje to także serwery proxy, które mogą przechowywać poprzednie odpowiedzi. Podejście to ma ograniczone zastosowania i brakuje mu skalowalności.
2. Oparet o usługę DNS (*DNS-based*). Ta sytuacja zwykle zawiera mapowanie pojedynczego URL do wielu różnych adresów IP (*Internet Adress*), każdy na innej maszynie, poprzez użycie usługi DNS (*Domain Name System*). Podejście to jest rozwiązaniem niskiego poziomu, aczkolwiek maszyny końcowe pozostają niewidoczne dla świata zewnętrznego i użytkowników.
3. Oparte o rozdzielacz zadań (*dispatcher-based*). W tym wypadku ruch wszystkich połączeń klient-serwer zarządzany jest przez pojedynczy scentralizowany komponent. Pozwala to na kontrolę znacznie wyższego poziomu, ale może

prowadzić do problemów z wydajnością, gdyż pakiety sieciowe zostają nadpisane adresem rzeczywistego serwera końcowego. Decentralizacja może również doprowadzić do problemu wąskiego gardła (*bottleneck problem*), ponieważ rozdzielacz jest jedynym komponentem posiadającym wiedzę o serwerach końcowych i wszystkie żądania muszą zostać przez niego przetworzone.

4. Po stronie serwera (*server-side*). W tym podejściu serwery same mają zdolność do rozdzielania żądań i przeprowadzania równoważenia obciążenia. Wstępnie, żądanie wysyłane jest do jednego z serwerów końcowych przez użycie jednego z poprzednich podejść. Różnica polega na tym, że wybrany serwer może zdecydować się przekazać zadanie do innego serwera. Oznacza to, że to podejście jest przykładem systemu rozproszonego, gdzie serwery końcowe mają wiedzę o pozostałych serwerach i mogą wspólnie rozwiązywać zadania.

Pierwsze i ostatnie podejście cechuje niska skalowalność lub kontrola, więc nie będziemy brać ich pod uwagę w dalszej części tej pracy. Czwarte podejście wymaga dodatkowego przemyślenia. Systemy rozproszone, gdzie zadania mogą migrować pomiędzy różnymi węzłami, są całkiem powszechne i dokonują rozdzielania zadań na różne sposoby. Niektóre z nich (zobacz [7] dla przykładu) zakładają, że węzły współpracują ze sobą, więc obciążony węzeł może przekazać część swoich zadań do innego lub kilku innych węzłów. Jeśli taki „pomocniczy" węzeł nie jest obecnie dostępny, specjalne scentralizowane węzły są wykorzystywane jako tymczasowe równoważniki obciążenia. Inne rozwiązania zmuszają węzły do rywalizacji między sobą: w podejściu prezentowanym w pracy [5], każdy węzeł ma ograniczony budżet i próbuje uzyskać zadania poprzez wykupywanie ich (jak podczas normalnej aukcji). W ten sposób zadania mogą zmieniać węzły, ale muszą wrócić do węzła startowego na samym końcu. Węzły są zainteresowane wyłącznie własnymi zyskami, ale to z kolei prowadzi do poprawy wydajności całego systemu.

W tej pracy skupiamy się na ostatnim z wymienionych typów rozdzielania zadań, gdzie cały proces równoważenia obciążenia i szeregowania wykonuje się na jednym, dedykowanym, komponencie sieciowym (lub, bardziej generalnie, jednym takim komponencie dla każdej grupy serwerów końcowych), co upraszcza cały problem szeregowania sieciowego. W tym wypadku również istnieją pewne podejścia. Na przykład Cheng i inni [3] próbowali rozdysponować obciążenie i utrzymać zadany poziom QoS (dokładnie rzecz biorąc, średnią przepustowość i opóźnienie) poprzez użycie mechanizmów adaptacji oprogramowania. Zmiany w parametrach systemu są i pociągają za sobą pewne akcje adaptacji. Wśród tych akcji znajdują się: uruchamianie nowego serwera, wyłączanie bezczynnych serwerów oraz przekierowywanie użytkowników do innej grupy serwerów. Pozwala to na zwiększenie wybranych parametrów QoS bez potrzeby bezpośredniej interwencji w procesy rozporządzania zadań. Poza tym, są pewne inne podejścia, które proponują rozwiązania pomiędzy scentralizowanymi i rozproszonymi systemami rozdzielania zadań. Jeden taki przykład zaprezentowano w pracy [10], gdzie równoważenie obciążeń pozornie jest scentralizowane, ale węzły w sieci stacji roboczych lub klastra komputerów mogą wymieniać dane lub zadania. Praca ta analizuje również różnice między scentralizowanym i rozproszonym podejściem do równoważenia obciążenia, a także wskazuje różnicę pomiędzy globalnym i lokalnym równoważeniem obciążeń, gdzie każda lokalna grupa węzłów jest zarządzana przez pojedynczy scentralizowany rozdzielacz zadań.

W kwestii kryteriów optymalizacji, większość powyższych podejść przyjmuje tylko jedno kryterium, takie jak: średni czas odpowiedzi, czas wykonywania zadań lub zużycie energii. Wielokryterialne podejścia są znacznie mniej pospolite, jednak również występują. Na przykład, Ben-Bassat i Borovits [1] rozważają dwa oddzielne kryteria: maksymalna liczba zadań wykonany na jednostkę czasu oraz maksymalny czas bezczynności na maszynę. Jednakże, wzięli oni pod uwagę także trzecie kryterium, będące kombinacją dwóch powyższych. Jako inny przykład niech posłuży praca Garga i Singha [6], którzy biorą pod uwagę dwa kryteria: czas wykonania oraz całkowity koszt wykonania. Rozważają oni system gridowy i rozwiązują problem szeregowania za pomocą metaheurystyki NSPSO (*Non-dominated Sort Particle Swarm Optimization*) z wyborem ostatecznego rozwiązania pozostawionym użytkownikowi, będący formą wspomagania decyzji wielokryterialnych.

3. Zastosowane podejście

Rozważamy system komputerowy z $M = \{1, ..., m\}$ maszynami reprezentującymi serwery końcowe (węzły) oraz pojedynczą maszyną pracującą jako rozdzielacz zadań, która wykonuje równoważenie obciążenia oraz szeregowanie. Serwery nie mogą wymieniać się danymi pomiędzy sobą, a każdy z nich jest podłączony do rozdzielacza poprzez sieć komputerową. Następnie, dostajemy do wykonania $N = \{1, ..., n\}$ zadań (żądań) do wykonania. Zakładamy, że zadania wykonują się w systemie wsadowym (*batch mode*), bez ingerencji użytkownika. Wszystkie zadania przechodzą przez rozdzielacz i muszą zostać przydzielone do jednego z serwerów końcowych. Każde zadanie j-te posiada: czas przybycia A_j (tzn. czas w którym zadanie pojawiło się w systemie oraz najwcześniejszy czas kiedy może zostać rozdzielone i wykonane), żądany termin wykonania D_j (*deadline*, czyli ostateczny termin wykonania zadania bez kary za spóźnienie) oraz czas wykonania E_j (czas potrzebny na wykonanie zadania na serwerze końcowym). Serwery są identyczne, więc każde z zadań może zostać wykonane na dowolnym serwerze, a czas wykonania nie zależy od maszyny na której zadanie się wykonuje. Zakładamy również, że zadania nie mogą być przerywane, zawieszane (z wyjątkiem opisanym poniżej), wznawiane lub przesyłane na inną maszynę.

Czasy przybycia oraz ostatecznego wykonania łatwo określić, ale czas wykonania jest trudniejszy do ustalenia. Niewiele zadań ma przewidywalny czas wykonania, więc zakładamy że zadania z nieznanym czasem wykonania posiadają pewien limit dostępnego serwera (zależny na przykład od priorytetu). Jeśli nie zostaną ukończone przed tym czasem, zostają zawieszone i wznowione później. Zakładamy również, że czas potrzebny na przesłanie danych do serwera końcowego oraz przesłanie odpowiedzi do klienta są wliczone w czas wykonywania zadania.

Czas C_j jest terminem zakończenia wykonywania zadania j-tego (tzn. odpowiedź na żądanie dotarła do klienta). Kara za niewykonanie j-tego zadania w terminie przedstawiona jest wzorem: $P_j = max\ (0, C_j - D_j)$. Natomiast czas odpowiedzi $R_j = C_j - A_j$. Zakładamy również, że $L_j = 1$ gdy zadanie j-te jest spóźnione ($P_j > 0$) oraz $L_j = 0$ w przeciwnym wypadku. W związku z powyższym rozważamy następujące kryteria optymalizacji:

1. Średnia kara za spóźnienia: $\frac{1}{n}\sum_{j=1}^{N} P_j$.

2. Średni czas odpowiedzi: $\frac{1}{n}\sum_{j=1}^{N} R_j$.
3. Czas wykonania wszystkich zadań: $\max C_j$.
4. Liczba spóźnionych zadań: $\sum_{j=1}^{N} L_j$.
5. Maksymalna kara: $\max P_j$.
6. Maksymalny czas odpowiedzi: $\max R_j$.

Kryteria te są wysokopoziomowe (w porównaniu do niskopoziomowych parametrów QoS sieci, takich jak przepustowość) i mogą być użyte do sprecyzowania wymagań użytkownika końcowego, jak również wymagań zarządców systemu sieciowego. Niektóre z powyższych kryteriów są istotne dla klienta (na przykład średni czas odpowiedzi oraz średnia kara), podczas gdy pozostałe mają większe znacznie dla właścicieli usługi (np. czas wykonania, maksymalna i średnia kara). W naszej pracy, kilka funkcji celu składających się z dwóch lub trzech różnych w/w kryteriów.

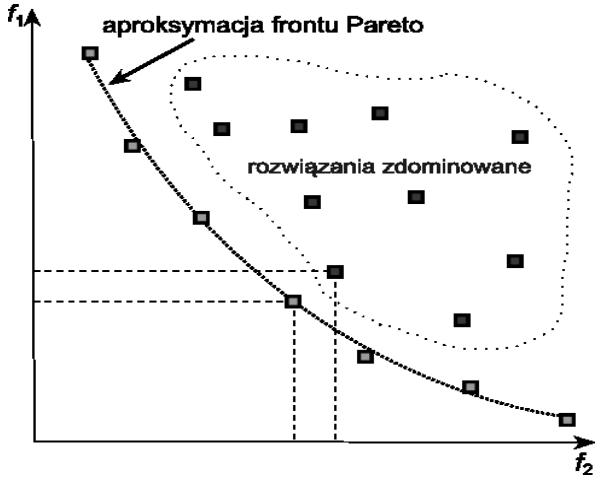
Kluczowym założeniem naszej pracy jest to, że wykonywane zadania są czasochłonne, tzn. ich czas wykonania jest dostatecznie długi, by móc zastosować przy ich rozdzielaniu złożone algorytmy szeregowania. Ta sytuacja nie ma miejsca, gdy zadania składają się z prostych żądań jak obsługa strony HTTP, gdzie wykonujemy niewiele obliczeń lub przesyłamy małe ilości danych. Jednakże, możemy również wziąć pod uwagę złożone obliczenia inżynierskie lub symulacje (szczególnie teraz, gdy wiele usług migruje do obliczeń w chmurze i w związku z tym jest wykonywanych zdalnie). W takim wypadku, wykonanie zadań może zająć minuty lub godziny i nasze podejście znajduje realne zastosowanie. W związku z tym, dodajemy przychodzące zadania do kolejki nie przydzielonych zadań i czekamy na moment, w którym nie można już dłużej opóźniać przydziału zadań. Używamy wtedy algorytmu memetycznego bazującego na algorytmie NSGA-II w celu znalezienia uszeregowania wszystkich zebranych w kolejce zadań.

4. Opis algorytmu

W pracy [4] Deb i inni zasugerowali metaheurystykę NSGA-II (*Elitist Non-dominated Sorting Genetic Algorithm*), opartą na NSGA (*Non-dominated Sorting* GA), którą krytykowano z powodu dużej złożoności obliczeniowej sortowania rozwiązań niezdominowanych, brak elitarności i potrzebę sprecyzowania parametru udziału. Algorytm NSGA-II został pozbawiony powyższych wad poprzez dodanie metody szybkiego podziału rozwiązań niezdominowanych, estymatora gęstości oraz operatora porównania. Pozwoliło to na przyśpieszenie działania algorytmu oraz ulepszenie procesu znajdywania jednolicie rozłożonej aproksymacji frontu Pareto, patrz Rys. 1.

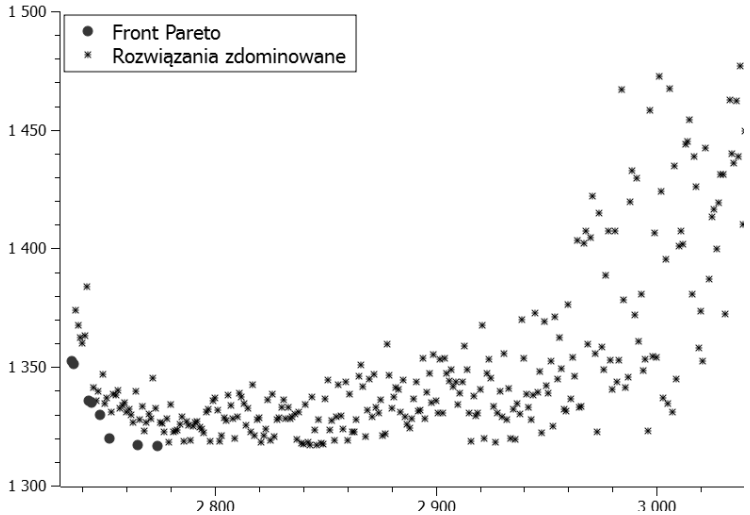
Szybkie sortowanie rozwiązań niezdominowanych dzieli zbiór rozwiązań, uzyskanych w obecnej iteracji algorytmu memetycznego, na fronty Pareto (patrz Rys. 2.) ze złożonością obliczeniową nie większą niż $O(mN^2)$. Każde z rozwiązań posiada dwie własności: a) liczbę rozwiązań dominujących dane rozwiązanie - n_i oraz b) listę rozwiązań zdominowanych przez dane rozwiązanie – S_i. Najpierw, wszystkie rozwiązania, których liczba n_i jest równa zero, są przenoszone do pierwszego frontu. Następnie, dla każdego rozwiązania w pierwszym froncie, zmniejszamy o jeden liczby n_i dla każdego rozwiązania

zdominowanego przez obecne. Nowe rozwiązania z liczbą n_i równą zero zostają przeniesione do frontu drugiego i ponownie sprawdzamy każde z nich, zmniejszając o jeden liczby n_i zdominowanych przez nie rozwiązań. Proces powtarzany jest, aż wszystkie rozwiązania zostaną przypisane do odpowiednich frontów.



Rys. 1. Rozwiązania niezdominowane - aproksymacja frontu Pareto

Następnie liczone są estymatory gęstości dla każdego z rozwiązań. We froncie, dla każdego rozwiązania poza pierwszym i ostatnim, liczone są powierzchnie jakie roztaczają najbliżsi sąsiedzi nad punktem *i*-tym. Wartość tę przypisujemy i-temu punktowi, jako wartość estymatora gęstości tego punktu. Im większa wartość estymatora, tym lepiej, gdyż szukamy punktów jak najbardziej różniących się od siebie w danym froncie, by móc poddać go klasteryzacji.



Rys. 2. Przykładowy rozkład rozwiązań dla problemu szeregowania zadań

Proces selekcji wymaga użycia operatora porównania, który steruje procesem poszukiwania w kolejnych iteracjach algorytmu, w celu znalezienia jednolicie rozłożonej aproksymacji frontu Pareto. W przypadku dwóch rozwiązań o różnych rangach (frontach w których się znajdują) wybieramy to o niższym froncie (tzn. wpierw to z pierwszego, następnie drugiego, itd), natomiast w przypadku gdy oba rozwiązania należą do tego samego frontu, preferujemy rozwiązanie o większej wartości estymatora gęstości. Pozwala nam to wybrać przede wszystkim rozwiązania, znajdując się w oddalonych od siebie rejonach przestrzeni, a tym samym bardziej różnorodnych.

Podczas pracy nad algorytmem memetycznym, zdecydowano się wykorzystać wspomniane wcześniej szybkie sortowanie, w celu obniżenia złożoności obliczeniowej algorytmu. Ze względu na zastosowanie, końcowa postać funkcji celu (a więc i ocena rozwiązań z frontów) przybiera postać znormalizowanej sumy kryteriów cząstkowych. Pozwoliło to na wybór jednego z wielu rozwiązań niezdominowanych bez konieczności angażowania decydenta (*decision maker*) oraz bez analizy wcześniej podjętych decyzji. W ten sposób wyeliminowaliśmy dodatkową złożoność obliczeniową, która wymagana była do oceny rozwiązań w oryginalnej pracy. Każde z rozwiązań reprezentuje pewną liczbę zbiorów rozwiązań, odpowiadającą liczbie węzłów w sieci i poddawane jest operacjom genetycznym, tzn. mutacji, krzyżowaniu i selekcji. Do krzyżowania użyto schematu PMX (*partially matched crossover*) oraz funkcji zamiany zadań między węzłami, w roli operatora mutacji [11]. Dodatkowo, użyto selekcji turniejowej przeprowadzanej na losowanych listach rozwiązań. Rozwiązania wybierane były poprzez leksykograficzne porównanie frontu oraz znormalizowanej sumy kryteriów, w obu przypadkach wygrywała niższa wartość.

Zróżnicowane zbiory rozwiązań, dostarczone przez operatory algorytmu genetycznego, nie są wstępnie uszeregowane ani nie dostarczają dobrych rozwiązań. Dla każdego osobnika w populacji potomnej uruchamiany jest algorytm konstrukcyjny. Poprzez metodę wstawień (patrz [8]) przygotowane są, uwzględniając wartości funkcji celu, uszeregowania dla każdego z węzłów. Wpierw ustawiane są zadania o najbliższym żądanym terminie zakończenia. Każde z zadań wstawiane jest we wszystkie możliwe pozycje w częściowym uszeregowaniu, a następnie wybierane jest wstawienie o najniższej znormalizowanej sumie funkcji celu [9]. Kolejne zadania ze zbioru dostępnego dla danego węzła w rozwiązaniu są wstawiane w ten sposób, aż wszystkie zadania zostaną uszeregowane.

W celu poprawy wydajności algorytmu, do początkowego zbioru rozwiązań dodawane są cztery rozwiązania bazujące na szybkich algorytmach konstrukcyjnych. W ten sposób, nasz algorytm jest sterowany w celu jak najszybszego znalezienia dobrych rozwiązań, które będą nie gorsze niż pozostałe rozwiązania. Sam algorytm jest elastyczny, co pozwala na zmianę liczby iteracji, rozmiaru populacji itp., w zależności od aktualnych potrzeb, wynikających ze zgromadzonych w kolejce do rozdzielenia żądań.

Sam algorytm działa on-line, ale dopóki nie ma przynajmniej kilku zadań na liście oczekujących, nie wykonuje części memetycznej, a rozdziela nadchodzące zadania do węzłów będących w stanie bezczynności. Gdy uruchomiony, bierze pod uwagę zajętość węzłów podczas sprawdzania wartości funkcji celu możliwych uszeregowań. Podczas testów, część memetyczna uruchamiana była co najmniej kilkukrotnie podczas symulacji każdej z przygotowanych instancji problemu.

5. Badania

Porównano wyniki naszego algorytmu memetycznego (*Memetic Algorithm* – MA) z rozwiązaniami dostarczonymi przez cztery znane algorytmy konstrukcyjne:

1. RR (*Round Robin*, algorytm karuzelowy) – rozdziela zadania, gdy tylko się pojawią, przydziela je do następnej maszyny na liście. Gdy nie ma już więcej maszyn, wraca na początek listy.
2. LBM (*Leasy Busy Machine*, najmniej zajęta maszyna) – rozdziela zadania gdy tylko pojawią się w systemie, przypisując je do najmniej zajętej maszyny. W przypadku naszego problemu, oznacza to maszynę, która najwcześniej ukończy wykonywanie przydzielonych jej aktualnie zadań.
3. SJF (*Shortest Job First*, najpierw najkrótsze zadanie) – dodaje przychodzące zadania do listy, a następnie rozdziela zebrane zadania tylko wtedy, gdy któraś z maszyn przejdzie do stanu bezczynności. Lista zadań posortowana jest niemalejąco po czasach wykonania, tak że najkrótsze zadania wykonywane są najpierw. Potem, algorytm SJF przydziela każde zadanie z listy kolejno do najmniej obciążonych maszyn.
4. EDF (*Earliest Deadline Firts*, najpierw najwcześniejszy wymagany termin zakończenia) – działa dokładnie jak algorytm SJF, ale sortuje zadania według rosnących żądanych terminów zakończenia, tak że najpilniejsze zadania rozdzielane są w pierwszej kolejności.

W celu porównania jakości prezentowanych przez algorytmy rozwiązań, postanowiliśmy wykorzystać poniższą metodykę. Posłużymy się instancją problemu I (zbiór zadań do wykonania). Wpierw dla każdego algorytmu A_i wyliczamy dwa wskaźniki, $B(A_i)$ oraz $W(A_i)$, zgodnie z poniższymi wzorami:

$$B(A_i) = \sum_{j=1}^{J} \sum_{c=1}^{C} d(A_i, A_j, c) \tag{1}$$

$$W(A_i) = \sum_{j=1}^{J} \sum_{c=1}^{C} d(A_j, A_i, c) \tag{2}$$

Gdzie: C – liczba kryteriów w funkcji celu,

J – liczba algorytmów (porównujemy 5, więc zawsze $J = 5$),

$d(A_i, A_j, c)$ – gdy algorytm i-ty ma lepszą wartość kryterium c od algorytmu j-tego (tzn. $A_i(c) < A_j(c)$) funkcja ta przyjmuje wartość 1, w przeciwnym wypadku 0.

Podsumowując, $B(A_i)$ oznacza ile razy dany algorytm był lepszy od pozostałych (pod względem zadanego kryterium), natomiast $W(A_i)$ określa ile razy ten sam algorytm był gorszy od pozostałych algorytmów. Współczynnik jakości algorytmu liczymy jako $Q(A_i) = B(A_i) - W(A_i)$, w ten sposób (dla danej instancji I) dla każdego algorytmu otrzymujemy pojedynczy współczynnik liczbowy, określający jego jakość.

Instancje problemu zostały skonstruowane w poniższy sposób: czas przybycia A_j zadania j-tego jest generowany losowo z normalnej dystrybuanty z transformacji Box-Mullera. Co więcej, $A_j \in [0, f \cdot N]$, gdzie N jest liczbą zadań oraz współczynnik f jest z przedziału [0.8, 1.2]. Czas wykonania E_j generowany z rozkładem jednostajnym w przedziale [10, 50]. Żądany termin zakończenia $D_j = A_j + k \cdot E_j$, gdzie współczynnik k jest generowane losowo zgodnie z rozkładem normalnym, na przedziale [1.1, 1.4]. Wzięto pod

uwagę instancje z 20, 50, 100 i 200 zadaniami (tzn. liczbą zadań, która została zebrana i rozdzielona jednorazowo). Liczba maszyn *M* była z przedziału od 5 do 15. Dla każdego przygotowanego typu wygenerowano 10 instancji [12]. Natomiast kryteria były zebrane w cztery zbiory testowe:

1. Średnia kara $\overline{P}$ oraz średni czas odpowiedzi $\overline{R}$.
2. Średni czas odpowiedzi $\overline{R}$ oraz maksymalny czas wykonania $\max C_j$.
3. Maksymalna kara $\max P_j$ oraz średni czas odpowiedzi $\overline{R}$.
4. Maksymalny czas odpowiedzi $\max R_j$, liczba spóźnionych zadań $|L|$ oraz maksymalny czas wykonania $\max C_j$.

Dla przykładu wyniki dla przypadku 10 instancji z 50 zadaniami i 5 maszynami oraz 4-tym zbiorem kryteriów przedstawione zostały w Tab. 1.

Tab. 1. Liczba rozwiązań Pareto-optymalnych

Instancja	**RR**	**LBM**	**SJF**	**EDF**	**MA**
1	0	-6	0	3	3
2	1	-9	2	3	3
3	-4	4	-4	2	2
4	-11	5	3	0	3
5	-11	-3	5	4	5
6	-11	-1	6	0	6
7	-2	3	0	-4	3
8	-11	-3	5	4	5
9	-8	3	4	-3	4
10	-2	3	0	-4	3
Suma	*-59*	*-4*	*21*	*5*	*37*

Przedstawione wyniki pokazują, że algorytmy konstrukcyjne są niepewne w przypadku przypisania wielu kryteriów do funkcji celu, ponieważ współczynnik jaki osiągnęły zależy w dużej mierze od instancji problemu, nawet gdy wszystkie instancje są tego samego typu. Jednakże, są również zbiory kryteriów (jak średni czas odpowiedzi i średnia kara), gdzie jeden algorytm konstrukcyjny w znacznym stopniu zdominował pozostałe algorytmy konstrukcyjne. Powyższa tabelka prezentuje również podsumowanie dla każdego z algorytmów oraz to, że zaproponowany algorytm najlepiej sobie radził z generowaniem rozwiązań niezdominowanych.

Następnie zaprezentowano wyniki badań w zależności od rozmiaru instancji oraz zbioru kryteriów w funkcji celu. Wyniki zaprezentowano w Tab. 2. Wzięto pod uwagę różne zbiory kryteriów (zaprezentowane wcześniej) i porównaliśmy całościowe wyniki dla wszystkich algorytmów. Jak łatwo zauważyć, algorytm karuzelowy (RR) wydaje się być najgorszym wyborem. Wśród algorytmów konstrukcyjnych, najlepiej spisującym się był algorytm SJF, niemniej został on zdominowany przez skonstruowany przez nas algorytm. W przypadku trzech kryteriów wszystkie, z wyjątkiem RR, algorytmy konstrukcyjne

spisały się podobnie. Wynika to z braku jednoznacznej i jawnej funkcji celu, przez co algorytmy te tracą na niezawodności, gdy funkcja celu składa się z więcej niż jednego kryterium. Wśród wyszczególnionych algorytmów konstrukcyjnych, najlepiej spisywał się algorytm SJF, który w przypadku pierwszych dwóch zbiorów niemalże dorównał algorytmowi MA.

Tab. 2. Podsumowanie wyników badań w relacji do zbioru kryteriów

Kryteria	**RR**	**LBM**	**SJF**	**EDF**	**MA**
Zbiór 1	-366	-171	264	-32	298
Zbiór 2	-355	-59	217	-39	226
Zbiór 3	-276	-12	64	78	146
Zbiór 4	-307	57	61	58	127

W Tab. 3. prezentuje się wyniki uzależnione od rozmiaru instancji. W tym przypadku zsumowane są wyniki dla wszystkich zbiorów rozwiązań. Ponownie, algorytm MA okazał się najlepszym ze wszystkich zaimplementowanych. Dodatkowo algorytm SJF również w tym wypadku okazał się najlepszym z konstrukcyjnych, a algorytm RR okazał się najmniej skutecznym.

Tab. 3. Podsumowanie wyników badań w zależności od instancji

Instancja	**RR**	**LBM**	**SJF**	**EDF**	**MA**
20 x 5	-214	-69	134	-8	157
50 x 5	-279	-80	141	35	182
100 x 5	-270	-8	116	21	140
100 x 10	-265	-32	151	-21	169
200 x 15	-273	7	82	41	143
Suma	-1301	-182	624	68	791

6. Wnioski

W pracy zaprezentowano wykorzystanie algorytmu memetycznego do wielokryterialnego szeregowania sieciowego przy założeniu zadań wymagających znacznych zasobów (zwłaszcza czasu). Kryteria szeregowania określone są przez użytkownika końcowego (jako część *Quality of Service*) lub zarządców systemu. W pracy porównano zmodyfikowaną wersję metheurystyki NSGA-II z czterema algorytmami konstrukcyjnymi pod względem ich zdolności do uzyskiwania niezdominowanych rozwiązań. Wyniki badań wykazują, że zastosowany przez nas algorytm z łatwością pokonuje algorytmy RR, LBM, EDF oraz jest zauważalnie lepszy niż algorytm SJF. Głównymi zaletami zaprezentowanego rozwiązania są: a) jawna postać funkcji celu, b) wykorzystanie algorytmów konstrukcyjnych do budowy populacji początkowej oraz c) duża elastyczność: algorytm może być łatwo rozbudowany o nowe formy operatorów krzyżowania / mutacji oraz strategii selekcji, a także o wsparcie dla ważonej funkcji celu i innych metod wspierających podejmowania decyzji. Czas działania algorytmu może być kontrolowany (od liczby iteracji, poprzez rozmiar populacji po wymuszone przerwanie algorytmu) w zależności od czasu dostępnego na ustalenie szeregowania. Co najważniejsze, rozwiązania dostarczone przez algorytm zawsze są co najmniej tak samo dobre jak te uzyskane dzięki algorytmom konstrukcyjnym i mogą podlegać dalszej poprawie, jednakże

możliwość ulepszenia zależy w dużej mierze od zastosowanych kryteriów jak i od profilu szeregowanych zadań.

Literatura

1. Ben-Bassat M., Borovits I.: Computer network scheduling. Omega, 3(1):119–123, February 1975.
2. Cardellini V., Colajanni M., Yu P. S.: Dynamic Load Balancing on Web-Server Systems. IEEE Internet Computing, 3(3):28–39, May 1999.
3. Cheng S.-W., Garlan D.,. Schmerl B. R, Sousa J. a. P., Spitznagel, Steenkiste P., Hu N.: Software Architecture-Based Adaptation for Pervasive Systems. Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing, ARCS'02, pp. 67–82, London, UK, UK, 2002. Springer-Verlag.
4. Deb K., Pratap A., Agarwal S., Meyarivan T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. Evolutionary Computation, IEEE Transactions on, 6(2):182 – 197, apr 2002.
5. Ferguson D., Yemini Y., Nikolaou C.: Microeconomic algorithms for load balancing in distributed computer systems. 8th International Conference on Distributed Computing Systems, pp. 491–499, 1988.
6. Garg R., Singh A. K.: Multi-objective workflow grid scheduling based on discrete particle swarm optimization. Proceedings of the Second international conference on Swarm, Evolutionary, and Memetic Computing - Volume Part I, SEMCCO'11, pp. 183–190, Berlin, Heidelberg, 2011. Springer-Verlag.
7. Jain G. D. P.: An Algorithm for Dynamic Load Balancing in Distributed Systems with Multiple Supporting Nodes by Exploiting the Interrupt Service. International Journal of Recent Trends in Engineering, 1(1):232–236, May 2009.
8. Nowicki E., Smutnicki C.: A fast taboo search algorithm for the job shop problem. Manage. Sci., 42(6):797–813, June 1996.
9. Pinedo M.: Scheduling: Theory, Algorithms and Systems. Prentice Hall, second edition.
10. Zaki M. J., Li W., Parthasarathy S.: Customized Dynamic Load Balancing for a Network of Workstations. Journal of Parallel and Distributed Computing, 43:156–162, 1995.
11. Zitzler E., Thiele L.: Multiobjective Optimization Using Evolutionary Algorithms - A Comparative Case Study. Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, PPSN V, pp. 292–304, London, UK, UK, 1998. Springer-Verlag.
12. Żelazny D., Rudy J.: Memetic algorithm approach for multi-criteria network scheduling, Proceedings of the International Conference on ICT Management, 247-261, 2012.

Mgr inż. Jarosław Rudy
Mgr inż. Dominik Żelazny
Instytut Informatyki Automatyki i Robotyki
Politechnika Wrocławska
50-370 Wrocław, ul. Janiszewskiego 11/17
Tel: (71) 320 27 45 / 321 26 77
e-mail: {jaroslaw.rudy, dominik.zelazny}@pwr.wroc.pl