Równoległe algorytmy szeregowania zadań produkcyjnych

Wojciech Bożejko

Wrocław University of Technology
Institute of Engineering
Janiszewskiego 11-17
50-372 Wrocław, Poland
wbo@ict.pwr.wroc.pl

marzec 2003

Spis treści

Streszczenie

Rozprawa dotyczy algorytmów szeregowania zadań dedykowanych do realizacji w środowiskach obliczeń równoległych, takich jak system wieloprocesorowy, klaster czy lokalna sieć komputerowa. Z jednej strony, silnie sekwencyjny charakter obliczeń wykonywanych przez znane obecnie algorytmy rozwiązywania problemów szeregowania stanowi znaczną przeszkodę w projektowaniu odpowiednich, wystarczająco efektywnych, algorytmów równoległych. Z drugiej strony, obliczenia równoległe oferują istotne korzyści w rozwiązywaniu trudnych problemów optymalizacji kombinatorycznej.

Praca podzielona jest zasadniczo na sześć części. Bardzo krótka część pierwsza zawiera wprowadzenie w problematykę szeregowania zadań, problemy szeregowania rozważane w pracy oraz klasyczne i najnowsze tendencje w optymalizacji dyskretnej. Nieco obszerniejsza część druga stanowi opracowane przez autora „state-of-art" dla dziedziny obliczeń równoległych, przygotowane w oparciu o bardzo obszerną bibliografię, Część ta omawia stosowane podejścia, modele, architektury, środowiska programowe i sprzęt do prowadzenia obliczeń równoległych, a także porusza zagadnienia projektowania i oceny algorytmów równoległych. Szczególną uwagę zwrócono na aspekty wykonywania w tym środowisku algorytmów optymalizacji kombinatorycznej. Następne cztery części stanowią trzon pracy i są całkowicie oryginalnymi rezultatami autora. Podział na części został dostosowany do strukturalnie odmiennych podejść, stosowanych przy projektowaniu algorytmów równoległych. Omawiane są kolejno: poszukiwania jednowątkowe, poszukiwania wielowątkowe, równoległe schematy B&B oraz wybrane narzędzia badania struktury przestrzeni rozwiązań.

W zakresie poszukiwań jednowątkowych, dedykowanych dla jednorodnych systemów wieloprocesorowych, zaproponowano szereg oryginalnych metod uwzględniających odmienne techniki projektowania algorytmów równoległych oraz różne potrzeby zgłaszane przez nowoczesne algorytmy optymalizacji dyskretnej (analiza pojedynczych rozwiązań, analiza lokalnych otoczeń, analiza rozwiązań rozproszonych). Szczególną uwagę zwrócono tu na problemy efektywności, kosztu oraz przyspieszenia obliczeń w zależności od typu problemu, jego wielkości oraz zastosowanego środowiska obliczeń równoległych. W analizach zwrócono uwagę na problemy obciążania procesorów w jedno, dwu i trójwymiarowych sieciach obliczeń. Dla poszczególnych algorytmów wyprowadzono teoretyczne oszacowania własności oraz przeprowadzono analizę porównawczą korzyści wynikających z zastosowania odpowiednich podejść.

W zakresie poszukiwań wielowątkowych, dedykowanych zarówno dla jednorodnych jak i niejednorodnych systemów wieloprocesorowych (takich

jak duże komputery typu *mainframe*, klastry, systemy rozproszone połączone sieciami komputerowymi) zaprojektowano i przebadano eksperymentalnie warianty równoległe najbardziej obiecujących aktualnie metod optymalizacji kombinatorycznej (poszukiwanie tabu, symulowane wyżarzanie, metody genetyczne) w zastosowaniu do wybranych problemów szeregowania zadań. Omówiono szczegółowo różne techniki realizacji wątków obliczeń oraz ich komunikacji, w tym w szczególności modele migracyjne (wyspowe) dla podejścia ewolucyjnego. Stwierdzono efekt występowania przyspieszenia ponadliniowego.

W zakresie wariantów równoległych schematu podziału i ograniczeń, dedykowanych dla jednorodnych i niejednorodnych systemów obliczeń równoległych, zaprojektowano i przebadano algorytmy tego typu dla wybranej klasy problemów szeregowania zadań. Metody te stanowią dobrą alternatywę dla sekwencyjnego schematu B&B, o zanikającej ostatnio popularności.

Ostatnia część pracy zawiera pomocnicze badania związane z analizą przestrzeni rozwiązań, wykorzystywane przy projektowaniu opisanych wcześniej algorytmów. Omówiono miary odległości w przestrzeniach kombinatorycznych, problemy wizualizacji przestrzeni, metody badania topologii i krajobrazu przestrzeni, metody analizy trajektorii poszukiwań wykonywanych przez różne algorytmy.

Wśród wielu dodatków zamieszczonych na końcu pracy, warto zwrócić uwagę na syntetyczne zestawienia wyników osiągniętych przez autora, dostępne w formie tabelarycznej w Dodatku C.

Abstract

The main issue discussed in the dissertation is the problem of using scheduling algorithms in parallel environments, such as multiprocessor systems, cluster or local network. On the one hand, sequential character of the scheduling algorithms' computation process is the obstacle in projecting enough effective parallel algorithms. On the other hand, parallel computations offer essential advantages of solving difficult problems of combinatorial optimization.

The dissertation is generally divided into six parts. The first one, which is very short, includes introduction to scheduling issues, scheduling problems considered in thesis and classical and the newest discreet optimization tendencies. Somewhat larger second part is the "state-of-art", work out for parallel computing by author, prepared basis on very extensive bibliography. This part discuss applied approaches, models, architecture, programming environments and computer hardware used in parallel computations. Issue of projecting and evaluation is placed in this part too, especially in context of performing combinatorial optimization algorithms in parallel environments. Next four parts make up the core of the work and are completely unique results of author. Dividing into parts is adjusted to structurally different approaches, applied to project parallel algorithms. There are single-thread search, multi-thread search, parallel branch and bound scheme and solution's space's research tools discussed in this part.

There is plenty of genuine single-thread search methods proposed in work, designed for homogeneous parallel systems. These methods take into consideration dissimilar techniques of parallel algorithm's projecting process and different necessities of modern algorithms of discreet optimization as well (analysis of one solution, analysis of local neighborhood, analysis of diffuse solutions). Efficiency, cost and computation's speedup depending on type of the problem, its size and environment of parallel system used is taken under special consideration in this part of the dissertation. Special focus is put on the problem of processors balance in one-, two- and three-dimensions computation's nets. Theoretical estimations of properties are derived for particular algorithms, comparing analysis of advantages result from application of different approaches has been done.

In the matter of multi-thread search, dedicated for homogeneous and heterogeneous multiprocessors systems (such as mainframe computers, clusters, diffuse systems connected by networks) some parallel variants of the most promising currently methods of combinatorial optimization (tabu search, simulated annealing, genetic methods) has been projected and researched experimentally, in the application of selected scheduling problems.

Different techniques of computation's threads' realization and its communication have been discussed, especially for migration models (so-called island models) in evolution methods. Effect of superlinear speedup has been observed.

In the matter of parallel variants of branch and bound scheme, dedicated for homogeneous and heterogeneous parallel systems, this type of algorithms has been projected and researched for selected class of scheduling problems. Such a methods are good alternative for sequential B&B scheme, which has been dying out recently.

The last part of the dissertation includes auxiliary researches connected with solutions space's analysis, used in projecting process of parallel algorithms described in previous parts of the work. Measures of the distance in combinatorial spaces, problems of spaces' visualization, methods of landscape and space's topology research and methods of analysis of algorithms search trajectory has been discussed.

There are many appendices at the end of the work, including synthesis of author's work's results presented in tabular form in Appendix C.

1

Wprowadzenie

1.1 Wstęp

Zdecydowana większość praktycznych zagadnień szeregowania zadań produkcyjnych należy do klasy problemów silnie NP-trudnych, czyli takich, dla których nie istnieją algorytmy rozwiązywania o wielomianowej złożoności obliczeniowej (jeśli $P \neq NP$). Póki co, istniejące algorytmy dokładne rozwiązywania NP-trudnych problemów szeregowania zadań mają z konieczności wykładniczą złożoność obliczeniową. Zwiększanie szybkości komputerów, nawet o rząd wielkości, jedynie w niewielkim stopniu wpływa na zwiększenie rozmiaru rozwiązywalnych problemów. Wobec tego możliwe są dwa, nie wykluczające się wzajemnie, podejścia pozwalające na rozwiązywanie przykładów problemów o dużych i bardzo dużych rozmiarach w akceptowalnym czasie:

1. algorytmy przybliżone,
2. algorytmy równoległe.

Jakość najlepszych rozwiązań wyznaczanych przez algorytmy przybliżone zależy, w dużym stopniu, od liczby analizowanych przez nie rozwiązań, a co za tym idzie od czasu obliczeń. Czas i jakość posiadają przeciwstawną tendencję zmian, w tym sensie że otrzymanie lepszego rozwiązania wymaga znaczącego wzrostu czasu obliczeń. Poprzez konstrukcję algorytmów równoległych można znacznie zwiększyć liczbę rozpatrywanych rozwiązań (w jednostce czasu) wykorzystując efektywnie wieloprocesorowe środowisko obliczeń.

Jak do tej pory, w literaturze pojawiło się niewiele prac dotyczących równoległych algorytmów szeregowania zadań produkcyjnych. Stan ten wynika z faktu, iż jest to dziedzina interdyscyplinarna, zawieszona niejako po-

między dwoma gałęziami nauki: automatyką – od strony zastosowań, a informatyką – od strony teorii algorytmów i obliczeń równoległych. Istniejące prace dotyczące algorytmów równoległych rzadko odnoszą się do problemu szeregowania zadań, a z kolei prace z dziedziny szeregowania zadań rzadko opisują stosowanie komputerów równoległych w procesie eksperymentów obliczeniowych. Zupełnie brak jest własności teoretycznych równoległych algorytmów szeregowania zadań. Podejście kompleksowe, a przy tym syntetyczne, pozwoliłoby na podsumowanie istniejącego stanu badań oraz wypełnienie tej luki badawczej. Praca niniejsza ma spełniać to zadanie.

1.2 Tezy pracy

Zasadniczym celem pracy jest wykazanie, że posługując się systemem obliczeń wieloprocesorowych, takim jak komputer równoległy, klaster lub rozległa sieć połączonych ze sobą komputerów, można zrealizować następujące obliczenia:

- wyznaczać rozwiązania problemów szeregowania zadań szybciej, niż czynią to obecnie znane metody; teza ta nie jest banalna ze względu na silnie sekwencyjny charakter obliczeń w aktualnie stosowanych algorytmach,
- mając ustalony czas obliczeń, można otrzymać lepsze jakościowo wyniki, to znaczy wyznaczyć rozwiązanie bliższe optymalnemu, niż czynią to algorytmy sekwencyjne,
- mając ustalony czas obliczeń T, możliwe jest wyznaczenie przez algorytm równoległy w czasie T na p procesorach rozwiązań lepszych, niż rozwiązania wyznaczone przez algorytm sekwencyjny w czasie $T \cdot p$ – czyli osiągnięcie tzw. przyspieszenia ponadliniowego (*superlinear speedup*),
- mając ustaloną liczbę procesorów p (lub rząd jej wielkości), posługując się teoretycznym modelem komputera równoległego, można wyznaczyć teoretyczną złożoność obliczeniową kluczowych elementów algorytmów szeregowania zadań w taki sposób, by cały proces obliczeń był kosztowo optymalny.

Powyżej sformułowane tezy zostaną w pracy udowodnione w następującym procesie badawczym:

- przedstawione zostaną modele matematyczne wybranych szczególnie trudnych problemów szeregowania zadań produkcyjnych,

- udowodnione zostaną szczególne własności algorytmów związane z zastosowaniem środowiska obliczeń równoległych do rozwiązania problemu,
- opracowane zostaną szybkie algorytmy równoległe rozwiązywania badanych zagadnień szeregowania,
- przeprowadzone zostaną eksperymenty obliczeniowe na reprezentatywnych przykładach testowych z literatury,
- przedstawione zostaną wnioski wynikające z przeprowadzonych badań.

Prowadzone badania mają charakter interdyscyplinarny. Obejmują one między innymi takie dziedziny jak: teorię i praktykę projektowania algorytmów, teorię i praktykę obliczeń równoległych, teorię i praktykę szeregowania zadań, metody dokładne i przybliżone rozwiązywania problemów optymalizacji kombinatorycznej, metody sztucznej inteligencji, teorię złożoności obliczeniowej.

1.3 Zakres pracy

Praca podzielona jest zasadniczo na sześć części. Część pierwsza (rozdziały 2 i 3) zawiera krótkie wprowadzenie w problematykę szeregowania, problemy szeregowania rozważane w pracy oraz klasyczne techniki stosowane w optymalizacji dyskretnej. Część druga (rozdziały 4 i 5) stanowi wprowadzenie w techniki, modele, środowiska i sprzęt do prowadzenia obliczeń równoległych oraz zagadnienia projektowania i oceny algorytmów równoległych. Rozdziały 2 – 5 są oparte w znaczącej części na wynikach literaturowych, przy czym syntezy zawarte w rozdziałach 4 i 5 wykonane przez autora w oparciu o bardzo obszerną bibliografię, mają charakter opracowania „state-of-art". Dalsze cztery części, rozdziały 6 – 9, stanowią trzon pracy i są całkowicie oryginalnymi rezultatami autora. Rozdział 6 dostarcza wielu oszacowań teoretycznych dla procesów poszukiwań jednowątkowych, uwzględniając odmienne techniki projektowania algorytmów równoległych oraz różne potrzeby zgłaszane przez nowoczesne algorytmy optymalizacji dyskretnej. Szczególną uwagę zwrócono w nim na problemy efektywności, kosztu oraz przyspieszenia obliczeń w zależności od typu problemu, jego wielkości oraz środowiska obliczeń równoległych. Rozdział 7 prezentuje alternatywne metody równoległych poszukiwań wielowątkowych wraz z obszernym materiałem eksperymentalnym. Przedstawiono w nim warianty

równoległe najbardziej obiecujących aktualnie metod optymalizacji kombinatorycznej. Rozdział 8 odnosi się do wariantów równoległych schematu podziału i ograniczeń, stanowiących alternatywę dla sekwencyjnego schematu B&B (o zanikającej popularności). Rozdział 9 zawiera pomocnicze badania związane z analizą przestrzeni rozwiązań, których wyniki były wykorzystywane przy projektowaniu opisanych wcześniej algorytmów. Spośród kilku dodatków zamieszczonych na końcu warto zwrócić uwagę na syntetyczne zestawienia wyników osiągniętych przez autora, dostępne w formie tabelarycznej w Dodatku C.

2

Problemy szeregowania zadań

W niniejszym rozdziale zostaną opisane problemy szeregowania zadań, które umożliwiają modelowanie i analizę zarówno prostych systemów wytwarzania, pracę pojedynczych stanowisk (problemy jednomaszynowe, patrz przegląd w [87]), jak i bardzo złożonych systemów produkcyjnych (problemy przepływowe, gniazdowe) mających aplikacje praktyczne na przykład w procesach przemysłu elektronicznego, samochodowego czy chemicznego (przegląd zastosowań w [171]). Wszystkie opisywane problemy należą do klasy problemów NP-trudnych.

Jednym z pierwszych etapów rozwiązywania problemu optymalizacyjnego dotyczącego szeregowania zadań produkcyjnych jest budowa jego modelu matematycznego. W procesie modelowania pojawiają się elementarne pojęcia: *zadanie* oraz *zasób*. Zadanie polega na wykonaniu ciągu pewnych czynności, zwanych operacjami, potrzebujących w procesie wykonywania określonych zasobów. Z zadaniem związane mogą być dane: termin gotowości, żądany termin zakończenia (*deadline*), przerywalność wykonania (czyli możliwość przerwania i późniejszego wznowienia wykonywania zadania), podzielność operacji (dekompozycja), sposoby wykonywania operacji (specyficzne żądania zasobowe, alternatywne sposoby wykonywania). Zasoby z kolei mogą być odnawialne (jak procesor, maszyna, pamięć operacyjna), nieodnawialne (materiały eksploatacyjne, surowce) oraz podwójnie ograniczone (energia, kapitał). Z zasobami związane są ograniczenia – dla odnawialnych jest to ograniczenie strumienia dostępności, dla nieodnawialnych – ograniczenie dostępnej ilości, dla podwójnie ograniczonych – oba rodzaje ograniczeń. Cechami zasobów są: dostępność (wyrażona przez przedziały czasowe), koszt, ilość, podzielność, przywłaszczalność. Konstruując model

matematyczny zagadnienia, wszystkie wyżej wymienione cechy należy sformalizować w sposób matematycznie jednoznaczny.

Oznaczmy przez $\mathcal{J} = \{1,2,...,n\}$ zbiór zadań, które mają być wykonane przy użyciu zbioru różnych typów maszyn $M = \{1, 2, \ldots, m\}$. Każde zadanie i jest ciągiem o_i operacji $O_i = (l_{i-1} + 1, l_{i-1} + 2, \ldots, l_i)$, $l_i = \sum_{k=1}^{i} o_k$, $l_0 = 0$. Operacje w obrębie zadania muszą być wykonane w określonym porządku technologicznym (w pewnej ustalonej kolejności), tzn. operacja j musi być wykonana po zakończeniu wykonywania operacji $j-1$, a przed wykonaniem operacji $j+1$. Dla uproszczenia zapisu zbiór operacji zadania i oznaczmy przez $\mathcal{O}_i$. Dla każdej operacji $j \in \mathcal{O}$, $\mathcal{O} = \bigcup_{i=1}^{n} \mathcal{O}_i$ określane są następujące pojęcia:

M_j – ciąg m_j podzbiorów maszyn, określający alternatywne sposoby wykonywania operacji; $M_j = (M_{1j}, M_{2j}, ..., M_{m_j,j}), M_{ij} \subseteq M$; operacja j potrzebuje do wykonania zestawu maszyn M_{ij} gdzie $1 \leqslant i \leqslant m_j$,

p_{ij} – czas wykonywania operacji j sposobem i-tym (np. na i- tej maszynie),

v_j – sposób wykonywania operacji (zmienna decyzyjna),

S_j – termin rozpoczęcia wykonywania operacji (zmienna decyzyjna),

C_j – termin zakończenia wykonywania operacji, $C_j = S_j + p_{v_j j}$.

Z kolei dla zadania i określone są między innymi następujące pojęcia:

o_i – liczba operacji w zadaniu,

r_i – najwcześniejszy możliwy termin rozpoczęcia wykonywania zadania,

d_i – żądany czas zakończenia wykonywania zadania,

$\mathcal{S}_i$ – termin rozpoczęcia wykonywania zadania, $\mathcal{S}_i = S_{l_{i-1}+1}$,

$\mathcal{C}_i$ – termin zakończenia wykonywania zadania, $\mathcal{C}_i = C_{l_i}$,

$\mathcal{L}_i$ – nieterminowość zakończenia zadania, $\mathcal{L}_i = \mathcal{C}_i - d_i$,

$\mathcal{T}_i$ – spóźnienie zakończenia zadania, $\mathcal{T}_i = \max\{0, \mathcal{C}_i - d_i\}$,

$\mathcal{E}_i$ – przyspieszenie rozpoczęcia zadania, $\mathcal{E}_i = \max\{0, r_i - \mathcal{S}_i\}$,

$f_i(t)$ – niemalejąca funkcja kosztu związana z zakończeniem wykonywania zadania i w chwili $t \geqslant 0$,

$\mathcal{F}_i$ – czas przepływu zadania i przez system, $\mathcal{F}_i = \mathcal{C}_i - r_i$,

$\mathcal{U}_i$ – jednostkowe spóźnienie zadania i definiowane jako

$$\mathcal{U}_i = \begin{cases} 0 & \text{gdy} \quad C_i \leqslant d_i, \\ 1 & \text{w przeciwnym przypadku.} \end{cases} \tag{2.1}$$

Większość problemów szeregowania zadania nie wymaga podania wszystkich wymienionych danych oraz zmiennych decyzyjnych. Zwykle używa się minimalnego zbioru pojęć wystarczającego do opisu problemu. Na przykład gdy dla każdego $j \in \mathcal{O}$ zachodzi $m_j = 1, |M_{1j}| = 1$ oznacza to, że problem

posiada maszyny dedykowane, bowiem zmienne decyzyjne v_j nie podlegają wyborowi. Wówczas v nie występuje w modelu problemu.

W celu precyzyjnego określenia problemu szeregowania opracowano w pracy [81] notację trójpolową $\alpha|\beta|\gamma$ rozwiniętą następnie w kolejnych pracach [116, 161] (podobna notacja istnieje także dla systemów masowej obsługi). W notacji tej przyjmuje się, że α – oznacza typ zagadnienia, β – dodatkowe specyficzne ograniczenia zagadnienia, γ – postać funkcji celu. Tutaj znaczenie tych symboli przyjęto za pracą [170] [1].

Symbol α jest złożeniem trzech symboli $\alpha_3\alpha_2\alpha_1$, mających następujące znaczenie. Symbol α_1 określa skończoną (daną) liczbę maszyn w systemie: 1,2,...; jeśli liczba ta nie jest określona z góry, to używa się symbolu pustego mającego sens dowolnej liczby maszyn m. Symbol α_2 określa sposób przejścia zadań przez system, przy czym wyróżniono następujące tradycyjne sposoby, w zależności od struktury procesu wytwarzania:

- F – przepływowy (*flow shop*), w którym wszystkie zadania posiadają jednakową marszrutę technologiczną, wymagają obsługi na wszystkich stanowiskach (maszynach), zaś każde stanowisko wymaga określenia oddzielnej sekwencji wprowadzania zadań,
- F* – przepływowy permutacyjny (*permutation flow shop*), model który ma takie same założenia jak F z dodatkowym wymaganiem, aby kolejność obsługi zadań na wszystkich maszynach była jednakowa (zgodna z kolejnością wprowadzania zadań do systemu),
- J – gniazdowy (*job shop*), w którym zadania mogą posiadać różne (co do liczby i kolejności odwiedzania gniazd) marszruty technologiczne,
- G – ogólny (*general shop*), w którym każde zadanie jest pojedynczą operacją, zaś zależności technologiczne są dane dowolnym grafem,
- I – równoległy (*parallel shop*), w którym każde zadanie jest pojedynczą operacją, oraz wszystkie operacje są wykonywane na dokładnie jednej z kilku równoległych (tego samego typu) maszyn,
- O – otwarty (*open shop*), w którym wszystkie operacje zadania mają być wykonane, lecz kolejność technologiczna operacji w zadaniu nie jest określona.

Symbol α_3 określa tryb realizacji poszczególnych operacji zadania. Jeśli α_3 jest symbolem pustym to przyjmuje się, że każda operacja ma jednoznacznie określoną (dedykowaną) maszynę, na której będzie wykonywana,

[1] Stosowany tam system oznaczeń jest pewną mutacją „klasycznego" literaturowego, umożliwiającą wygodny opis bardziej skomplikowanych zagadnień szeregowania.

to znaczy $m_j = 1$, $|M_{ij}| = 1$, $j \in \mathcal{O}$. Inaczej zakłada się, że zachodzi $m_j \geqslant 1$, $|M_{ij}| = 1$, $i = 1, 2, \ldots, m_j$, $j \in \mathcal{O}$, oraz operacja może być wykonywana na dokładnie jednej z identycznych maszyn równoległych (P), na jednej z maszyn jednorodnych (Q), lub jednej z maszyn niejednorodnych (R).

Symbol β określa istnienie dodatkowych założeń i ograniczeń, jak na przykład różnych terminów zgłoszeń (najwcześniejszych możliwych terminów rozpoczęcia wykonywania zadania, r_i), istnienia narzuconego częściowego porządku technologicznego wykonywania zadań (*prec*), ograniczenia bez czekania (*no wait*), bez magazynowania (*no store*), i inne. Pełniejszy wykaz dodatkowych założeń i ograniczeń znaleźć można w pracy [170].

Ostatni wymieniany parametr γ przyjmuje jedną z symbolicznych postaci funkcji kryterialnej. W teorii i praktyce szeregowania występują zasadniczo dwie klasy funkcji γ, a mianowicie

$$f_{\max} = \max_{1 \leqslant i \leqslant n} f_i(\mathcal{C}_i) \tag{2.2}$$

oraz

$$\sum f_i = \sum_{i=1}^{n} f_i(\mathcal{C}_i), \tag{2.3}$$

gdzie $f_i(t)$ są pewnymi funkcjami niemalejącymi. Klasy te obejmują, między innymi, szereg najczęściej stosowanych w praktyce kryteriów, takich jak na przykład: długość uszeregowania

$$C_{\max} = \max_{1 \leqslant i \leqslant n} \mathcal{C}_i, \tag{2.4}$$

średni czas przepływu

$$\sum F_i = \frac{1}{n} \sum_{i=1}^{n} \mathcal{F}_i, \tag{2.5}$$

czy też ważona suma spóźnień zadań

$$\sum w_i T_i = \sum_{i=1}^{n} w_i \mathcal{T}_i = \sum_{i=1}^{n} w_i \max\{0, \mathcal{C}_i - d_i\}. \tag{2.6}$$

Poniżej podamy dokładniejsze specyfikacje problemów rozważanych w niniejszej pracy, zwracając szczególną uwagę na charakter i złożoność obliczeniową elementów składowych odpowiednich algorytmów ich rozwiązywania. Problemy zostały wybrane z klasycznej teorii szeregowania tak, by pokryć najbardziej reprezentatywne klasy zagadnień, a mianowicie: (a) problemy o typowo sekwencyjnym procesie obliczeń (problemy jednomaszynowe), (b)

problemy o 2D, 3D regularnym procesie obliczeń (problemy przepływowe), (c) problemy o nieregularnym procesie obliczeń (problemy gniazdowe i hybrydowe).

2.1 Problemy jednomaszynowe

Problemy jednomaszynowe umożliwiają modelowanie pracy pojedynczych stanowisk wytwórczych w systemach wytwarzania oraz stanowią element składowy wielu bardziej zaawansowanych (wielomaszynowych) modeli szeregowania. W problemie tym każde z n zadań (indeksowanych $1, 2, \ldots, n$) należy wykonać na jednej maszynie sekwencyjnie, bez przerywania.

Rozwiązaniem opisanego zagadnienia jest harmonogram pracy stanowiska (maszyny) reprezentowany przez wektory terminów rozpoczęcia $S = (S_1, S_2, \ldots, S_n)$ oraz zakończenia wykonywania zadań $C = (C_1, C_2, \ldots, C_n)$; ze sformułowania problemu mamy natychmiast $\mathcal{C}_j \equiv C_j$, $j \in \mathcal{J}$. W praktyce, ponieważ $C_j = S_j + p_j$, $j \in \mathcal{J}$, zatem rozwiązanie jest całkowicie charakteryzowane przez jeden z tych wektorów. Jeśli funkcja celu jest regularna (niemalejąca), to rozwiązanie optymalne leży w klasie uszeregowań dosuniętych w lewo na osi czasu i może być jednoznacznie reprezentowane kolejnością wykonywania zadań na stanowisku, ta zaś jest przedstawiana za pomocą permutacji $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$ elementów zbioru $\mathcal{J}$. Dla danej permutacji π, terminy zakończenia wykonywania C_j wyznacza się w oparciu o wzór rekurencyjny

$$C_{\pi(j)} = C_{\pi(j-1)} + p_{\pi(j)}, \quad j = 1, 2, \ldots, n. \tag{2.7}$$

gdzie $\pi(0) = 0$ oraz $C_0 = 0$, lub nierekurencyjny wynikający z (2.7)

$$C_{\pi(j)} = \sum_{i=1}^{j} p_{\pi(j)}, \quad j = 1, 2, \ldots, n. \tag{2.8}$$

Realizacja wzoru (2.7) wymaga czasu $O(n)$, z (2.8) – $O(n^2)$.

Dalej, niech π będzie permutacją zadań ze zbioru $\mathcal{J}$, a Π zbiorem wszystkich takich permutacji. Dla problemu $1||\gamma$ należy wyznaczyć permutację $\pi^* \in \Pi$ taką, że:

$$F(\pi^*) = \min_{\pi \in \Pi} F(\pi) \tag{2.9}$$

gdzie dla $\gamma \in \{\sum f_i\}$ mamy

$$F(\pi) = \sum_{j=1}^{n} f_{\pi(j)}(C_{\pi(j)}). \tag{2.10}$$

zaś dla $\gamma \in \{f_{max}\}$

$$F(\pi) = \max_{1 \leqslant j \leqslant n} f_{\pi(j)}(C_{\pi(j)}). \tag{2.11}$$

Dość często spotykanym w praktyce jest uogólnienie poprzedniego problemu, poprzez wprowadzenie niezerowych terminów gotowości (najwcześniejszych możliwych terminów rozpoczęcia wykonywania zadania) r_j dla każdego zadania $j \in \mathcal{J}$. Terminy te można interpretować jako czasy pojawiania się zgłoszeń i są one znane a priori. Oznacza to, że każdy poszukiwany harmonogram pracy maszyny musi spełniać warunek $r_j \leqslant S_j$ dla $j \in \mathcal{J}$. Rozważane uogólnienie łączyć można ze wszystkimi wymienionymi wcześniej funkcjami kryterialnymi - na przykład rozważając funkcję celu $F(\pi) = \sum_{j=1}^{n} f_{\pi(j)}(C_{\pi(j)})$ i wprowadzając niezerowe terminy gotowości r_j otrzymamy problem oznaczony $1|r_i|\sum f_i$. Dla tak otrzymanego problemu terminy zakończenia wykonywania zadań dla ustalonej π można wyznaczyć ze wzoru rekurencyjnego

$$C_{\pi(j)} = \max\{C_{\pi(j-1)}, r_{\pi(j)}\} + p_{\pi(j)} \tag{2.12}$$

lub nierekurencyjnego

$$C_{\pi(j)} = \max_{1 \leqslant i \leqslant j}(r_{\pi(i)} + \sum_{s=i}^{j} p_{\pi(s)}), \tag{2.13}$$

dla $j = 1, 2, \ldots, n$. W praktyce do wyznaczania kryterium, dla danej π, jest stosowany wzór (2.12) o mniejszym koszcie $O(n)$, bowiem bezpośrednia realizacja wzoru (2.13) prowadzi do metody o złożoności $O(n^2)$.

2.2 Problemy przepływowe

Przytoczymy definicję zgodną z oznaczeniami z prac [141] i [170]. Dany jest zbiór n zadań $\mathcal{J}$={1,2,...,n} oraz zbiór m maszyn M={1,2,...,m}. Zadanie $j \in \mathcal{J}$ jest ciągiem m operacji O_{1j}, O_{2j},...,O_{mj}. Operację O_{ij} należy wykonać, bez przerywania, na maszynie i w czasie p_{ij}. Wykonywanie zadania na maszynie i (dla $i = 2, 3, \ldots, m$) może się rozpocząć dopiero po zakończeniu wykonywania tego zadania na maszynie $i - 1$.

Rozwiązaniem jest harmonogram pracy maszyn reprezentowany przez macierze terminów rozpoczęcia $S = (S_1, S_2, \ldots, S_n)$, gdzie $S_j = (S_{1j}, S_{2j}, \ldots, S_{mj})$ oraz zakończenia zadań $C = (C_1, C_2, \ldots, C_n)$, gdzie $C_j = (C_{1j}, C_{2j}, \ldots, C_{mj})$. Ze sformułowania mamy natychmiast $\mathcal{C}_j \equiv C_{mj}, j \in \mathcal{J}$. W praktyce, ponieważ $C_{ij} = S_{ij} + p_{ij}$, zatem rozwiązanie jest całkowicie

charakteryzowane przez jedną z tych macierzy. Jeśli funkcja jest regularna, to harmonogram optymalny jest dosunięty w lewo na osi czasu, zatem można go poszukiwać w zbiorze takich rozwiązań. W tym przypadku każde rozwiązanie może być jednoznacznie reprezentowane kolejnością wykonywania zadań na maszynie i, ta zaś z kolei jest reprezentowana permutacją $\pi_i = (\pi_i(1), \pi_i(2), \ldots, \pi_i(n))$ elementów ze zbioru $\mathcal{J}$. Jeśli permutacje π_i mogą być różne na różnych maszynach i, to odpowiedni problem jest oznaczany jako "ogólny" (F), jeśli zaś wszystkie permutacje π_i są takie same (czyli kolejność wykonywania zadań jest na wszystkich maszynach taka sama), to problem jest określany jako "permutacyjny" (F^*).

Permutacyjny problem przepływowy

Niech π =($\pi(1)$, $\pi(1)$,...,$\pi(n)$) będzie permutacją zadań {1,2,...,n}, a Π zbiorem wszystkich takich permutacji. Każda permutacja $\pi\in\Pi$ wyznacza jednoznacznie kolejność wykonywania zadań na maszynach (na każdej maszynie taką samą). W pracy omawiane będą problemy związane z kilkoma różnymi kryteriami: ogólnymi f_{max} czy $\sum f_i$ oraz szczególnymi: terminem zakończenia wszystkich zadań oraz sumą terminów zakończenia zadań.

Do wyznaczania C_{ij} używa się wzoru rekurencyjnego

$$C_{i\pi(j)} = \max\{C_{i-1,\pi(j)}, C_{i,\pi(j-1)}\} + p_{i\pi(j)},$$
$$i = 1, 2, \ldots, m, \quad j = 1, 2, \ldots, n, \tag{2.14}$$

z warunkiem początkowym

$$C_{i\pi(0)} = 0, \quad i = 1, 2, \ldots, m, \quad C_{0\pi(j)} = 0, \quad j = 1, 2, \ldots, n,$$

lub nierekurencyjnego

$$C_{i\pi(j)} = \max_{1=j_0\leqslant j_1\leqslant\ldots\leqslant j_i=j} \sum_{s=1}^{i} \sum_{j=j_{i-1}}^{j_i} p_{s\pi(j)}. \tag{2.15}$$

Ponieważ złożoność obliczeniowa metody ze wzoru (2.14) jest $O(mn)$, zaś tej ze wzoru (2.15) jest $O(\binom{j+i-2}{i-1}(j+i-1)) = O(\frac{(n+m)^{n-1}}{(n-1)!})$, w praktyce używany jest wyłącznie pierwszy z niech. Drugi wzór jest używany tylko do dowodzenia własności.

W teorii szeregowania rozpatrywane są najczęściej dwa NP-trudne problemy $F^*||f_{max}$ oraz $F^*||\sum f_i$. W obu należy wyznaczyć permutację $\pi^* \in \Pi$ taką, że:

$$F(\pi^*) = \min_{\pi\in\Pi} F(\pi) \tag{2.16}$$

Rysunek 2.1: Graf $G(\pi)$.

gdzie

$$F(\pi) = \sum_{j=1}^{n} f_{\pi(j)}(C_{m\pi(j)}), \quad F(\pi) = \max_{1 \leqslant j \leqslant n} f_{\pi(j)}(C_{m\pi(j)}) \tag{2.17}$$

dla $\gamma \in \{\sum f_i, f_{max}\}$ odpowiednio. Dla $r_j = const$ problem ten jest równoważny temu z kryterium średniego czasu przepływu.

Model grafowy

Wielkości C_{ij} ze wzorów (2.14) i (2.15) można także wyznaczyć posługując się modelem grafowym problemu. Dla danej kolejności wykonywania zadań $\pi \in \Pi$ tworzymy graf $G(\pi) = (M \times N,\ F^0 \cup F^*)$, gdzie $M = \{1, 2, \ldots, m\}$, $N = \{1, 2, \ldots, n\}$.

$$F^0 = \bigcup_{s=1}^{m-1} \bigcup_{t=1}^{n} \{((s,t),(s+1,t))\} \tag{2.18}$$

jest zbiorem łuków technologicznych (pionowych), zaś

$$F^* = \bigcup_{s=1}^{m} \bigcup_{t=1}^{n-1} \{((s,t),(s,t+1))\} \tag{2.19}$$

jest zbiorem łuków kolejnościowych (poziomych). Łuki grafu $G(\pi)$ nie posiadają obciążeń, natomiast obciążenie każdego węzła (s,t) wynosi $p_{s,\pi(t)}$. Czas C_{ij} zakończenia wykonywania zadania $\pi(j)$, j = 1,2,...,n, na maszynie i, i = 1,2,...,m, odpowiada długości najdłuższej ścieżki prowadzącej z wierzchołka (1,1) do wierzchołka (i,j), wraz z obciążeniem tego ostatniego. Dla problemu $F^*||C_{max}$, wartość funkcji kryterialnej dla ustalonego π jest równa długości ścieżki krytycznej w grafie $G(\pi)$.

2.3 Hybrydowe problemy przepływowe

Problem przepływowy z maszynami równoległymi (*flow shop with parallel machines*, FSPM), zwany także hybrydowym, lub ogólnym problemem przepływowym, może być opisany następująco: dany jest system składający się z m gniazd, każde gniazdo l złożone z $m_l \geqslant 1$ identycznych maszyn.

Niech zbiór gniazd będzie oznaczony przez $M = \{1,...,m\}$ a zbiór maszyn w gnieździe l przez $M_l = \{1,...,m_l\}$, $l \in M$. Zbiór n zadań opisany przez $\mathcal{J}=\{1,2,...,n\}$ ma zostać wykonany przez system. Każde zadanie przechodzi przez wszystkie gniazda w tym samym porządku 1,2,...,m. Zadanie $j \in \mathcal{J}$ składa się z m operacji związanych z wykonaniem zadania j w gnieździe l w czasie $p_{lj} > 0$. Wykonywanie zadania na maszynie nie może być przerywane. Dla każdego gniazda l zadanie j może być wykonywane na jednej z m_l maszyn. Każda maszyna może wykonywać co najwyżej jedno zadanie w danym momencie czasowym, oraz każde zadanie może być wykonywane co najwyżej na jednej maszynie w danym momencie czasowym. Uszeregowanie *dopuszczalne* jest zdefiniowane zbiorem par $(i_{lj}, C_{lj}), j \in \mathcal{J}, l \in M$, gdzie $i_{lj} \in M_l$ jest maszyną, na której jest wykonywane zadanie j w gnieździe l, a $C_{lj} > 0$ jest czasem zakończenia wykonywania zadania j w gnieździe l (na maszynie i_{lj}) w taki sposób, że powyższe ograniczenia są spełnione. Ze sformułowania problemu mamy $\mathcal{C}_j \equiv C_{mj}$.

Omawiany problem może być matematycznie sformalizowany w następujący sposób. Pojęcie wsadu (*batch*) niech oznacza podzbiór zadań przypisanych do gniazda. Ponieważ maszyny w gniazdach są identyczne, a każde zadanie przypisane do gniazda może wykonywać się na każdej z jej maszyn, więc wsad nie jest związany z żadną maszyną ani kolejnością. W każdym gnieździe l zbiór $\mathcal{J}$ musi być rozbity na m_l wsadów $\mathcal{J}_{li} \subset n$, $i \in M_l$; $n_{li} = |\mathcal{J}_{li}|$, $i \in M_l$, $l \in M$. Kolejność wykonywania zadań wsadu $\mathcal{J}_{li}$ może być opisana permutacją

$$\pi_{li} = (\pi_{li}(1), \pi_{li}(2), ..., \pi_{li}(n)) \in P(\mathcal{J}_{li}), \tag{2.20}$$

gdzie $\pi_{li}(k)$ oznacza element $\mathcal{J}_{li}$ który występuje na pozycji k w π_{li} a $P(\mathcal{J}_{li})$ jest zbiorem wszystkich permutacji zbioru $\mathcal{J}_{li}$. Proces wykonywania zadań w gnieździe l może być dokładnie opisana przez zbiór m_l permutacji $\pi_l = (\pi_{l1}, \pi_{l2},...,\pi_{lm_l})$, każda permutacja odpowiadająca jednemu wsadowi. Kolejność wykonywania zadań jest zdefiniowana przez m-kę $\pi = (\pi_1, \pi_2,..., \pi_m)$. Zbiór wszystkich takich kolejności wykonywania to

$$\Pi = \{\pi = (\pi_1, \pi_2, ..., \pi_m) : \pi_l = (\pi_{l1}, \pi_{l2}, ..., \pi_{lm_l}),$$

$$(\pi_{li} \in P(\mathcal{J}_{li}), i \in M_l, l \in M)\}. \tag{2.21}$$

Dla danej $\pi \in \Pi$, czasy zakończenia zadań mogą być wyznaczone z pomocą następującego wzoru rekurencyjnego, dla k=1,2,...,n_{li}, i=1,2,...,m_l, l=1,2,...,m:

$$C_{l,\pi_{li}(k)} = \max\{C_{l,\pi_{li}(k-1)}, C_{l-1,\pi_{li}(k)}\} + p_{l,\pi_{li}(k)}, \tag{2.22}$$

Rysunek 2.2: Przykład grafu $G(\pi)$ dla problemu przepływowego hybrydowego.

gdzie $\pi_{li}(0) = 0$, $i = 1,2,...,m_l$, $C_{l,0} = 0$, $l = 1,2,...,m$, $C_{0,j} = 0$, $j = 1,2,...,n$. Rozpatrywać będziemy dwa rodzaje funkcji kryterialnej, a mianowicie f_{max} i $\sum f_i$. Dla kryterium f_{max} naszym celem jest wyznaczenie kolejności wykonywania zadań π^* takiej, że

$$f_{\max}(\pi^*) = \min_{\pi\in\Pi} f_{\max}(\pi), \quad \text{gdzie} \quad f_{\max}(\pi) = \max_{j\in N} f_j(C_{m,j}). \tag{2.23}$$

Problem ten w dalszej części pracy oznaczany będzie przez $FP||f_{max}$.

Z kolei dla kryterium $\sum f_i$ należy wyznaczyć π^*, takie że

$$F(\pi^*) = \min_{\pi\in\Pi} F(\pi), \quad \text{gdzie} \quad F(\pi) = \sum_{j=1}^{n} f_j(C_{m,j}). \tag{2.24}$$

Problem ten w dalszej części pracy oznaczany będzie przez $FP||\sum f_i$.

Aby opisać uszeregowanie dopuszczalne dla danej kolejności wykonywania zadań π należy przypisać obciążenia $\mathcal{J}_{li}$, $i \in M_l$ maszyn w gnieździe l, $l \in M$. Ponieważ maszyny w gnieździe są identyczne, rozkład obciążeń jest jednoznacznie określony rozbiciem zbioru $\mathcal{J}$. Uszeregowanie dopuszczalne (i_{lj}, C_{lj}), $j \in \mathcal{J}$, $l \in M$, można otrzymać z kolejności wykonywania zadań π w następujący sposób: niech $i_{lj} = i$, gdzie $i \in M_l$, $j \in \mathcal{J}_{li}$, a C_{lj} można wyznaczyć z (2.22).

Model grafowy

Wielkości C_{lj} dla danej kolejności wykonywania zadań $\pi \in \Pi$ można także wyznaczyć posługując się poniższym modelem grafowym. Dla danej kolejności π tworzymy graf $G(\pi) = (O,\ E^* \cup E(\pi))$ ze zbiorem wierzchołków $O = M \times \mathcal{J}$ oraz zbiorem łuków $E^* \cup E(\pi)$, gdzie

$$E^* = \bigcup_{j=1}^{n} \bigcup_{l=1}^{m-1} \{((l,j),(l+1,j))\}, \tag{2.25}$$

$$E(\pi) = \bigcup_{l=1}^{m} \bigcup_{i=1}^{m_l} \bigcup_{k=1}^{n_{li}-1} \{((l,\pi_{li}(k)),(l,\pi_{li}(k+1)))\}. \tag{2.26}$$

Łuki ze zbioru E^* reprezentują drogę zadań poprzez gniazda, podczas gdy łuki ze zbioru $E(\pi)$ reprezentują kolejność wykonywania wsadów. Każdy węzeł $(l, j) \in O$ reprezentuje l –tą operację zadania j i ma wagę p_{lj}. Łuki mają obciążenia zerowe. Terminy zakończenia C_{lj} wyznaczone z równania rekurencyjnego (2.22) odpowiadają długości najdłuższych ścieżek dochodzących do wierzchołka (l, j) (wliczając w to wartość p_{lj}) w grafie $G(\pi)$. W tak skonstruowanym grafie wartość funkcji kryterialnej $C_{max}(\pi)$ jest długością najdłuższej drogi (ścieżki krytycznej) w $G(\pi)$. Z kolei wartość kryterium $C_{sum}(\pi) = \sum_{j=1}^{m} C_{mj}$ można obliczyć sumując po wszystkich $j \in \mathcal{J}$ długości najdłuższych ścieżek pomiędzy wierzchołkami (1,j) i (m,j). Przykład grafu $G(\pi)$ dla pewnej ustalonej kolejności wykonywania zadań π przedstawiono na rysunku 2.2.

2.4 Problemy gniazdowe

Dany jest zbiór zadań $\mathcal{J} = \{1,2,...,n\}$, zbiór maszyn (gniazd) $M = \{1,2,...,m\}$ i zbiór operacji $\mathcal{O} = \{1,2,...,o\}$. Zbiór $\mathcal{O}$ jest dekomponowany na podzbiory odpowiadające zadaniom. Zadanie j składa się z sekwencji o_j operacji indeksowanych kolejno przez (l_{j-1}+1, l_{j-1}+2,...l_j), które powinny zostać wykonane w zadanej kolejności, gdzie $l_j = \sum_{i=1}^{j} o_i$ jest całkowitą liczbą operacji pierwszych j zadań, j = 1,2,...,n, $l_0 = 0$, $\sum_{i=1}^{n} o_i = o$. Operacja i musi być wykonana na maszynie $v_i \in M$ w nieprzerywalnym czasie $p_i > 0$, $i \in \mathcal{O}$. Każda maszyna może wykonywać co najwyżej jedną operację w dowolnej chwili czasu. Rozwiązaniem dopuszczalnym jest wektor terminów rozpoczęcia wykonywania operacji S= (S_1, S_2,...,S_o) taki, że spełnione są powyższe ograniczenia, tzn:

$$S_{l_{j-1}+1} \geqslant 0, \quad j = 1, 2, ..., n, \tag{2.27}$$

$$S_i + p_i \leqslant S_{i+1}, \quad i = l_{j-1} + 1, l_{j-1} + 2, ..., l_j - 1, \quad j = 1, 2, ..., n, \tag{2.28}$$

$$S_i + p_i \leqslant S_j \quad \text{albo} \quad S_j + p_j \leqslant S_i, \quad i, j \in O, \quad v_i = v_j, \quad i \neq j. \tag{2.29}$$

Oczywiście, $C_j = S_j + p_j$. Do powyższych ograniczeń należy jeszcze dołączyć odpowiednią funkcję kryterialną. Najczęściej spotykane są następujące dwa kryteria: minimalizacja terminu zakończenia wykonywania wszystkich zadań oraz minimalizacja sumy czasów zakończeń wykonywania poszczególnych zadań. Ze sformułowania problemu mamy $\mathcal{C}_j \equiv C_{l_j}$, $j \in \mathcal{J}$.

Pierwsze kryterium, termin zakończenia wykonywania zadań

$$C_{\max}(S) = \max_{1 \leqslant j \leqslant n} C_{l_j}, \tag{2.30}$$

Rysunek 2.3: Przykład grafu dysjunktywnego dla problemu gniazdowego.

odpowiada problemowi oznaczanemu w literaturze przez $J||C_{max}$. Drugie kryterium, suma terminów zakończenia wykonywania zadań:

$$C(S) = \sum_{j=1}^{n} C_{l_j}, \tag{2.31}$$

odpowiada problemowi oznaczanemu w literaturze przez $J||\sum C_i$.

Oba opisywane problemy są NP-trudne i chociaż są podobnie modelowane, to drugi z nich jest powszechnie uznawany za trudniejszy z uwagi na brak pewnych szczególnych własności (tzw. własności blokowych, patrz [141]) wykorzystywanych w optymalizacji czasu działania algorytmów rozwiązywania powyższych problemów.

Model dysjunktywny

Model dysjunktywny opiera się na pojęciu grafu dysjunktywnego $G = (O^*, U^* \cup V)$. Graf ten posiada zbiór wierzchołków $O^* = O \cup \{0\}$ reprezentujących operacje (powiększony o dodatkową początkową operację sztuczną (0), dla której $p_0 = 0$), zbiór łuków koniunktywnych (skierowanych) przedstawiających kolejność technologiczną wykonywania operacji

$$U^* = U \cup U^0 = \bigcup_{j=1}^{n} \bigcup_{i=l_{j-1}+1}^{l_j-1} \{(i, i+1)\} \cup \bigcup_{j=1}^{n} \{(0, l_{j-1}+1)\} \tag{2.32}$$

oraz zbiór łuków dysjunktywnych (nieskierowanych) przestawiających możliwe kolejności realizacji operacji na maszynach

$$V = \bigcup_{i,j \in O, i \neq j, v_i = v_j} \{(i,j),(j,i)\}. \tag{2.33}$$

Przykład grafu dysjunktywnego przedstawiony jest na Rys. 2.3. Łuki dysjunktywne $\{(i,j),(j,i)\}$ są w istocie parami łuków skierowanych w odwrotnych do siebie kierunkach łączących wierzchołki i oraz j. Węzeł $i \in O$ posiada obciążenie p_i równe czasowi wykonania operacji O_i. Łuki mają obciążenia zerowe. Wybór dokładnie jednego łuku ze zbioru $\{(i,j),(j,i)\}$ odpowiada ustaleniu kolejności wykonywania operacji – "i przed j" lub "j przed i". Podzbiór $W \subset V$ zawierający wyłącznie łuki skierowane, co najwyżej jeden

Rysunek 2.4: Przykład grafu koniunktywnego dla problemu gniazdowego.

z każdej pary $\{(i,j),(j,i)\}$, nazywamy *reprezentacją* łuków dysjunktywnych. Reprezentacja taka jest kompletna, jeśli wszystkie łuki dysjunktywne posiadają określoną orientację. Reprezentacja kompletna, definiując relację poprzedzania wykonywania zadań na tej samej maszynie, generuje jedno rozwiązanie - niekoniecznie dopuszczalne, jeśli zawierać będzie cykle. Rozwiązanie dopuszczalne jest generowane przez reprezentację kompletną W taką, że graf $G(W) = (O, U \cup W)$ jest acykliczny. Dla uszeregowania dopuszczalnego wartości S_i wektora terminów rozpoczęcia wykonywania operacji S= (S_1, S_2,...,S_o) można wyznaczyć jako najdłuższe drogi dochodzące do wierzchołków grafu i (nie licząc samego p_i jako elementu sumy będącej długością drogi). Ponieważ graf $G(W)$ zawiera o wierzchołków i $O(o^2)$ krawędzi, zatem wyznaczenie wartości funkcji celu dla danej reprezentacji W wymaga czasu rzędu $O(o^2)$.

Model kombinatoryczny

W przypadku wielu zastosowań lepszą od modelu dysjunktywnego techniką modelowania problemu gniazdowego jest kombinatoryczna reprezentacja rozwiązań. Jest ona pozbawiona redundantości, charakterystycznej dla grafu dysjunktywnego. Zbiór operacji $\mathcal{O}$ może być zdekomponowany na podzbiory operacji wykonywanych na jednej, ustalonej maszynie $k \in M$, $M_k = \{i \in \mathcal{O} : v_i = k\}$ i niech $m_k = |M_k|$. Kolejność wykonywania operacji na maszynie k jest określona permutacją $\pi_k = (\pi_k(1), \pi_k(2), \ldots, \pi_k(m_k))$ elementów ze zbioru M_k, $k \in M$, gdzie $\pi_k(i)$ oznacza ten element z M_k, który jest na pozycji i w π_k. Niech $\Pi(M_k)$ będzie zbiorem wszystkich permutacji elementów M_k. Kolejność wykonywania operacji na wszystkich maszynach jest definiowana jako $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$, gdzie $\pi \in \Pi$, $\Pi = \Pi(M_1) \times \Pi(M_2) \times \ldots \times \Pi(M_m)$. Dla kolejności π, tworzymy graf skierowany (digraf)) $G(\pi) = (\mathcal{O}, U \cup E(\pi))$ ze zbiorem węzłów $\mathcal{O}$ i zbiorem łuków $U \cup E(\pi))$, gdzie U jest zbiorem łuków stałych reprezentujących kolejność wykonywania operacji w zadaniu, zaś zbiorem łuków reprezentującym kolejność wykonywania operacji na maszynach jest

$$E(\pi) = \bigcup_{k=1}^{m} \bigcup_{i=1}^{m_k - 1} \{(\pi_k(i), \pi_k(i+1))\} \tag{2.34}$$

Każdy węzeł $i \in \mathcal{O}$ ma wagę p_i, każdy łuk ma wagę zero. Kolejność wykonywania π jest dopuszczalna, jeśli tylko graf $G(\pi)$ nie zawiera cyklu. Na

Rys. 2.4 pokazano przykład grafu $G(\pi)$ dla takich samych danych, dla jakich skonstruowano graf dysjunktywny przedstawiony na Rys. 2.3 - widoczne jest, że graf $G(\pi)$ posiada bardziej przejrzystą strukturę, jest pozbawiony redundantości związanej z nadmiarowymi łukami w reprezentacji dysjunktywnej. Łatwo zauważyć, że graf $G(\pi)$ dla reprezentacji kombinatorycznej może być wzbogacony o dodatkowy węzeł początkowy (0) analogicznie, jak dla reprezentacji dysjunktywnej. Dla danej dopuszczalnej kolejności wykonywania π wyznaczenie wartości funkcji celu w oparciu o model kombinatoryczny wymaga czasu rzędu $O(o)$, zatem mniejszego niż w przypadku reprezentacji dysjunktywnej.

3

Metody optymalizacji dyskretnej

Niezależnie od przyjętego kryterium, większość omawianych problemów optymalizacyjnych jest formułowana jako zagadnienia, w których część bądź wszystkie zmienne – takie jak czasy wykonywania zadań, pożądane terminy zakończenia i inne – przyjmują wartości dyskretne, (zwykle całkowitoliczbowe lub binarne). Zadania takie należą do wyjątkowo trudnych z obliczeniowego punktu widzenia, z uwagi na mnogość występowania ekstremów lokalnych i często brak własności wypukłości, ciągłości i różniczkowalności funkcji kryterialnej. Istnienie wielu ekstremów lokalnych, kłopotliwe już w przypadku optymalizacji ciągłej, w połączeniu z olbrzymimi rozmiarami badanych przestrzeni dyskretnych powoduje, że wyznaczenie optymalnego rozwiązania badanego problemu jest dość trudne.

W celu uniknięcia niedogodności związanych z otrzymywaniem rozwiązania dokładnego (optymalnego) problemu optymalizacyjnego, próbuje się wyznaczyć pewne jego rozwiązanie przybliżone. Metody heurystyczne (przybliżone) charakteryzują się dużym stopniem przystosowania do problemu, który rozwiązują oraz do kryterium, które mają minimalizować. Często jednak metoda przybliżona doskonale sprawdzająca się w rozwiązywaniu jednego problemu, czy wręcz konkretnej grupy przykładów, nie sprawdza się w innym. Wynika to w dużej mierze ze specjalizacji algorytmów heurystycznych – parametry ich działania muszą być doskonale dostrojone. Pewnym wyjściem jest tu zastosowanie mechanizmu automatycznego strojenia (na przykład w metodzie symulowanego wyżarzania do dostrojenia temperatury początkowej). Innym rozwiązaniem jest zastosowanie równoległych algorytmów przybliżonych, które poprzez wykonywanie współbieżnie procesów o różnych wartościach parametrów strojących mogą

być z powodzeniem stosowane do szerszej grupy przykładów.

Rozwiązując problem optymalizacyjny należy znaleźć takie jego rozwiązanie x^*, aby zminimalizować wartość zadanego kryterium K:

$$K(x^*) = \min_{x \in \mathcal{X}} K(x) \tag{3.1}$$

gdzie x jest pewnym rozwiązaniem badanego problemu, $K(x)$ – wartością funkcji kryterialnej, a $\mathcal{X}$ zbiorem rozwiązań dopuszczalnych określonym przez zadane ograniczenia problemu. W szczególności rozwiązanie dopuszczalne $x \in \mathcal{X}$ może być reprezentowane za pomocą różnych konstrukcji matematycznych, np. wektora, macierzy, itp. W problemach szeregowania, będących zagadnieniami optymalizacji dyskretnej, dość często x ma postać kombinatoryczną: permutacji, wektora lub ciągu permutacji, podziału zbioru, wektora binarnego lub całkowitoliczbowego, itp. Funkcja kryterialna K reprezentuje pewną miarę jakości rozwiązania x dla danego problemu. Dla problemów szeregowania zadań $K(x)$ łączy elementarne funkcje kosztu zdefiniowane dla poszczególnych zadań, takie jak spóźnienia, czasy przepływu zadań przez system i inne zaprezentowane w poprzednim rozdziale.

3.1 Metody dokładne

Metoda dokładna wyznacza rozwiązanie globalnie optymalne, tzn. rozwiązanie $x^* \in X$ takie, że

$$K(x^*) = \min_{x \in X} K(x). \tag{3.2}$$

Spośród metod dokładnych służących do rozwiązywania problemów optymalizacyjnych wymienić można:

1. metody oparte o schemat podziału i ograniczeń (*Branch and Bound*, B&B),
2. metody przeglądu sterowanego (np. algorytm Balasa),
3. metody płaszczyzn odcinających (np. algorytmy Gomory'ego),
4. metody oparte o schemat programowania dynamicznego (PD),
5. metody subgradientowe,
6. efektywne algorytmy dedykowane do rozwiązywania specyficznego problemu.

Zastosowanie metod dokładnych jest w praktyce ograniczone do problemów o niedużych rozmiarach, z uwagi na ich czasochłonność. Stosuje się je jednak z powodzeniem na przykład do wyznaczania minimalnego czasu

cyklu w powtarzającym się procesie produkcyjnym. Dla niewielkiego zbioru zadań można wtedy – nawet kosztem dużego czasu obliczeń – wyznaczyć rozwiązanie, którego nawet niewielki zysk (w stosunku do rozwiązania przybliżonego) otrzymywany w jednym cyklu zwielokrotni się o liczbę wykonanych cykli. Innym, równie popularnym zastosowaniem metod dokładnych jest wyznaczanie rozwiązań referencyjnych, do których następnie porównuje się wyniki algorytmów przybliżonych badając ich jakość.

3.2 Metody przybliżone i błąd przybliżenia

Metoda przybliżona A wyznacza rozwiązanie x^A bliskie optymalnemu, to znaczy takie, że wartość funkcji kryterialnej $K(x^A)$ różni się niewiele od wartości optymalnej $K(x^*)$. Sposobów obliczania błędów przybliżenia jest kilka. Można wyznaczać błąd bezwzględny (różnica wartości funkcji kryterialnej) jak i względny (procentowy). Dokładniej błąd przybliżenia można sformalizować następująco: niech Z będzie zbiorem danych liczbowych dla konkretnego przykładu badanego problemu optymalizacyjnego. Oznaczmy przez $X(Z)$ zbiór wszystkich rozwiązań problemu, dla tego przykładu, zaś przez $K(x;\, Z)$ wartość kryterium K dla rozwiązania x przykładu Z. Rozwiązanie $x^* \in X(Z)$ takie, że $K(x^*;\, Z) = \min_{x \in X(Z)} K(x;\, Z)$ jest nazywane rozwiązaniem optymalnym dla przykładu Z. Niech $x^A \in X(Z)$ oznacza rozwiązanie przybliżone generowane przez algorytm A dla przykładu Z. Za błąd przybliżenia algorytmu A można przyjąć jedną za następujących wielkości

$$B^A(Z) = \left| K(x^A; Z) - K(x^*; Z) \right|, \tag{3.3}$$

$$S^A(Z) = K(x^A; Z)/K(x^*; Z), \tag{3.4}$$

$$T^A(Z) = \frac{K(x^A; Z) - K(x^*; Z)}{K(x^*; Z)}, \tag{3.5}$$

$$U^A(Z) = \frac{K(x^A; Z) - K(x^*; Z)}{K(x^A; Z)}. \tag{3.6}$$

Oczywiście wartości błędów $T^A(Z)$ i $U^A(Z)$ nie mogą być wyznaczane, jeśli wartość mianownika wyniesie 0. Niektóre z metod przybliżonych służących do rozwiązywania problemów optymalizacyjnych wymienione zostały w Tab. 3.1.

Większość metod przybliżonych działa w oparciu o pojęcie *przeszukiwania* przestrzeni rozwiązań dopuszczalnych wykorzystując przy tym pewne

Nazwa polska	nazwa angielska	skrót
poszukiwanie zstępujące	*descending search*	DS
poszukiwanie losowe	*random search*	RS
poszukiwanie snopowe	*beam search*	BS
poszukiwanie progowe	*threshold search*	TA
poszukiwanie ewolucyjne	*evolutionary search*,	
	genetic search,	GS
	genetic algorithm	GA
ewolucja różnicowa	*differentail evolution*	
podejście immunologiczne	*artificial immune system*	
poszukiwanie biochemiczne	*DNA method*	
symulowane wyżarzanie	*simulated annealing*	SA
symulowane wstrząsanie	*simulated jumping*	SJ
poszukiwanie z zakazami	*tabu search*	TS
poszukiwanie z pamięcią		
adaptacyjną	*adaptive memory search*	AMS
metoda geometryczna	*geometric search*	
poszukiwanie mrówkowe	*ant search*	AS
poszukiwanie rozproszone	*scatter search*	SS
spełnianie ograniczeń	*constraint satisfaction*	CS
poszukiwanie ścieżkowe	*path search*	PS
algorytmy naśladujące	*memetic algorithms*	MA
algorytmy kulturowe	*cultural algorithms*	CA
sieci neuronowe	*neural networks*	NN
metody hybrydowe		

Tablica 3.1: Przybliżone metody rozwiązywania problemów optymalizacji dyskretnej.

dodatkowe informacje o rozwiązywanym problemie i stanie poszukiwań – czyli heurezy. Dokładniej pojęcie przeszukiwania heurystycznego zostanie omówione w rozdziale następnym.

3.3 Najnowsze tendencje w optymalizacji

Przeszukiwanie jest jednym z bardziej uniwersalnych sposobów rozwiązywania zadań optymalizacji dyskretnej. Rozwój metod przeszukiwania jest ściśle związany z postępem w dziedzinie sztucznej inteligencji – najwcześniejszymi algorytmami w tej dziedzinie były algorytmy przeszukiwania. Z kolei żywy rozwój metod sztucznej inteligencji wpłynął także na zainteresowanie się coraz to nowymi metodami przeszukiwania, od strategii „ślepych", nie wykorzystujących informacji o dziedzinie rozwiązywanego problemu (takie jak strategie w głąb i wszerz), po najbardziej dopasowane do problemu, wykorzystujące jego specyficzne własności.

Idea poszukiwań z wykorzystaniem strategii heurystycznych wywodzi się z obserwacji, iż dla większości problemów ich przestrzeń stanów (rozwiązań) zawiera dodatkowe informacje. Koszt wyznaczenia tych informacji jest niewielki, a pozwalają one dodatkowo klasyfikować stany i łatwiej wybierać najlepsze kierunki przeszukiwania. W niniejszej pracy uwagę skupiono na pewnej strategii heurystycznego przeszukiwania, w której informacja heurystyczna wyrażona jest przez funkcje numeryczne.

Pojęcie „przeszukiwania heurystycznego" pojawiło się we wczesnych latach sześćdziesiątych [19]. Heurystyka była narzędziem eliminowania (podczas przeszukiwania) niektórych rozwiązań z dużego zbioru wszystkich możliwych rozwiązań. Następnie zasady przeszukiwania heurystycznego były w różny sposób formalizowane. W uproszczeniu przyjmuje się, że rozwiązywany problem można opisać za pomocą obiektów (stanów) i operatorów (przejść pomiędzy stanami). Operatory zastosowane do obiektów generują – jeśli mogą – nowe obiekty. Stany początkowe i operatory definiują graf obiektów. Heurystyczne przeszukiwanie jest procesem poszukiwania żądanego stanu, drogi do żądanego stanu lub podgrafu spełniającego zadane warunki.

Przy wyborze najbardziej obiecujących operatorów przeszukiwanie heurystyczne posługuje się różnymi środkami (analogie, uproszczenia), których celem jest ograniczenie zbioru przeszukiwanych obiektów – w zadaniach praktycznych bowiem zazwyczaj występują bardzo duże przestrzenie stanów. Z drugiej strony heurystyka nie gwarantuje znalezienia rozwiązania optymalnego, chociaż „dobra" metoda heurystyczna generuje w krótkim czasie rozwiązania niewiele różniące się od optymalnego, a więc w pełni

zadawalające z praktycznego punktu widzenia.

Głównym zadaniem heurystyki jest usprawnienie algorytmu rozwiązywania danego problemu. Najważniejsze znaczenie ma eliminowanie z dalszych rozważań części niesprawdzonych jeszcze dokładnie obiektów, które nie rokują wyznaczenia dobrego rozwiązania. Od jakości stosowanej heurystyki zależy złożoność algorytmu rozwiązania problemu. Kierowanie procesem decyzyjnym bezpośrednio (przez wskazywanie najlepszych kierunków poszukiwania rozwiązania) lub pośrednio (przez eliminowanie najmniej obiecujących kierunków) uważane jest za najważniejszą funkcję heurystyki.

Heurystyki nie gwarantują znalezienia rozwiązania optymalnego badanego problemu. Jednak dla wielu autorów właśnie ta cecha ma zalety. Według ich opinii postęp w dziedzinie metod sztucznej inteligencji, a w dziedzinie przeszukiwania w szczególności, jest opóźniany nadmiernym wykorzystywaniem zupełnych reguł decyzyjnych, tzn. reguł, które nie eliminują z rozważań żadnych kierunków prowadzących do rozwiązania. Przekonanie, że proces myślowy matematyków czy ekspertów w ogóle ma cechę zupełności często nie ma uzasadnienia.

Jak już wspomniano, dołączenie heurystyki do algorytmu rozwiązującego dany problem ma na celu poprawienie jego efektywności. Lepsza jakość rozwiązania jest często wynikiem zastosowania praktycznych reguł – często intuicyjnych i empirycznych – ściśle związanych z dziedziną, z której pochodzi problem.

3.3.1 Algorytmy poszukiwań lokalnych

W konstrukcji wielu algorytmów przybliżonych stosowana jest metoda iteracyjnego polepszania bieżącego rozwiązania poprzez lokalne przeszukiwanie. Rozpoczyna się ona od pewnego rozwiązania początkowego (startowego). Następnie generuje się jego otoczenie (sąsiedztwo) oraz wyznacza najlepsze rozwiązanie z tego otoczenia, które przyjmuje się za rozwiązanie startowe w kolejnej iteracji.

Opis algorytmów poszukiwań lokalnych ograniczony zostanie do pewnej klasy problemów szeregowania zadań, w której zbiór rozwiązań może być reprezentowany zbiorem permutacji. Podana technika może być także stosowana do analizowana innych problemów optymalizacji kombinatorycznej posiadających tak samo reprezentowaną przestrzeń rozwiązań, a mianowicie do problemu komiwojażera (TSP), jedno-maszynowych i m-maszynowych przepływowych problemów szeregowania z różnymi ograniczeniami (*flow-shop*), niektórych systemów przepływowo-równoległych, itp.

Przez Π będzie oznaczony zbiór wszystkich permutacji zbioru zadań $\mathcal{J}$. Niech π będzie dowolnym rozwiązaniem $\pi \in \Pi$, a N_π jego otoczeniem, czyli

zbiorem wszystkich permutacji możliwych do wygenerowania z permutacji π za pomocą pojedynczego ruchu. Sposób określania otoczenia jest jednym z podstawowych elementów algorytmu. Najczęściej stosowane są trzy rodzaje otoczeń: wymiana par przyległych (API), wymiana par dowolnych (NPI), technika przenoszenia i wstawiania (INS).

Wymiana par przyległych (API). Otoczenie N_π permutacji π zawiera wszystkie permutacje wygenerowane przez zamianę miejscami w π dwóch *przyległych* elementów. Oznaczając przez σ permutację otrzymywaną z π poprzez zamianę par przyległych na pozycjach a i a+1 ($1 \leqslant a \leqslant n-1$) dostaniemy z permutacji

$$\pi = (\pi(1), \pi(2), ..., \pi(a), \pi(a+1), ..., \pi(n))$$

permutację

$$\sigma = (\pi(1), \pi(2), ..., \pi(a+1), \pi(a), ..., \pi(n)).$$

Otoczenie API zawiera $n-1$ permutacji.

Wymiana dowolnych par (NPI). Otoczenie N_π permutacji π zawiera wszystkie permutacje wygenerowane przez zamianę miejscami w π dwóch *dowolnych* elementów. Stosując zamianę na pozycjach a i b ($1 \leqslant a, b \leqslant n$) z permutacji

$$\pi = (\pi(1), \pi(2), ..., \pi(a), ..., \pi(b), ..., \pi(n))$$

dostaniemy permutację

$$\sigma = (\pi(1), \pi(2), ..., \pi(b), ..., \pi(a), ..., \pi(n)).$$

Otoczenie NPI zawiera $\frac{n(n-1)}{2}$ permutacji.

Wstawienie (INS). Dowolna permutacja z otoczenia N_π jest wygenerowana z π poprzez wyjęcie pewnego elementu permutacji π z pozycji a, oraz wstawienie tak by zajmował on pozycję b w nowej permutacji (część elementów permutacji należy wówczas przesunąć). Posługując się tą techniką z permutacji

$$\pi = (\pi(1), \pi(2), ..., \pi(a-1), \pi(a), \pi(a+1), ..., \pi(b-1), \pi(b), \pi(b+1), ..., \pi(n))$$

dostaniemy permutację

$$\sigma = (\pi(1), \pi(2), ..., \pi(a-1), \pi(a+1), ..., \pi(b-1), \pi(b), \pi(a), \pi(b+1), ..., \pi(n)).$$

Otoczenie INS zawiera $(n-1)^2$ permutacji.

Z bardziej zaawansowanych technik generacji sąsiedztwa stosowane są: odwrócenie podciągu (INV), permutacje przyległych k elementów (k-AI), permutacje k dowolnych elementów (k-NI). Liczebności tych otoczeń wynoszą odpowiednio $O(n)$ dla otoczenia INV, O($nk!$) dla otoczenia k-AI oraz $O(n^k)$ dla otoczenia k-NI.

Jako osobną klasę należy wyróżnić tzw. otoczenie z niezależnymi wymianami (DPI, *dynasearch swap*) definiowane jako wszystkie permutacje, które można otrzymać z π stosując dowolny ciąg wymian parami niezależnych. Pary wymian (a,b) i (c,d) są niezależne, jeśli $\max\{a,b\} \leqslant \min\{c,d\}$ lub $\min\{a,b\} \geqslant \max\{c,d\}$. Sąsiedztwo to zawiera $2^{n-1}-1$ rozwiązań. W pracy [46] zostało pokazane, że dla problemu jednomaszynowego $1\|\sum f_i$ wykorzystanie schematu programowania dynamicznego pozwala wyznaczyć najlepszy element tak zdefiniowanego otoczenia w czasie $O(n^3)$, a więc przeglądnąć wykładniczą liczbę elementów otoczenia w czasie wielomianowym.

3.3.2 Przeszukiwanie tabu

Metoda tabu została pierwotnie zaproponowana przez Glovera [76–78] i rozwijana dalej przez innych autorów [140, 141]. Jest ona modyfikacją metody lokalnych poszukiwań. Dopuszcza się możliwość zwiększania wartości funkcji celu (przy wyznaczaniu nowego rozwiązania generującego otoczenie), aby w ten sposób zwiększyć szansę na osiągnięcie minimum globalnego. Takie ruchy „w górę" należy jednak kontrolować, ponieważ w przeciwnym razie po osiągnięciu minimum lokalnego nastąpiłby szybki do niego powrót. W celu uniknięcia „zapętlenia", skierowania poszukiwań w obiecujące regiony przestrzeni oraz umożliwienie wyjścia z ekstremum lokalnego wprowadza się tzw. mechanizm zabronień. Wykonując ruchy zapamiętuje się rozwiązania, atrybuty rozwiązań lub ruchów na tzw. liście tabu. Generując otoczenie nie rozpatrujemy rozwiązań znajdujących się na tej liście chyba, że spełniają kryterium aspiracji, to jest warunki, przy których ograniczenia tabu można pominąć. Podstawowymi elementami metody tabu search są:

ruch: funkcja, która transformuje jedno rozwiązanie w drugie,

otoczenie: zbiór rozwiązań możliwych do otrzymania z jednego ustalonego rozwiązania za pomocą pewnej klasy ruchów,

lista tabu: lista zawierająca atrybuty ruchów lub rozwiązań dla pewnej liczby ostatnio rozpatrywanych rozwiązań, skojarzona z operatorami

jej modyfikacji (dodawanie i usuwanie atrybutów) oraz operatorami badania zabronień,

kryterium aspiracji: warunki, przy których można pominąć ograniczenia wprowadzone przez listę tabu. Odpowiednie stosowanie takiego kryterium jest szczególnie ważne, gdy otoczenie zawiera dużo elementów,

warunek zakończenia: algorytm zazwyczaj kończy działanie, jeżeli: (a) wykonał z góry określoną liczbę iteracji, (b) w kolejnych iteracjach nie uzyskano poprawy wartości funkcji kryterialnej, (c) aktualne otoczenie jest zbiorem pustym.

Niech $x \in X$ będzie dowolnym rozwiązaniem dopuszczalnym, *LTS* listą tabu, a x^* najlepszym do tej pory znalezionym rozwiązaniem (na początek przyjmujemy za x^* rozwiązanie początkowe x).

Algorytm tabu search

repeat
Wyznaczyć otoczenie N_x rozwiązania x;
Usunąć z N_x rozwiązania zakazane przez listę *LTS*,
uwzględniając kryterium aspiracji;
Znaleźć rozwiązanie $y \in N_x$ takie, że:
$F(y) = \min\{F(z)\colon z \in N_x\}$;
if $F(y) < F(x^*)$ **then** $x^* \leftarrow y$;
Umieścić atrybuty y na liście *LTS*;
$x \leftarrow y$;
until Warunek_Końca

Złożoność obliczeniowa algorytmu opartego na metodzie tabu search w dużej mierze zależy od wyboru rodzaju poszczególnych jego elementów, tj. metody wyznaczania sąsiedztwa, rodzaju elementów przechowywanych na liście tabu jak również rodzaju oraz długości tej listy, sposobu wyliczania wartości funkcji celu oraz warunku zakończenia.

3.3.3 Symulowane wyżarzanie

Metoda symulowanego wyżarzania, ze względu na prostotę implementacji oraz uniwersalność, jest z powodzeniem stosowana do rozwiązywania wielu problemów optymalizacyjnych. Została ona zaprezentowana w pracy [99], Kirkpatrick i inni, a jej podstawowe idee pochodzą z termodynamiki. Posiada ona pewne analogie z procesem wyżarzania (chłodzenia) ciała stałego,

stąd w jej opisie używa się pojęć z tej właśnie dziedziny. Cechą charakterystyczną metody jest stopniowe obniżanie parametru kontrolnego zwanego temperaturą. Podobnie, jak w metodzie przeszukiwania tabu , z otoczenia N_x rozwiązania x wyznacza się dowolne rozwiązanie y (jeżeli $F(y) < F(x^*)$, to y jest przyjmowane za najlepsze do tej pory wyznaczone rozwiązanie x^*. Rozwiązanie y, z pewnym prawdopodobieństwem (wynosi ono 1, gdy $F(y) \leqslant F(x)$), przyjmuje się za rozwiązanie startowe x w następnej iteracji. Konstrukcja algorytmu opartego na metodzie symulowanego wyżarzania wymaga określenia następujących elementów:

funkcji akceptacji: funkcja prawdopodobieństwa z jakim są przyjmowane (za rozwiązania startowe w następnej iteracji) elementy otoczenia,

schemat chłodzenia: funkcja określająca zmianę funkcji akceptacji w zależności od parametru kontrolnego zwanego temperaturą (zwykle zależnego od numeru iteracji algorytmu).

Niech $x \in X$ będzie dowolnym rozwiązaniem dopuszczalnym, a x^* najlepszym do tej pory znalezionym rozwiązaniem (na początek przyjmujemy za x^* rozwiązanie początkowe x). Przez $\varphi(t)$ oznaczmy schemat chłodzenia (t – temperatura) oraz przez $\Psi(\pi,\varphi(t))$ funkcję akceptacji.

Algorytm symulowanego wyżarzania

repeat
  repeat
    Wylosuj rozwiązanie $y \in N_x$;
    if $F(y) < F(x^*)$ **then** $x^* \leftarrow y$;
    if $F(y) \leqslant F(x)$ **then** $x \leftarrow y$;
    else
      if $\Psi(\delta,\varphi(t))$ **then** $x \leftarrow y$;
  until Pora zmieniać parametr kontrolny;
  Zmień parametr kontrolny;
until *Warunek_Końca*

Parametr kontrolny (temperatura) zmienia się zazwyczaj co ustaloną liczbę iteracji algorytmu według jednego ze schematów chłodzenia. W literaturze wymienianych jest kilka schematów, zapewniających zbieżność algorytmu z prawdopodobieństwem 1 do rozwiązania optymalnego, a mianowicie: (a) geometryczny $t_{i+1} = \lambda_i t_i$ (λ_i jest często *const*), (b) logarytmiczny $t_{i+1} = t_i/(1+\lambda_i t_i)$, (c) oparty na twierdzeniu Hajek'a [88] $t_i = \frac{A}{\ln(i+2)}$,

gdzie A jest parametrem liczbowym określającym liczbę kroków niezbędną do opuszczenia dowolnego ekstremum lokalnego.

Przy implementacji algorytmu należy tak ustalić jego parametry, aby po wykonaniu pewnej liczby iteracji prawdopodobieństwo akceptacji rozwiązań znacznie różniących się od najlepszego do tej pory wyznaczonego malało, przez co szybko dochodzimy do pewnego minimum lokalnego. Po jego osiągnięciu przywraca się początkowe wartości parametrom (zwiększając prawdopodobieństwo akceptacji „gorszych" rozwiązań) umożliwiając w ten sposób znaczne oddalenie się od bieżącego minimum.

3.3.4 Algorytmy ewolucyjne

Określenia „algorytmy ewolucyjne" oraz „algorytmy genetyczne" obejmują grupę metod obliczeniowych, których wspólną cechą jest korzystanie, przy rozwiązywaniu danego problemu, z mechanizmu opartego na zjawisku naturalnej ewolucji gatunków. Są one bezpośrednią adaptacją tego zjawiska, stąd w ich opisie używa się pojęć z genetyki. Ich twórcą jest Holland [90,91].

W metodach tych, na wyróżnionych podzbiorach zbioru rozwiązań dopuszczalnych (populacji osobników), wykonywane są cyklicznie trzy podstawowe operacje:

selekcja: polegająca na wyborze, z bieżącej populacji, pewnego podzbioru osobników najlepiej przystosowanych (najbardziej obiecujących) zwanych rodzicami, na bazie których zostanie utworzone następne pokolenie,

krzyżowanie: polegająca na wygenerowaniu z odpowiednio dobranych par rodziców nowych osobników (potomstwa),

mutacja: wykonywana z niewielkim prawdopodobieństwem na zbiorze potomków, w celu zapobiegnięcia stagnacji w procesie poszukiwań.

Selekcja i krzyżowanie są najmocniejszymi operatorami na zbiorze osobników. Dzięki nim cały proces ten ma charakter ewolucyjny i prowadzi do wygenerowania podzbioru zawierającego „najbardziej obiecujące" rozwiązania.

Działanie algorytmu genetycznego rozpoczyna się od utworzenia populacji początkowej $P_0 \in X$, której liczebność jest zazwyczaj stała przez cały czas działania algorytmu. Niech k będzie kolejnym numerem iteracji algorytmu. Nowa k+1 generacja (tj. zbiór P_{k+1}) jest tworzona w następujący sposób. Z bieżącej populacji P_k wybierana jest pewna ilość najlepszych osobników (rodziców, podzbiór P'_k) - operacja selekcji. Z nich, poprzez mechanizm krzyżowania, generuje się nowych osobników (potomstwo,

zbiór P''_k). Na części z nich jest dokonywana operacja mutacji, której celem jest większe zróżnicowanie populacji. Tak wygenerowane potomstwo zastąpi najgorszych osobników w bieżącej populacji (wymiana), tworząc w ten sposób nową generacje. Algorytm zazwyczaj kończy działanie po wygenerowaniu z góry ustalonej liczby osobników.

Klasyczny algorytm genetyczny

$k \leftarrow 0$;
$P_k \leftarrow$ Losowa_Populacja;
repeat
  {*Wybór rodziców*}
  Selekcja(P_k, P'_k);
  {*Generowanie potomstwa*}
  Krzyżowanie(P'_k, P''_k);
  Mutacja(P''_k);
  {*Nowa populacja*}
  $P_{k+1} \leftarrow P''_k$;
  $k \leftarrow k + 1$;
until *Warunek_Końca*

Algorytm kończy działanie (*Warunek_Końca*) po wygenerowaniu z góry określonej liczby iteracji. Złożoność obliczeniową algorytmu zależy przede wszystkim od maksymalnej liczby wygenerowanych osobników oraz liczebności populacji.

4

Obliczenia równoległe

Proces projektowania i implementacji algorytmu równoległego bardzo ściśle powiązany jest z architekturą komputera, na którym będzie uruchamiany. W przypadku algorytmów sekwencyjnych przeznaczonych dla komputerów posiadających jeden procesor i nierozproszoną pamięć operacyjną już wiele lat temu powstały jednoznaczne modele, dla których konstruuje się takie algorytmy. Językiem opisu algorytmu sekwencyjnego może być pseudo-Pascal lub pseudo-C, lub jeszcze inny język strukturalny wysokiego poziomu – proces tłumaczenia tak zapisanego algorytmu na język wewnętrzny maszyny jest jednoznaczny i nie budzi wątpliwości. Inaczej jest w przypadku algorytmów równoległych przeznaczonych dla komputerów posiadających wiele procesorów. Procesory te mogą posiadać wspólną pamięć operacyjną, ale mogą też posiadać pamięci lokalne – każdy procesor swoją. Każda taka pamięć lokalna może być pamięcią typu buforowego (typu *cache*), ale może też być częścią większej pamięci rozproszonej. Pojawia się problem zgodności wspólnych danych wykorzystywanych przez różne procesory.

Mnogość rozwiązań technicznych umożliwiających urównoleglenie algorytmów zmusza do stworzenia pewnej ogólnej klasyfikacji. W tej części pracy przedstawiono najbardziej rozpowszechnione obecnie typy architektur równoległych. Opisano także model komputera równoległego PRAM na którego przykładzie podano sposoby odwoływania się przez procesory do wspólnej pamięci dzielonej maszyny.

4.1 Równoległe algorytmy poszukiwań

Równoległe implementacje algorytmów poszukiwań wydają się być naturalną alternatywą służącą przyspieszeniu procesu wyznaczania dobrych rozwiązań przybliżonych w problemach optymalizacji kombinatorycznej. Nie

tylko pozwalają na rozwiązywanie problemów o większych rozmiarach lub poprawianiu jakości rozwiązań w stosunku do ich sekwencyjnych odpowiedników, ale także można je traktować jako nowe, silniejsze algorytmy, niemożliwe do sekwencyjnego zasymulowania. Równoległość może być bowiem drogą służącą nie tylko zredukowaniu czasu obliczeń algorytmu (na przykład lokalnego przeszukiwania), ale także poprawieniu jego efektywności. Cel taki może zostać osiągnięty, na przykład, przez użycie różnych strategii poszukiwania oraz różnych ustawień parametrów na każdym z procesorów, pozwalając otrzymać wysokiej jakości rozwiązanie dla różnych klas trudności problemów, bez czasochłonnego procesu dostrajania algorytmu. Z tego punktu widzenia, algorytmy równoległe są narzędziem daleko bardziej uniwersalnym i stabilnym (mając na uwadze stabilność jakości otrzymywanych rozwiązań, ich dyspersję) w stosunku do algorytmów sekwencyjnych – zmiana instancji problemu, czy całej ich grupy, nie spowoduje drastycznego pogorszenia efektywności tak skonstruowanego algorytmu równoległego, jak to się dzieje w przypadku szeregu algorytmów sekwencyjnych przystosowanych do rozwiązania pewnych konkretnych, często mało typowych, przykładów testowych.

4.2 Teoretyczne modele architektur równoległych

Architektury komputerów współbieżnych możemy podzielić na kilka podgrup, biorąc za podstawę podziału liczbę wykonywanych równolegle strumieni rozkazów oraz liczbę strumieni danych, przekształcanych w wyniki.

Typowy jednoprocesorowy komputer sekwencyjny posiada jeden strumień rozkazów, wykonywanych przez procesor. Posiada także pamięć operacyjną, będącą pojedynczym strumieniem danych.

W komputerze równoległym wiele jednostek przetwarzających wykonuje pewne operacje równolegle. Mogą to być takie same rozkazy, ale wykonywane dla różnych danych (model SIMD) – mamy wtedy do czynienia z pojedynczym strumieniem rozkazów i wielokrotnym strumieniem danych. Inną możliwością jest wykonywanie przez wiele jednostek różnych instrukcji na różnych danych (model MIMD). Praca jednostek przetwarzających w modelu SIMD ma charakter synchroniczny – każda z nich wykonuje w danym momencie ten sam rozkaz. Model MIMD nie nakłada na pracę systemu równoległego żadnych ograniczeń – jest to najogólniejszy, asynchroniczny model architektury równoległej.

Biorąc za podstawę klasyfikacji liczbę strumieni danych i liczbę strumieni rozkazów można wyróżnić 4 typy organizacji architektur równoległych:

Rysunek 4.1: Architektura równoległa SIMD.

Rysunek 4.2: Architektura równoległa MIMD.

- SISD (*Single Instruction stream, Single Data stream*; pojedynczy strumień rozkazów, pojedynczy strumień danych) – to architektura komputera sekwencyjnego; pojedyncza jednostka arytmetyczno-logiczna wykonuje pojedynczy ciąg rozkazów.
- SIMD(*Single Instruction stream, Multiple Data stream*; pojedynczy strumień rozkazów, wielokrotny strumień danych) – architektura ta cechuje się zwielokrotnieniem jednostek przetwarzających realizujących ten sam strumień rozkazów (zwykle dekodowanych przez pojedynczą, wspólną jednostkę sterującą).
- MIMD(*Multiple Instruction stream, Multiple Data stream*; wielokrotny strumień rozkazów, wielokrotny strumień danych)
- MISD(*Multiple Instruction stream, Single Data stream*; pojedynczy strumień rozkazów, wielokrotny strumień danych) – architektura ta znalazła się w klasyfikacji jedynie dla kompletności; brak jest na razie jej praktycznych zastosowań – procesory musiałyby wykonywać różne rozkazy na tych samych danych.

Powyższy podział był jedną z pierwszych klasyfikacji systemów równoległych i pochodzi od M. Flynna [68].

4.3 Ziarnistość

Komputer równoległy może być zbudowany z niewielkiej liczby potężnych procesorów (silnych w sensie mocy obliczeniowej, o skomplikowanej budowie, dużym zbiorze wykonywanych instrukcji maszynowych), lub dużej liczby względnie słabych procesorów (o nieskomplikowanej budowie, niewielkim zbiorze wykonywanych instrukcji). Pierwszy z wymienionych typów zwany jest systemem gruboziarnistym (*coarse-grain*), drugi – drobnoziarnistym (*fine-grain*). Komputery o strukturze gruboziarnistej, jak na przykład Cray serii Y-MP posiadają niewielką liczbę procesorów (od 8 do 16), każdy o mocy obliczeniowej kilku GFlops [1]. Odmiennie, komputery drobnoziarniste,

[1] 1 GFlops określa moc obliczeniową umożliwiającą wykonywanie 10^9 operacji zmiennoprzecinkowych w ciągu sekundy.

takie jak na przykład CM-2, MasPar MP-1 czy MasPar MP-2, oferują dużą liczbę relatywnie wolnych i prostych procesorów (na przykład CM-2 składa się z maksymalnie 65,536 jednobitowych procesorów, a MasPar MP-1 – z maksymalnie 16,384 procesorów czterobitowych). Pomiędzy wymienionymi systemami dostępne jest całe spektrum komputerów o strukturze pośredniej, zwanych średnioziarnistymi (*medium-grain*), takich jak na przykład CM-5, Paragon XP/S czy nCUBE 2, posiadających od kilkudziesięciu do kilkuset procesorów, każdy o mocy obliczeniowej przeciętnej stacji roboczej.

Ziarnistość (*granularity*) komputera równoległego może być zdefiniowana jako stosunek czasu potrzebnego do wykonania podstawowej operacji komunikacji do czasu wykonania podstawowej operacji obliczeniowej. Komputery równoległe, dla których współczynnik ten jest niewielki, są najlepiej przystosowane do wykonywania algorytmów wymagających częstej komunikacji. Z kolei komputery równoległe o dużej wartości współczynnika ziarnistości są raczej przeznaczone do wykonywania algorytmów nie wymagających zbyt częstej komunikacji.

Architektura maszyny równoległej ma więc ścisły związek z *ziarnistością obliczeń* wykonywanych na tej maszynie, tzn. współczynnikiem wyrażającym się ilorazem czasu obliczeń uruchamianych aplikacji oraz czasu komunikacji pomiędzy procesami. Ziarnistość obliczeń może być także rozumiana jako ilość obliczeń (w sensie czasu lub ilości iteracji) wykonanych pomiędzy kolejnymi momentami komunikacji. Jeżeli architektura systemu równoległego pozwala na szybką i częstą komunikację (na przykład korzysta z bardzo szybkich sieci lub pamięci współdzielonej) to w takim systemie uruchamiane mogą być aplikacje drobnoziarniste . W przeciwnym przypadku zastosowane muszą być aplikacje średnio- lub gruboziarniste, wymagające rzadszej komunikacji.

4.4 Model komputera równoległego PRAM

Poniżej opisany został teoretyczny model komputera równoległego opartego na pamięci dzielonej, do której dostęp posiada każdy z procesorów. Model ten określa się jako PRAM (*Parallel Random Access Machine*) i jest on uogólnieniem sekwencyjnego modelu RAM. Składa się on ze zbioru (nieskończonego) rejestrów globalnych, z których każdy może przechowywać dowolną liczbę całkowitą (podobnie jak w modelu RAM) oraz ze zbioru (także nieskończonego) identycznie zaprogramowanych procesorów (czyli maszyn RAM), z których każdy ma swój własny zestaw rejestrów – lokalnych dla każdego procesora. Obliczenia są rozpoczynane przez pierwszy procesor po załadowaniu łańcucha wejściowego do początkowych rejestrów

globalnych. W każdym kroku procesory mogą wykonywać jedną ze standardowych operacji lub też uaktywnić nowy, następny procesor, który będzie wykonywał swoje instrukcje równolegle i synchronicznie z wszystkimi pozostałymi uaktywnionymi aktywnymi w danym momencie procesorami.

Podstawowym założeniem – przyjmowanym przy badaniu wydajności algorytmów dla modelu PRAM – jest, że czas działania może być mierzony liczbą równoległych dostępów do pamięci, wykonywanych podczas działania algorytmu, przy czym czas dostępu do pamięci pojedynczego procesora jest jednostkowy. To założenie jest prostym uogólnieniem założenia dla sekwencyjnej maszyny o dostępie swobodnym (maszyny RAM), w której czas działania mierzony liczbą dostępów do pamięci jest asymptotycznie taki sam jak czas działania mierzony jakąkolwiek inną miarą.

Każdy z procesorów w modelu PRAM może próbować odczytać bądź zapisać komórkę pamięci równolegle z innym procesorem. We wczesnym okresie istnienia tego modelu zabraniano równoczesnego zapisu do tej samej komórki pamięci przez różne procesory, widząc w takiej sytuacji potencjalne zagrożenie dla stabilności systemu [18], później jednak rozwiązano ten problem poprzez rozszerzenie modelu i podział modelu PRAM na podklasy zależnie od tego, jak obsługiwane są takie równoczesne odwołania do tej samej komórki pamięci [56, 106].

1. Exclusive-read, exclusive-write (EREW) PRAM – w tej klasie zabroniony jest równoległy dostęp do pamięci przez kilka procesorów; odczyt i zapis musi odbywać się dla danej komórki pamięci sekwencyjnie. Jest to najsłabszy (najmniej wymagający dla sprzętu) model pamięci komputera PRAM.

2. Concurrent-read, exclusive-write (CREW) PRAM – dopuszcza równoległy odczyt, zapis nadal musi odbywać się sekwencyjnie.

3. Exclusive-read, concurrent-write (ERCW) PRAM – dopuszcza równoległy zapis; odczyt jest sekwencyjny.

4. Concurrent-read, concurrent-write (CRCW) PRAM – umożliwia równoległy odczyt oraz zapis komórek pamięci. Jest to najsilniejszy z modeli pamięci komputera PRAM.

W przypadku modeli ERCW oraz CRCW problemem okazuje się umożliwienie w pełni równoległego zapisu do komórki pamięci. Stosowanych jest tu kilka metod, z których najpopularniejsze to:

- zaopatrzenie procesorów w priorytety; zapis do komórki pamięci odbywa się według priorytetów, tj. zapisuje tylko procesor o najwyższym priorytecie,

- zezwala się na zapis procesorom tylko wtedy, gdy wszystkie chcą zapisać tę samą wartość (rozwiązanie spotykane np. w środowisku Express – współbieżnych bibliotek języka C),
- wyróżnienie procesu – arbitra, który może wykonywać operacje zapisu, dla pozostałych procesów jest to operacja zabroniona,
- zapisywana jest suma wartości dostarczonych przez procesory.

Model EREW PRAM posiada najszersze ograniczenie spośród wymienionych. Jest on jednak najbliższy praktycznym realizacjom komputerów równoległych, w których trudno jest umożliwić prawdziwie równoległy zapis lub odczyt komórek pamięci. Z uwagi na to stworzono szereg opracowań symulacji modeli CREW, ERCW oraz CRCW przy pomocy maszyny EREW PRAM.

4.5 Rzeczywiste architektury równoległe

Biorąc pod uwagę teoretyczną klasyfikację architektur komputerów równoległych według ilości wykonywanych równolegle strumienie rozkazów oraz strumieni danych, w realnych systemach równoległych występują najczęściej modele MIMD oraz SIMD. Rodzina MIMD (*Multiple Instruction, Multiple Data*) rozciąga się od komputerów kilkuprocesorowych ze wspólną pamięcią (*shared memory*) po rozproszone systemy z setkami czy nawet tysiącami procesorów powiązanych sieciami o różnych topologiach, takich jak architektura pierścieniowa (*ring*) systemu KSR, dwuwymiarowa krata w Intel Paragon, trójwymiarowy torus w komputerze CRAY T3D/E, wielopoziomowe przełączniki (*multi-stage switches*) w systemie IBM-SP. Synchroniczne ze swej natury systemy SIMD (*Single Instruction, Multiple Data*) to duże komputery z 65536 4- lub 8-bitowymi procesorami takie jak MasPar MP-1 i MP-2 czy komputery Connection Machines CM-1 i CM-2. Ich procesory posiadają małą pamięć operacyjną (na przykład 16 KB w MasPar MP-1) i są połączone specyficzną siecią, jak na przykład dwuwymiarowa krata w MasPar i hiperkostka w CM-1 i CM-2.

Pod koniec lat dziewięćdziesiątych nastąpił powrót maszyn opartych na dzielonej pamięci (*Symmetric MP*, w skrócie SMP) posiadających od dwóch do kilkuset procesorów (na przykład Silicon Graphics Origin). Najważniejszą jednak innowacją tego okresu było łączenie maszyn SMP poprzez szybkie sieci (Myrinet, SCI, ATM, GigaEthernet) tworząc grupy - klastry (*cluster*) będące bardzo silnymi (w sensie mocy obliczeniowej) maszynami równoległymi. Oprócz skalowalności i dużej tolerancji na uszkodzenia poszczególnych elementów systemu, podejście takie oferuje doskonały

Lp	Nazwa serii	Dostawca
1	S3600	Hitachi
2	Quadrics Qx, QHx	Alenia
3	Gamma II Plus	The Cambridge Parallel Processing
4	MasPar MP-1, MP-2	Thinking Machine
5	CM-1, CM-2	Thinking Machine

Tablica 4.1: Komputery równoległe oparte na modelu SIMD.

Lp	Nazwa serii	Dostawca
1	Cray J90, T90,T3D/E, MTA, Y-MP	Cray Research Inc.
2	AlphaServer 8200, 8400 Cluster	The Digital Equipment Corp.
3	Exemplar SPP-2000	HP/Convex
4	SX-4, Cenju-3	Nec
5	AP3000, VX, VPP300, VPP700	Fujitsu
6	SR2201, S3800	Hitachi
7	IBM9076 SP2	IBM
8	Computing Surface 2	Meiko
9	E10000 Starfire	Sun
10	Paragon	Intel
11	Origin 200, 2000, 3800	Silicon Graphics

Tablica 4.2: Komputery równoległe oparte na modelu MIMD.

współczynnik kosztu do wydajności w porównaniu do dużych, jednorodnych komputerów równoległych. Klastry mogą być tworzone poprzez łączenie tanich komputerów klasy PC, na przykład w oparciu o system Linux, sieci takie jak Ethernet, SCI czy Myrinet i odpowiednie biblioteki takie jak PVM czy MPI.

4.6 Języki programowania równoległego

Wiele nowych narzędzi programistycznych zostało stworzonych by konstruować programy współbieżne. Każde z nich przystosowane jest do typu architektury systemu, w którym aplikacja ma być uruchamiana, lub do specyfiki rozwiązywanego problemu. Narzędziami tym są zarówno języki programowania równoległego, jak i biblioteki procedur rozszerzających możliwości klasycznych języków programowania. Istnieją zasadniczo trzy strategie tworzenia programów równoległych.

Pierwsza metoda to użycie *języka programowania równoległego*, który jest w istocie językiem programowania sekwencyjnego wzbogaconym o zbiór specjalnych odwołań systemowych. Odwołania takie umożliwiają synchronizacje procesów, ich tworzenie, przekazywanie komunikatów itp. Tego typu języki to na przykład Ada [59, 123, 173], BCPL, CSP, CODE, Concordia, CUBL, Concurent Pascal, Dapple, Forth, Fortran 90, Glish, HPF (*High Performance Fortran*) [89,101,102,130,131], Handel-C, KROC, Linda [38,39,123,166], mpC, Modula-2, NESL, Occam, Odyssey, OpenMP [60], Parallaxis-III, Parallel C++, pC++/Sage++, Voyager i wiele innych. Pisząc w języku programowania równoległego, programista nie ma bezpośredniego wpływu na sposób komunikacji, co z jednej strony może być uważane za wadę, ale z drugiej pozwala na programowanie wysokopoziomowe.

Drugie rozwiązanie służące programowaniu równoległemu to użycie *bibliotek komunikacyjnych* takich jak PVM (*Parallel Virtual Machine*) [86] lub MPI (*Message-Passing Interface*) [69, 84, 85, 134–136, 186]. Aby zwiększyć efektywność i przeciwdziałać równoczesnemu wykonywaniu na używanych procesorach wielu procesów (*contex swapping*) każdy procesor zwykle wykonuje jeden proces. Tego typu narzędzia są przeznaczone przede wszystkim do tworzenia aplikacji gruboziarnistych uruchamianych na klastrach lub stacjach roboczych połączonych siecią. PVM jest tu pionierem, pozwalając na działanie programów równoległych wykorzystujących mechanizm wymiany komunikatów (*message passing*) na maszynach o różnej wydajności i budowie (niejednorodnych, tzn. heterogenicznych). Programista tworzy program w języku C, C++ czy Fortranie dodając biblioteki procedur PVM. Wiele równoległych implementacji algorytmów metaheury-

stycznych wykorzystuje właśnie tą bibliotekę, np. [6, 8, 12, 15, 34, 49, 62, 66, 74, 143, 150–153, 178, 179, 181, 182]. MPI jest z kolei propozycją standaryzacji interfejsu wymiany komunikatów dla systemów z pamięcią rozproszoną oraz możliwością przenoszenia tak stworzonych aplikacji pomiędzy różnymi platformami sprzętowymi i systemowymi. MPI definiuje bowiem jedynie bibliotekę procedur przesyłania komunikatów, a nie całe otoczenie programistyczne. Istnieje wiele implementacji standardu MPI, wszystkie oparte na wirtualnej maszynie równoległej złożonej z połączonych komputerów heterogenicznych. Każdy z komputerów wykonuje (w tle) proces służący przesyłaniu komunikatów pomiędzy komputerami. Istnieje wiele implementacji metaheurystyk opartych na MPI, np. [7, 16, 64, 100, 125, 126, 128, 162, 164].

Rosnąca rola klastrów SMP zwiększa ostatnio rolę trzeciego modelu programowania równoległego – opartego na *lekkich procesach* takich jak wątki(*threads*) POSIX [32, 94] czy wątki Java [142]. Wątki nie są generalnie pojęciem wyłącznie związanym z programowaniem równoległym – historycznie są elementem systemów operacyjnych - lecz z uwagi na ich wagę w procesie implementacji programu równoległego w systemie operacyjnym zrozumienie koncepcji wątków jest zasadnicze dla programowania współbieżnego. Wątki są bowiem związane z procesem i do komunikacji wykorzystują pamięć globalną tego procesu. Zaletą takiego rozwiązania są krótkie czasy tworzenia i przełączania wątków (*contex swapping*). Ich użycie jest związane z aplikacjami średnio- i gruboziarnistymi. Ostatnio koncepcja wątków została rozszerzona na maszyny z pamięcią rozproszoną (*distributed memory*) z narzędziami programistycznymi takimi jak Clik [172], Charm++/Converse [30,96–98], Athapascan, PM2 [149], i wątki Java [118].

4.7 Typy współbieżności a modele programowania.

Istnieją zasadniczo dwa typy współbieżności wykorzystywane przy tworzeniu programów rónoległych: oparte na równoległości danych (*data parallelism*) i równoległości funkcjonalnej (*functional parallelism*) [69, 106, 127, 133]. Typ równoległych danych to model w którym ten sam program jest wykonywany synchronicznie przez procesory, ale na różnych danych – co jest łatwe w implementacji szczególnie w systemach z pamięcią współdzieloną. W przypadku równoległości funkcjonalnej program równoległy złożony jest ze współdziałających procesów opartych na różnych kodach programów, wykonywanych asynchronicznie.

Niezależnie od architektury systemu równoległego i otoczenia programistycznego (języki, biblioteki) wyróżnić można dwa modele programowania

równoległego: *scentralizowany* i *rozproszony*. Model *scentralizowany*, także zwany modelem „klient – serwer" lub „*master – slave"* charakteryzuje się wyróżnieniem wyspecjalizowanego procesora centralnego (*master,* serwer) z którym komunikują się pozostałe procesory podrzędne (*slave,* klient) aby otrzymać zadania do wykonania i odesłać otrzymane wyniki. Dane w tym modelu mogą być przechowywane przez procesor centralny, lub znajdować się w pamięci współdzielonej. Procesor centralny jest odpowiedzialny za zbalansowanie obciążenia procesorów podrzędnych (*load balancing*), może także wykonywać obliczenia. W modelu *rozproszonym* nie istnieją dane globalne, tak współdzielone jak i przechowywane centralnie - dane są lokalne dla każdego procesora. Informacje są rozsyłane poprzez system komunikatów wymienianych pomiędzy procesorami. Modelem bazującym na podejściu rozproszonym jest SPMD (*Single Program, Multiple Data*) związany z programowaniem maszyn MIMD, gdzie ten sam kod jest wykonywany na różnego typu procesorach i na różnych danych [129, 155]. Jest on szereoko stosowany z uwagi na łatwość implementacji - jeden kod jest uruchamiany na różnych procesorach, bez użycia centralnego mechanizmu synchronizacji (jak to mam miejsce w modelu SIMD). Z kolei model MPMD (*Multiple Program, Multiple Data*), także związany z modelem programowania rozproszonego, wymaga zaimplementowania oddzielnego kodu programu dla każdego z procesorów równoległych, co w praktyce jest dość kłopotliwe, a przez to rzadziej stosowane.

4.8 Komputery biologiczne

Obliczenia biochemiczne są techniką rozwiązywania trudnych problemów, takich jak problemy kombinatoryczne, za pomocą kodowania złożonych substancji biochemicznych, takich jak kwas dezyksorybonukleinowy (DNA), rybonukleinowy (RNA) czy cząsteczki protein (białek). Technika ta, jak i cała koncepcja równoległych obliczeń biologicznych, jest w swej istocie zupełnie odmienna od zaprezentowanych w niniejszym rozdziale architektur i modeli programowania równoległego, opartych na „klasycznych" procesorach elektronicznych, z uwagi jednak na potencjalną moc obliczeniową oraz liczbę możliwych do wykorzystania „procesorów" biologicznych otwiera olbrzymie możliwości zastosowania algorytmów równoległych do tej pory uważanych za bezużyteczne w praktyce, bazujących na bardzo dużej liczbie procesorów, na przykład rzędu 10^{20}.

Proces biologicznych obliczeń równoległych polega na losowym łączeniu bądź przekształcaniu się cząsteczek, w warunkach laboratoryjnych (na przykład w próbówce). W literaturze [4, 107] znaleźć można dwa podsta-

wowe podejścia dotyczące obliczeń biologicznych. Pierwsze, zaproponowane przez Adlemana [4] opiera się na idei kodowania rozwiązań (punktów przestrzeni rozwiąza) za pomocą molekuł – łańcuchów DNA. Ilość molekuł, jaką dysponuje badacz – rzędu 1 mola ($6,02 \cdot 10^{23}$) pozwala zakodować wszystkie rozwiązania problemu kombinatorycznego, o rozmiarach spotykanych w praktyce, za pomocą sekwencji DNA w formie łańcuchów z lepkimi końcami, pasującymi do siebie w ściśle określony sposób. Rozwiązaniem jest sekwencja połączonych ze sobą łańcuchów elementarnych o określonej długości. Poszukiwanie rozwiązania polega na równoległym losowym sklejaniu się łańcuchów DNA. Zakłada się, że z dużym prawdopodobieństwem każde rozwiązanie pamiętane jest za pomocą przynajmniej jednej molekuły - łańcucha DNA. Następnie, za pomocą złożonych procesów chemicznych, znajdowana jest cząsteczka kodująca rozwiązanie najlepsze. Zaproponowany przez Adlemana algorytm sprowadza się więc do wygenerowania i sprawdzenia wszystkich rozwiązań, i jest algorytmem probabilistycznym – to znaczy działa z dużym prawdopodobieństwem. W 1994 roku Adleman korzystając z metody opartej na rekombinacji DNA rozwiązał siedmio-wierzchołkowy problem komiwojażera (TSP). Przeprowadzenie koniecznych procesów laboratoryjnych związanych z kodowaniem danych i odczytywaniem wyników zajęło co prawda 7 dni, lecz szybki postęp w dziedzinie biochemii daje nadzieje na znaczne skrócenie tego czasu w przyszłości.

Odmienne podejście zaprezentowane jest w pracy Kurtz, Mahaney, Royer, Simon [107]. Autorzy proponują zastosowanie jednocząsteczkowych „procesorów" zwanych przez autorów CNA (*computational nucleic acid*, obliczeniowy kwas nukleinowy), a będących pojedynczymi molekułami o cechach RNA, DNA i protein. Cząsteczki CNA mogą kodować rozwiązania, mogą też się w określony sposób replikować wykonując „iteracje" algorytmu. Poprzez zredukowanie rozmiarów procesora do pojedynczej cząsteczki, możliwe jest wykorzystanie w obliczeniach równoległych olbrzymiej liczby (rzędu 1 mola to jest około $6,02 \cdot 10^{23}$) procesorów. Nawet, jeśli pojedynczy procesor nie jest zbyt szybki (w porównaniu do procesorów elektronicznych), równoległy komputer biologiczny i tak dysponuje potencjalną mocą obliczeniową wielokrotnie przewyższającą moc obliczeniową współczesnych superkomputerów. W modelu CNA ilość „instrukcji" wykonywanych przez pojedynczy procesor w ciągu sekundy można oszacować częstotliwością syntezy protein, wynoszącą 10 do 100 nukleotydów na sekundę. Moc równoległego komputera biologicznego dysponującego ilością procesorów rzędu 1 mola jest więc setki milionów razy większa od mocy najszybszych obecnie superkomputerów (posiadających moc obliczeniową rzędu 1 TFlops =

10^{12} operacji zmiennoprzecinkowych na sekundę). Co więcej, częstotliwość wykonywanych w ciągu sekundy instrukcji nie maleje wraz ze wzrostem długości cząsteczki – łańcucha kodującego, toteż opisany model obliczeń jest odporny na zmianę skali rozmiarów rozwiązywanych problemów.

5

Projektowanie algorytmów równoległych

Projektowanie algorytmów równoległych jest procesem dużo bardziej złożonym w porównaniu do projektowania odpowiedników sekwencyjnych, bowiem wymaga właściwego doboru zdecydowanie większej liczby współdziałających elementów składowych. Co więcej, obiektywna ocena charakterystyki numerycznej algorytmu równoległego jest również bardziej złożona. Istotnie, algorytm sekwencyjny jest oceniany zasadniczo przez dwa parametry mające skrajnie przeciwstawne tendencje: złożoność obliczeniową (czas pracy algorytmu) oraz dokładność. Algorytm równoległy, oceniany w trzech kategoriach: przyspieszenia, kosztu i efektywności, wymaga dodatkowo sprecyzowania strategii współbieżności (jedno-, wielowątkowa), mechanizmów komunikacji i kooperacji (dla wielu wątków poszukiwań). Znajomość pojęć i zachodzących zależności pozwala na projektowanie nowych algorytmów o zaskakująco dobrych własnościach numerycznych.

5.1 Pojęcia podstawowe

W procesie projektowania algorytmów równoległych występuje kilka parametrów oceny "dobroci" proponowanych konstrukcji, takich jak przyśpieszenie, efektywność i koszt. Parametry te oceniają zachowanie się algorytmu w relacji do jego odpowiednika sekwencyjnego. W wielu przypadkach pożądane jest zaprojektowanie algorytmu równoległego, którego koszt wykonania jest identyczny z kosztem wykonania algorytmu sekwencyjnego rozwiązującego ten sam problem. Algorytm taki jest określany mianem kosztowo optymalnego. Poniżej zostaną podane precyzyjne definicje kryteriów oceny algorytmów równoległych.

5.1.1 Przyspieszenie

Dany jest problem P, algorytm równoległy A_p oraz maszyna równoległa M z q identycznymi procesorami. Oznaczmy przez $T_{A_p,M}(p)$ czas obliczeń algorytmu A_p potrzebny do rozwiązania problemu P na maszynie M z użyciem $p \leqslant q$ procesorów. Oznaczmy przez T_{A_s} czas obliczeń najlepszego (najszybszego) znanego algorytmu sekwencyjnego A_s rozwiązującego ten sam problem P na maszynie sekwencyjnej z procesorem identycznym jak te na maszynie równoległej M. Definiujemy *przyspieszenie*

$$s_{A_p,M}(p) = \frac{T_{A_s}}{T_{A_p,M}(p)}. \tag{5.1}$$

Należy zaznaczyć, że jeśli chcemy aby powyższa definicja była precyzyjna, oba algorytmy (sekwencyjny i równoległy) muszą zawsze znajdować dokładnie to samo rozwiązanie problemu P. Ponieważ jednak założenie to jest często trudne do spełnienia, szczególnie przez algorytmy metaheurystyczne, bądź też trudno jest ustalić najlepszy algorytm sekwencyjny, wartość T_{A_s} w powyższym wzorze jest zamieniana i szacowana przez $T_{A_p,M}(1)$, czyli czas wykonania algorytmu równoległego A_p z użyciem tylko jednego procesora maszyny równoległej M.

Przyspieszenie jest miarą skrócenia czasu działania algorytmu używającego p procesorów w stosunku do algorytmu sekwencyjnego.

5.1.2 Efektywność

Efektywność $\eta_{A_p,M}(p)$ maszyny równoległej M jest definiowana przez

$$\eta_{A_p,M}(p) = \frac{s_{A_p,M}(p)}{p} \tag{5.2}$$

i określa średnią frakcję efektywnego czasu pracy każdego z procesorów. Wartość efektywności zawiera się w przedziale [0,1]. Idealna wartość efektywności wynosi jeden (mówimy wtedy o liniowym przyspieszeniu) i oznacza, że każdy z procesorów jest używany przez tak długi czas, jak to tylko możliwe – czyli brak jest przestojów procesorów.

Spotykane są problemy optymalizacyjne, w których pojawia się anomalia polegająca na tym, że efektywność algorytmu równoległego jest większa niż jeden. Powodem jest zwykle fakt podejmowania przez algorytm sekwencyjny złych decyzji dotyczących kierunku przeszukiwań przestrzeni rozwiązań, podczas gdy algorytm równoległy, badając te kierunki równocześnie, nie popełnia błędu. Efekt taki opisywany był w literaturze już w latach osiemdziesiątych a dotyczył algorytmu podziału i ograniczeń (*branch and*

bound) [27, 109, 110, 120, 124]. Jeśli chodzi o najnowsze wyniki to Porto i Ribeiro [150] oraz Bożejko i Wodecki [26] opisywali wspomniany efekt dla algorytmu tabu search. Przyspieszenie ponadliniowe pojawiało się także w przy badaniu równoległego algorytmu symulowanego wyżarzania oraz równoległego algorytmu ewolucyjnego – Bożejko i Wodecki [23, 29].

5.1.3 Koszt

Kosztem rozwiązania problemu w systemie obliczeń równoległych jest iloczyn czasu wykonania i ilości procesorów użytych przez algorytm równoległy rozwiązujący dany problem. Koszt odzwierciedla sumę czasów poświęconych przez procesory na rozwiązanie problemu. Efektywność może być także definiowana jako stosunek czasu wykonania najszybszego znanego algorytmu sekwencyjnego do kosztu rozwiązania tego samego problemu na p procesorach.

Dla algorytmów sekwencyjnych kosztem rozwiązania problemu jest czas wykonania najszybszego znanego algorytmu używającego jednego procesora. O algorytmie równoległym mówimy że jest *kosztowo optymalnym* jeśli koszt rozwiązania problemu przez ten algorytm w systemie równoległym jest proporcjonalny do czasu wykonania najszybszego znanego algorytmu sekwencyjnego na jednym procesorze. Ponieważ efektywność jest stosunkiem kosztu sekwencyjnego do kosztu równoległego, algorytm kosztowo optymalny posiada efektywność $O(1)$. Z drugiej strony, jeśli efektywność systemu równoległego wynosi $O(1)$ to stosunek kosztów sekwencyjnego do równoległego jest stały (są one proporcjonalne) a więc system taki jest kosztowo optymalny.

Niektórzy autorzy (na przykład [56]) wprowadzają pojęcie *pracy* wykonywanej przez algorytm równoległy. Jest ono równoważne pojęciu kosztu rozwiązania problemu przez system równoległy, czyli iloczynowi czasu działania algorytmu równoległego i liczby użytych procesorów. Spotykane jest także pojęcie *sekwencyjnej efektywności* algorytmu równoległego, równoważne pojęciu kosztowej optymalności.

5.2 Strategie współbieżności

Metaheurystyki oparte na metodzie lokalnych poszukiwań mogą być przedstawione jako procesy badania grafu, w którym wierzchołki stanowią punkty przestrzeni rozwiązań (np. permutacje), a łuki odpowiadają relacji sąsiedztwa – łączą wierzchołki będące rozwiązaniami sąsiednimi w tej przestrzeni. Poruszanie się po takim grafie definiuje pewną *drogę* (lub inaczej

trajektorię). Równoległe algorytmy metaheurystyczne używają wielu procesorów do współbieżnego generowania lub przeglądania grafu sąsiedztwa.

Można zdefiniować dwa podejścia do zrównoleglania procesu lokalnego poszukiwania, w zależności od ilości trajektorii badanych współbieżnie w grafie sąsiedztwa.

1. Pojedyncza trajektoria: algorytmy drobno- i średnioziarniste.
2. Wiele trajektorii: algorytmy średnio- i gruboziarniste.

Podejścia te stawiają przed algorytmem pewne wymagania dotyczące częstotliwości komunikacji, co implikuje rodzaj ziarnistości. Algorytmy drobnoziarniste odpowiadają podejściu z częstszą komunikacją, gruboziarniste – z rzadszą.

Algorytmy jednościeżkowe

Algorytmy jednościeżkowe badają pojedynczą trajektorię, jednak mogą czynić to współbieżnie poprzez podział procesu badania otoczenia na kilka procesorów, z których każdy bada pewną część otoczenia szukając najlepszego ruchu. Idea ta została zaproponowana najwcześniej dla sekwencyjnych algorytmów poszukiwań, patrz Nowicki i Smutnicki [141] pod nazwą metody reprezentantów (*representatives*). Pochodzenie nazwy jest ściśle związane z działaniem metody, bowiem z każdej części otoczenia zostaje wybrany reprezentant, a dopiero spośród reprezentantów najlepszy jako następny punkt trajektorii poszukiwań. Odpowiedniki równoległe metody reprezentantów pojawiły się w literaturze później.

Zrównoleglona może być także proces obliczania wartości funkcji celu aktualnie badanego punktu przestrzeni rozwiązań. Często jest to na przykład algorytm wyznaczania najdłuższej drogi w pewnym grafie, lub też maksymalnego czy minimalnego przepływu. Fakt ten zostanie wykorzystany i rozwinięty w dalszej części pracy.

Algorytmy wielościeżkowe

Algorytmy wykorzystujące model wielościeżkowy badają współbieżnie przestrzeń rozwiązań za pomocą równolegle działających wątków poszukiwań. Algorytmy te można dodatkowo podzielić na podklasy ze względu na wymieniane informacje o aktualnym stanie poszukiwań:

- Niezależne procesy poszukiwań.
- Kooperujące procesy poszukiwań.

W sytuacji gdy współbieżnie działające procesy poszukiwań nie wymieniają pomiędzy sobą żadnych informacji, mówimy o niezależnych (*independent*) procesorach poszukiwań. Jeśli zaś informacja zdobyta w trakcie eksploracji trajektorii przez proces poszukiwań jest przekazywana innemu procesowi i przez niego wykorzystywana, to można mówić o procesach kooperujących (*cooperative*). Spotykany jest także model mieszany, tzw. półniezależny (*semi-independent*) [59] wykonujący niezależne procesy poszukiwań przy zachowaniu pewnych wspólnych parametrów.

5.2.1 Równoległe obliczenia dla jednej trajektorii

Celem tej metody jest po prostu przyspieszenie przejścia grafu sąsiedztwa poprzez zrównoleglenie najbardziej czasochłonnych operacji – czyli obliczania wartości funkcji celu, bądź zrównoleglenie procesu generowania sąsiadów. W przypadku zrównoleglania obliczania wartości funkcji celu przyspieszenie obliczeń może być uzyskane przy zachowaniu identycznej trajektorii przejścia przez graf, jak trajektoria algorytmu sekwencyjnego. W drugim przypadku – dekompozycji generowania otoczenia na procesory równoległe – zaistnieć może sytuacja w której algorytm, sprawdzając równolegle większą ilość sąsiadów niż to czyni wersja sekwencyjna (najczęściej zaopatrzona w mechanizm redukcji rozmiarów otoczenia), poruszać się będzie po trajektorii lepszej niż sekwencyjny odpowiednik, wyznaczając korzystniejszą trasę przejścia przez graf i tym samym dochodząc do lepszych rezultatów obliczeń (wartości funkcji celu).

Zrównoleglanie przeszukiwania jednościeżkowego posiada średnią lub drobną ziarnistość i wymaga częstej komunikacji i synchronizacji. Jest ono ściśle związane z charakterystyką rozwiązywanego problemu i zdefiniowanej struktury sąsiedztwa. Pierwsze aplikacje bazujące na opisywanym modelu pojawiły się w kontekście zrównoleglania metody symulowanego wyżarzania i algorytmu genetycznego. Chociaż równoległa dekompozycja sąsiedztwa nie zawsze prowadzi do redukcji czasu obliczeń, jest często stosowana do zwiększania rozpatrywanego sąsiedztwa. Tego typu algorytm równoległy tabu search dla problemu komiwojażera został zaproponowany przez Fiechtera [67]. Synchroniczny tabu search był także badany przez Porto i Ribeiro [150–152]. W pracy [153] prezentują oni równoległe algorytmy tabu oparte na modelu *master - slave* z dynamiczną strategią obciążania procesorów.

Aarts i Verhoeven [2,184] różnicują klasę jednościeżkowych algorytmów równoległego przeszukiwania na dwie podklasy. Klasa jednokrokowa (*single – step*) obejmuje algorytmy, w których badanie otoczenia jest dzielone pomiędzy równoległe procesory, ale jako wynik wybierany jest jeden ruch. W

klasie wielokrokowej (*multiple – step*) sekwencja kolejnych ruchów w grafie sąsiedztwa jest wykonywana współbieżnie.

5.2.2 Równoległe obliczenia dla wielu trajektorii

Implementacje algorytmów opartych na równoległym wielościeżkowym przeszukiwaniu przestrzeni rozwiązań są aplikacjami gruboziarnistymi, czyli wymagającymi rzadkiej komunikacji. Są one łatwiejsze w zastosowaniu w systemach rozproszonych, jak na przykład klastrach komputerów klasy PC, dysponujących korzystnym wskaźnikiem ilorazu mocy obliczeniowej do ceny. Oprócz przyspieszenia obliczeń, uzyskać można także poprawę jakości otrzymywanych rozwiązań. Procesy poszukiwań mogą być niezależne lub kooperujące.

Niezależne procesy poszukiwań

W tej kategorii rozróżnić możemy dwa podstawowe podejścia:

1. Badanie przestrzeni rozwiązań za pomocą wielu trajektorii. Każdy z procesorów startuje z różnych rozwiązań początkowych (lub różnych populacji w przypadku algorytmu genetycznego). Wątki poszukiwań mogą stosować ten sam lub różne algorytmy lokalnego poszukiwania, z takimi samymi lub różnymi wartościami parametrów strojących (np. długość listy tabu, wielkość populacji, itp.). Trajektorie mogą przecinać się w jednym lub wielu miejscach grafu sąsiedztwa.

2. Równoległe badanie podgrafów grafu sąsiedztwa otrzymanych przez dekompozycję problemu na kilka podproblemów (np. ustalenie zmiennych). Podgrafy grafu sąsiedztwa są badane współbieżnie bez przecinania się trajektorii. Otrzymujemy całkowitą dekompozycję grafu sąsiedztwa na rozłączne podgrafy.

Pierwsza równoległa implementacja algorytmu tabu opartego na wielościeżkowym badaniu przestrzeni rozwiązań została zaproponowana przez Taillarda i dotyczyła kwadratowego zagadnienia przydziału (*quadric assgnment problem*) [175] i problemu gniazdowego (*job shop*) [176]. Zrównoleglenie algorytmu genetycznego z użyciem niezależnych wątków poszukiwań nawiązuje do tak zwanego modelu *wyspowego*, bez komunikacji pomiędzy podpopulacjami zamieszkującymi poszczególne wyspy [31,103]. Chociaż zauważono pewne przyspieszenie, nie otrzymano poprawy otrzymywanych w ten sposób rozwiązań w stosunku do wyników sekwencyjnego algorytmu genetycznego z jedną dużą populacją. Fakt ten można wytłumaczyć

szybką stagnacją podpopulacji (brakiem dalszej poprawy średniej wartości funkcji celu po pewnej liczbie wykonanych iteracji) na każdym z procesorów pozbawionym komunikacji z pozostałymi.

Strategia wielościeżkowej trajektorii opartej na niezależnych procesach przeszukiwania przestrzeni rozwiązań jest łatwa w implementacji i może osiągać dobre wartości przyspieszeń, pod warunkiem poprawnej dekompozycji przestrzeni ze względu na procesy poszukiwań. Jeśli dekompozycja ta zostanie przeprowadzona nieprawidłowo, algorytm równoległy może wykonywać wielokrotne przeszukiwania tych samych punktów przestrzeni (redundancja poszukiwań).

Kooperujące procesy poszukiwań

Model ten jest najogólniejszym i najbardziej obiecującym typem strategii przeszukiwania przestrzeni rozwiązań przez równoległy algorytm metaheurystyczny, wymaga jednak większej wiedzy programistycznej i znajomości specyfiki rozwiązywanego problemu. Kooperacja oznacza w tym wypadku wymianę informacji – doświadczeń dotyczących dotychczasowego procesu przeszukiwania przestrzeni przez równoległe procesy, a wymieniać należy specyficzne informacje, charakterystyczne dla problemu i metody – np. najlepsze znalezione rozwiązania, rozwiązania elitarne (mało różniące się od najlepszych znanych), częstotliwości ruchów, listy tabu, podpopulacje i ich rozmiary, i inne. Informacja współdzielona przez procesy poszukiwań może być implementowana jako globalne zmienne trzymane w pamięci współdzielonej (*shared memory*) lub jako pola w lokalnej pamięci dedykowanego procesora centralnego, który komunikuje się z wszystkimi pozostałymi procesorami udostępniając im na żądanie wymagane dane. W modelu, w którym procesy kooperują ze sobą a informacja zdobyta podczas poruszania się po przestrzeni wzdłuż pewnej trajektorii jest wykorzystywana do poprawy innych trajektorii, należy spodziewać się nie tylko zbieżności takiego algorytmu do rozwiązania optymalnego, ale także znalezienia w tym samym czasie rozwiązania lepszego niż algorytm pozbawiony komunikacji.

Pierwszym tego typu algorytmem heurystycznym był asynchroniczny algorytm tabu przedstawiony przez Crainic'a, Toulouse i Gendreau [52]. Pakiety takie jak EBSA [70] czy ASA [93] oferują gotowe implementacje algorytmu symulowanego wyżarzania opartego na kooperujących procesach poszukiwań, podobnie biblioteka ParSA [100]. Strategia współdziałania jest także bardzo wydajną implementacją równoległego algorytmu genetycznego (w sensie jakości otrzymywanych rozwiązań). Dostępne są gotowe biblioteki takie jak PGAPack [13] i POOGAL [31]. Większość kooperujących implementacji algorytmu genetycznego bazuje na *migracyjnym modelu wy-*

spowym. Każdy z procesorów posiada swoją własną podpopulację wymieniając co jakiś czas osobniki (zwykle najlepsze) z pozostałymi procesorami [31, 51]. Bubak i Sowa [31] zastosowali migracyjny model wyspowy do implementacji równoległego algorytmu genetycznego dla problemu komiwojażera (*TSP*) na komputerze HP/Convex Exemplar SPP1600 z 16 procesorami oraz na heterogenicznym klastrze złożonym z komputerów Hewlett Packard (D-370/2 oraz 712/60) i IBM (RS6000/520 oraz RS6000/320).

6

Poszukiwanie jednowątkowe

Metody te dopuszczają istnienie tylko jednego procesu (wątku) zarządzającego poszukiwaniami. Proces ten wykonuje cyklicznie iteracje, na które składają się

- obliczenia numeryczne (np. wyznaczanie wartości funkcji celu),
- funkcje zarządzania (np. wybór rozwiązania, realizacja pamięci obliczeń, akceptacja rozwiązań).

Czynności związane z zarządzaniem zajmują statystycznie 1-3% czasu trwania iteracji, zatem ich przyspieszanie poprzez realizację w środowisku równoległym daje znikomo mało korzyści. Odmiennie, przyspieszenie działania obliczeń numerycznych poprzez zaimplementowanie ich w środowisku wieloprocesorowym może znacznie poprawić wydajność działania algorytmu przeszukującego przestrzeń rozwiązań. Ważny jest przy tym aspekt teoretyczny przyspieszenia – użyte metody i własności powinny prowadzić do powstania algorytmów kosztowo optymalnych, czyli takich, których koszt rozwiązania problemu w systemie równoległym (iloczyn czasu wykonania i ilości użytych procesorów) jest proporcjonalny do czasu wykonania najszybszego znanego algorytmu rozwiązującego ten problem na jednym procesorze. Dla pewnych problemów uzyskanie takiej własności może być trudne – zaakceptować wówczas można także metody nie będące kosztowo optymalnymi, jeśli dają one duży zysk czasowy. Innym zagadnieniem może też być stworzenie metod, nastawionych wyłącznie na uzyskanie maksymalnego zysku związanego ze skróceniem czasu obliczeń – bez względu na liczbę użytych procesorów (czyli stawiające tezę: „mając dowolną ilość procesorów można obliczyć ... w czasie ...”). Warto zwrócić uwagę, że co prawda maksymalna liczba równolegle pracujących procesorów we współczesnych systemach obliczeń równoległych jest rzędu 10^5 procesorów, lecz metody obliczeń

biologicznych (patrz rozdział 4.8) dopuszczają równoległe wykorzystanie nawet 10^{23} procesorów. Procesory takie są wyspecjalizowanymi cząstkami (np. DNA lub białek), pozwalającymi wykonywać specjalnie spreparowane algorytmy służące do rozwiązywania problemów optymalizacji kombinatorycznej. Metoda ta znalazła już swą praktyczną realizację (rozwiązanie problemu komiwojażera za pomocą rekombinacji DNA, Adleman [4]). Tak więc w kontekście obliczeń biologicznych stworzenie szybkich metod, bazujących nawet na ogromnych ilościach procesorów, może być praktycznie użyteczne.

W analizie działania algorytmów poszukiwania można wyróżnić następujące elementy związane z oceną rozwiązań:

1. obliczanie wartości funkcji oceny pojedynczego rozwiązania (np. algorytmy SA, SJ, RS),
2. obliczanie wartości funkcji oceny dla grupy rozwiązań sąsiednich (np. algorytmy TS, AMS, LS),
3. obliczanie wartości funkcji oceny dla grupy rozwiązań rozproszonych (np. algorytmy GA, SS).

Omówimy i zanalizujemy wymienione przypadki bardziej szczegółowo, podając noweoryginalne algorytmy równoległe wyznaczania funkcji kryterialnej odpowiednio dla punktów (1), (2), (3).

Studiując literaturę można stwierdzić, że w zakresie metod (1), (3) używane są wyłącznie metody sekwencyjne oraz nie podano żadnych skutecznych metod przyspieszania tego typu obliczeń. W zakresie punktu (2) podano dla pewnej wąskiej klasy problemów z kryterium C_{max}, tzw. (sekwencyjne) *akceleratory*, patrz [170], które przyspieszają proces obliczeń poprzez umiejętną agregację i dekompozycję sekwencyjnego procesu obliczeń.

W dowodach twierdzeń dotyczących złożoności obliczeniowej algorytmów równoległych wykorzystamy następujące powszechnie znane fakty:

Fakt 1 *Ciąg sum prefiksowych* $(y_1, y_2, ..., y_n)$ *ciągu wejściowego* $(x_1, x_2, ..., x_n)$ *takich, że*

$$y_k = y_{k-1} + x_k = x_1 + x_2 + ... + x_k \text{ dla } k = 2, 3, ..., n$$

gdzie $y_1 = x_1$ *można wyznaczyć na maszynie EREW PRAM za pomocą* $O(n/\log n)$ *procesorów* [1].

[1] Notacja asymptotyczna O oszacowuje funkcję z góry, z dokładnością do stałego współczynnika, zobacz np. [56].

Bezpośrednio z powyższego wynika następujący fakt.

Fakt 2 *Sumę ciągu wejściowego* $(x_1, x_2, ..., x_n)$ *można wyznaczyć na maszynie EREW PRAM w czasie* $O(\log n)$ *za pomocą* $O(n/\log n)$ *procesorów.*

Kolejnym powszechnie znanym rezultatem jest Fakt 3.

Fakt 3 *Wartość minimalną i maksymalną ciągu wejściowego* $(x_1, x_2, ..., x_n)$ *można wyznaczyć na maszynie EREW PRAM w czasie* $O(\log n)$ *za pomocą* n *procesorów.*

Cook, Dwork i Reishuk [47] udowodnili, że wyznaczenie maksimum (minimum) ciągu n – elementowego na maszynie CREW PRAM wymaga czasu $\Omega(\log n)$ [2]. Stąd mamy, że algorytm skojarzony z Faktem 3 działa na maszynie PRAM bez równoczesnego zapisu w czasie najkrótszym możliwym.

Nieco mniej znany jest wynik mówiący, że zagadnienie przedstawione w Fakcie 3 można policzyć za pomocą liczby procesorów mniejszego rzędu, zachowując tą samą złożoność obliczeniową. Mówi o tym Fakt 4.

Fakt 4 *Wartość minimalną i maksymalną ciągu wejściowego* $(x_1,x_2,...,x_n)$ *można wyznaczyć na maszynie EREW PRAM za pomocą* $O(n/\log n)$ *procesorów w czasie* $O(\log n)$.

Zupełnie oczywisty jest Fakt 5.

Fakt 5 *Wartość* $y = (y_1, y_2, ..., y_n)$ *gdzie* $y_i = f(x_i)$, $x = (x_1,x_2,...,x_n)$ *można policzyć na maszynie CREW PRAM w czasie* $O(c) = O(1)$ *na* n *procesorach, gdzie* c *jest czasem potrzebnym do wyznaczenia pojedynczej wartości* $y_i = f(x_i)$.

Natomiast nieco mniej oczywisty jest Fakt 6, wynikający z podanych wcześniej rezultatów.

Fakt 6 *Zagadnienie sformułowane w Fakcie 5 można rozwiązać w czasie* $O(\log n)$ *na* $O(n/\log n)$ *procesorach.*

Czasami zachodzi potrzeba realizacji algorytmu na maszynie zawierającej mniejszą od wymaganej liczbę procesorów. W rozwiązaniu tego problemu będziemy korzystać z następującego faktu.

[2]Notacja asymptotyczna Ω oszacowuje funkcję z dołu, z dokładnością do stałego współczynnika, zobacz np. [56].

Fakt 7 *Jeśli algorytm A działą na p – procesorowej maszynie PRAM w czasie t, to dla każdego $p' < p$ istnieje algorytm A' dla tego samego problemu działający na p' – procesorowej maszynie PRAM w czasie $O(pt/p')$.*

Fakty 1 – 4 i 7 można znaleźć na przykład w pracy [56]. Ogólnie znane są metody obliczania sum prefiksowych, a także wartości minimalnych i maksymalnych ze zbioru n elementów, w czasie logarytmicznym z pomocą n procesorów. Nieco mniej znany jest rezultat mówiący, że na maszynie PRAM można to także zrobić za pomocą liczby procesorów mniejszego rzędu, a mianowicie $\mathrm{O}(n/\log n)$, nie tracąc rzędu złożoności obliczeniowej $O(\log n)$ całej metody. Sposób konstrukcji takich algorytmów znaleźć można także w [56], a ich idea oparta jest na ogólnym twierdzeniu Brenta. Silnie wykorzystywane jest tam założenie dotyczące maszyny PRAM o stałym czasie dostępu do pamięci współdzielonej przez wszystkie procesory.

Twierdzenia zawarte w dalszej części tego rozdziału zostały sformułowane dla modelu CREW maszyny PRAM, to jest z możliwością równoległego odczytu komórek pamięci (z zabronieniem równoległego zapisu). Ułatwiło to przedstawienie idei algorytmów zawartych w dowodach, choć część twierdzeń prawdziwa jest także dla modelu EREW. Bardziej szczegółowo temat ten omówiony jest w uwagach zamieszczonych na końcu rozdziału.

6.1 Analiza pojedynczego rozwiązania

Teoria szeregowania zadań w procesie wieloletniego rozwoju dopracowała się rutynowych technik *sekwencyjnego* obliczania wartości funkcji celu pojedynczego rozwiązania. Podejście to, jak należy przypuszczać, jest ściśle związane z domyślnym wykonywaniem zadań po kolei przez system obsługi oraz wynikającym stąd następstwem zdarzeń w systemie. Z tego punktu widzenia proces zrównoleglania obliczeń jest „wbrew naturze" zjawiska, choć, jak pokażemy dalej, może prowadzić do zaskakujących wyników teoretycznych. W tym rozdziale podamy rezultaty otrzymane przez autora rozprawy dotyczące dedykowanych algorytmów równoległych dla wybranych problemów szeregowania. Podane wyniki nie wyczerpują listy możliwości, należy je jednak traktować jako szkic kierunków i podejść.

6.1.1 Problemy jednomaszynowe

Rozpoczniemy od najprostszych rezultatów dotyczących problemów jednomaszynowych. Przypomnijmy, że złożoność obliczeniowa najlepszych sekwencyjnych metod wyznaczania wartości funkcji celu dla pojedynczego rozwiązania w problemach $1||\gamma$ oraz $1|r_j|\gamma$, $\gamma \in \{\sum f_i, f_{max}\}$ jest $O(n)$.

Twierdzenie 1 *Dla ustalonej permutacji π, wartość kryterium w problemie $1||\sum f_i$ można policzyć na maszynie CREW PRAM w czasie $O(\log n)$ za pomocą $O(n/\log n)$ procesorów.*

Dowód Bez straty ogólności rozważań możemy założyć, że $\pi(i) = i$, $i = 1,2,...,n$. Rezygnujemy świadomie z rekurencyjnego wzoru (2.7) na rzecz nierekurencyjnego (2.8) o większej złożoności obliczeniowej. Zgodnie z nim, wszystkie C_j, $j = 1,2,...,n$ można policzyć jako sumy prefiksowe, patrz Fakt 1, w czasie $O(\log n)$ używając $O(n/\log n)$ procesorów. Następnie, korzystając z Faktu 6, wartości $f_j(C_j)$, $j = 1,2,...,n$ można policzyć w czasie $O(\log n)$ używając $O(n/\log n)$ procesorów. Ostatecznie, ponieważ wartość kryterium jest równa $\sum f_i = \sum_{j=1}^{n} f_j(C_j)$, zatem z Faktu 2, możemy sumę $\sum f_i$ policzyć w czasie $O(\log n)$ używając $O(n/\log n)$ równolegle pracujących procesorów.■

Wniosek 1 Przyspieszenie metody bazującej na Tw. 1 wynosi $O(\frac{n}{\log n})$. Koszt wynosi $O(n)$. Przedstawiona metoda jest kosztowo optymalna. Jej efektywność jest $O(1)$. □

Dla praktycznych rozmiarów $n \leqslant 1000$ wymagana liczba procesorów $p \approx 100$, co jest realizowalne technicznie. Jeśli natomiast rzeczywista liczba procesorów jest mniejsza od wymaganej $O(n/\log n)$, poprzez Fakt 7 otrzymujemy złożoność obliczeniową metody $O(\frac{n}{p})$, zachowując jej kosztową optymalność.

Bezpośrednio z dowodu Tw. 1 wynika, że dla ustalonej permutacji π, wartość kryterium w problemie $1||f_{max}$ można policzyć na maszynie CREW PRAM w czasie $O(\log n)$ korzystając z $O(n/\log n)$ procesorów. Istotnie, jedyną różnicą jest sposób obliczenia funkcji celu w końcowym etapie; liczona jest wartość $\max_{1\leqslant j\leqslant n} f_j(C_j)$ zamiast $\sum_{j=1}^{n} f_j(C_j)$. Zgodnie z Faktem 3, obliczenie maksimum z n liczb można wykonać za pomocą $O(n/\log n)$ procesorów w czasie $O(\log n)$. Cała metoda jest kosztowo optymalna z efektywnością $O(1)$. Ponieważ problem $1||f_{max}$ jest wielomianowy, równoległy wariant liczenia kryterium ma mniejsze zastosowanie. Przeciwnie, Tw. 1 może być stosowane już dla problemów z nieliniowymi funkcjami $f_i(t)$, takich jak na przykład NP-trudny problem $1||\sum w_i T_i$.

Wraz ze wzrostem stopnia trudności problemów, zmienia się odpowiednio ocena algorytmów. Problem $1|r_j|\sum f_i$ jest NP-trudny nawet dla najprostszych postaci funkcji $f_i(t)$.

Twierdzenie 2 *Dla ustalonej permutacji π, wartość kryterium w problemie $1|r_j|\sum f_i$ można policzyć na maszynie CREW PRAM w czasie $O(\log n)$ korzystając z $O(n^2/\log n)$ procesorów.*

Dowód Bez straty ogólności rozważań zakładamy, że $\pi(i) = i$, $i = 1, 2, \ldots, n$. Rezygnujemy ze wzoru rekurencyjnego (2.12) na rzecz nierekurencyjnego (2.13), który można przedstawić następująco:

$$C_j = \max_{1 \leqslant u \leqslant j} (r_u + P_j - P_{u-1}) \quad \text{dla} \quad j = 1, 2, \ldots, n, \tag{6.1}$$

gdzie $P_s = \sum_{i=1}^{s} p_i$.

Zaczynamy od policzenia sum prefiksowych P_j, $j = 1, 2, \ldots, n$. Za pomocą $O(n/\log n)$ procesorów można to zrobić w czasie $O(\log n)$, co wynika z Faktu 1. Następnie w czasie $O(\log n)$ liczymy dla $j = 1, 2, \ldots, n$, $u = 1, 2, \ldots, j$ wielkości

$$R_{uj} = r_u + P_j - P_{u-1} \tag{6.2}$$

korzystając w tym kroku z $O(\frac{n(n-1)}{2 \log n}) = O(n^2/\log n)$ procesorów. Wartość C_j wyrazić można jako

$$C_j = \max_{1 \leqslant u \leqslant j} R_{uj} \quad \text{dla} \quad j = 1, 2, ..., n. \tag{6.3}$$

Obliczenie pojedynczej wartości C_j na bazie wzoru (6.3) można zrealizować w czasie $O(\log j) = O(\log n)$ za pomocą $O(n/\log n)$ procesorów, stąd wszystkie wielkości C_j można policzyć niezależnie w czasie $O(\log n)$ za pomocą $O(n^2/\log n)$ procesorów. Sumę $\sum f_i = \sum_{j=1}^{n} f_j(C_j)$ można obliczyć w czasie $O(\log n)$ za pomocą $O(n/\log n)$ procesorów (Fakt 2). Cały proces wymaga więc $O(n^2/\log n)$ procesorów.■

Wniosek 2 Przyspieszenie metody bazującej na Tw. 2 wynosi $O(\frac{n}{\log n})$. Koszt wynosi $O(n^2)$, zaś efektywność $O(\frac{1}{n})$. □

Efektywność metody maleje bardzo szybko wraz ze wzrostem wielkości n, co należy uznać za fakt niekorzystny. Oferowane przyspieszenie jest dla $n = 10$ równe 3, zaś dla $n = 1000$ około 100. Niestety efektywność metody dla $n = 1000$ jest poniżej 0.1% [3].

6.1.2 Problem przepływowy

Złożoność obliczeniowa najlepszego sekwencyjnego algorytmu wyznaczającego wartość funkcji celu dla pojedynczego rozwiązania w problemie $F^*||\gamma$, $\gamma \in \{f_{max}, \sum f_i\}$ jest $O(nm)$. Korzystając z różnych podejść możliwe jest otrzymanie różnych algorytmów równoległych o odmiennych charakterystykach.

[3] oszacowanie prawdziwe z dokładnością do stałego mnożnika

Rysunek 6.1: Kolejność obliczania C_{ij}.

Twierdzenie 3 *Dla ustalonej permutacji π, wartość kryterium w problemie $F^*||\sum f_i$ można wyznaczyć na maszynie CREW PRAM w czasie $O(n+m)$ wykorzystując m procesorów.*

Dowód Bez straty ogólności możemy przyjąć, że $\pi = (1, 2, \ldots, n)$. Skorzystamy z zależności rekurencyjnej (2.14). Proces równoległego wyznaczania wartości $C_{i,j}$ wykonujemy następująco. W pierwszym takcie procesor 1 wyznacza wartość $C_{1,1}$. W drugim takcie procesory 1 i 2 wyznaczają równolegle wartości odpowiednio $C_{2,1}$ i $C_{1,2}$. W takcie k-tym procesor numer t wyznacza wartość $C_{k-t+1,t}$ jeśli $1 \leqslant k-t+1 \leqslant n$. Opisywana kolejność jest przedstawiona na Rys. 6.1 za pomocą linii przerywanych, na tle grafu $G(\pi)$, zdefiniowanego w Rozdz. 2.2 (w dalszej części pracy ten charakterystyczny graf będziemy nazywać grafem siatkowym).

Opisane postępowanie wymaga wykonania $n+m-1$ kroków. Dla obliczenia wartości kryterium $\sum f_i = \sum_{j=1}^{n} f_j(C_{m,j})$ należy jeszcze dodać n wartości $f_j(C_{m,j})$, co można wykonać sekwencyjnie w n iteracjach lub równolegle, za pomocą m procesorów przy złożoności $O(n/m + \log n)$. Ostatecznie złożoność obliczeniowa wyznaczenia kryterium w problemie $F^*||\sum f_i$ wynosi więc $O(n+m)$ przy pomocy m procesorów.■

Wniosek 3 Przyspieszenie metody bazującej na Tw. 3 wynosi $O(\frac{nm}{n+m})$, efektywność wynosi $O(\frac{n}{n+m})$. □

Zaprezentowana metoda nie jest więc kosztowo optymalna. Jej efektywność pogarsza się stosunkowo wolno wraz ze wzrostem m. Przykładowo, dla $n = 10, m = 3$ efektywność wynosi około 77%, zaś gdy $m = 10$ mamy efektywność 50% [4]. Oczywiście, dla $n \gg m$ efektywność zbliża się do 100%. Dla $n \ll m$ efektywność szybko maleje do 0. Na Rys. 6.2 zaprezentowano zmianę efektywności metody, w funkcji ilości zadań n, dla kilku wybranych wartości parametru m (ilości maszyn).

Opisana metoda wymaga „wymuszonej" liczby procesorów $p = m$, co nie powinno sprawiać kłopotów w praktyce (zwykle $m \leqslant 20$). Zastosowanie Faktu 7 może doprowadzić do realizacji opisanego algorytmu równoległego w środowisku zawierającym $p < m$ procesorów. Wtedy złożoność obliczeniowa takiego algorytmu wyniesie $O((n+m)m/p)$, choć problem konstrukcji odpowiedniego algorytmu pozostaje otwarty. Oczywiście, jeśli $p \geqslant m$, to $(p-m)$ procesorów będzie zawsze bezczynnych.

[4] oszacowanie prawdziwe z dokładnością do stałego mnożnika

Rysunek 6.2: Efektywność metody zaproponowanej w Tw. 3.

Pokażemy dalej inną metodę, dla problemu $F^*||\sum f_i$, o zdecydowanie lepszej charakterystyce.

Twierdzenie 4 *Dla ustalonej permutacji π, wartość kryterium w problemie $F^*||\sum f_i$ można wyznaczyć w czasie $O(n+m)$ na $O(\frac{nm}{n+m})$ procesorowej maszynie CREW PRAM.*

Dowód Bez straty ogólności możemy przyjąć, że $\pi = (1, 2, \ldots, n)$. Opieramy się na schemacie obliczeń zaprezentowanym na Rys. 6.3. Niech $p \leqslant m$ będzie ilością użytych procesorów. Proces obliczeń będzie przeprowadzany dla *poziomów* $k = 1, 2, \ldots, n + m - 1$. Na poziomie k przeprowadzane są obliczenia wszystkich wartości $C_{i,j}$ takich, że $i + j - 1 = k$. Wszystkie obliczenia na poziomie k przeprowadzane są po obliczeniu wszystkich wartości $C_{i,j}$ na poziomie $k - 1$. Głębokość całego schematu obliczeń (ilość poziomów) niech oznaczona będzie przez d i wynosi $d = n + m - 1$. Niech n_k będzie liczbą wartości $C_{i,j}$ obliczanych na poziomie k. Wobec tego

$$\sum_{k=1}^{d} n_k = nm \tag{6.4}$$

ponieważ ilość wszystkich obliczanych wartości $C_{i,j}$ na wszystkich poziomach wynosi nm. Rozważmy n_k elementów na poziomie k. Grupujemy je w $\left\lceil \frac{n_k}{p} \right\rceil$ grup, z których pierwsze $\left\lfloor \frac{n_k}{p} \right\rfloor$ grup zawiera po p elementów, a pozostałe elementy (jest ich najwyżej p) niech należą do ostatniej grupy (podział na grupy na poszczególnych poziomach zaznaczony jest na Rys. 6.3). Obliczenia równoległe maszyny PRAM na k-tym poziomie wykonywane są więc w czasie $O(\left\lceil \frac{n_k}{p} \right\rceil)$. Łączny czas obliczeń jest równy sumie czasów wykonywania na wszystkich poziomach i jest rzędu

$$\sum_{k-1}^{d} \left\lceil \frac{n_k}{p} \right\rceil \leqslant \sum_{k-1}^{d} \left(\frac{n_k}{p} + 1 \right) = \frac{nm}{p} + d = \frac{nm}{p} + n + m - 1. \tag{6.5}$$

Poszukujemy wartości p, $1 \leqslant p \leqslant m$, dla której efektywność algorytmu równoległego jest $O(1)$, co gwarantuje jego kosztową optymalność. Stąd mamy wymaganie

$$\eta_{A_p,M}(p) = \frac{1}{p} \, \frac{nm}{\frac{nm}{p} + n + m - 1} = c = O(1) \tag{6.6}$$

dla pewnej stałej $c < 1$, co pozwala nam wyznaczyć drogą prostych przekształceń

$$p = \frac{nm}{n+m-1}\left(\frac{1}{c} - 1\right) = O(\frac{nm}{n+m}) \tag{6.7}$$

Przyjmując $p = O(\frac{nm}{n+m})$, łączny czas obliczeń wartości C_{ij} wyniesie

$$O(\frac{nm}{p} + n + m - 1) = O(\frac{nm}{\frac{nm}{n+m}} + n + m) = O(n+m). \tag{6.8}$$

Obliczenie wartości kryterium $\sum f_i = \sum_{j=1}^{n} f_j(C_{m,j})$ można wykonać za pomocą jednego procesora w czasie $O(n)$. Zauważmy, że można wykonać to szybciej, za pomocą większej ilości procesorów, ale w rozpatrywanym przypadku, chcąc uzyskać złożoność $O(n+m)$, wystarczy to zrobić sekwencyjnie. Ostatecznie łączny czas obliczeń wyniesie $O(n+m+n) = O(n+m)$.■

Wniosek 4 Przyspieszenie metody bazującej na Tw. 4 wynosi $O(\frac{nm}{n+m})$. Koszt wynosi $O(nm)$. Ilość używanych procesorów

$$O(\frac{nm}{n+m}) = O(\frac{1}{\frac{1}{m} + \frac{1}{n}}) \tag{6.9}$$

jest rzędu średniej harmonicznej liczb n i m. Zaproponowana metoda jest kosztowo optymalna. □

Przedstawiona metoda pozwala zarówno na sterowanie efektywnością i szybkością poprzez wybór liczby procesorów, jak i dostosowanie tych parametrów do liczby procesorów istniejących w systemie rzeczywistym. Niezależnie od tego, można podać „optymalną" liczbę procesorów zapewniającą kosztową optymalność metody. Jej wyznaczenie było możliwe poprzez elastyczne dostosowanie liczby zaangażowanych procesorów do obu rozmiarów problemu, to znaczy do n i m równocześnie. Przykładowo, dla $n \gg m$ mamy $p \approx m$, dla $n \ll m$ mamy $p \approx n \ll m$, dla $n \approx m$ mamy $p \approx n/2$.

Z dowodu Tw. 4 wynika, że po obliczeniu czasów zakończenia wykonywania zadań $C_{i,j}$, $i = 1,2,...,m$, $j = 1,2,...,n$, wielkość $\max_{1 \leqslant j \leqslant n} f_j(C_{m,j})$ jest wartością funkcji kryterialnej dla problemu $F^*||f_{max}$. Stąd otrzymujemy natychmiast następujące twierdzenie nie wymagające dowodu. Zachowana zostaje przy tym taka sama złożoność obliczeniowa oraz rząd ilości użytych procesorów, jak w Tw. 4.

Twierdzenie 5 *Dla ustalonej permutacji π, wartość kryterium w problemie $F^*||f_{max}$ można wyznaczyć w czasie $O(n+m)$ na $O(\frac{nm}{n+m})$ procesorowej maszynie CREW PRAM.*

Rysunek 6.3: Kolejność obliczania C_{ij}. Wyróżniono obliczenia przyporządkowane puli p procesorów.

Wniosek 5 Metoda skojarzona z Tw. 5 jest kosztowo optymalna. □

W Tw. 3, 4 oraz 5 korzystaliśmy z rekurencyjnego schematu (2.14) obliczania terminów C_{ij}. W Rodz. 6.1.1 pokazano, że nierekurencyjne techniki obliczania wartości funkcji celu zwykle lepiej poddają się procesowi zrównoleglania. W przypadku problemu przepływowego wzór nierekurencyjny (2.15) jest używany tylko do dowodzenia własności problemu, nie zaś do obliczania wartości funkcji celu explicite, ze względu na jego znaczną złożoność obliczeniową. Pokażemy dalej, że wykorzystanie wzoru (2.15) może prowadzić do ciekawych rezultatów teoretycznych.

Twierdzenie 6 *Dla ustalonej permutacji π, wartość kryterium w problemie $F^*||C_{max}$, można wyznaczyć w czasie $O(m + \log n)$ na $O(\frac{(n+m)^{n-1}}{m(n-1)!})$ procesorowej maszynie CREW PRAM.*

Dowód Bez straty ogólności rozważań możemy założyć, że $\pi(i) = i$, $i =$ 1,2,...,n. Rezygnujemy świadomie z rekurencyjnego wzoru (2.14) na rzecz nierekurencyjnego (2.15), który możemy zapisać w postaci

$$C_{ij} = \max_{1=j_0 \leqslant j_1 \leqslant \ldots \leqslant j_i = j} \sum_{s=1}^{i} \left(P_{s,j_s} - P_{s,j_{s-1}-1} \right), \tag{6.10}$$

gdzie $P_{s,t} = \sum_{k=1}^{t} p_{sk}$, $t = 1, 2, \ldots, n$, są sumami prefiksowymi. Korzystając z Faktu 1, dla ustalonego s wartości $P_{s,t}$, $t = 1, 2, \ldots, n$, można obliczyć w czasie $O(\log n)$, co wymaga $O(n/\log n)$ procesorów. Zatem wszystkie $P_{s,t}$ dla $t = 1, 2, \ldots, n$, $s = 1, 2, \ldots, m$ możemy wyznaczyć za pomocą $O(mn/\log n)$ procesorów w czasie $O(\log n)$ na etapie wstępnym. Do wyznaczenia wartości kryterium potrzebna jest wielkość C_{mn}. Korzystamy z faktu, że ilość wszystkich podciągów $(j_0, j_1, \ldots, j_m)$ spełniających warunek $1 = j_0 \leqslant j_1 \leqslant \ldots \leqslant j_m = n$, odpowiada liczbie kombinacji $m-1$ elementowych z powtórzeniami ze zbioru $n-2$ elementowego, i wynosi $\binom{n+m-2}{m-1}$. Sposób wygenerowania wszystkich takich podciągów w czasie $O(m)$ za pomocą $\binom{n+m-2}{m-1}$ procesorów znaleźć można w Lemacie 1 zamieszczeonym w Dodatku B. Następnie za pomocą $\binom{n+m-2}{m-1}$ procesorów można wyznaczyć sekwencyjnie wszystkie sumy $\sum_{s=1}^{m} \left(P_{s,j_s} - P_{s,j_{s-1}-1} \right)$ ze wzoru (6.10) dla wszystkich podciągów w czasie $O(m)$. Dla wyznaczenie wartości C_{mn}, należy obliczyć maksimum z $\binom{n+m-2}{m-1}$ obliczony sum, co zgodnie Faktem 4 możemy wykonać w czasie $O(\log\binom{n+m-2}{m-1})$ za pomocą $O(\binom{n+m-2}{m-1}/\log\binom{n+m-2}{m-1})$

procesorów. Złożoność obliczeniowa tego kroku, poprzez nierówność

$$\binom{n+m-2}{m-1} = \frac{m(m+1)\dots(n+m-2)}{(n-1)!} \leqslant \frac{(n+m)^{n-1}}{(n-1)!} \tag{6.11}$$

oraz zależność $\log(n!) = \Theta(n \log n)$, wynosi [5]

$$O(\log \frac{(n+m)^{n-1}}{(n-1)!}) = O(n \log(1 + \frac{m}{n})). \tag{6.12}$$

Zauważmy, że (6.12) na podstawie nierówności

$$n \log(1 + \frac{m}{n}) \leqslant \lim_{n\to\infty}(n \log(1 + \frac{m}{n})) = \log \lim_{n\to\infty}(1 + \frac{m}{n})^n = \log e^m = m \log e \tag{6.13}$$

jest $O(m)$. Ponieważ złożoność obliczeniowa pozostałych kroków algorytmu nie przekracza $O(\max(m, \log n)) = O(m + \log n)$, taka też jest ostateczna złożoność obliczeniowa algorytmu. Liczba użytych procesorów wynosi

$$O(\max(\frac{mn}{\log n}, \frac{\binom{n+m-2}{m-1}}{\log\binom{n+m-2}{m-1}})) = O(\frac{(n+m)^{n-1}}{m(n-1)!}).\blacksquare$$

Wniosek 6 Dla metody bazującej na Tw. 6 mamy

$$s_{A_p,M}(p) = O(\frac{mn}{m + \log n}). \square \tag{6.14}$$

Powyższy wynik ma raczej teoretyczne znaczenie, bowiem liczba procesorów wymagana do obliczeń rośnie wykładniczo ze wzrostem n, pogarszając jednocześnie bardzo szybko efektywność.

Łatwo zauważyć, że możliwe jest rozszerzenie otrzymanego rezultatu na problemy $F^*||\gamma$, $\gamma \in \{\sum f_i, f_{max}\}$. Wyznaczenie wszystkich wartości C_{mj}, $j = 1, 2, \dots, n$ ma złożoność obliczeniową $O(m+\log n)$ przy n razy większej liczbie procesorów, niż ta wymieniona w Tw. 6.

Dwa następne twierdzenia także odnoszą się do problemu wyznaczania wartości funkcji kryterialnej dla problemów przepływowych z kryterium $\sum f_i$ i f_{max}. Pozwalają osiągnąć mniejszą złożoność obliczeniową (rzędu logarytmicznego), kosztem zwiększenia ilości procesorów i utraty własności kosztowej optymalności (podobnie jak poprzednio), korzystając z odmiennych technik prowadzenia obliczeń.

[5] Notacja Θ określa asymptotyczne oszacowanie funkcji, z dokładnością do stałego współczynnika, zobacz np. [56].

Twierdzenie 7 *Dla ustalonej permutacji π, wartość kryterium w problemie $F^*||\sum f_i$ można wyznaczyć w czasie $O(\log(n+m)\log(nm))$ za pomocą $O(\frac{(nm)^3}{\log(nm)})$ procesorowej maszyny CREW PRAM.*

Dowód Zaproponowana zostanie metoda równoległa oparta na sekwencyjnym algorytmie Floyda-Warshalla [56], dedykowanym do wyznaczania najkrótszych ścieżek w grafie. Bez straty ogólności można założyć, że $\pi = (1, 2, \ldots, n)$. Obliczenia wartości C_{ij} przeprowadzimy z użyciem grafu siatkowego $G(\pi)$ opisanego w Rozdz. 2.2.

Aby uzyskać większą czytelność oznaczeń, oryginalny graf $G(\pi)$ przekształcono do równoważnego grafu $G^*(\pi)$ dokonując przenumerowania wierzchołków. Dla uzyskania jednolitej (złożonej z jednego, a nie pary indeksów) numeracji wierzchołków nadajemy im numery $1, 2, \ldots, nm$ kolejnymi wierszami od lewego górnego wierzchołka do prawego dolnego. Stąd definiujemy nowy graf $G^*(\pi)$ o zbiorze wierzchołków

$$W = \{u : u = 1, 2, ..., nm\}. \tag{6.15}$$

Wierzchołkowi (i, j) w grafie $G(\pi)$ odpowiada w transformacji wzajemnie jednoznacznej wierzchołek

$$u = (i-1)n + j \tag{6.16}$$

nowego grafu $G^*(\pi)$. Wierzchołkowi $u \in W$ odpowiada wierzchołek (i, j) grafu $G(\pi)$ taki, że

$$i = \left\lfloor \frac{u-1}{n} \right\rfloor + 1, \quad j = u - \left\lfloor \frac{u-1}{n} \right\rfloor n. \tag{6.17}$$

W ten sposób otrzymujemy graf skierowany

$$G^*(\pi) = (W, E^0 \cup E^*), \tag{6.18}$$

gdzie zbiory łuków pionowych i poziomych wyglądają następująco.

$$E^0 = \bigcup_{u=1}^{nm-n} \{(u, u+n)\},\ E^* = \bigcup_{k=1}^{m} \bigcup_{u=(k-1)n}^{kn-1} \{(u, u+1)\}. \tag{6.19}$$

Wierzchołek $u \in W$ otrzymuje wagę (oznaczoną dalej p_u) odpowiadającą wadze p_{ij} wierzchołka z nim skojarzonego w grafie $G(\pi)$. Graf $G^*(\pi)$ przedstawiony jest na Rys. 6.4.

Rysunek 6.4: Graf $G^*(\pi)$.

Dla grafu $G^*(\pi)$ wprowadzamy macierz odległości $A = [a_{u,v}]$ o rozmiarze $nm \times nm$, gdzie $a_{u,v}$ jest najdłuższą drogą pomiędzy wierzchołkami u i v. Wartości $a_{u,v}$ inicjujemy następująco:

$$a_{u,v} = \begin{cases} p_u & \text{jeśli} \quad (u,v) \in E^0 \cup E^* \\ 0 & \text{jeśli} \quad (u,v) \notin E^0 \cup E^* \end{cases} \tag{6.20}$$

Macierz A posłuży do obliczenia długości najdłuższych ścieżek w grafie $G^*(\pi)$, które są także długościami najdłuższych ścieżek w grafie $G(\pi)$. Początkowe wartości macierzy A można wyznaczyć w czasie $O(1)$ za pomocą $(nm)^2$ procesorów, ponieważ operacja ta polega na wykonaniu $(nm)^2$ niezależnych instrukcji przypisania.

Problem wyznaczenia wartości funkcje celu dla problemu przepływowego $F^*||\sum f_i$ wymaga znalezienia długości najdłuższych dróg z wierzchołka $1 \in W$ do wierzchołków $(m-1)n+1, (m-1)n+2, \ldots, mn$ (co odpowiada wyznaczeniu następujących wartości czasów zakończeń wykonywania zadań: $C_{m,1}$, $C_{m,2}$,...,$C_{m,n}$). Dla wyznaczenia długości ścieżek wystarczy wykonać $\lceil \log(n+m-1) \rceil$ równoległych kroków, bowiem w każdym kroku $k = 1, 2, \ldots, \lceil \log(n+m-1) \rceil$ opisany poniżej algorytm aktualizuje długości najdłuższych ścieżek pomiędzy wierzchołkami odległymi (w sensie liczby wierzchołków) o $1, 2, 4, 8, ..., 2^{\log(n+m-1)}$ (zobacz także [106]). Po wykonaniu $\lceil \log(n+m-1) \rceil$ kroków macierz A zawiera informacje o długościach dróg pomiędzy wierzchołkami odległymi (w sensie liczby wierzchołków) o $2^{\log(n+m-1)} = (n+m-1)$, czyli pomiędzy *wszystkimi* wierzchołkami – bowiem liczba wierzchołków na najdłuższej (w sensie liczby wierzchołków) ścieżce, przebiegającej z wierzchołka 1 do nm, wynosi $(n+m-1)$.

Dla potrzeb algorytmu, definiowana jest dodatkowo trójwymiarowa tablica $T = [t_{u,w,v}]$, o rozmiarze $nm \times nm \times nm$, służąca do obliczania tranzytywnego domknięcia długości ścieżek w grafie $G^*(\pi)$. Algorytm wymaga wykonania $\lceil \log(n+m) \rceil$ razy następujących identycznych kroków:

1. aktualizacja $t_{u,w,v}$ dla wszystkich trójek (u,w,v) zgodnie z zależnością $t_{u,w,v} = a_{u,w} + a_{w,v}$,
2. aktualizacja $a_{u,v}$ dla wszystkich par (u,v) na podstawie zależności $a_{u,v} = \max\{a_{u,v}, \max_{1 \leqslant w \leqslant nm} t_{u,w,v}\}$.

Krok 1 wykonywany na $(nm)^3$ procesorach mógłby być zrealizowany w czasie $O(1)$. Na $\lceil (nm)^3 / \log(nm) \rceil$ procesorach obliczenia trzeba wy-

Rysunek 6.5: Porównanie przyspieszenia metod bazujących na Tw. 5, Tw. 6 i Tw. 8 dla $m = 10$.

konać $\lceil \log(nm) \rceil$ - krotnie, a więc złożoność wykonania kroku 1 wynosi $O(\log(nm))$.

Krok 2 polega na wyznaczeniu maksimum $nm + 1$ wartości, co można wykonać na $O(nm/\log(nm))$ procesorach w czasie $O(\log(nm))$ (patrz Fakt 3). Ponieważ maksimum takie należy wyznaczyć dla $(nm)^2$ par $(u, v) \in W$ ($|W| = nm$), a obliczenia te są niezależne, więc należy je powtórzyć $\lceil \log(n+m) \rceil$ - krotnie, stąd cała opisana metoda wymaga ilości procesorów rzędu

$$(nm)^2 O(nm/\log(nm)) = O((nm)^3/\log(nm))$$

Złożoność obliczeniowa opisanego powyżej fragmentu algorytmu wynosi

$$\lceil \log(n+m) \rceil O(\log(nm)) = O(\log(n+m)\log(nm)).$$

Ostatecznie, dla obliczenia wartości kryterium, należy jeszcze zsumować n wartości $f_j(C_{mj})$, gdzie C_{mj} odpowiada odległości pomiędzy wierzchołkiem $1 \in W$, a wierzchołkiem $(m-1)n + j$, $j = 1, 2, \ldots, n$. Można to zrobić w czasie $O(\log n)$ za pomocą $O(n/\log n)$ procesorów (patrz Fakt 2), co zachowuje złożoność obliczeniową $O(\log(n+m)\log(nm))$ całej opisanej metody oraz liczbę procesorów $O(\frac{(nm)^3}{\log(nm)})$.■

Wniosek 7 Dla metody bazującej na Tw. 7 mamy

$$s_{A_p,M}(p) = O(\frac{nm}{\log(n+m)\log(nm)}),\ \eta_{A_p,M}(p) = O(\frac{1}{(nm)^2\log(n+m)}).\ \square \tag{6.21}$$

Efektywność przedstawionej metody jest bardzo daleka od optymalnej $O(1)$. Jedyny zysk związany jest ze znacznym zmniejszeniem złożoności obliczeniowej, a mianowicie z rzędu liniowego do logarytmicznego (do kwadratu) względem rozmiarów zadania. Niestety, odbywa się to kosztem znacznego zwiększenia ilości użytych procesorów. Analiza zysków prowadzi do zaskakujących wyników. Przykładowo, dla $n = 10$, $m = 3$ efektywność metody jest około 0.03% przy przyspieszeniu 1.65. Dla $n = 20$, $m = 3$ efektywność jest poniżej 0.01%, zaś przyspieszenie wynosi 2.25.

Korzystając z dowodu Tw. 7 można sformułować w pełni analogiczne twierdzenie dla kolejnej klasy problemów przepływowych.

Rysunek 6.6: Porównanie czasów działania metod bazujących na Tw. 5, Tw. 6 i Tw. 8 dla $m = 10$.

Rysunek 6.7: Porównanie czasów działania metod bazujących na Tw. 5, Tw. 6 i Tw. 8 zależnie od liczby procesorów.

Twierdzenie 8 *Dla ustalonej permutacji π, wartość kryterium w problemie $F^*||f_{max}$ można wyznaczyć w czasie $O(\log(n+m)(\log(nm)))$ korzystając z $O((nm)^3/\log(nm))$ procesorowej maszyny CREW PRAM.*

Dowód wynika z dowodu Tw. 7.

Wniosek 8 Oceny algorytmu skojarzonego z Tw. 8 są takie same jak we Wniosku 6. □

Interesujące jest porównanie wyników Tw. 5, Tw. 6 i Tw. 8. Nie bacząc na liczbę potrzebnych procesorów, otrzymane przyspieszenie i czasy działania przedstawiono na Rys. 6.5 oraz Rys. 6.6 [6] dla ustalonego $m = 10$. Można zaobserwować występowanie „anomalii" dla małych n oraz m, dla których metoda z Tw. 8 jest nie tylko mniej efektywna, ale także wolniejsza od tej z Tw. 5. Z kolei metoda z Tw. 6, przy ustalonej wartości m, oferuje niespotykanie krótki czas działania, oraz przyspieszenie szybko wzrastające dla dużych n.

Z kolei na Rys. 6.7 oraz Rys. 6.8 przedstawiono zależność od liczby procesorów złożoności obliczeniowej oraz przyspieszenia odpowiednich algorytmów, wykorzystując Tw. 5, 6 i 8 oraz Fakt 7. Przedstawione rysunki dotyczą małych rozmiarów n oraz m ($n = 10, m = 5$), bowiem dla dużych wartości n, m wymagane liczby procesorów dla Tw. 6 oraz 8 rosną bardzo szybko. Na rysunkach tych widać także anomalię pomiędzy Tw. 5 oraz 8.

Tw. 6 oraz 8 burzą, w pewnym sensie, intuicję mówiącą, że skoro dla problemu $F^*||C_{max}$ na najdłuższej ścieżce (ścieżce krytycznej) znajduje się $(n + m-1)$ wierzchołków związanych zależnością rekurencyjną, to należy wykonać co najmniej $O(n + m)$ iteracji. Okazuje się, że można osiągnąć złożoność obliczeniową niższego rzędu, a mianowicie $O(\log(n+m)\log(nm))$

[6] oszacowania prawdziwe z dokładnością do stałego mnożnika

Rysunek 6.8: Porównanie przyspieszenia metod bazujących na Tw. 5, Tw. 6 i Tw. 8 zależnie od liczby procesorów.

lub $O(m+\log n)$. Dla ustalonej ilości maszyn m można więc otrzymać odpowiednio złożoności $O(\log^2 n)$ (metoda z Tw. 8) czy nawet $O(\log n)$ (metoda z Tw. 6). Otwartym pozostaje problem, czy jest to wartość graniczna.

6.1.3 Problem gniazdowy

W pracy [168], Steinhöfel i inni, pokazano, że wartość funkcji celu pojedynczego rozwiązania dla problemu gniazdowego $J||C_{max}$ można wyznaczyć w czasie $O(\log^2 o)$ na $O(o^3)$ procesorowej maszynie PRAM (gdzie o jest liczbą operacji danego problemu gniazdowego). Poniżej pokazany zostanie mocniejszy rezultat, a mianowicie: do przeprowadzenia obliczeń wystarczy liczba procesorów rzędu $O(\frac{o^3}{\log o})$, przy zachowaniu złożoności $O(\log^2 o)$. Nowo proponowany algorytm jest $O(\log o)$ razy efektywniejszy, przy czym ma on zastosowanie zarówno dla problemu $J||\sum f_i$ jak i $J||f_{max}$.

Twierdzenie 9 *Dla ustalonej dopuszczalnej kolejności wykonywania operacji π, wartość kryterium w problemie $J||\sum f_i$ można wyznaczyć w czasie $O(\log^2 o)$ na $O(\frac{o^3}{\log o})$ procesorowej maszynie CREW PRAM.*

Dowód Stosujemy schemat postępowania analogiczny, do opisanego w dowodzie Tw. 7. Także tu odwołujemy się do algorytmu Floyda-Warshalla z tym, że obliczenia przeprowadzamy dla grafu $G(\pi)$ skojarzonego z rozwiązaniem dopuszczalnym problemu gniazdowego, patrz Rozdz. 2.4.

Odpowiednik kroku 1 wykonywany na o^3 procesorach mógłby być wykonany w czasie $O(1)$. Na $\lceil o^3/\log o\rceil$ procesorach obliczenia trzeba wykonać $\lceil \log o\rceil$ - krotnie, a więc złożoność wykonania kroku 1 wynosi $O(\log o)$.

Odpowiednik kroku 2 polega na wyznaczeniu maksimum z $o+1$ wartości, co można wykonać na $O(o/\log o)$ procesorach w czasie $O(\log o)$ (patrz Fakt 3). Ponieważ maksimum takie należy wyznaczyć dla o^2 par (u,v), a obliczenia te są niezależne i należy je powtórzyć $\lceil \log o\rceil$ - krotnie, więc za pomocą $p = O(o^3/\log o)$ procesorów cała opisana procedura ma złożoność

$$T_{A_p,M}(p) = \lceil \log o\rceil O(\log o) = O(\log^2 o). \tag{6.22}$$

Ostatecznie, dla obliczenia wartości kryterium, należy jeszcze zsumować wielkości $f_j(C_{l_j})$, gdzie $C_{l_j} = a_{0,l_j}$ należy pobrać z tablicy A (zobacz dowód Tw. 7). Można to zrobić w czasie $O(\log n)$ za pomocą $O(n/\log n)$ procesorów (patrz Fakt 2), co zachowuje górne ograniczenie złożoności obliczeniowej $O(\log^2 o)$ oraz liczbę procesorów $O(o^3/\log o)$ całej opisanej metody, ponieważ ilość zadań n jest mniejsza bądź równa liczbie operacji o.■

Wniosek 9 Dla metody bazującej na Tw. 9 mamy

$$s_{A_p,M}(p) = O(\frac{o}{\log^2 o}), \quad \eta_{A_p,M}(p) = O(\frac{1}{o^2 \log o}). \square \tag{6.23}$$

Efektywność zaprezentowanej metody równoległej maleje szybko wraz ze wzrostem rozmiarów zadania. Nie jest więc ona kosztowo optymalna, choć złożoność obliczeniowa $O(\log^2 o)$ daje spory zysk czasowy względem sekwencyjnej złożoności $O(o)$, tym bardziej, że za notacją O w przypadku proponowanej metody nie kryją się duże stałe. Porównanie szybkości wzrostu funkcji $f(o) = o$ oraz $f(o) = \log^2 o$ zostało przedstawione w Tab. 6.1.

o	$\lceil \log^2 o \rceil$
10	12
100	45
1,000	100
10,000	177
100,000	276
1,000,000	398
10,000,000	541

Tablica 6.1: Szybkość wzrostu funkcji $f(o) = o$ oraz $f(o) = \lceil \log^2 o \rceil$.

Twierdzenie 10 *Dla ustalonej dopuszczalnej kolejności wykonywania zadań π, wartość kryterium w problemie $J||C_{max}$ można wyznaczyć w czasie $O(\log o \log \Lambda)$ na $O(o^3/\log o)$ procesorowej maszynie CREW PRAM, gdzie Λ jest górnym ograniczeniem wartości funkcji celu.*

Dowód Dowód opiera się na konstrukcji dowodu Tw. 9. Główną pętlę wystarczy powtarzać $\lceil \log \Lambda \rceil$ zamiast $\lceil \log o \rceil$ razy, ponieważ maksymalna liczba wierzchołków na drodze krytycznej jest nie większa niż Λ/p_{min}, gdzie p_{min} jest najkrótszym czasem wykonywania operacji spośród zbioru wszystkich operacji O. Wartości p_{min} są całkowitoliczbowe, minimalna wartość wynosi 1, stąd maksymalna liczba wierzchołków na drodze krytycznej wynosi Λ. Aby wyznaczyć długość ścieżki krytycznej wystarczy wykonać $\lceil \log \Lambda \rceil$ równoległych kroków (porównaj z Tw. 7) co zmniejsza złożoność obliczeniową całej metody do $O(\log o \log \Lambda)$ zachowując liczbę procesorów $O(o^3/\log o)$.■

Wniosek 10 Dla metody bazującej na Tw. 10 mamy

$$s_{A_p,M}(p) = O(\frac{o}{\log o \log \Lambda}),\ \eta_{A_p,M}(p) = \frac{s_{A_p,M}(p)}{p} = O(\frac{1}{o^2 \log \Lambda}).\ \square \quad (6.24)$$

Gdy znana jest wartość górnego ograniczenia funkcji kryterialnej Λ, wówczas wartość $\log \Lambda$ jest stała i nie ma wpływu na złożoność obliczeniową. W przypadku jednak gdy nie wiadomo nic o wartości funkcji celu, algorytm równoległy musi wykonać $\lceil \log o \rceil$ zamiast $\lceil \log \Lambda \rceil$ iteracji głównej pętli, co oznacza, że poprawę rzędu wielkości złożoności obliczeniowej można otrzymać tylko w przypadkach, które były już w jakiś sposób badane (rozwiązywane). W sytuacji ogólnej, mając do czynienia z problemem zupełnie nieznanym, takiej poprawy rzędu oszacowania nie można zakładać. Konieczne jest wtedy albo zastosowanie metody z twierdzenia poprzedniego, o złożoności $O(\log^2 o)$, albo dokonanie wstępnego oszacowania wartości Λ. Pierwsze podejście polega na zauważeniu, że metoda badająca wartość kryterium $\sum C_i$ umożliwia wyznaczenie wartość $C_{\max}$ według tego samego schematu postępowania (wyznaczając jedną, najdłuższą ścieżkę krytyczną), zachowując tę samą złożoność obliczeniową i liczbę procesorów. Drugie podejście polega na wstępnym oszacowaniu wartości Λ, na przykład poprzez skonstruowanie losowego rozwiązania dopuszczalnego, bądź stosując prosty algorytm priorytetowy, lub też jeszcze inną metodą. Trzeba mieć przy tym świadomość, że dla problemu gniazdowego rozwiązanie losowe ma średni błąd względny rzędu 130%, zaś konwencjonalne algorytmy priorytetowe wyznaczają wartości funkcji celu z błędem rzędu 30% względem wartości optymalnej.

Na podstawie badań nad znanymi przykładami testowymi dla problemu gniazdowego (Fisher i Thomson [138], Lawrence [112], Yamada i Nakano [189], Storer i inni [169]) w pracy [168] wysunięto przypuszczenie, że dla rozwiązań optymalnych wartość funkcji celu Λ spełnia zależność $\Lambda \leqslant p_{max}(n+m)$, gdzie p_{max} jest najdłuższym czasem wykonywania operacji spośród zbioru wszystkich operacji O, n jest liczbą zadań a m liczbą gniazd. Może to oznaczać, że wartość Λ jest rzędu $O(n+m)$, czyli zamieniając złożoność obliczeniową metody równolegle wyznaczającej wartość funkcji celu z $O(\log^2 o)$, opisanej w twierdzeniu poprzednim, na $O(\log o \log \Lambda)$ redukuje się złożoność obliczeniową z $O(\log^2(nm))$ do $O(\log(nm)\log(n+m))$.

6.1.4 Problem przepływowy hybrydowy

Podejście stosowane dla hybrydowego problemu przepływowego jest zasadniczo takie samo jak dla problemu gniazdowego z tym, że wszystkie rozwiązania π są dopuszczalne.

Twierdzenie 11 *Dla ustalonej kolejności wykonywania zadań π, wartość kryterium w problemie $FP||\sum f_i$ można wyznaczyć w czasie $O(\log^2(mn))$ na $O((mn)^3/\log(mn))$ procesorowej maszynie CREW PRAM.*

Dowód Stosujemy schemat postępowania analogiczny, do opisanego w dowodzie Tw. 7. Także tu odwołujemy się do algorytmu Floyda-Warshalla z tym, że obliczenia przeprowadzamy dla grafu $G(\pi)$ skojarzonego z rozwiązaniem dopuszczalnym problemu gniazdowego, zobacz Rozdz. 2.4.

Złożoność obliczeniowa wykonania kroku 1 wynosi $O(\log(mn))$ na $\lceil(mn)^3/\log(mn)\rceil$ procesorach. Krok 2 polega na wyznaczeniu maksimum $mn+1$ wartości, co można wykonać na $O((mn)/\log(mn))$ procesorach w czasie $O(\log(mn))$ (patrz Fakt 3). Ponieważ maksimum takie należy wyznaczyć dla $(mn)^2$ par (u,v), a obliczenia te są niezależne i należy je powtórzyć $\lceil\log(mn)\rceil$ - krotnie, więc za pomocą $O((mn)^3/\log(mn))$ procesorów cała opisana procedura ma złożoność $O(\log^2(mn))$.

W końcu, obliczenie wartości kryterium (suma n liczb) można zrobić w czasie $O(\log n)$ za pomocą $O(n/\log n)$ procesorów (patrz Fakt 2) co zachowuje górne ograniczenie złożoności obliczeniowej $O(\log^2(mn))$ oraz liczbę użytych procesorów $O((mn)^3/\log(mn))$ całej metody.■

Wniosek 11 Dla metody bazującej na Tw. 11 mamy

$$s_{A_p,M}(p) = O(\frac{mn}{\log^2(mn)}), \quad \eta_{A_p,M}(p) = O(\frac{1}{(mn)^2\log(mn)}). \square \qquad (6.25)$$

Metoda równoległa nie jest więc kosztowo optymalna, ale złożoność obliczeniowa $O(\log^2(mn))$ daje zysk czasowy względem sekwencyjnej złożoności $O(mn)$. Przyspieszenie metody silnie wzrasta wraz ze wzrostem iloczynu mn, patrz także Tab. 6.1 dla $o = mn$.

Kolejne twierdzenie odnosi się do tego samego problemu przepływowego hybrydowego, lecz z kryterium C_{max}. Podobnie jak dla problemu gniazdowego z tym samym kryterium, możliwe jest zredukowanie złożoności obliczeniowej $O(\log^2(mn))$ do $O(\log(mn)\log\Lambda)$.

Twierdzenie 12 *Dla ustalonej kolejności wykonywania zadań π, wartość kryterium w problemie $FP||C_{max}$ można wyznaczyć w czasie $O(\log(mn)\log\Lambda)$ na $O((mn)^3/\log(mn))$ procesorowej maszynie CREW PRAM.*

Aby wykazać Tw. 12 należy zastosować metodę zaprezentowaną w dowodzie twierdzenia poprzedniego. Powtarzając główną pętlę $\lceil\log\Lambda\rceil$ zamiast $\lceil\log(mn)\rceil$ razy, zmniejsza się złożoność obliczeniową całej metody do $O(\log(mn)\log\Lambda)$ zachowując liczbę procesorów rzędu $O((mn)^3/\log(mn))$.

Można dokonać takiej redukcji, ponieważ maksymalna liczba wierzchołków na drodze krytycznej jest nie większa niż Λ/p_{min}, gdzie p_{min} jest najkrótszym czasem wykonywania zadania ze zbioru wszystkich zadań N. Wartości p_{min} są całkowitoliczbowe, minimalna wartość wynosi 1, stąd maksymalna liczba wierzchołków na drodze krytycznej wynosi Λ. Aby wyznaczyć długość ścieżki krytycznej wystarczy wykonać $\lceil \log \Lambda \rceil$ równoległych kroków (uzasadnienie tego faktu zostało przedstawione w dowodzie Tw. 7).

Wniosek 12 Dla metody bazującej na Tw. 11 mamy

$$s_{A_p,M}(p) = O(\frac{mn}{\log(mn)\log\Lambda}), \quad \eta_{A_p,M}(p) = O(\frac{1}{(mn)^2\log\Lambda}). \square \qquad (6.26)$$

Tw. 12 jest w pełni analogiczne do Tw. 10 (przy założeniu $o = mn$), zatem wnioski tam wyciągnięte pozostają w mocy także dla przedstawionej metody.

6.1.5 Wnioski i uwagi

Przyspieszenia, złożoności obliczeniowe oraz ilości użytych procesorów zaproponowanych metod równoległego wyznaczania wartości funkcji kryterialnej dla wybranych problemów szeregowania zadań produkcyjnych są przedstawione w Tab. C.1 oraz C.2, znajdujących się w Dodatku C.

W grupie metod równoległych dedykowanych dla problemów szeregowania metody oparte na analizie grafu siatkowego jako pewnego układu kombinacyjnego o strukturze drzewiastej prowadzą do algorytmów kosztowo optymalnych. Mogą one być stosowane przy względnie niedużej (aktualnie technicznie dostępnej) liczbie procesorów. Metoda podana w Tw. 6 oraz podejścia oparte na algorytmie sekwencyjnym Floyda-Warshalla prowadzą do metod o niskiej efektywności, lecz bardzo szybkich. Wyniki te można uważać za pierwsze przybliżenie granicznej wartości przyspieszenia przy dowolnej dostępnej liczbie procesorów(dążącej do nieskończoności).

Część wyników zaprezentowanych w tym rozdziale można rozszerzyć na model EREW PRAM z wyłącznością odczytu. Wystarczy zauważyć, że głównym problemem jest wykorzystywanie w procesie dowodowym Faktów 5 oraz 6, odwołujących się do modelu CREW, a wykorzystywanych do obliczania wartości funkcji celu. Jeśli zapewnimy sobie wyłączność odczytu przez procesory takich wielkości jak czasy wykonywania zadań, żądane czasy zakończenia, itp., kopiując je n – krotnie, wówczas możliwe będzie sformułowanie odpowiednich twierdzeń dla modelu EREW. Operacja kopiowania wymaga czasu $O(\log n)$ oraz $O(n/\log n)$ procesorów i może być wykonana w fazie wstępnej.

Powyższa uwaga pozwala rozszerzyć na model EREW PRAM wyniki Tw. 1-2, 4 i 5 dotyczących wyznaczania wartości funkcji celu dla problemów jednomaszynowych $1||\sum f_i$, $1||\sum f_{max}$, $1|r_j|\sum f_i$, oraz przepływowych $F^*||\sum f_i$ i $F^*||f_{max}$.

6.2 Analiza zbioru rozwiązań sąsiednich

Znaczna część algorytmów poszukiwań lokalnych dokonuje analizy zbioru rozwiązań sąsiednich (bliskich) względem pewnego ustalonego rozwiązania. Zbiór rozwiązań sąsiednich nazywany jest krótko otoczeniem. Celem analizy jest *przeglądnięcie otoczenia*, tzn. wyznaczenie rozwiązania najlepszego w sensie wartości funkcji kryterium. W tym rozdziale przedstawimy różne metody równoległego przeglądania otoczeń.

6.2.1 Problem jednomaszynowy. Otoczenie API

Niech π będzie permutacją n – elementową reprezentującą rozwiązanie problemu $1||\sum f_i$. Wykonując zamianę $v = (a, a{+}1)$ pary przyległych zadań w permutacji π wartość funkcji celu otrzymanej (po zamianie) permutacji $\pi_{(v)}$ można wyznaczyć w czasie $O(1)$ poprzez zmodyfikowanie wartości funkcji celu $F(\pi)$ rozwiązania π generującego otoczenie, według wzoru

$$F(\pi_{(v)}) = F(\pi) - f_{\pi(a)}(x - p_{\pi(b)}) - f_{\pi(b)}(x) + f_{\pi(b)}(x - p_{\pi(a)}) + f_{\pi(a)}(x) \quad (6.27)$$

gdzie $x = C_{\pi(b)}$ jest terminem zakończenia wykonywania zadania $\pi(b)$, $b = a + 1$ w permutacji π.

Zgodnie z powyższym wzorem, całe otoczenie API można przeglądnąć sekwencyjnie w czasie $O(n)$.

Twierdzenie 13 *Pełne otoczenie API dla problemu $1||\sum f_i$ można przeglądnąć w czasie $O(\log n)$ na maszynie CREW PRAM za pomocą $O(n)$ procesorów.*

Dowód Stosujemy $n - 1 = O(n)$ procesorów przydzielając każdemu wyliczenie w czasie $O(1)$ wartości $F(\pi_{(v)})$ według (6.27). Problemem pozostaje wybranie najlepszego ruchu – co wymaga przynajmniej $O(\log n)$ czasu i $O(n/\log n)$ procesorów (Fakt 3). Podobnie, $O(\log n)$ czasu i $O(n/\log n)$ procesorów wymaga wyliczenie wartości funkcji kryterialnej $F(\pi)$ rozwiązania π generującego otoczenie (Tw. 1). Cały proces ma więc złożoność $O(\log n)$ i wymaga użycia $O(\max\{n, n/\log n\}) = O(n)$ procesorów.■

Wniosek 13 Dla metody bazującej na Tw. 13 mamy

$$s_{A_p,M}(p) = O(\frac{n}{\log n}), \quad \eta_{A_p,M}(p) = O(\frac{1}{\log n}). \square \tag{6.28}$$

Efektywność metody spada stosunkowo wolno wraz ze wzrostem n. Dla $n = 100$ przyspieszenie wynosi 15.1, zaś efektywność 15%. Dla $n = 1000$ przyspieszenie jest 100, efektywność 10% [7].

Poniżej zostanie zaprezentowana metoda, pozwalająca przeglądnąć otoczenie API w problemie $1||\sum f_i$ w czasie identycznym jak w twierdzeniu poprzednim, lecz używająca mniejszej liczby procesorów, a mianowicie $O(n/\log n)$. Metoda ta będzie, w odróżnieniu od zaprezentowanej w Tw. 13, kosztowo optymalna.

Twierdzenie 14 *Pełne otoczenie API dla problemu $1||\sum f_i$ można przeglądnąć w czasie $O(\log n)$ na maszynie CREW PRAM za pomocą $O(n/\log n)$ procesorów.*

Dowód Przeglądanie otoczenia można podzielić na grupy, po $\left\lceil \frac{n}{p} \right\rceil$ zamian pozycji w permutacji π w każdej grupie, gdzie $p = \left\lceil \frac{n}{\log n} \right\rceil$ jest liczbą użytych procesorów. Obliczenia wartości funkcji kryterialnej w każdej grupie są od siebie niezależne. Każdy procesor $k = 1,2,...,p$ będzie przeglądał część otoczenia otrzymaną z wymiany par przyległych

$$v = ((k\text{-}1)\left\lceil \tfrac{n}{p} \right\rceil + a,\ (k\text{-}1)\left\lceil \tfrac{n}{p} \right\rceil + a{+}1),\ a = 1,2,...,\ \left\lceil \tfrac{n}{p} \right\rceil$$

dla $k = 1, 2, ..., p{-}1$, oraz

$$v = ((k\text{-}1)\left\lceil \tfrac{n}{p} \right\rceil + a,\ (k\text{-}1)\left\lceil \tfrac{n}{p} \right\rceil + a{+}1),\ a = 1,2,...,\ n - (p{-}1)\left\lceil \tfrac{n}{p} \right\rceil - 1$$

dla $k = p$. Ostatnia grupa może pozostać niepełna. Korzystając z własności (6.27) można wyliczyć wartość $F(\pi_{(v)})$ na podstawie $F(\pi)$ dla pojedynczej wymiany par przyległych $v = (a,a{+}1)$ w czasie $O(1)$. Wstępne obliczenie $F(\pi)$ można przeprowadzić w czasie $O(\log n)$ za pomocą $O(n/\log n)$ procesorów (Tw. 1). Ponieważ proces wyznaczenia wszystkich wartości $F(\pi_{(v)})$ w pojedynczej grupie ma złożoność

$$\left\lceil \frac{n}{p} \right\rceil O(1) = O(\frac{n}{p}) = O(\frac{n}{\frac{n}{\log n}}) = O(\log n) \tag{6.29}$$

[7] oszacowanie prawdziwe z dokładnością do stałego mnożnika

więc taka też będzie złożoność obliczeniowa wyznaczenia wszystkich wartości $F(\pi_{(v)})$ dla wszystkich ruchów v. Każdy procesor, obliczając sekwencyjnie swoją porcję wartości $F(\pi_{(v)})$, może pamiętać wartość najlepszą – w tym celu należy dodatkowo wykonać liczbę porównań równą liczbie elementów w grupie pomniejszoną o 1, czyli

$$\left\lceil \frac{n}{p} \right\rceil - 1 = O(\frac{n}{p}) = O(\frac{n}{\frac{n}{\log n}}) = O(\log n) \tag{6.30}$$

porównań co zachowuje rząd złożoności $O(\log n)$. W celu wyznaczenia najlepszego ruchu z całego otoczenia API należy wyznaczyć element minimalny z $p = \left\lceil \frac{n}{\log n} \right\rceil$ najlepszych wartości funkcji kryterialnej pamiętanych dla każdej grupy, co zgodnie z Faktem 3 można zrobić w czasie $O(\log n)$, za pomocą p procesorów, ponieważ $p \leqslant n$. Wobec tego złożoność obliczeniowa całej metody wynosi $O(\log n)$ a liczba użytych procesorów

$$p = \left\lceil \frac{n}{\log n} \right\rceil = O(\frac{n}{\log n}).\blacksquare \tag{6.31}$$

Wniosek 14 Dla metody bazującej na Tw. 14 mamy

$$s_{A_p,M}(p) = O(\frac{n}{\log n}), \quad \eta_{A_p,M}(p) = O(1). \ \square \tag{6.32}$$

Metoda ta jest kosztowo optymalna, przy przyspieszeniu takim samym jak w Tw. 13. Zauważmy, że wymagania dotyczące liczby procesorów mogą być w praktyce trudne do spełnienia, np. dla $n = 1000$ metoda z Tw. 13 wymaga 1000 procesorów, metoda z Tw. 14 około 100 procesorów [8]. W każdym z wymienionych przypadków możliwe jest dostosowanie metody do realnej liczby procesorów w systemie obliczeniowym poprzez zastosowanie Faktu 7.

6.2.2 Problem jednomaszynowy. Otoczenie INS

Złożoność obliczeniowa przeglądania otoczenia typu INS dla problemu jednomaszynowego $1||\sum f_i$, bez wykorzystania podobieństwa poszczególnych elementów otoczenia, wynosi $O(n^3)$. Każdy ruch typu *insert* polegający na wyjęciu zadania z pewnej pozycji w permutacji π i wstawieniu go na inną pozycję w tej permutacji, wyrazić można jako złożenie ruchów typu *wymiana par przyległych*. Fakt ten pozwala zmniejszyć złożoność obliczeniową procesu przeglądania całego otoczenia INS do $O(n^2)$ poprzez generowanie

[8] oszacowanie prawdziwe z dokładnością do stałego mnożnika

elementów otoczenia kolejnymi wymianami par przyległych i stosowanie wzoru (6.27).

Twierdzenie 15 *Pełne otoczenie INS dla problemu* $1||\sum f_i$ *można przeglądnąć w czasie* $O(n)$ *za pomocą* $O(n)$ *procesorów na maszynie CREW PRAM.*

Dowód Przeglądanie otoczenia można podzielić na grupy, po n zamian pozycji w permutacji π w każdej grupie, gdzie $p = n$ jest ilością użytych procesorów. Obliczenia wartości funkcji kryterialnej w każdej grupie są od siebie niezależne. Każdy procesor $k = 1,2,...,n$ będzie przeglądał część otoczenia otrzymaną z wymiany par $v = (k, b)$, $b = 1,2,...,n$, $b \neq k$. Korzystając z (6.27) można wyliczyć wartość $F(\pi_{(v)})$ na podstawie $F(\pi)$ dla pojedynczej wymiany par przyległych $v = (a,a+1)$ w czasie $O(1)$. Każdą wymianę (k,b) można wyrazić jako ciąg wymian par przyległych

$$(k,k+1),\ (k+1,k+2),...(b-2,b-1),\ (b-1,b)$$

jeśli $k < b$, lub

$$(b,b+1),\ (b+1,b+2),...(k-2,k-1),\ (k-1,k)$$

jeśli $k > b$. Ruch $v = (k, b)$, $k = b$ pomijamy. Wobec tego, dla danego k, wykonujemy najpierw wszystkie wymiany par przyległych „na lewo" od k, tzn. $(k-1,k)$, $(k-2,k-1)$,..., (2,3), (1,2), a następnie „na prawo" od k, czyli $(k,\ k+1)$, $(k+1,\ k+2)$,..., $(n-2,n-1)$, $(n-1,n)$. W ten sposób wygenerujemy wszystkie wymiany $v = (k, b)$, $b = 1,2,...,n$, $b \neq k$, a dla każdej z nich, według wzoru (6.27), możemy wyliczyć wartość funkcji $F(\pi_{(v)})$ w czasie $O(1)$. Wstępne obliczenie $F(\pi)$ można przeprowadzić w czasie $O(\log n)$ za pomocą $O(n/\log n)$ procesorów (patrz Tw. 1). Ponieważ proces wyznaczenia wszystkich wartości $F(\pi_{(v)})$ w pojedynczej grupie ma złożoność $O(n)$, a procesorów jest n i każdy przegląda swoją część otoczenia niezależnie, więc złożoność obliczeniowa wyznaczenia wszystkich wartości $F(\pi_{(v)})$ dla wszystkich ruchów v będzie wynosiła także $O(n)$. Każdy procesor, obliczając sekwencyjnie swoją porcję wartości $F(\pi_{(v)})$, może pamiętać wartość najlepszą – w tym celu należy dodatkowo wykonać liczbę porównań równą liczbie elementów w grupie pomniejszoną o 1, czyli $n - 1$. porównań, co zachowuje rząd złożoności $O(n)$. W celu wyznaczenia najlepszego ruchu z całego otoczenia INS należy wyznaczyć element minimalny z $p = n$ najlepszych wartości funkcji kryterialnej pamiętanych dla każdej grupy, co zgodnie z Faktem 3 można zrobić w czasie $O(\log n)$, za pomocą n procesorów. Wobec

tego złożoność obliczeniowa całej metody wynosi $O(\max\{\log n, n\}) = O(n)$, a liczba użytych procesorów $n = O(n)$.■

Wniosek 15 Dla metody bazującej na Tw. 15 mamy

$$s_{A_p,M}(p) = O(\frac{n^2}{n}) = O(n), \quad \eta_{A_p,M}(p) = \frac{s_{A_p,M}(p)}{p} = O(1). \square \tag{6.33}$$

Metoda jest więc kosztowo optymalna, ze znacznym n – krotnym przyspieszeniem.

Twierdzenie 16 *Pełne otoczenie INS dla problemu $1||\sum f_i$ można przeglądnąć w czasie $O(\log n)$ za pomocą $O(n^3/\log n)$ procesorów na maszynie CREW PRAM.*

Dowód Każdemu z $n(n-1)$ elementów otoczenia INS przydzielamy pulę $O(n/\log n)$ procesorów, za pomocą której wyliczana jest wartość funkcji kryterialnej $\sum f_i$ dla każdego elementu otoczenia w czasie $O(\log n)$ (mowa jest o tym w Tw. 1). W celu wskazania najlepszego ruchu z całego otoczenia INS należy wyznaczyć element minimalny z $n(n-1) < n^2$ najlepszych wartości funkcji kryterialnej pamiętanych dla każdej grupy, co można zrobić w czasie $O(\log n^2) = O(2\log n) = O(\log n)$, za pomocą n^2 procesorów (Fakt 3). Wobec tego złożoność obliczeniowa całej metody wynosi $O(\log n)$, a liczba użytych procesorów $n^2 O(n/\log n) = O(n^3/\log n)$.■

Wniosek 16 Dla metody bazującej na Tw. 16 mamy

$$s_{A_p,M}(p) = O(\frac{n^2}{\log n}), \quad \eta_{A_p,M}(p) = O(\frac{1}{n}). \square \tag{6.34}$$

Metoda nie jest więc kosztowo optymalną, ale dysponując odpowiednio dużą liczbą procesorów pozwala przeglądać otoczenie INS w czasie logarytmicznym. Dla $n = 100$ przyspieszenie jest ponad 1500 razy przy efektywności około 15% [9]. Następne twierdzenie pokazuje, że można to zrobić za pomocą mniejszej liczby procesorów $O(n^2/\log n)$ i jest to metoda kosztowo optymalna.

Twierdzenie 17 *Pełne otoczenie INS dla problemu $1||\sum f_i$ można przeglądnąć w czasie $O(\log n)$ za pomocą $O(n^2/\log n)$ procesorów na maszynie CREW PRAM.*

[9] oszacowanie prawdziwe z dokładnością do stałego mnożnika

Dowód Przeglądanie otoczenia można podzielić na grupy, przydzielając każdej zamianie $v = (a, b)$, dla każdego a = 1,2,..., n, pulę $p = \left\lceil \frac{n-1}{\log n} \right\rceil$ procesorów. Obliczenia wartości funkcji kryterialnej w każdej grupie są od siebie niezależne, toteż skupmy się na obliczeniu wartości funkcji kryterialnych dla pewnej grupy związanej z ustalonym a. Ilość elementów w tak ustalonej części otoczenia wynosi $n-1$. Proces przeglądania takiego fragmentu można podzielić na grupy, po $\left\lceil \frac{n-1}{p} \right\rceil$ zamian pozycji w permutacji π w każdej grupie, gdzie $p = \left\lceil \frac{n-1}{\log n} \right\rceil$ jest liczbą użytych procesorów. Każdy procesor k = 1,2,...,p będzie przeglądał część otoczenia związaną z ruchami

$$v = (a, (k-1)\left\lceil \frac{n-1}{p} \right\rceil + s),\ s = 1, 2, ..., \left\lceil \frac{n-1}{p} \right\rceil \tag{6.35}$$

dla $k = 1, 2, ..., p-1$, oraz

$$v = (a, (k-1)\left\lceil \frac{n-1}{p} \right\rceil + s),\ s = 1, 2, ..., n-(p-1)\left\lceil \frac{n-1}{p} \right\rceil \tag{6.36}$$

dla $k = p$. Ostatnia grupa pozostanie niepełna. Jeśli w pierwszym kroku w każdej grupie policzymy wartość funkcji celu dla pierwszego ruchu z grupy, czyli

$$v = (a, (k-1)\left\lceil \frac{n-1}{p} \right\rceil + 1) \quad \text{dla } k = 1, 2, \ldots, p, \tag{6.37}$$

to korzystając z własności (6.27) można wygenerować w czasie $O(1)$ wartość funkcji celu dla każdego elementu grupy wykonując kolejne wymiany par przyległych (patrz dowód Tw. 15). Wygenerowanie wartości funkcji celu pierwszego elementu grupy wykonujemy w czasie $O(\log n)$ za pomocą $O(\frac{n}{\log n})$procesorów przypisanych grupie (Tw. 1). W grupie należ wykonać $\left\lceil \frac{n-1}{p} \right\rceil$ zamian par przyległych, co daje złożoność czasową $\left\lceil \frac{n-1}{p} \right\rceil O(1)$, co z kolei na podstawie ciągu nierówności

$$\left\lceil \frac{n-1}{p} \right\rceil < \frac{n-1}{p} + 1 = \frac{n-1}{\left\lceil \frac{n-1}{\log n} \right\rceil} + 1 < \frac{n-1}{\frac{n-1}{\log n}} + 1 = \log n + 1 \tag{6.38}$$

daje

$$\left\lceil \frac{n-1}{p} \right\rceil O(1) = O(\log n + 1) = O(\log n). \tag{6.39}$$

Taka też będzie złożoność obliczeniowa wyznaczenia wszystkich wartości $F(\pi_{(v)})$ dla wszystkich ruchów v. Każdy procesor, obliczając sekwencyjnie

swoją porcję wartości $F(\pi_{(v)})$, może pamiętać wartość najlepszą – w tym celu należy dodatkowo wykonać liczbę porównań równą liczbie elementów w grupie pomniejszoną o 1, czyli

$$\left\lceil \frac{n-1}{p} \right\rceil - 1 = \quad O(\frac{n}{p}) = O(\frac{n}{\frac{n}{\log n}}) = O(\log n) \tag{6.40}$$

porównań co zachowuje rząd złożoności $O(\log n)$. W celu wyznaczenia najlepszego ruchu z całego otoczenia INS należy wyznaczyć element minimalny z $np = n \left\lceil \frac{n-1}{\log n} \right\rceil$ najlepszych wartości funkcji kryterialnej pamiętanych dla każdej grupy dla każdego a= 1,2,...,n, co zgodnie z Faktem 3 można zrobić w czasie $O(\log n^2) = O(2 \log n) = O(\log n)$, za pomocą np procesorów, ponieważ $np \leqslant n^2$. Wobec tego złożoność obliczeniowa całej metody wynosi $O(\log n)$, a liczba użytych procesorów

$$O(\max\{\left\lceil \frac{n-1}{\log n} \right\rceil, \frac{n}{\log n}\}) = O(\frac{n}{\log n}) \tag{6.41}$$

dla każdej z n grup przypisanych do ustalonego a = 1,2,...,n, oraz np = $O(\frac{n^2}{\log n})$ procesorów użytych przy wyznaczaniu elementu minimalnego z wszystkich grup, co daje ostatecznie $O(\frac{n^2}{\log n})$ procesorów.■

Wniosek 17 Dla metody bazującej na Tw. 17 mamy

$$s_{A_p,M}(p) = O(\frac{n^2}{\log n}), \quad \eta_{A_p,M}(p) = \frac{s_{A_p,M}(p)}{p} = O(1). \ \square \tag{6.42}$$

Zaprezentowana metoda jest kosztowo optymalna, przy takim samym przyspieszeniu jak w Tw. 16.

6.2.3 Problem jednomaszynowy. Otoczenie NPI

Dla otoczenia NPI w problemie jednomaszynowym $1||\sum f_i$ nie są znane sekwencyjne metody przyspieszania procesu przeglądania otoczenia. Standardowymi metodami należy więc wyznaczyć (w czasie $O(n)$) wartości funkcji celu wszystkich elementów otocznia, a jest ich $\frac{n(n-1)}{2}$, po czym wskazać element o minimalnej wartości. Wobec tego, złożoność obliczeniowa sekwencyjnego przeglądania otoczenia typu NPI dla problemu $1||\sum f_i$ wynosi $O(n^3)$.

Jak zostanie pokazane poniżej, rozkładając proces przeglądania otoczenia na równoległe procesory maszyny PRAM, możliwe jest znaczne przyspieszenie procesu przeglądnięcie całego otoczenia NPI, a mianowicie wykonanie przeglądu w czasie $O(\log n)$, przy zachowaniu kosztowej optymalności metody.

Twierdzenie 18 *Pełne otoczenie NPI dla problemu* $1||\sum f_i$ *można przeglądnąć w czasie* $O(\log n)$ *za pomocą* $O(n^3/\log n)$ *procesorów na maszynie CREW PRAM.*

Dowód Do każdego z $\frac{n(n-1)}{2}$elementów otoczenia NPI przydzielamy pulę $O(n/\log n)$ procesorów, za pomocą których każdy procesor wylicza wartość funkcji kryterialnej $\sum f_i$ w czasie $O(\log n)$ (na podstawie Tw. 1). W celu wyznaczenia najlepszego ruchu z całego otoczenia NPI należy wyznaczyć element minimalny z $\frac{n(n-1)}{2} < n^2$ najlepszych wartości funkcji kryterialnej pamiętanych dla każdej grupy, co zgodnie z Faktem 3 można zrobić w czasie $O(\log n^2) = O(2\log n) = O(\log n)$, za pomocą n^2 procesorów. Wobec tego złożoność obliczeniowa całej metody wynosi $O(\log n)$, a liczba użytych procesorów $n^2 O(n/\log n) = O(n^3/\log n)$.■

Wniosek 18 Dla metody bazującej na Tw. 18 mamy

$$s_{A_p,M}(p) = O(\frac{n^3}{\log n}), \quad \eta_{A_p,M}(p) = O(1). \square \tag{6.43}$$

Metoda jest kosztowo optymalna. Dla $n = 100$ oferowane przyspieszenie jest 150,000 [10]. Niestety, taka też jest liczba wymaganych procesorów, co ogranicza praktyczne zastosowanie metody. Wskazane jest zastosowanie w tym przypadku Faktu 7.

6.2.4 Problem przepływowy. Otoczenie API

Otoczenie API jest jednym z najprostszych i najczęściej używanych. Przyspieszanie sekwencyjnych algorytmów przeglądania otoczenia API dla problemów przepływowych dotyczy tylko kryterium C_{max}. Realizuje je tak zwany (sekwencyjny) akcelerator, dokonujący odpowiedniej dekompozycji i agregacji obliczeń, korzystając z pokrewieństwa rozwiązań, patrz np. [170]. Ponieważ niektóre z twierdzeń udowadnianych w tym rozdziale bazują na konstrukcji akceleratora, dlatego przedstawimy go poniżej.

Niech π będzie pewną permutacją generującą otoczenie API, a $v = (a, a+1)$ parą przyległych pozycji, których wymiana w permutacji π prowadzi do wygenerowania nowego rozwiązania $\pi_{(v)}$. Dla permutacji π wyznaczamy

$$r_{st} = \max\{r_{s-1,t}, r_{s,t-1} + p_{s\pi(t)}\} \tag{6.44}$$

dla $t = 1, 2, \ldots, a-1$, $s = 1, 2, \ldots, m$, oraz

$$q_{st} = \max\{q_{s+1,t}, q_{s,t+1} + p_{s\pi(t)}\} \tag{6.45}$$

[10] oszacowanie prawdziwe z dokładnością do stałego mnożnika

dla $t = a-1, a-2, \ldots, 1$, $s = m, m-1, \ldots, 1$, gdzie $r_{0t}= 0 = q_{jt}$, t =1,2,...,n, $r_{s0}= 0 = q_{s,m+1}$, s =1,2,...,m. Wartość r_{st} jest długością najdłuższej drogi w grafie $G(\pi)$ dochodzącej do wierzchołka (s,t), wraz z obciążeniem tego wierzchołka, natomiast q_{st} jest długością najdłuższej drogi w grafie $G(\pi)$ wychodzącej z wierzchołka (s,t), wraz z obciążeniem tego wierzchołka. Graf siatkowy $G(\pi)$ opisano w Rozdz. 2.2. Obciążenie węzła (s,t) wynosi $p_{s,\pi(t)}$. Następnie każda wartość $C_{max}(\pi_{(v)})$ dla pojedynczej wymiany pary przyległych zadań $v = (a, a+1)$ może być znaleziona w czasie $O(m)$ ze wzorów

$$C_{\max}(\pi_{(v)}) = \max_{1 \leqslant s \leqslant m} (d'_s + q_{s,a+2}), \tag{6.46}$$

gdzie

$$d'_s = \max\{d'_{s-1}, d_s\} + p_{s,\pi(a)}, \quad s = 1, 2, ..., m, \tag{6.47}$$

reprezentuje długość najdłuższej drogi dochodzącej do wierzchołka $(s,a+1)$ w grafie $G(\pi)$ oraz

$$d_s = \max\{d_{s-1}, r_{s,a-1}\} + p_{s\pi(a+1)}, \quad s = 1, 2, ..., m \tag{6.48}$$

reprezentuje długość najdłuższej drogi dochodzącej do wierzchołka (s,a) w grafie $G(\pi_{(v)})$. Odpowiednie warunki początkowe są następujące: $d'_0 = d_0 = 0$, $r_{s0} = 0 = q_{s,n+2}$, $s = 1, 2, \ldots, m$. Otoczenie API zawiera $n-1$ rozwiązań sąsiednich.

Przeglądanie całego otoczenia API, dla danej permutacji π, ma na celu wyznaczenie jednej permutacji spośród $\pi_{(v)}, v = (a, a+1), a = 1, 2, \ldots, n-1$ takiej, dla której wartość kryterium jest minimalna. Dla problemu $F^*||C_{max}$ proces ten ma złożoność $O(n^2m)$, natomiast stosując opisany powyżej akcelerator sekwencyjny uzyskać można złożoność $O(nm)$.

Twierdzenie 19 *Pełne otoczenie API w problemie $F^*||\gamma$, $\gamma \in \{f_{max}, \sum f_i\}$ można przeglądnąć w czasie $O(n+m)$ na maszynie CREW PRAM za pomocą $O(\frac{n^2m}{n+m})$ procesorów.*

Dowód Nie bacząc na pokrewieństwo, przydzielamy każdemu rozwiązaniu z otoczenia $O(\frac{nm}{n+m})$ procesorów, co pozwala policzyć wartość kryterialną pojedynczego rozwiązania w czasie $O(n+m)$, patrz Tw. 4. Pozostaje wybrać minimalną wartość spośród $n-1$ obliczonych wartości funkcji kryterialnych. Można to zrobić na przykład sekwencyjnie w czasie $O(n)$ (lub w czasie logarytmicznym za pomocą $O(n/\log n)$ procesorów), co zachowuje złożoność $O(n+m)$, a liczba użytych procesorów wynosi $(n-1)O(\frac{nm}{n+m}) = O(\frac{n^2m}{n+m})$.■

Powstaje dylemat, do jakiej wersji algorytmu sekwencyjnego należy odnieść otrzymany algorytm równoległy. Z jednej strony, realizując $n-1$ razy obliczanie pojedynczej wartości funkcji celu $F^*||C_{max}$ otrzymujemy złożoność przeglądania otoczenia API $O(n^2m)$. Z drugiej strony, akcelerator API ma złożoność $O(nm)$. Zatem, ocena „dobroci" metody z Tw. 19 dla problemu $F^*||C_{max}$ może być dokonywana tylko w stosunku do najlepszego algorytmu, tzn. akceleratora. Stąd otrzymujemy następujące wnioski.

Wniosek 19 Przyspieszenie metody bazującej na Tw. 19 dla problemu $F^*||C_{max}$ wynosi

$$s_{A_p,M}(p) = O(\frac{nm}{n+m}), \tag{6.49}$$

a efektywność

$$\eta_{A_p,M}(p) = \frac{s_{A_p,M}(p)}{p} = O\left(\frac{\frac{nm}{n+m}}{\frac{n^2m}{n+m}}\right) = O(\frac{1}{n}). \ \square \tag{6.50}$$

Zaprezentowana metoda dla $F^*||C_{max}$ nie jest więc kosztowo optymalna. Jej efektywność pogarsza się szybko ze wzrostem n. Zauważmy, że algorytm przedstawiony w dowodzie Tw. 19, a odnoszący się do problemu $F^*||\sum C_i$ oraz otoczenia API, prowadzi do metody kosztowo optymalnej z efektywnością $O(1)$, bowiem dla tego problemu akcelerator API nie ma zastosowania.

Kolejne twierdzenie, w odróżnieniu od poprzedniego, wykorzystuje silnie fakt pokrewieństwa rozwiązań należących do sąsiedztwa, oraz akcelerator API, pozwalając na otrzymanie zdecydowanie mocniejszego rezultatu.

Twierdzenie 20 *Pełne otoczenie API w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(n+m)$ na maszynie CREW PRAM za pomocą $O(\frac{nm}{n+m})$ procesorów.*

Dowód Niech $v = (a, a+1)$ będzie parą przyległych pozycji. Przy projektowaniu algorytmu równoległego skorzystamy z sekwencyjnej wersji akceleratora API. Wartości r_{st}, q_{st} generuje się jednokrotnie, na początku procesu przeglądania otoczenia API, w czasie $O(n+m)$ na maszynie PRAM z $O(\frac{nm}{n+m})$ procesorami w sposób analogiczny jak opisano w dowodzie Tw. 4 i jest to metoda kosztowo optymalna.

Proces przeglądania otoczenia API można podzielić na grupy, po $\left\lceil \frac{n}{p} \right\rceil$ zamian pozycji w każdej grupie, gdzie $p = \left\lceil \frac{nm}{n+m} \right\rceil$ jest liczbą użytych procesorów. Obliczenia wartości funkcji kryterialnej w każdej grupie są od siebie

niezależne. Każdy procesor $k = 1,2,...,p$ będzie przeglądał część otoczenia otrzymaną za pomocą v ruchów postaci

$$v = ((k-1)\left\lceil\frac{n}{p}\right\rceil + a, (k-1)\left\lceil\frac{n}{p}\right\rceil + a + 1),$$

gdzie

$$a = 1, 2, \ldots, \left\lceil\frac{n}{p}\right\rceil \tag{6.51}$$

dla $k = 1, 2, ..., p-1$, oraz ruchów v postaci

$$v = ((p-1)\left\lceil\frac{n}{p}\right\rceil + a, (p-1)\left\lceil\frac{n}{p}\right\rceil + a + 1),$$

gdzie

$$a = 1, 2, \ldots, n - (p-1)\left\lceil\frac{n}{p}\right\rceil - 1 \tag{6.52}$$

dla $k = p$. Ostatnia grupa może pozostać niepełna. Ponieważ proces wyznaczenia wszystkich wartości $C_{max}(\pi_{(v)})$ w pojedynczej grupie ma złożoność

$$\left\lceil\frac{n}{p}\right\rceil O(m) = O(\tfrac{nm}{p}) = O(\tfrac{nm}{\frac{nm}{n+m}}) = O(n+m),$$

zatem taka też będzie złożoność obliczeniowa wyznaczenia wszystkich wartości $C_{max}(\pi_{(v)})$ dla wszystkich ruchów v. Każdy procesor, obliczając sekwencyjnie swoją porcję wartości $C_{max}(\pi_{(v)})$, może pamiętać wartość najlepszą. W tym celu należy dodatkowo wykonać liczbę porównań równą liczbie elementów w grupie pomniejszoną o 1, czyli

$$\left\lceil\frac{n}{p}\right\rceil - 1 = O(\frac{n}{p}) = O(\frac{n}{\frac{nm}{n+m}}) = O(\frac{n+m}{m}), \tag{6.53}$$

co zachowuje złożoność obliczeniową całej metody $O(n+m)$. W celu wyznaczenia najlepszego ruchu z całego otoczenia API należy znaleźć element minimalny z p najlepszych wartości funkcji kryterialnej pamiętanych dla każdej grupy. Zgodnie z Faktem 3 można zrobić w czasie $O(\log p)$, za pomocą $p = O(\frac{nm}{n+m})$ procesorów. Złożoność $O(\log p)$ tego etapu, poprzez następujący ciąg nierówności,

$$\log p = \log\left\lceil\frac{nm}{n+m}\right\rceil < \log(\frac{nm}{n+m} + 1) =$$

$$= \log(\frac{nm+n+m}{n+m}) = \log(\frac{(n+1)(m+1)-1}{n+m}) =$$

$$= (\log((n+1)(m+1)-1) - \log(n+m) < \log((n+1)(m+1)) =$$
$$= \log(n+1) + \log(m+1) < n+1+m+1 \tag{6.54}$$

nie zwiększa złożoności obliczeniowej $O(n+m)$ całej metody.■

Wniosek 20 Dla metody bazującej na Tw. 20 mamy

$$s_{A_p,M}(p) = O(\frac{nm}{n+m}), \quad \eta_{A_p,M}(p) = O(1). \quad \square \tag{6.55}$$

Zaprezentowana metoda jest więc kosztowo optymalna.

Poniżej zaprezentowana zostanie inna metoda przeglądania otoczenie API dla problemu $F^*||C_{max}$, działająca w czasie krótszego rzędu, a mianowicie $O(\log(n+m)\log(nm))$. Niestety, odbywa się to kosztem zwiększenia rzędu ilości procesorów.

Twierdzenie 21 *Pełne otoczenie API w problemie $F^*||\gamma$, $\gamma \in \{f_{max}, \sum f_i\}$, można przeglądnąć w czasie $O(\log(n+m)(\log(nm)))$ na $O(n^4m^3/\log(nm))$ procesorowej maszynie CREW PRAM.*

Dowód Pomijamy fakt pokrewieństwa rozwiązań. Przydzielamy każdemu z $n-1$ elementów otoczenia pulę $O(n^3m^3/\log(nm))$ procesorów (czyli wykorzystujemy łącznie $O(n^4m^3/\log(nm))$ procesorów), obliczając dla każdego elementu otoczenia wartość funkcji celu w czasie $O(\log(n+m)(\log(nm)))$ metodą opisaną w dowodach Tw. 7 lub 8, odpowiednio. Po wykonaniu niezależnego obliczenia $n-1$ wartości funkcji kryterialnych należy wybrać najlepszy ruch wyznaczając element minimalny spośród $n-1$ wartości, co zgodnie z Faktem 2 można zrobić w czasie $O(\log(n))$ za pomocą n procesorów. Ostatecznie cały metoda ma złożoność $O(\log(n+m)(\log(nm)))$ oraz wykorzystuje $O(n^4m^3/\log(nm))$ procesorów.■

Wniosek 21 Dla metody bazującej na Tw. 21 mamy

$$s_{A_p,M}(p) = O(\frac{nm}{\log(n+m)\log(nm)}),\ \eta_{A_p,M}(p) = O(\frac{1}{n^3m^2\log(n+m)}).\ \square \tag{6.56}$$

Metoda nie jest więc kosztowo optymalna, ale daje duży zysk czasowy, redukując złożoność sekwencyjną $O(nm)$ przeglądania otoczenia API do złożoności $O(\log(n+m)(\log\ nm))$. Zastosowanie w praktyce algorytmu wymagającego tak dużej liczby procesorów, rzędu $O(n^4m^3\ /\ \log(nm))$, jest raczej dedykowane do obliczeń w środowiskach biologicznych.

Fakt pokrewieństwa rozwiązań wykorzystany jest w twierdzeniu następnym, redukując liczbę procesorów potrzebnych do przeglądnięcia otoczenia API w problemie $F^*||C_{max}$ z $O(n^4m^3/\log(nm))$ do $O(n^3m^3/\log(nm))$, przy zachowaniu tej samej złożoności obliczeniowej.

Twierdzenie 22 *Pełne otoczenie API w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(\log(n+m)(\log(nm)))$ na $O(n^3m^3/\log(nm))$ procesorowej maszynie CREW PRAM.*

Dowód Korzystamy z grafu siatkowego $G(\pi)$ opisanego w Rozdz. 2.2 oraz podstawowej idei akceleratora API z odmienną techniką prowadzenia obliczeń. Wartości r_{st}, q_{st} reprezentujące długości najdłuższych dróg odpowiednio dochodzące i wychodzące z wierzchołka (s,t) można wyznaczyć w czasie $O(\log(n+m)(\log(nm)))$ na $O(n^3m^3\ /\ \log(nm))$ procesorowej maszynie PRAM wykorzystując metodę wyznaczania wszystkich najdłuższych dróg pomiędzy parami wierzchołków przedstawioną w dowodzie Tw. 7. Z własności grafu siatkowego wynika, że najdłuższa droga dochodząca do wierzchołka (s,t) rozpoczyna się w wierzchołku (1,1), a najdłuższa droga wychodząca z wierzchołka (s,t) kończy się w wierzchołku (n,m). Wobec tego po wyznaczeniu tablicy najdłuższych dróg A opisanej w dowodzie Tw. 7 można z niej bezpośrednio pobrać odpowiednie wartości r_{st}, q_{st} dla każdego wierzchołka (s,t), s = 1,2,...,m, t = 1,2,...,n. Następnie przydzielamy każdemu z $n-1$ rozwiązań z otoczenia API pulę $O(m^2/\log m)$ procesorów, które korzystając ze wzorów (6.46), (6.47), (6.48) w czasie $O(\log m)$ wyznaczają wartość funkcji kryterialnej dla pojedynczego elementu otoczenia – wymiany $v=(a,a+1)$. Proces ten wykonać można w sposób następujący. Wzór (6.48) zapisujemy w postaci

$$d_s = \max\{r_{s,a-1} + p_{s,\pi(a+1)},\ r_{s-1,a-1} + p_{s-1,\pi(a+1)} + p_{s,\pi(a+1)}, \ldots$$

$$\ldots, r_{1,a-1} + p_{1,\pi(a+1)} + p_{2\pi(a+1)} + \ldots + p_{s\pi(a+1)}\} =$$

$$\max_{1\leqslant k\leqslant s}(r_{k,a-1} + \sum_{t=k}^{s} p_{t,\pi(a+1)}) = \max_{1\leqslant k\leqslant s}(r_{k,a-1} + P^s_{k,\pi(a+1)}), \tag{6.57}$$

gdzie

$$P^s_{k,j} = \sum_{t=k}^{s} p_{t,j}, \quad k = 1,2,\ldots,s \tag{6.58}$$

są sumami prefiksowymi, które za pomocą puli $O(m/\log m)$ procesorów dla ustalonego s zgodnie z Faktem 1 można obliczyć w czasie $O(\log m)$. Ponieważ potrzebne są wartości wyrażeń $P^s_{k,j}$ dla wszystkich $s = 1,2,\ldots,m$,

można je wyznaczyć równolegle za pomocą $O(m^2/\log m)$ procesorów. Po tej operacji będziemy dysponowali wartościami $P^s_{k,j}$ dla każdego $s = 1, 2, \ldots, m$ i $k = 1, 2, \ldots, s$ przy ustalonym zadaniu $j = \pi(a{+}1)$. Sumy $r_{k,a-1}{+}P^s_{k,\pi(a+1)}$ występujące we wzorze (6.57), jako że jest ich najwyżej m, można policzyć w czasie $O(\log m)$ na $O(m/\log m)$ (patrz Fakt 6). Ostatecznie dla każdego s = 1,2,...,m wartość d_s można obliczyć w czasie $O(\log m)$ na $O(m/\log m)$ procesorach, czyli wszystkie te wartości wyznacza się w równolegle w czasie $O(\log m)$ za pomocą $O(m^2/\log m)$ procesorów. Identycznie rozwinąć można wzór (6.47):

$$d'_s = \max\{d'_{s-1}, d_s\} + p_{s,\pi(a)} =$$

$$\max\{d_s{+}p_{s\pi(a)}, d_{s-1}{+}p_{s-1,\pi(a)}{+}p_{s\pi(a)}, \ldots, d_1{+}p_{1,\pi(a)}{+}p_{2,\pi(a)}{+}...{+}p_{s\pi(a)}\} =$$

$$\max_{1\leqslant k\leqslant s}(d_k + \sum_{t=k}^{s} p_{t,\pi(a)}) = \max_{1\leqslant k\leqslant s}(d_k + P^s_{k,\pi(a)}) \tag{6.59}$$

Ponieważ wszystkie potrzebne sumy prefiksowe $P^s_{k,\pi(a)}$ wyznaczyć możemy w czasie $O(\log m)$ za pomocą $O(m^2/\log m)$ procesorów, a wartości d_s są już wyznaczone wcześniej, obliczenia wszystkich wartości d'_s, s = 1,2,...,m można dokonać równolegle w czasie $O(\log m)$ za pomocą $O(m^2/\log m)$ procesorów według zasad podanych przy wyznaczaniu d_s. Ostatecznie więc, $C_{\max}(\pi_{(v)}) = \max_{1\leqslant s\leqslant m}(d'_s + q_{s,a+2})$ (patrz wzór (6.47)) wyznaczymy w czasie $O(\log m)$ za pomocą $O(m/\log m)$ procesorów - operacja ta polega na wykonaniu m dodawań $d'_s + q_{s,a+2}$, $s = 1, 2, \ldots, m$, co wykonać można w czasie $O(\log m)$ na $O(m/\log m)$ procesorach (patrz Fakt 6). Następnie dla wyznaczenia (6.47), to jest obliczenia maksimum ze zbioru m elementowego, co wymaga czasu $O(\log m)$ (patrz Fakt 4), angażujemy $O(m/\log m)$ procesorów. Tak więc ostatecznie za pomocą $(n{-}1)O(m^2/\log m) = O(nm^2/\log m)$ procesorów można w czasie $O(\log m)$ wyznaczyć wartości funkcji kryterialnej dla wszystkich elementów otoczenia API. Następnie należy znaleźć element otoczenia o minimalnej wartości funkcji kryterialnej, co można zrobić w czasie $O(\log n)$ za pomocą $n - 1$ procesorów. Cała metoda potrzebuje

$$O(\max\{n-1, \frac{nm^2}{\log m}, \frac{n^3m^3}{\log(nm)}\}) = O(\frac{n^3m^3}{log(nm)}) \tag{6.60}$$

procesorów, a jej złożoność obliczeniowa wynosi

$$O(\max\{\log m, \log n, \log(n+m)(\log(nm))\}) = O(\log(n+m)\log(nm)).\blacksquare \tag{6.61}$$

Wniosek 22 Dla metody bazującej na Tw. 22 mamy

$$s_{A_p,M}(p) = O(\frac{nm}{\log(n+m)\log(nm)}),\ \eta_{A_p,M}(p) = O(\frac{1}{(nm)^2\log(n+m)}).\ \square \tag{6.62}$$

Łatwo zauważyć, że otoczenie API możemy przeglądnąć równolegle w czasie tego samego rzędu, co wyznaczenie wartości funkcji celu pojedynczego rozwiązania, porównaj Tw. 19 – 22 z twierdzeniami 4, 6 – 8. W klasie omawianych metod istnieje algorytm kosztowo optymalny, patrz Tw. 20.

6.2.5 Problem przepływowy. Otoczenie INS

Przeglądnięcie otoczenia INS w ogólnym przypadku implikuje złożoność obliczeniową $O(n^3m)$. Dla otoczenia INS oraz kryterium C_{max} znany jest sekwencyjny akcelerator, patrz np. praca [170]. Złożoność obliczeniowa sekwencyjnego przeglądania otoczenia INS z akceleratorem sekwencyjnym w problemie przepływowym $F^*||C_{max}$ wynosi $O(n^2m)$. Pokażemy mocniejsze wyniki dla algorytmów równoległych.

Twierdzenie 23 *Pełne otoczenie INS w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(n+m)$ na maszynie CREW PRAM za pomocą $O(\frac{n^2m}{n+m})$ procesorów.*

Dowód Niech $v = (a, b)$, $a \neq b$ określa otoczenie INS polegające na następującej modyfikacji permutacji π: usunięciu zadania $\pi(a)$ oraz wstawieniu go do π tak, aby było na pozycji b w permutacji $\pi_{(v)}$. Niech r_{st}, q_{st}, $s = 1, 2, ..., m$, $t = 1, 2, ..., n-1$, będą wielkościami wyznaczonymi ze wzorów (6.47) dla permutacji $(n-1)$ elementowej otrzymanej z π przez usunięcie zadania $\pi(a)$. Dla każdej pozycji a = 1,2,...,n wartości r_{st}, q_{st} można wyznaczyć w czasie $O(n+m)$ na maszynie PRAM, z $O(\frac{nm}{n+m})$ procesorami zgodnie z metodą zaprezentowaną w Tw. 4. Dysponując $O(\frac{n^2m}{n+m})$ procesorami operację tę można wykonać w czasie $O(n+m)$ dla wszystkich permutacji $(n-1)$ elementowych otrzymanych z π przez usunięcie zadania $\pi(a)$, $a = 1, 2, ..., n$. Dla każdego ustalonego a wartość $C_{\max}(\pi_{(v)})$ otrzymaną przez wstawienie zadania $\pi(a)$ na pozycje b = 1,2,...,n, $b \neq a$ można obliczyć, korzystając z zależności (6.46), w czasie $O(m)$. Dzielimy proces wyznaczania wartości funkcji kryterialnej elementów otoczenia na $p = \left\lceil \frac{n^2m}{n+m} \right\rceil$ grup przyporządkowanym pojedynczym procesorom. Korzystając z opisanej powyżej własności i z faktu, że otoczenie INS zawiera $(n-1)^2 = O(n^2)$ rozwiązań, złożoność obliczeniowa wyznaczenia wszystkich wartości $C_{\max}(\pi_{(v)})$ wynosi

$$\left\lceil \frac{(n-1)^2}{p} \right\rceil O(m) = O(n+m).$$

Następnie należy znaleźć element otoczenia o minimalnej wartości funkcji kryterialnej, co można zrobić w czasie $O(\log(n^2)) = O(2\log n) = O(\log n)$ za pomocą n procesorów. Cała metoda ma więc złożoność $O(n+m+\log n) = O(n+m)$ i potrzebuje $O(\frac{n^2 m}{n+m})$ procesorów.■

Wniosek 23 Dla metody bazującej na Tw. 23 mamy

$$s_{A_p,M}(p) = O(\frac{n^2 m}{n+m}), \quad \eta_{A_p,M}(p) = O(1). \square \tag{6.63}$$

Zaproponowana metoda jest kosztowo optymalna.

Poniżej zostanie zaprezentowany inny sposób przeglądania otoczenia INS w problemie $F^*||C_{max}$. Mimo że proponowany algorytm nie jest kosztowo optymalny, pozwala skrócić rząd złożoności obliczeniowej do $O(\log(n+m)(\log(nm)))$, kosztem zwiększenia liczby procesorów.

Twierdzenie 24 *Pełne otoczenie INS w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(\log(n+m)(\log(nm)))$ na $O(n^5 m^3/\log(nm))$ procesorowej maszynie CREW PRAM.*

Dowód Analogicznie jak w Tw. 21, metoda opiera się na wykorzystaniu takiej liczby procesorów, która pozwoli przeglądnąć niezależnie całe otoczenie wyznaczając wartość funkcji kryterialnej każdego elementu otoczenia w czasie $O(\log(n+m)\log(nm))$. Odpowiada to przydzieleniu każdemu z $(n-1)^2$ elementów otoczenia puli $O(n^3 m^3/\log(nm))$ procesorów (czyli wykorzystaniu $O(n^5 m^3/\log(nm))$ procesorów), dla każdego elementu otoczenia obliczając wartość funkcji celu w czasie rzędu $O(\log(n+m)\log(nm))$ metodą opisaną w dowodzie Tw. 7. Po wykonaniu niezależnego obliczenia n^2 wartości funkcji kryterialnych należy wybrać najlepszy ruch wyznaczając element minimalny spośród n^2 wartości, co zgodnie z Faktem 2 można zrobić w czasie $O(\log(n^2)) = O(2\log n) = O(\log n)$ za pomocą n^2 procesorów. Zachowuje to złożoność $O(\log(n+m)(\log(nm)))$ oraz rząd $O(n^5 m^3/\log(nm))$ ilości procesorów używanych w opisywanej metodzie.■

Wniosek 24 Dla metody bazującej na Tw. 24 mamy

$$s_{A_p,M}(p) = O(\frac{n^2 m}{\log(n+m)\log(nm)}),\ \eta_{A_p,M}(p) = O(\frac{1}{n^3 m^2 \log(n+m)}). \square \tag{6.64}$$

Otrzymany rezultat ma, w chwili obecnej, bardziej teoretyczny niż użytkowy charakter. Ze względów technicznych układy elektroniczne z tak dużą liczbą procesorów (rzędu $O(n^5m^3/\log(nm))$) są dotychczas niespotykane. Nie stanowi to jednak bariery dla systemów obliczeń biologicznych.

Metoda nie jest kosztowo optymalna, ale daje duży zysk czasowy, redukując złożoność sekwencyjną $O(n^2m)$ przeglądania otoczenia INS do złożoności $O(\log(n+m)(\log\ nm))$. Należy ją (a także Tw. 21 dotyczące otoczenia API) traktować raczej jako następujące stwierdzenie: mając dowolnie dużą liczbę procesorów można przeglądnąć otoczenie INS (API) w czasie $O(\log(n+m)(\log\ nm))$. Problem, czy jest to granica możliwości, jest otwarty.

Stosując bardziej wyrafinowaną metodę możliwe jest uzyskanie złożoności obliczeniowej $O(m+\log(n+m)\log(nm))$, a nawet $O(\log(n+m)\log(nm))$ za pomocą o liczby procesorów rzędu mniejszego o n^2, o czym traktują dwa następne twierdzenia. Taka liczba procesorów – rzędu $O(n^3m^3\ /\ \log(nm))$ – dla przykładów testowych cytowanych w literaturze mieści się w granicach możliwości technicznych współczesnych systemów obliczeń równoległych, co podkreślają autorzy prac proponujących algorytmy wymagające nawet liczby procesorów rzędu $O((nm)^3)$ (zobacz [168]).

Twierdzenie 25 *Pełne otoczenie INS w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(m+\log(n+m)(log(nm)))$ na $O(n^3m^3/\log(nm))$ procesorowej maszynie CREW PRAM.*

Dowód Niech $G(\pi)$ będzie grafem siatkowym zdefiniowanym w Rozdz. 2.2 dla rozwiązania π generującego otoczenie. Niech r_{st}, q_{st}, $s = 1,2,...,m$, $t = 1,2,...,n-1$, będą wielkościami wyznaczonymi ze wzorów (6.47) dla permutacji $(n-1)$ elementowej otrzymanej z π przez usunięcie zadania $\pi(a)$. Wartości r_{st}, q_{st} reprezentujące długości najdłuższych dróg odpowiednio dochodzące i wychodzące z wierzchołka (s,t) można wyznaczyć w czasie $O(\log(n+m)(\log(nm)))$ na $O(n^3m^3/\log(nm))$ procesorowej maszynie PRAM wykorzystując metodę wyznaczania wszystkich najdłuższych dróg pomiędzy parami wierzchołków zaprezentowaną w dowodzie Tw. 7. Z własności grafu siatkowego wynika, że najdłuższa droga dochodząca do wierzchołka (s,t) rozpoczyna się w wierzchołku (1,1), a najdłuższa droga wychodząca z wierzchołka (s,t) kończy się w wierzchołku (n,m). Wobec tego po wyznaczeniu tablicy najdłuższych dróg $\max_{dist}$ opisanej w dowodzie Tw. 7. można z niej bezpośrednio pobrać odpowiednie wartości r_{st}, q_{st} dla każdego wierzchołka (s,t), s = 1,2,...,m, t = 1,2,...,n. Następnie przydzielamy każdemu z $O(n^2)$ elementów otoczenia INS pojedynczy procesor, który korzystając z zależności

$$C_{\max}(\pi_{(v)}) = \max_{1 \leqslant i \leqslant m} (d_i + q_{i,b+1}), \tag{6.65}$$

gdzie

$$d_i = \max\{r_{i,b}, d_{i-1}\} + p_{i\pi(a)},\ i = 1, 2, ..., m, \tag{6.66}$$

może w czasie $O(m)$ wyznaczyć wartość funkcji kryterialnej dla pojedynczego elementu otoczenia odpowiadającego wykonaniu ruchu $v = (a,b)$, $a \neq b$. Za pomocą $O(n^2)$ procesorów można w czasie $O(m)$ wyznaczyć wartości funkcji kryterialnej niezależnie dla wszystkich elementów otoczenia INS. Następnie należy znaleźć element otoczenia o minimalnej wartości funkcji kryterialnej, co można zrobić w czasie $O(\log n^2) = O(2 \log n) = O(\log n)$ za pomocą $O(n^2)$ procesorów. Cała metoda potrzebuje

$$O(\max\{n^2, n^3m^3 \ / \ \log(nm)\}) = O(n^3m^3 \ / \ \log(nm))$$

procesorów, a jej złożoność obliczeniowa wynosi

$$O(max\{m, \log(n+m)(\log(nm))\}) = O(m+\log(n+m)(\log\ (nm))).\blacksquare$$

Wniosek 25 Dla metody bazującej na Tw. 25 mamy

$$s_{A_p,M}(p) = O(\frac{n^2m}{m + \log(n+m)\log(nm)}), \tag{6.67}$$

$$\eta_{A_p,M}(p) = O(\frac{\log(nm)}{nm^2(m + \log(n+m)\log(nm))}).\ \square \tag{6.68}$$

Następne twierdzenie pokazuje, jak złożoność $O(m+\log(n+m(log(nm)))$ metody zaprezentowanej w Tw. 25 można zredukować do złożoności $O(\log(n+m)(log(nm)))$, przy zachowaniu tej samej liczby procesorów maszyny PRAM, rzędu $O(n^3m^3/\log(nm))$.

Twierdzenie 26 *Pełne otoczenie INS w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(\log(n+m)(\log(nm)))$ na $O(n^3m^3/\log(nm))$ procesorowej maszynie CREW PRAM.*

Dowód Postępujemy analogicznie jak w dowodzie twierdzenia poprzedniego. Posługując się wzorami (6.65) oraz (6.66), równolegle wyznaczamy wartość funkcji $C_{\max}(\pi_{(v)})$ dla każdego z $n(n-1)$ elementów otoczenia INS.

Wcześniej, w czasie $O(\log(n+m)(\log nm))$ na $O(n^3m^3 \ / \log(nm))$ procesorach, obliczamy wartości r_{st}, q_{st}, s = 1,2,...,m, t = 1,2,...,n-1, będą wielkościami wyznaczonymi ze wzorów (6.47) dla permutacji $(n-1)$ elementowej otrzymanej z π przez usunięcie zadania $\pi(a)$. Następnie do obliczenia każdego z $n(n-1)$ elementów otoczenia INS przydzielamy pulę $O(m^2/\log m)$ procesorów. Wzór (6.66) rozwijamy do postaci

$$d_i = \max\{r_{i,b}, d_{i-1}\} + p_{i,\pi(a)} =$$

$$\max\{r_{i,b} + p_{i,\pi(a)}, r_{i-1,b} + p_{i-1,\pi(a)} + p_{i,\pi(a)}, ...$$

$$..., r_{1,b} + p_{1,\pi(a)} + p_{2,\pi(a)} + ... + p_{i,\pi(a)}\} =$$

$$\max_{1 \leqslant k \leqslant i} (r_{k,b} + \sum_{t=k}^{i} p_{t,\pi(a)}) = \max_{1 \leqslant k \leqslant i} (r_{k,b} + P^i_{k,\pi(a)}) \tag{6.69}$$

gdzie

$$P^i_{k,j} = \sum_{t=k}^{i} p_{t,j}, \quad k = 1, 2, ..., i \tag{6.70}$$

są sumami prefiksowymi które, zgodnie z Faktem 1, dla puli $O(m/\log m)$ procesorów, można dla ustalonego i obliczyć w czasie $O(\log m)$. Ponieważ potrzebne są $P^i_{k,j}$ dla wszystkich i = 1,2,...,m, toteż można je wyznaczyć równolegle na etapie wstępnym za pomocą $O(m^2/\log m)$ procesorów (czyli puli jaką dysponuje proces wyznaczania wartości funkcji kryterialnej dla jednego elementu otoczenia INS). Po tej operacji będziemy dysponowali wartościami $P^i_{k,j}$ dla każdych i = 1,2,...,m, k = 1,2,...,i. Sumy $r_{k,b} + P^i_{k,\pi(a)}$, k = 1,2,...,i we wzorze (6.69), jako że jest ich najwyżej m, można policzyć w czasie $O(\log m)$ na $O(m/\log m)$ procesorach (patrz Fakt 6). Ostatecznie dla każdego i = 1,2,...,m wartość d_i można obliczyć w czasie $O(\log m)$ na $O(m/\log m)$ procesorach, czyli wszystkie te wartości wyznacza się w równolegle w czasie $O(\log m)$ za pomocą $O(m^2/\log m)$ procesorów. Następnie dla wyznaczenia $C_{\max}(\pi_{(v)}) = \max\limits_{1 \leqslant i \leqslant m} (d_i + q_{i,b+1})$ dokonujemy równolegle m dodawań $d_i + q_{i,b+1}$, i = 1,2,...,m oraz obliczenia maksimum ze zbioru m elementowego co wymaga także $O(m/\log m)$ procesorów i czasu $O(\log m)$ (patrz odpowiednio dla m dodawań Fakt 6, dla maksimum m liczb Fakt 4). Tak więc ostatecznie, za pomocą $n(n-1)O(m^2/\log m) = O(n^2m^2/\log m)$ procesorów można w czasie $O(\log m)$ wyznaczyć wartości

funkcji kryterialnej dla wszystkich elementów otoczenia INS. Następnie należy znaleźć element otoczenia o minimalnej wartości funkcji kryterialnej, co można zrobić w czasie $O(\log n^2) = O(2\log n) = O(\log n)$ za pomocą n^2 procesorów. Cała metoda potrzebuje

$$O(\max\{n^2, \frac{n^2m^2}{\log m}, \frac{n^3m^3}{\log(nm)}\}) = O(\frac{n^3m^3}{\log(nm)})$$

procesorów, a jej złożoność obliczeniowa wynosi

$$O(\max\{\log m, \log n, \log(n+m)(\log(nm))\}) = O(\log(n+m)\log(nm)).\blacksquare$$

Wniosek 26 Dla metody bazującej na Tw. 26 mamy

$$s_{A_p,M}(p) = O(\frac{n^2m}{\log(n+m)\log(nm)}),\ \eta_{A_p,M}(p) = O(\frac{1}{nm^2\log(n+m)}).\ \square \tag{6.71}$$

Otoczenie INS możemy przeglądnąć równolegle w czasie tego samego rzędu, co wyznaczenie wartości funkcji celu pojedynczego rozwiązania, porównaj Tw. 24 – 26 z twierdzeniami 6 – 8. W klasie omawianych metod istnieje algorytm kosztowo optymalny, patrz Tw. 23.

6.2.6 Problem przepływowy. Otoczenie NPI

Rozpoczniemy od opisu sekwencyjnego akceleratora podanego w [170], a wykorzystywanego w algorytmach równoległych opisanych w tym rozdziale. Algorytm bezpośredni wykorzystujący otoczenie NPI ma złożoność obliczeniową $O(n^3m)$. Akcelerator NPI dla problemu $F^*||C_{max}$ ma złożoność obliczeniową $O(n^2m)$.

Niech v= (a,b), $a \neq b$ określa parę zadań ($\pi(a)$, $\pi(b)$), których wymiana generuje nową permutację $\pi_{(v)}$. Bez straty ogólności rozważań możemy przyjąć $a < b$, ze względu na symetrię. Dalej niech r_{st}, q_{st}, s = 1,2,...,m, t = 1,2,...,n będą wielkościami wyznaczonymi ze wzorów (6.45) dla permutacji $n-$elementowej π. Oznaczmy przez D_{st}^{xy} długość najdłuższej drogi pomiędzy węzłami (s,t) oraz (x,y) w grafie siatkowym $G(\pi)$. Sposób obliczania $C_{\max}(\pi_{(v)})$ można wyrazić poprzez następujące rozumowanie. Najpierw obliczamy długość najdłuższej drogi dochodzącej do wierzchołka (s,a), dołączając zadanie $\pi(b)$ przestawione na mocy zamiany v na pozycję a

$$d_s = \max\{d_{s-1}, r_{s,a-1}\} + p_{s,\pi(b)},\ s = 1,2,...,m, \tag{6.72}$$

gdzie $d_0 = 0$. Następnie obliczamy długość najdłuższej drogi dochodzącej do wierzchołka $(s, b-1)$, dołączając fragment grafu zawarty pomiędzy zadaniami na pozycjach od $a+1$ do $b-1$ włącznie, niezmienny względem $G(\pi)$

$$d'_s = \max_{1 \leqslant w \leqslant s} (d_w + D^{s,b-1}_{w,a+1}),\ s = 1, 2, ..., m. \tag{6.73}$$

Kolejno obliczamy długość najdłuższej drogi dochodzącej do wierzchołka (s,b), dołączając zadanie $\pi(a)$ przestawione w wyniku zamiany na pozycję b

$$d''_s = \max\{d''_{s-1}, d'_s\} + p_{s,\pi(a)},\ s = 1, 2, ..., m, \tag{6.74}$$

gdzie $d''_0 = 0$. W końcu otrzymujemy

$$C_{\max}(\pi_{(v)}) = \max_{1 \leqslant s \leqslant m} (d''_s + q_{s,b+1}). \tag{6.75}$$

Obliczenie $C_{\max}(\pi_{(v)})$ jest możliwe pod warunkiem, że dysponujemy odpowiednimi wartościami D^{xy}_{st}. Te ostatnie można obliczać rekurencyjnie, dla ustalonych t oraz $y = t+1,\ t+2,...,n$, korzystając z zależności

$$D^{x,y+1}_{st} = \max_{s \leqslant k \leqslant x} (D^{ky}_{st} + \sum_{i=k}^{x} p_{i\pi(y+1)}) \tag{6.76}$$

gdzie $D^{xt}_{st} = \sum_{i=s}^{x} p_{i\pi(t)}$. Formuła ta może być zapisana inaczej w postaci

$$D^{s,t+1}_{st} = D^{st}_{st} + p_{s,\pi(t+1)}, \quad D^{x0}_{st} = D^{0y}_{st} = 0, \tag{6.77}$$

$$D^{x,y+1}_{st} = \max\{D^{xy}_{st}, D^{x-1,y}_{st}\} + p_{x,\pi(y+1)}, \tag{6.78}$$

$x = 1, 2, ..., m,\ y = 1, 2, ..., n$, pozwalającej dla ustalonych (s,t) wyznaczyć wszystkie D^{xy}_{st}, x =1,2,...,m, y= 1,2,...,n w czasie $O(nm)$. Ostatecznie obliczenie sekwencyjne wszystkich $O(n^2)$ wartości $C_{\max}(\pi_{(v)})$, a wcześniej wszystkich D^{xy}_{st}, x, s =1,2,...,m, y, t= 1,2,...,n można wykonać sekwencyjnie w czasie $O(n^2m^2)$ (patrz także [141]).

Twierdzenie 27 *Pełne otoczenie NPI w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(nm)$ na $O(n^2m)$ procesorowej maszynie CREW PRAM.*

Dowód Podamy odpowiednik równoległy akceleratora sekwencyjnego. Niech każdy z $\frac{n(n-1)}{2}$ elementów otoczenia będzie związany z pulą $O(m)$ procesorów. Dla ustalonych (s,t) wartość D^{xy}_{st} i wszystkich x =1,2,...,m, y=

1,2,...,n można wyliczyć sekwencyjnie w czasie $O(nm)$. Używając $O(nm)$ procesorów możemy wyznaczyć D_{st}^{xy} dla wszystkich x, s =1,2,...,m, y, $t = 1, 2, ..., n$ w czasie $O(nm)$ i zrobić to można jednorazowo, na wstępnym etapie, dla wszystkich elementów otoczenia. Skupimy się na wyliczeniu wartości $C_{\max}(\pi_{(v)})$ dla jednego ustalonego elementu otoczenia związanego z dowolnym ruchem v. Obliczenie wartości d_s w (6.72) wykonujemy sekwencyjnie, w czasie $O(m)$. Wyliczenia maksimum m wartości w formule (6.73) można wykonać równolegle, dla wszystkich s, za pomocą $O(m)$ procesorów w czasie $O(m)$. Formułę (6.74) wyliczamy dla każdego s sekwencyjnie, w czasie $O(m)$. Wyliczenie wartości $C_{\max}(\pi_{(v)})$ w (6.75) jest równoznaczne z wykonaniem niezależnym m dodawań a następnie wyznaczeniu maksimum z m wartości. Wykonujemy to sekwencyjnie w czasie $O(m)$. Ostatecznie obliczenie równoległe wszystkich $O(n^2)$ wartości $C_{\max}(\pi_{(v)})$ można wykonać w czasie $O(m)$ za pomocą $O(n^2m)$ procesorów. Ponieważ jednak proces generowania D_{st}^{xy} miał złożoność $O(nm)$ więc taka też będzie złożoność obliczeniowa całej metody.■

Wniosek 27 Dla metody bazującej na Tw. 27 mamy

$$s_{A_p,M}(p) = O(\frac{n^2m^2}{nm}) = O(nm),\ \eta_{A_p,M}(p) = O(\frac{nm}{n^2m}) = O(\frac{1}{n}).\ \square \quad (6.79)$$

Następne trzy twierdzenia odnoszą się do tego samego zagadnienia, do którego odnosiło się twierdzenie poprzednie - przeglądania otoczenia NPI dla problemu przepływowego $F^*||C_{max}$. W Tw. 28 zostanie zaproponowana metoda pozwalająca przeglądać otoczenie NPI szybciej niż przestawiono w twierdzeniu poprzednim (Tw. 27), w czasie rzędu $O(m + \log(n + m)(\log(nm)))$. Natomiast dalej, w Tw. 30, pokazane zostanie, że przegląd ten można wykonać w czasie $O(\log(n+m)(\log(nm)))$, zachowując ten sam rząd liczby procesorów, czyli $O(n^3m^3/\log(nm))$. Z kolei w Tw. 29 zaprezentowana zostanie metoda przeglądania otoczenia NPI badanego problemu działająca dłużej, w czasie $O(n + m)$, mająca jednak własność kosztowej optymalności, przy wykorzystaniu $O(\frac{n^2m^2}{n+m})$ procesorów.

Twierdzenie 28 *Pełne otoczenie NPI w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(m + \log(n + m)(\log(nm)))$ na $O(n^3m^3/\log(nm))$ procesorowej maszynie CREW PRAM.*

Dowód Stosujemy rozumowanie podobne jak w dowodzie twierdzenia poprzedniego z tym, że wartości D_{st}^{xy}, czyli długości najdłuższych dróg pomiędzy wierzchołkami (s, t) i (x, y) wyznaczamy za pomocą $O(n^3m^3/\log(nm))$ procesorów w czasie $O(\log(n + m)(\log\ nm))$ (patrz dowód Tw. 7). Mając

D_{st}^{xy} można obliczyć wartości każdego $C_{\max}(\pi_{(v)})$ w czasie $O(m)$ za pomocą $O(m)$ procesorów. Biorąc do tego etapu $O(n^2m)$ procesorów złożoność czasowa wyznaczania całego otoczenia NPI wynosi

$$O(\max\{m, \log(n+m)\log(nm)\}) = O(m + \log(n+m)\log(nm)),$$

a liczba procesorów

$$O(\max\{n^2m, n^3m^3/\log(nm)\}) = O(n^3m^3/\log(nm)).\blacksquare$$

Wniosek 28 Dla metody bazującej na Tw. 28 mamy

$$s_{A_p,M}(p) = O(\frac{n^2m^2}{m + \log(n+m)(\log(nm)}), \tag{6.80}$$

$$\eta_{A_p,M}(p) = \frac{s_{A_p,M}(p)}{p} = O(\frac{\frac{n^2m^2}{m+\log(n+m)\log(nm)}}{\frac{n^3m^3}{\log(nm)}}) =$$

$$= O(\frac{\log(nm)}{nm(m + \log(n+m)\log(nm))}). \square \tag{6.81}$$

Twierdzenie 29 *Pełne otoczenie NPI w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(n + m)$ na $O(\frac{n^2m^2}{n+m})$ procesorowej maszynie CREW PRAM.*

Dowód Stosując schemat przedstawiony w dowodzie Tw. 4 i wykorzystując rekurencyjną definicję (6.78) dla D_{st}^{xy} można używając $O(\frac{nm}{n+m})$ procesorów otrzymać D_{st}^{xy} dla ustalonej pary (s,t) oraz wszystkich x =1,2,...,m, y= 1,2,...,n w czasie $O(n + m)$. Stosując nm razy więcej procesorów, czyli $O(\frac{n^2m^2}{n+m})$, można wyliczyć równolegle D_{st}^{xy}, x =1,2,...,m, y= 1,2,...,n także dla wszystkich s =1,2,...,m, t= 1,2,...,n zachowując złożoność czasową $O(n+m)$. Dalej stosujemy rozumowanie identyczne jak w dowodzie twierdzenia poprzedniego – z tym, że proces równoległego wyznaczania wartości $C_{\max}(\pi_{(v)})$ dla poszczególnych elementów otoczenia NPI rozdzielamy na $p = \left\lceil \frac{n^2m}{n+m} \right\rceil$ grup, z których każda dysponuje $O(m)$ procesorami. Tak więc wszystkich procesorów jest $O(pm)$ = O($\frac{n^2m^2}{n+m}$). Proces generowania pojedynczej wartości $C_{\max}(\pi_{(v)})$ dla ustalonego ruchu v niech będzie wykonywany tak jak opisano w dowodzie Tw. 28, czyli w czasie $O(m)$ za pomocą $O(m)$ procesorów. Złożoność obliczeniowa wyznaczenia wszystkich n^2

wartości $C_{\max}(\pi_{(v)})$, jeśli proces obliczeń podzieliliśmy na p równoległych niezależnych grup (wątków), wyniesie

$$\frac{n^2}{p}O(m) = O(\frac{n^2}{\lceil \frac{n^2 m}{n+m} \rceil}m) = O(n+m) \tag{6.82}$$

i taka też jest złożoność obliczeniowa całej metody. Ponieważ każdy z p wątków dysponował pulą $O(m)$ procesorów, więc liczba użytych procesorów wynosi $O(pm) = \mathrm{O}(\frac{n^2m^2}{n+m})$.■

Wniosek 29 Dla metody bazującej na Tw. 29 mamy

$$s_{A_p,M}(p) = O(\frac{n^2m^2}{n+m}),\ \eta_{A_p,M}(p)O(\frac{\frac{n^2m^2}{n+m}}{\frac{n^2m^2}{n+m}}) = O(1).\ \square \tag{6.83}$$

Metoda jest kosztowo optymalna.

Twierdzenie 30 *Pełne otoczenie NPI w problemie $F^*||C_{max}$ można przeglądnąć w czasie $O(\log(n+m)(\log(nm)))$ na $O(n^3m^3/\log(nm))$ procesorowej maszynie CREW PRAM.*

Dowód Niech wartości D_{st}^{xy}, czyli długości najdłuższych dróg pomiędzy wierzchołkami (s,t) i (x,y) będą zdefiniowane jak w (6.76). Długości najdłuższych dróg pomiędzy wierzchołkami (s,t) i (x,y) w grafie siatkowym $G(\pi)$ można wyznaczyć za pomocą $O(n^3m^3\ /\ \log(nm))$ procesorów w czasie $O(\log(n+m)(\log(nm)))$ (patrz dowód Tw. 7). Niech każdy z n^2 elementów otoczenia będzie związany z pulą $O(m^2/\log m)$ procesorów. Skupiamy się na wyliczeniu wartości $C_{\max}(\pi_{(v)})$ dla jednego ustalonego elementu otoczenia związanego z dowolnym ruchem v. Obliczenie wartości d_s w (6.72) wykonujemy równolegle, w czasie $O(\log m)$, za pomocą $O(m^2/\log m)$ procesorów. Wcześniej rozwijamy wzór (6.72) do postaci

$$\begin{aligned} d_s = \max\{&r_{s,a-1} + p_{s\pi(b)},\ r_{s-1,a-1} + p_{s-1,\pi(b)} + p_{s\pi(b)}, ... \\ &..., r_{1,a-1} + p_{1,\pi(b)} + p_{2\pi(b)} + ... + p_{s\pi(b)}\} = \\ = \max_{1\leqslant k\leqslant s}&(r_{k,a-1} + \sum_{t=k}^{s} p_{t,\pi(b)}) = \max_{1\leqslant k\leqslant s}(r_{k,a-1} + P^s_{k,\pi(b)}) \end{aligned} \tag{6.84}$$

gdzie

$$P^s_{k,j} = \sum_{t=k}^{s} p_{t,j}, \quad k = 1,2,...,s \tag{6.85}$$

są sumami prefiksowymi które zgodnie z Faktem 1 można dla ustalonego s obliczyć w czasie $O(\log m)$ korzystając z puli $O(m/\log m)$ procesorów. Ponieważ potrzebne są $P^s_{k,j}$ dla wszystkich s = 1,2,...,m, więc można je wyznaczyć równolegle za pomocą $O(m^2/\log m)$ procesorów. Po tej operacji będziemy dysponowali wartościami $P^s_{k,j}$ dla każdego $s = 1, 2, ..., m$ i $k = 1, 2, ..., s$ przy ustalonym zadaniu $j = \pi(b)$. Sumy $r_{k,a-1} + P^s_{k,\pi(b)}$ występujące we wzorze (6.84), jako że jest ich co najwyżej m, można policzyć w czasie $O(\log m)$ na $O(m/\log m)$ procesorach (patrz Fakt 6). Ostatecznie dla każdego s = 1,2,...,m wartość d_s można obliczyć w czasie $O(\log m)$ na $O(m/\log m)$ procesorach, czyli wszystkie te wartości wyznacza się w równolegle w czasie $O(\log m)$ za pomocą $O(m^2/\log m)$ procesorów.

Wyliczenia maksimum m wartości w formule (6.73) można wykonać równolegle, dla wszystkich s, za pomocą $O(m^2/\log m)$ procesorów w czasie $O(\log m)$. Wcześniej wyliczamy m sum $d_w + D^{s,b-1}_{w,a+1}$ w czasie $O(\log m)$ za pomocą $O(m/\log m)$ procesorów (Fakt 6). Formułę (6.74) rozwijamy do postaci

$$d''_s = \max\{d''_{s-1}, d'_s\} + p_{s,\pi(a)} =$$

$$= \max\{d'_s + p_{s,\pi(a)}, d'_{s-1} + p_{s-1,\pi(a)} + p_{s,\pi(a)}, \dots$$

$$\dots, d'_1 + p_{1,\pi(a)} + p_{2,\pi(a)} + ... + p_{s,\pi(a)}\} =$$

$$= \max_{1\leqslant k\leqslant s}(d'_k + \sum_{t=k}^{s} p_{t,\pi(a)}) = \max_{1\leqslant k\leqslant s}(d'_k + P^s_{k,\pi(a)}) \tag{6.86}$$

Ponieważ wszystkie potrzebne sumy prefiksowe $P^s_{k,\pi(a)}$ wyznaczyć możemy w czasie $O(\log m)$ za pomocą $O(m^2/\log m)$ procesorów, a wartości d'_s są już wyznaczone wcześniej, obliczenia wszystkich wartości d''_s, s = 1,2,...,m można dokonać równolegle w czasie $O(\log m)$ za pomocą $O(m^2/\log m)$ procesorów według zasad podanych przy wyznaczaniu d'_s. Ostatecznie

$$C_{\max}(\pi_{(v)}) = \max_{1\leqslant s\leqslant m}(d'_s + q_{s,a+2}) \tag{6.87}$$

wyznaczamy w czasie $O(\log m)$ za pomocą $O(m/\log m)$ procesorów - operacja ta polega na wykonaniu m dodawań $d'_s + q_{s,a+2}$, s=1,2,...,m, co wykonać można w czasie $O(\log m)$ na $O(m/\log m)$ procesorach (patrz Fakt 6). Następnie dla wyznaczenia $C_{\max}(\pi_{(v)}) = \max_{1\leqslant s\leqslant m}(d'_s + q_{s,a+2})$ obliczamy maksimum ze zbioru m elementowego, co wymaga także $O(m/\log m)$ procesorów i czasu $O(\log m)$ (patrz Fakt 4). Tak więc ostatecznie za pomocą

$$(n^2)O(m^2/\log m) = O(n^2m^2/\log m) \tag{6.88}$$

procesorów można w czasie $O(\log m)$ wyznaczyć wartości funkcji kryterialnej dla wszystkich elementów otoczenia NPI. Następnie należy znaleźć element otoczenia o minimalnej wartości funkcji kryterialnej, co można zrobić w czasie $O(\log n^2) = O(2\log n) = O(\log n)$ za pomocą n^2 procesorów. Cała metoda potrzebuje

$$O(\max\{n^2, \frac{n^2m^2}{\log m}, \frac{n^3m^3}{\log(nm)}\}) = O(\frac{n^3m^3}{\log(nm)}) \tag{6.89}$$

procesorów, a jej złożoność obliczeniowa wynosi

$$O(\max\{\log m, \log n, \log(n+m)(log(nm))\}) = O(\log(n+m)(\log(nm))).\blacksquare$$

Wniosek 30 Dla metody bazującej na Tw. 30 mamy

$$s_{A_p,M}(p) = O(\frac{n^2m^2}{\log(n+m)(\log(nm)}), \tag{6.90}$$

$$\eta_{A_p,M}(p) = O(\frac{\frac{n^2m^2}{\log(n+m)\log(nm)}}{\frac{n^3m^3}{\log(nm)}}) = O(\frac{1}{nm\log(n+m)}).\ \square \tag{6.91}$$

Otoczenie NPI możemy przeglądnąć równolegle w czasie tego samego rzędu, co wyznaczenie wartości funkcji celu pojedynczego rozwiązania. W klasie omawianych metod istnieje algorytm kosztowo optymalny, patrz Tw. 29.

6.2.7 Wnioski i uwagi

Przyspieszenia, złożoności obliczeniowe oraz liczyb użytych procesorów dla zaproponowanych metod równoległego przeglądania otoczenia dla problemu $1||\sum f_i$ są zaprezentowane w tablicach C.3 oraz C.4, zamieszczonych w Dodatku C. Wyróżniono metody kosztowo optymalne.

Z kolei przyspieszenia, złożoności obliczeniowe oraz liczby użytych procesorów zaproponowanych metod równoległego przeglądania otoczenia dla problemu $F^*||C_{max}$ są zaprezentowane w tablicach C.5, C.6, zamieszczonych w Dodatku C. W tablicach C.7, C.8 i C.9 przedstawiono porównanie szybkości wzrostu złożoności obliczeniowej oraz ilości wykorzystywanych procesorów różnych metod przeglądania otoczeń problemu $F^*||C_{max}$ dla

wielkości n i m wziętych z przykładów testowych Taillarda [177]. Podobnie wyróżniono metody kosztowo optymalne.

Generalnie przeglądanie całych otoczeń API, NPI, INS jest możliwe w czasie tego samego rzędu, co wyznaczenie wartości funkcji celu pojedynczego rozwiązania, przy odpowiednio zwiększonej liczbie procesorów. Wielkość tego czasu wydaje się być naturalną wartością graniczną. Akceleratory sekwencyjne były oryginalnie projektowane z zamierzeniem zmniejszenia czasu obliczeń poprzez zmniejszenie kosztu obliczeń. Równoległe odpowiedniki akceleratora sekwencyjnego także redukują koszt obliczeń. Jednakże odmiennie, zmniejszenie czasu obliczeń następuje tylko do wielkości progowej, po czym redukcja kosztu obliczeń uzyskiwana jest poprzez zmniejszenie liczby angażowanych procesorów.

Większość wyników zaprezentowanych w tym rozdziale można rozszerzyć na model EREW PRAM z wyłącznością odczytu. Wystarczy zauważyć, że równoległy algorytm przeglądania otoczenia może, choć nie musi, wymagać wygnerowania explicite pełnego otoczenia N_x rozwiązania x w pamięci maszyny PRAM, to jest skopiowania rozwiązania x k – krotnie (np. dla otoczenia API $k = n-1$ – krotnie), a następnie odpowiedniego zmodyfikowania (dla otoczenia API wykonania wymiany pewnej pary przyległej). Za pomocą modelu EREW PRAM operację tę można wykonać w czasie $O(\log k)$, za pomocą $O(nk/\log k)$ procesorów, posługując się drzewiastą strukturą przesyłania rozwiązań od jednego do k procesorów. Zabieg ten pozbawia nas wymagania równoległego odczytu rozwiązania x przez procesory, a więc modelu CREW PRAM, na rzecz przejścia do szerszego modelu EREW. W ten sam sposób zapewnić sobie można wyłączność odczytu takich wielkości jak czasy wykonywania zadań, żądane czasy zakończenia, itp., kopiując je k – krotnie i zapewniając sobie przez to wyłączność ich odczytu przez procesory przydzielone danemu elementowi otoczenia, co wymaga czasu $O(\log k)$, oraz $O(k/\log k)$ procesorów.

Powyższa uwaga pozwala rozszerzyć na model EREW PRAM wyniki Tw. 13-18 dotyczących przeglądania otoczeń w problemach $1||\sum f_i$, oraz Tw. 19, 23, 27, dotyczących problemów $F^*||C_{max}$.

6.3 Analiza zbioru rozwiązań rozproszonych

Algorytmy populacyjne (np. GA, SS) bazują na zbiorze rozwiązań rozproszonych o niewielkiej liczności, nazywanych populacją $\mathcal{P} \subset \mathcal{X}$. Proces przetwarzania wymaga, między innymi, obliczania funkcji przystosowania rozwiązań z $\mathcal{P}$ do środowiska. Dość często funkcją oceny jest funkcja celu postawionego problemu szeregowania. Zakładając ignorowanie jakichkolwiek

cech podobieństwa pomiędzy rozwiązaniami z $\mathcal{P}$, można rozproszyć obliczenia na $|\mathcal{P}|$ niezależnych podprocesów, realizowanych równolegle. Tak też zrobiono w Tw. 19 dla problemu $F^*||\gamma$, $\gamma \in \{\sum f_i, f_{max}\}$.

W niniejszym rozdziale podjęto próbę odpowiedzi na pytanie, czy wykorzystanie faktu pokrewieństwa rozwiązań w $\mathcal{P}$ może poprawić efektywność równoległych implementacji obliczeń dla tej populacji. Naszym celem będzie zaprojektowanie algorytmu równoległego dla problemu $F^*||\gamma$ przy wykorzystaniu pokrewieństwa rozwiązań rozproszonych. Dla tego problemu populacja $\mathcal{P}$ zawiera pewien zestaw permutacji n – elementowych. Proces projektowania będzie składał się z dwóch faz: (1) ustalenie pokrewieństwa w $\mathcal{P}$, oraz (2) wykorzystanie pokrewieństwa do redukcji kosztu obliczeń.

6.3.1 Pokrewieństwo rozwiązań

Niech $\mathcal{P}$ będzie populacją o liczności k, tzn $|\mathcal{P}| = k$. Rozpoczniemy od ustalenia drzewa genealogicznego rozwiązań należących do $\mathcal{P}$. Permutację $\pi \in \mathcal{P}$ można interpretować jako *linię rodową* $\pi(1) \to \pi(2) \to \ldots \to \pi(n)$ zawierającą ciąg *potomków* tej linii w kolejnych pokoleniach $i = 1, 2, \ldots, n$. Potomka i – tego w linii rodowej π określamy trójką (a, i, π) taką, że $\pi(i) = a$. Linię π możemy przedstawić jako graf $H(\pi) = (V(\pi), E(\pi))$, gdzie zbiór wierzchołków

$$V(\pi) = \{(a, i, \pi) : \ a = \pi(i), \ i = 1, 2, \ldots, n\} \tag{6.92}$$

reprezentuje potomków w kolejnych pokoleniach, zaś zbiór łuków

$$E(\pi) = \{((a, i, \pi), (b, i+1, \pi)) : \ a = \pi(i), \ b = \pi(i+1), \ i = 1, 2, \ldots, n\}, \tag{6.93}$$

reprezentuje chronologiczną kolejność potomków. Z definicji wynika, że graf $H(\pi)$ jest łańcuchem.

Dla powiązania linii rodowych o wspólnych przodkach wprowadzimy relację identyczności ($\equiv$). Dla dwóch linii rodowych $\pi, \sigma \in \mathcal{P}$ węzły (a, i, π) oraz (a, i, σ) są identyczne [11], jeśli $\pi(j) = \sigma(j)$, $j = 1, 2, \ldots, i$; fakt ten będziemy oznaczać $(a, i, \pi) \equiv (a, i, \sigma)$ (każde dwa węzły identyczne są de facto jednym i tym samym węzłem). Zauważmy, że jeśli $(a, i, \pi) \equiv (a, i, \sigma)$, to permutacje π i σ posiadają wspólną część prefiksową $\pi(1), \pi(2), \ldots, \pi(i)$.

Drzewem genealogicznym populacji $\mathcal{P}$ jest graf

$$H(\mathcal{P}) = (V(\mathcal{P}), E(\mathcal{P})), \tag{6.94}$$

$$V(\mathcal{P}) = \bigcup_{\pi \in \mathcal{P}} V(\pi), \quad E(\mathcal{P}) = \bigcup_{\pi \in \mathcal{P}} E(\pi) \tag{6.95}$$

[11] dotyczą tej samej osoby

Rysunek 6.9: Przykładowe drzewo genealogiczne $H(\mathcal{P})$ permutacji z populacji $\mathcal{P}$.

z relacją identyczności w zbiorze $V(\mathcal{P})$. Na Rys. 6.9 pokazano drzewo genealogiczne dla populacji złożonej z 4 permutacji 9 – cio elementowych

$$\mathcal{P} = \{\pi^1 = (3,1,2,4,6,5,8,7,9), \pi^2 = (3,1,2,4,8,9,5,6,7),$$

$$\pi^3 = (3,1,2,4,8,9,5,7,6), \pi^4 = (3,1,2,4,6,5,8,9,7)\}.$$

Zauważmy, że dla większości populacji graf $H(\mathcal{P})$ jest lasem.

Skupieniem populacji $\mathcal{P}$ nazywamy wielkość

$$c(\mathcal{P}) = \frac{kn}{|V(\mathcal{P})|}. \tag{6.96}$$

Z definicji mamy $1 \leqslant c(\mathcal{P}) \leqslant k$. Wartość $c(\mathcal{P}) = 1$ odpowiada k całkowicie różnym rozwiązaniom. Wartość $c(\mathcal{P}) = k$ odpowiada k jednakowym rozwiązaniom.

Podana definicja drzewa genealogicznego nie precyzuje ani algorytmu budowy drzewa, ani tym bardziej kosztu budowy takiego drzewa. Pokażemy poniżej pewną metodę budowy drzewa genealogicznego. Dla populacji $\mathcal{P} = \{\pi^1, \pi^2, \ldots, \pi^k\}$ przyjmujemy, że permutacje są uporządkowane leksykograficznie. Porządek leksykograficzny elementów możemy uzyskać metodą równoległą na maszynie EREW PRAM w czasie $O(\log(\max(n,k)))$ za pomocą kn procesorów (patrz [56]). Budowę drzewa możemy rozpocząć od permutacji π^1. Następnie, krok po kroku, dodajemy kolejne permutacje tak, by połączyć wspólne części prefiksowe. Można to wykonać sekwencyjnie w czasie $O(kn)$. Nie będziemy zajmowali się równoległą wersją tego problemu ponieważ, jak pokażemy dalej, niezależnie od kosztu budowy drzewa prowadzenie obliczeń z jego użyciem wydaje się być mało przydatne.

6.3.2 Agregacja obliczeń

Praktyczne wykorzystanie drzewa genealogicznego pozwala na agregację obliczeń, poprzez unikanie ich powielania dla wspólnych części linii rodowych. Obliczenia wartości funkcji celu dla rozwiązań należących do $\mathcal{P}$ wymagają wyliczenia dla każdego j wielkości C_{ij}, $i = 1,2,\ldots,m$. Stąd, cały schemat obliczeń można przedstawić siecią przestrzenną, patrz Rys.6.10, otrzymaną poprzez powielenie „w głąb" drzewa genealogicznego, porównaj

Rysunek 6.10: Przestrzenna sieci obliczeń dla drzewa z Rys. 6.9 dla problemu $F^*||\gamma$.

Rysunek 6.11: Drzewo genealogiczne rozwiązań generowanych w otoczeniu API z permutacji naturalnej.

z Rys. 6.9 (kierunek prowadzenia obliczeń zaznaczono strzałkami). Prowadzenie obliczeń równoległych na sieci przestrzennej jest niewątpliwie nowym jakościowo podejściem. Pokażemy dalej, że w omawianym przypadku charakteryzuje się ono względnie niską efektywnością.

Łatwo zauważyć, że posługując się przestrzennym schematem zależności pomiędzy C_{ij} można „zaoszczędzić" na obliczeniach poprzez pominięcie liczby węzłów równej

$$mkn - m|V(\mathcal{P})| = (1 - \frac{1}{c(\mathcal{P})})mkn = \alpha(\mathcal{P})mkn. \tag{6.97}$$

Aby ocenić korzyści wynikające z tego podejścia należałoby dokonać oceny wartości współczynnika $\alpha(\mathcal{P})$, mającego sens frakcji liczby pomijanych węzłów. Oczywiście, dla $c(\mathcal{P}) = 1$ nic nie oszczędzamy.

Rozpocznijmy analizę od populacji pochodzących z otoczeń regularnych API, NPI, INS. Mimo, iż znane są dla tych otoczeń akceleratory zakładamy, że prowadzimy obliczenia dla klasy funkcji kryterialnych, dla których akcelerator nie ma zastosowania. Przykłady drzew (lasów) genealogicznych dla otoczeń API, NPI, INS pokazano na Rys. 6.11, 6.12 oraz 6.13. Dla uproszczenia rachunków, do wszystkich rozważanych populacji $\mathcal{P}$ włączyliśmy permutację generującą otoczenie.

Z analizy regularnej struktury drzewa genealogicznego otoczenia API (patrz Rys. 6.11) mamy $k = n$ oraz

$$\alpha(\mathcal{P}) = \frac{(n-1)(n-2)}{2n^2}. \tag{6.98}$$

Ponieważ $\alpha(\mathcal{P})$ jest funkcją rosnącą względem n, zatem

$$\alpha(\mathcal{P}) \leqslant \lim_{n\to\infty} \alpha(\mathcal{P}) = \frac{1}{2}, \tag{6.99}$$

Rysunek 6.12: Drzewo genealogiczne rozwiązań generowanych w otoczeniu NPI z permutacji naturalnej.

Rysunek 6.13: Drzewo genealogiczne rozwiązań generowanych w otoczeniu INS z permutacji naturalnej.

co implikuje $c(\mathcal{P}) \leqslant 2$, niezależnie od n. Oznacza to, że maksymalna teoretycznie osiągalna redukcja obliczeń dla otoczenia API nie zależy od k (jest stała) i nie może przekroczyć połowy kosztu ponoszonego w Tw. 19.

Wyprowadzenie analogicznych rezultatów dla otoczeń NPI i INS prowadzi do dużo bardziej skomplikowanych wzorów finalnych, które pominiemy, przytaczając tylko rezultat otrzymany poprzez analizę asymptotyczną odpowiednich wyrażeń. Dla otoczenia NPI mamy $\alpha(\mathcal{P}) \to \frac{1}{3}$, zaś dla otoczenia INS otrzymujemy $\alpha(\mathcal{P}) \to \frac{1}{3}$.

Dla populacji nieregularnych, na przykład występujących w metodach genetycznych, nie jest możliwe oszacowanie $\alpha(\mathcal{P})$ na drodze teoretycznej, bowiem struktura $\mathcal{P}$ jest nieznana a priori. Stąd wartości współczynnika $\alpha(\mathcal{P})$ oszacowano na drodze eksperymentalnej. Dla populacji występującej w algorytmie genetycznym z Rozdz. 7.3.5 rozwiązującym problem $F^*||\sum C_i$, wartość $\alpha(\mathcal{P})$ kształtuje się na poziomie od 5% do 10%.

Ostatecznie w czterech analizowanych przypadkach zysk z agregacji obliczeń był asymptotycznie stały. Stąd koszt równoległego przetwarzania pokrewnych rozwiązań rozproszonych będzie różnił się co najwyżej o stałą, od kosztu obliczeń równoległych wykonanych według schematu pomijającego fakt pokrewieństwa rozwiązań. Rząd wielkości złożoności obliczeniowej pozostanie taki sam. Biorąc pod uwagę dodatkowy koszt związany z sortowaniem leksykograficznym populacji $\mathcal{P}$ oraz budową drzewa genealogicznego, złożoność obliczeniowa metody wykorzystującej agregację obliczeń może być nawet gorsza od złożoności metody „naiwnej", pomijającej fakt pokrewieństwa rozwiązań z $\mathcal{P}$.

6.3.3 Wnioski i uwagi

Generalnie, wykorzystanie pokrewieństwa rozwiązań w problemach $F^*||\gamma$ nie wnosi *jakościowych* popraw w stosunku do podejścia ignorującego podobieństwa między rozwiązaniami. Faktycznie, wykorzystanie pokrewieństwa nie ma wpływu na złożoność obliczeniową (w pewnych przypadkach może ją zwiększać), oferując w zamian zmniejszenie liczby procesorów z dokładnością do stałej. Ostatecznie, liczba angażowanych procesorów jest tego samego rzędu co powoduje, że zyski z takiego podejścia są niewspółmiernie małe w stosunku do stopnia skomplikowania organizacji procesu obliczeniowego.

7

Poszukiwanie wielowątkowe

W rozdziale niniejszym opisane zostaną algorytmy równoległe, stosujące strategię rozproszenia procesu poszukiwań optimum globalnego w przestrzeni rozwiązań poprzez zastosowanie współbieżnie działających wątków poszukiwań. Algorytmy te mogą być stosowane do obliczeń za pomocą niejednorodnych klastrów jak i sieci rozproszonych. Wątki obliczeniowe mogą być niezależne (nie wymieniać informacji), jak i kooperujące. Analizowane będą wielowątkowe równoległe algorytmy poszukiwań: z zabronieniami (*parallel tabu search*, pTS), metodą symulowanego wyżarzania (*parallel simulated annealing*, pSA), metodą poszukiwania ewolucyjnego (*parallel genetic algorithm*, pGA). Przedstawiony zostanie aktualny stan badań oparty na przykładach zastosowań z literatury, oraz przedstawione zostaną przykłady implementacji algorytmów poszukiwań wielowątkowych zastosowanych do rozwiązywania problemów szeregowania zadań.

7.1 Równoległe metody tabu

Jednościeżkowe implementacje równoległej metody tabu w zastosowaniu do problemów optymalizacji kombinatorycznej bazujące na dekompozycji otoczenia znaleźć można w pracach [40,41,53,72,150–153,175]. W pracach tych stosowane są różnorodne środowiska i maszyny obliczeń równoległych takie jak sieci komputerów Sun SPARC, sieci transputerów, IBM SP-1, Silicon Graphics Origin 2000, Meiko z 16 transputerami T-800, Connection Machine CM-2. Czasy obliczeń są krótsze niż w przypadku ich sekwencyjnych odpowiedników, otrzymywane jest też prawie-liniowe przyspieszenie.

Badeau i inni [15] zaimplementowali równoległy algorytm tabu dla zagadnienia ustalania tras pojazdów (*vehicle routing*), łącząc strategie współbieżnego jedno- i wielościeżkowego przeglądania przestrzeni rozwiązań. Na

początku, procesor główny dystybuuje dane problemu pomiędzy procesory podrzędne. Każdy z nich konstruuje swoje rozwiązanie startowe i zwraca je do procesora głównego, który łączy otrzymane rozwiązania częściowe (tzn. trasy (*routes*)) generując jedno rozwiązanie początkowe, składające się ze zbioru tras. Tak powstałe rozwiązanie startowe jest rozsyłane do procesorów podrzędnych, które rozpoczynają dekompozycję problemu na podproblemy rozwiązywane przy pomocy sekwencyjnych algorytmów tabu . Rozwiązania podproblemów otrzymane współbieżnie działającymi algorytmami tabu są łączone w rozwiązanie problemu i odsyłane do procesora głównego, który porównuje wartości kryteriów dla poprzedniego i nowego rozwiązania, aktualizując je w miarę potrzeby. Proces jest kontynuowany aż do osiągnięcia kryterium zatrzymania. Obliczenia były wykonywane przez autorów na 5 stacjach roboczych Sun SPARC połączonych siecią.

Wielościeżkowy równoległy algorytm tabu został zaproponowany przez Taillarda [176] do rozwiązania problemu gniazdowego (*job shop*). W implementacji tej użyta została technika równoległych niezależnych wątków poszukiwań.

Model kooperujący wielościeżkowy jest najbardziej obiecującą metodą zrównoleglenia algorytmu tabu z uwagi na efektywność algorytmu i jakość uzyskiwanych rozwiązań. W większości implementacji procesory startują z różnych rozwiązań i wykonują swoje procesy poszukiwań za pomocą sekwencyjnych algorytmów tabu. Wymiana najlepszych rozwiązań jest dokonywana wyłącznie poprzez centralną pulę rozwiązań przechowywaną w pamięci współdzielonej.

Inną możliwością kooperacji jest użycie metody ścieżek łączących (*path relinking*). Ścieżki łączyłyby aktualne rozwiązania z nowymi znalezionymi przez współbieżnie działające wątki poszukiwań. Strategia ta, zastosowana jednak nie dla algorytmu tabu, ale dla metody GRASP (*greedy randomized adaptive search procedure*) przedstawiona została w pracach [7, 37].

Crainic, Gendreau oraz Crainic, Toulouse i Gendreau [50, 52] zaimplementowali kilka równoległych wersji algorytmu tabu, jako niezależne wątki poszukiwań a także kooperujące poszukiwanie wielościeżkowe porównując wydajności różnych technik zrównoleglania. Są oni także autorami bardziej szczegółowej klasyfikacji równoległych algorytmów tabu, w porównaniu do klasyfikacji Voss'a [185]. Klasyfikacje te przedstawione zostaną w następnym rozdziale.

7.1.1 Klasyfikacja równoległych algorytmów tabu

Pierwszą próbę sklasyfikowania równoległych algorytmów tabu podjął Voss [185] nawiązując do klasycznego podziału algorytmów równoległych Flynn'a

[68] na modele SIMD, MIMD, MISD i SISD. Klasyfikacja Voss'a umiejscowiona jest niejako „obok" ogólnej klasyfikacji równoległych metod metaheurystycznych biorących za przedmiot podziału ilość ścieżek, ziarnistość czy kooperację. Voss zaproponował podział równoległych algorytmów tabu na cztery kategorie ze względu to czy równoległe wątki poszukiwań startują z tych samych czy różnych rozwiązań startowych i czy stosują te same czy różne strategie przeszukiwań. Klasyfikacja ta została następnie rozszerzona przez Crainic'a, Toulouse i Gendreau [52,54] i w takiej formie przedstawiam ją poniżej:

- SPSS (*Single (Initial) Point Single Strategy*) – jedno rozwiązanie początkowe, jedna strategia przeszukiwania; model pozwalający na zrównoleglenie jedynie na najniższym poziomie, np. liczenia funkcji celu lub równoległego przeglądania otoczenia (zagadnienia opisane w rozdziale 6 dotyczącym poszukiwań jednowątkowych),
- SPDS (*Single (Initial) Point Different Strategies*) – wszystkie procesory startują z tego samego rozwiązania początkowego, ale stosują różne strategie przeszukiwania (np. różne długości listy tabu, inne elementy pamiętane na liście tabu, itp.),
- MPSS (*Multiple (Initial) Point Single Strategy*) – procesory rozpoczynają działanie z różnych rozwiązań początkowych stosując tę samą strategię przeszukiwania,
- MPDS (*Multiple (Initial) Point Different Strategies*) – procesory rozpoczynają działanie z różnych rozwiązań początkowych stosując różne strategie przeszukiwania; jest to najszersza klasa obejmujące wszystkie poprzednie jako jej szczególne przypadki.

Klasyfikacja Crainic'a, Toulouse i Gendreau oprócz podziału algorytmów równoległych tabu ze względu na punkty startowe i rodzaje strategii wprowadza jeszcze dwie klasyfikacje: (1) ze względu na liczbę procesorów sprawujących kontrolę nad działaniem algorytmu i (2) sposób kontroli działania i typu komunikacji. Ze względu na liczbę procesorów kontrolujących (1) wyróżnione są dwa modele:

- *1-control* – wyróżniony procesor centralny kontroluje działania algorytmu,
- *p-control* – kontrola działania jest rozproszona pomiędzy p procesorów działających współbieżnie.

Sposób kontroli i typ komunikacji (2) rozróżniony jest poprzez podział na klasy:

- synchronizacja sztywna (*rigid synchronization*),
- synchronizacja z rozproszoną bazą wiedzy (*knowledge synchronization*),
- zespołowa (*collegial*),
- zespołowa z rozproszoną bazą wiedzy (*knowledge collegial*).

Kontrola współbieżnego wykonania programu równoległego może być powierzona jednemu procesorowi, zwykle nazywanemu *master*, *server* lub *main processor*, lub też może być rozdzielona pomiędzy kilka procesorów. Stąd, zdefiniować można dwie kategorie.

Pierwsza kategoria, nazywany *1-control*, nawiązuje do podejścia sekwencyjnego. W implementacji równoległej jeden wyróżniony procesor wykonuje algorytm tabu, powierzając część pracy innym procesorom. Procesor centralny rozsyła zadania obliczeniowe i zbiera wyniki decydując kiedy zakończyć przeszukiwanie. Zadania obliczeniowe procesorów podrzędnych mogą mieć charakter przeglądania części otoczenia badanego punktu przestrzeni, lub też tylko obliczania wartości funkcji celu.

Drugą kategorię stanowi model, w którym kontrola jest rozproszona pomiędzy $p, (p > 1)$ procesorów działających współbieżnie - model ten określany przez *p-control*. Każdy z procesorów wykonuje swój własny algorytm tabu przeszukiwania przestrzeni rozwiązań komunikując się z pozostałymi procesorami - przy czym nie ma wyróżnionego procesora sterującego (centralnego). Cały proces przeszukiwania zatrzymuje się, gdy zakończą się wszystkie procedury przeszukujące wykonywane współbieżnie na procesorach. Koordynacja wymiany informacji oraz zaprojektowanie komunikacji tak, by wymagana informacja była dostępna, gdy jest potrzebna, jest kluczowym zagadnieniem w tym modelu i wymaga sprecyzowania typu kontroli i koordynacji.

7.1.2 Problemu przepływowy. Algorytm pTS

Równoległy algorytm tabu w wersji wielościeżkowej, asynchronicznej, został zaimplementowany dla przepływowego problemu szeregowania zadań z kryterium minimalizacji maksymalnego czasu zakończenia zadań $F^*||C_{max}$.

W pracy [73] udowodniono, że jeżeli liczba maszyn wynosi trzy lub więcej, wówczas problem jest silnie NP-trudny. Algorytmy dokładne jego rozwiązywania oparte na schemacie podziału i ograniczeń przedstawiono między innymi w pracach [79, 92, 108]. Pozwalają jednak one na rozwiązywanie, w rozsądnym czasie, przykładów o rozmiarze nie większym niż 20 zadań i 5 maszyn. W ostatnich latach opublikowano wiele algorytmów

rozwiązywania omawianego problemu opartych na metodach lokalnej optymalizacji, w tym: na metodzie tabu search [141, 156, 174], symulowanego wyżarzania [144, 146] oraz genetycznej [95, 157]. Generalnie, jakość wyznaczanych przez te algorytmy rozwiązań zależy w dużym stopniu od czasu ich działania. Im ten czas jest większy, tym przeglądany jest (bezpośrednio) większy podzbiór zbioru rozwiązań dopuszczalnych. Zwiększa to szansę na znalezienie dobrego rozwiązania. Zastosowanie metod programowania równoległego pozwala czas ten znacznie skrócić.

Własności problemu

Przy projektowaniu algorytmu dla problemu $F^*||C_{max}$ odwołujemy się do jego modelu grafowego. Niech $G(\pi)$ będzie grafem siatkowym zdefiniowanym w rozdziale 2.2. Najdłuższa droga w grafie $G(\pi)$, tj. najdłuższa droga z wierzchołka (1,1) do (m, n) jest zwana *ścieżką krytyczną*. Jej długość jest równa $C_{\max}(\pi)$. Ścieżka ta może być reprezentowana przez ciąg (j_0,j_1,....j_m). Ciąg ten określa fragmenty odpowiadające ścieżkom poziomym (i, j_i-1),$(i,j_{i-1}+1)$,...,(i,j_i), dla i= 1,2,...,m, oraz krawędziom pionowym $((i,j_i)$,$(i+1,j_i)$, dla $i = 1, 2, \ldots, m-1$. Niech $u = (u_0, u_1,...u_m)$, gdzie $u_0 = 1$ i $u_m = n$, będzie ciągiem, który definiuje ścieżkę krytyczną w $G(\pi)$, to znaczy ścieżka krytyczna jest postaci

$$(1,u_0),..., (1,u_1), (2,u_1),..., (2,u_2),... (m,u_{m-1}),..., (m,u_m).$$

Dla każdej maszyny k, podścieżka pozioma (k, u_{k-1}), $(k, u_{k-1}+1)$,..., (k, u_k), odpowiada ciągowi zadań $\pi(u_{k-1})$, $\pi(u_{k-1}+1)$,... $\pi(u_k)$ wykonywanych kolejno na maszynie $k-$tej, k = 1,2,...,m. Ciąg ten będziemy nazywać *blokiem* i będziemy oznaczali przez B_k, przy czym:

- $B_k = (\pi(f_k), \pi(f_k+1), ..., \pi(l_k-1), \pi(l_k))$, $f_k \leqslant l_k$, f_1=1, $l_m = n$, oraz $\pi(l_k) = \pi(f_{k+1}), \quad k = 1, 2, ..., m-1$,
- B_k zawiera operacje wykonywane na tej samej maszynie, $k = 1, 2, \ldots, m$,
- dwa sąsiednie bloki B_k, B_{k+1} zawierają dokładnie jedno wspólne zadanie, $\pi(l_k) = \pi(f_{k+1})$.

Blok jest więc maksymalnym podzbiorem operacji z drogi krytycznej, wykonywanych na tej samej maszynie. Operacje $\pi(f_k)$ i $\pi(l_k)$ w bloku B_k zwane są odpowiednio *pierwszą* i *ostatnią*.

Poniższe twierdzenie pochodzące z [79] jest podstawą konstrukcji wielu algorytmów, zarówno dokładnych jak i przybliżonych, rozwiązywania rozpatrywanego problemu.

Twierdzenie 31 *Niech B_k, k =1, 2,...,m będą blokami w grafie $G(\pi)$. Jeśli graf $G(\omega)$ został wygenerowany z $G(\pi)$ przez przestawienie pewnego zadania w permutacji π oraz $C_{max}(\omega) < C_{max}(\pi)$, wówczas w grafie $G(\omega)$ co najmniej jedno zadanie $j \in B_k$ poprzedza $\pi(f_k)$, dla pewnego k = 2,..., m, lub co najmniej jedno zadanie $j \in B_k$ występuje za $\pi(l_k)$, dla pewnego k = 1, 2,...,$m-1$.*

Twierdzenie to jest podstawą metody przestawiania zadań (generowania permutacji), umożliwiającej otrzymanie lepszej permutacji (grafu $G(\pi)$ o mniejszej długości drogi krytycznej) czyli takiej, że $C_{max}(\omega) < C_{max}(\pi)$.

Niech $B_k^f = B_k \backslash \{\pi(f_k)\}$ oraz $B_k^l = B_k \backslash \{\pi(l_k)\}$ będą podblokami zadań z k-tego bloku, odpowiednio bez pierwszego i ostatniego zadania. Generując permutację ω z permutacji π taką, że $C_{max}(\omega) < C_{max}(\pi)$ będziemy przestawiali zadania z B_k^f przed pierwsze zadanie tego bloku $\pi(f_k)$, oraz zadania z bloku B_k^l za ostatnie zadanie $\pi(l_k)$, k=1,2,...,m.

Sekwencyjny algorytm tabu search

Obecnie najlepsze znane w literaturze *efektywne* algorytmy rozwiązywania permutacyjnego problemu przepływowego są oparte na metodzie tabu search [80, 141]. Generalnie, metoda polega na iteracyjnym polepszaniu bieżącego rozwiązania poprzez lokalne przeszukiwanie. Rozpoczyna się od pewnego rozwiązania początkowego (startowego). Następnie generuje się jego otoczenie (sąsiedztwo) oraz wyznacza najlepsze rozwiązanie z tego otoczenia, które przyjmuje się za rozwiązanie startowe w następnej iteracji. Dopuszcza się możliwość zwiększania wartości funkcji celu (przy wyznaczaniu nowego rozwiązania startowego), aby w ten sposób zwiększyć szansę na osiągnięcie minimum globalnego. Takie ruchy „w górę" należy jednak w pewien sposób kontrolować, ponieważ w przeciwnym razie po osiągnięciu minimum lokalnego nastąpiłby szybki do niego powrót. Aby zapobiec generowaniu w nowych iteracjach rozwiązań niedawno rozpatrywanych (powstawaniu cykli), zapamiętuje się je (ich atrybuty) na liście rozwiązań zakazanych, tzw. liście tabu (pamięć krótkoterminowa, [140]).

Algorytm sTS

Niech $\pi \in \Pi$ będzie dowolną permutacją, ΛT listą tabu, a π^* najlepszym do tej pory znalezionym rozwiązaniem (na początek przyjmujemy za π^* permutację π).

Krok 1. Wyznaczyć otoczenie N_π permutacji π, nie zawierające elementów zabronionych przez listę ΛT;

Krok 2. Znaleźć permutację $\delta \in \mathcal{N}_\pi$ taką, że:
$C_{max}\ (\delta) = \min\{C_{max}\ (\beta)\colon \beta \in N_\pi\}$;

Krok 3. Jeśli $C_{max}(\delta) < C_{max}(\pi^*)$, to $\pi^* \leftarrow \delta$;
Umieść atrybuty δ na liście ΛT;
$\pi \leftarrow \delta$;

Krok 4. Jeżeli zachodzi **Warunek_Zakończenia**, to STOP;
inaczej idź do Kroku 1.

Sposób wyznaczania otoczenia, organizacja listy tabu i warunek zatrzymania są podstawowymi elementami metody. Niech B_k, będzie k - tym blokiem w permutacji π, B_k^f i B_k^l - blokami wewnętrznymi, $k = 1, 2, \ldots, m$. Dla zadania $j \in B_k^f$ niech $N_k^f(j)$ będzie zbiorem permutacji wygenerowanych przez wstawienie zadania j na początek bloku B_k (przed $\pi(f_k)$ - pierwsze zadanie bloku). Analogicznie, dla zadania $j \in B_k^f$ niech $N_k^l(j)$ będzie zbiorem permutacji wygenerowanych przez wstawienie zadania j na koniec bloku B_k (za $\pi(l_k)$ - ostatnie zadanie bloku). Definiujemy otoczenie N_π permutacji π przez:

$$N_\pi = \bigcup_{k=1}^{m} \bigcup_{j \in B_k} (N_k^f(j) \cup N_k^l(j)). \tag{7.1}$$

Lista tabu jest cykliczna i zawiera atrybuty pewnej liczby ostatnio rozpatrywanych rozwiązań, oraz ma z góry określoną maksymalną długość. Jeżeli lista nie jest pełna, to nowy element jest do niej dopisywany. Jeśli natomiast dopisanie elementu spowodowałoby, że lista przekroczyłaby maksymalną długość, wówczas usuwany jest z niej element pozostający na liście najdłużej.

Warunek zakończenia - algorytm kończy działanie po wykonaniu z góry określonej liczby *Max_iter* iteracji.

Dodatkowo, w algorytmie zastosowano mechanizm powrotu (*lista tabu długoterminowa*, patrz [140]. Pamiętana jest pewna liczba najlepszych (perspektywicznych) rozwiązań startowych takich, z których generowane permutacje dawały poprawę wartości funkcji celu. Po wykonaniu pewnej liczby iteracji, bez poprawy rozwiązania, następuje powrót (skok) do ostatniego pamiętanego na tej liście rozwiązania.

Złożoność obliczeniowa algorytmu opartego na metodzie tabu search zależy od sposobu realizacji jego elementów, tj. metody wyznaczania sąsiedztwa, organizacji oraz długości listy tabu, sposobu wyliczania wartości funkcji celu oraz warunku zakończenia.

Algorytm równoległy pTS

Równoległa wersja algorytmu została zaprojektowana dla modelu MIMD komputera równoległego bez pamięci współdzielonej. Do rozproszenia procesów poszukujących najlepszego rozwiązania posłużyła lista powrotów, wspólna dla wszystkich procesorów, przechowywana i aktualizowana przez specjalnie w tym celu wyselekcjonowany procesor. Ten sam procesor przechowuje także najlepsze znalezione do tej pory rozwiązanie.

Algorytm pTS

parfor $j = 1, 2, \ldots, p$ (dla każdego procesora)
begin

$\pi \in \Pi$ - rozwiązanie początkowe dla pierwszego procesora (j=1); pozostałe procesory pobierają rozwiązanie początkowe z listy powrotów (gdy tylko się pojawią nowe rozwiązania);

π^* – najlepsze znane rozwiązanie (zmienna lokalna);

T – lista tabu (lokalna);

ε – parametr używany w mechanizmie powrotów;

$\pi^* \leftarrow \pi$;

Krok 1. Wyznaczyć otoczenie N_π permutacji π, wyłączyć z otoczenia elementy zabronionych przez listę T oprócz $\beta \in N_\pi$ takich że $C_{max}(\beta) < C_{max}(\pi^*)$;

Krok 2. for $\beta \in N_\pi$ takich że $\frac{C_{\max}(\beta) - C_{\max}(\pi^*)}{C_{\max}(\pi^*)} < \varepsilon$

begin

Dodać β wraz z listą ruchów zabronionych T do listy powrotów poprzez wysłanie β oraz T do procesora nadrzędnego przechowującego dane współdzielone;

end;

Krok 3. Znaleźć permutację $\delta \in N_\pi$ taką, że:

$C_{max}\ (\delta) = \min\{C_{max}\ (\beta)\colon \beta \in N_\pi\}$;

Krok 4. if $C_{max}(\delta) < C_{max}(\pi^*)$ **then**

begin

$\pi^* \leftarrow \delta$;

Wysłać π^* i $C_{max}(\pi^*)$ do procesora przechowującego dane;

Pobrać aktualne wartości π^* oraz $C_{max}\ (\pi^*)$ (jeśli są inne od posiadanych lokalnie);

end;

Umieścić atrybuty δ na liście T;

$\pi \leftarrow \delta$;

Krok 5. if (w ciągu ostatnich L iteracji nie znaleziono lepszej permutacji niż π^*) **then**

begin

Pobrać nową permutację π z listy powrotów procesora przechowującego dane współdzielone;
Pobrać listę tabu T dla permutacji π, od procesora przechowującego dane;

end;

Krok 6. if (*Warunek_Zakończenia*) **then** STOP,

else go to Krok 1.

end of parfor.

Algorytm został tak zaprojektowany, aby komunikacja z procesorem przechowującym dane współdzielone nie była zbyt czasochłonna – wysyłanie i pobieranie danych odbywa się względnie rzadko, szczególnie w późniejszej fazie działania algorytmu (po znalezieniu rozwiązania bliskiego optymalnemu).

Eksperymenty obliczeniowe

Cel eksperymentów. Eksperymenty obliczeniowe przeprowadzono w celu zbadania, jak zmienia się jakość otrzymywanych rozwiązań wraz ze wzrostem ilości równoległych wątków poszukiwań (procesorów), przy ustalonym koszcie obliczeń.

Implementacja. Algorytm został zaimplementowane w języku Ada95 i uruchomione na komputerze Sun Enterprise 4x400 MHz działającym pod systemem operacyjnym Solaris 7. Zadania języka Ada95 były uruchamiane równolegle jako wątki systemowe. Użyte zostały następujące wartości parametrów strojących:

$\|T\|$	=	10	–	długość listy tabu,
ε	=	0.25%	–	stała używana przez mechanizm powrotów
L	=	10	–	liczba iteracji bez poprawy rozwiązania,
p	=	$1, 2, 4$	–	liczba procesorów,

dla 2000 iteracji:

Max_iter	=	2000	–	dla implementacji 1 – procesorowej
	=	1000	–	dla 2 procesorów,
	=	500	–	dla 4 procesorów.

dla 4000 iteracji:

Max_iter = 4000 – dla implementacji 1 – procesorowej
= 2000 – dla 2 procesorów,
= 1000 – dla 4 procesorów.

Liczba iteracji algorytmu wynosiła 2000 i 4000 dla implementacji jednoprocesorowej i odpowiednio $(2000/p)$ oraz $(4000/p)$, dla każdego procesora w przypadku implementacji p - procesorowej. Zapewniło to porównywalny koszt wykonania algorytmów równoległych oraz algorytmu sekwencyjnego – algorytm równoległy względnie rzadko znajduje nowe lepsze rozwiązanie, rozsyłane do pozostałych wątków poszukiwań, przez co częstotliwość komunikacji jest mała i nie ma wpływu na złożoność algorytmu równoległego (w ciągu całego czasu działania komunikacja następuje najwyżej kilka – kilkanaście razy, przy kilku tysiącach iteracji).

Przykłady testowe. Algorytm był testowany na wielu przykładach o różnych rozmiarach i poziomie trudności:

1. 50 przykładach o 12 rozmiarach z 100,. . . ,500 operacjami ($n \times m$=20×5, 20×10, 20×20, 50×5, 50×10, 50×20, 100×5) opublikowanych przez Taillarda [177], (OR-Library: [145]).

2. 100 przykładach o 5 rozmiarach z 2000,. . . ,10000 operacjami (200×5, 200×10, 200×20, 200×25, 200×50).

Wyniki obliczeń. W Tab. 7.1 i 7.2 przedstawione są wyniki porównawcze rozwiązań opisanego w poprzednim rozdziale algorytmu równoległego oraz najlepszego obecnie znanego w literaturze algorytmu konstrukcyjnego NEH (Nawaz, Enscore, Ham [139]). Mierzono procentowy błąd względny, zdefiniowany wzorem (3.5), rozwiązań x^A generowanych przez badane algorytmy, przyjmując za rozwiązania referencyjne x^* najlepsze znane wyniki pobrane z OR-Library [145]. Pogrubiono najlepsze wyniki.

Jak można zauważyć, w przypadku algorytmu równoległego wyniki są znacznie lepsze. Możemy stwierdzić że dla p procesorów, w pewnym sensie, przyspieszenie jest nawet większe od p (algorytm równoległy 4-procesorowy potrzebuje *mniej* niż 500 iteracji aby osiągnąć wynik podobny, jak algorytm sekwencyjny przy 2,000 iteracji). Algorytm równoległy wykazał także znaczną poprawę wyników po zwiększeniu liczby iteracji z 2,000 do 4,000. Wskazuje to na korzystną tendencję zbieżności do rozwiązania optymalnego. Cechy tej nie wykazuje tak wyraźnie sekwencyjna wersja algorytmu.

Wyniki algorytmu równoległego zamieszone w Tab. 7.1 i 7.2 są zdecydowanie najlepsze dla dużych wartości współczynnika n/m (20/5, 50/5,

$n \times m$	1 procesor	2 procesory	4 procesory	NEH
20×5	0,96%	0,67%	**0,45%**	2,87%
20×10	3,03%	**1,28%**	1,41%	4,74%
20×20	2,02%	1,10%	**1,05%**	3,69%
50×5	0,33%	**0,08%**	0,15%	0,89%
50×10	2,86%	2,35%	**2,25%**	4,53%
50×20	3,71%	3,52%	**3,19%**	5,24%
100×5	0,25%	0,16%	**0,12%**	0,46%
średnia	1,88%	1,31%	**1,23%**	3,20%

Tablica 7.1: Procentowy błąd względny algorytmu tabu search i NEH względem najlepszych rozwiązań, dla 2000 iteracji.

$n \times m$	1 procesor	2 procesory	4 procesory	NEH
20×5	1,96%	0,43%	**0,42%**	2,87%
20×10	2,84%	**1,09%**	1,30%	4,74%
20×20	1,82%	0,62%	**0,62%**	3,69%
50×5	0,33%	**0,08%**	0,14%	0,89%
50×10	2,81%	2,10%	**1,81%**	4,53%
50×20	3,49%	3,02%	**2,86%**	5,24%
100×5	0,25%	0,16%	**0,09%**	0,46%
średnia	1,79%	1,07%	**1,03%**	3,20%

Tablica 7.2: Błąd algorytmu tabu search i NEH względem najlepszych rozwiązań, dla 4000 iteracji.

100/5). W tych przypadkach długości bloków są bardzo korzystne. Generalnie, poprawa wyników algorytmu równoległego 4-procesorowego w stosunku do rozwiązań algorytmu sekwencyjnego wyniosła 35%, dla 2000 iteracji. Algorytm równoległy wykonywał taką samą ilość iteracji jak algorytm sekwencyjny (licząc ilość iteracji algorytmu jako sumę iteracji wykonanej przez każdy procesor). Po zwiększeniu liczby iteracji z 2000 do 4000, współczynnik poprawy jakości rozwiązań algorytmu równoległego względem algorytmu sekwencyjnego wzrósł do 42%.

7.1.3 Wnioski i uwagi

Przedstawiono konstrukcję algorytmu rozwiązywania permutacyjnego problemu przepływowego opartą na równoległej metodzie tabu search z wielościeżkowymi kooperującymi wątkami przeglądania przestrzeni rozwiązań. Przy implementacji algorytmu zastosowano ideę skoku powrotnego. Pozwoliło to na znacznie szybsze opuszczanie obszarów zbioru rozwiązań, które były mało perspektywiczne (nie rokowały poprawy wartości funkcji celu). Wyniki obliczeniowe wskazują, że algorytm równoległy jest znacznie efektywniejszy od sekwencyjnego. Otrzymane wyniki (po niewielkiej liczbie iteracji) tylko nieznacznie różnią się od najlepszych obecnie znanych.

7.2 Równoległe metody symulowanego wyżarzania

Większość wczesnych implementacji równoległego algorytmu symulowanego wyżarzania oparta była na strategii jednościeżkowego (*single-walk*) przeglądania przestrzeni rozwiązań bazującej na *przyspieszeniu ruchu* oraz wykonywaniu równolegle kilku ruchów (*ruchach równoległych*). W pierwszym przypadku wykonanie każdego ruchu jest dzielone na kilka pod-zadań wykonywanych równolegle: wybór możliwych ruchów, obliczenia wartości funkcji celu, akceptacja bądź odrzucenie ruchu, aktualizacja zmiennych globalnych. Kravitz i Rutenbar [104, 163] opisali jeden z pierwszych przykładów zrównoleglenia algorytmu symulowanego wyżarzania opartego na tej technice. W drugim przypadku (ruchów równoległych) każdy z procesorów generuje, szacuje i akceptuje (bądź odrzuca) swoją porcję ruchów. Synchronizacja może być wykonywana co jedną lub co kilka iteracji. Technika ta jest używana zwykle w połączeniu z dzieleniem otoczenia na (najlepiej rozłączne) podzbiory przydzielane do procesorów. Allwright i Carpenter zastosowali wyżej wymieniony model dla problemu komiwojażera [11], podobnie Felten, Karlin i Otto [65] dla modelu MIMD procesorów połączonych siecią

o topologii hiperkostki.

Wariant metody jednościeżkowej został przedstawiony w pracy [188] dla zagadnienia przydziału zadań (*task assignment problem*). Podstawowa idea polegała na równoczesnym rozwijaniu dwóch możliwych scenariuszy: po akceptacji ruchu oraz po jego odrzuceniu, przed podjęciem ostatecznej decyzji dotyczącej poprzedniego ruchu. Niestety takie podejście dla badanego problemu nie dało zadowalających wyników.

Wielościeżkowe (*multiple-walk*) przeszukiwanie przestrzeni rozwiązań z rzadką komunikacja (model gruboziarnisty) to następny typ zrównoleglenia metody symulowanego wyżarzania. Metoda teoretycznie jest oparta na łańcuchach Markowa (*multiple Markov chains*). Synchroniczną implementację takiego algorytmu równoległego zaprezentowali Aarts i inni [1]. Komunikacja dotycząca wymiany informacji (np. o najlepszym znalezionym rozwiązaniu) jest dokonywana w każdym punkcie synchronizacji, a długość łańcucha Markowa (czyli ilość iteracji) jest redukowana globalnie. Lee i Lee [113–115] zaprezentowali interpretację powyższej strategii dla modelu z asynchroniczną komunikacją. Odpowiednia wymiana informacji pomiędzy współbieżnymi wątkami przeszukującymi przestrzeń rozwiązań jest w tym wypadku zagadnieniem kluczowym dla efektywności takiego podejścia.

Inna implementację podejścia wielościeżkowego, bazująca na wykonywaniu równolegle kooperujących wątków symulowanego wyżarzania z różnymi temperaturami, została zaprezentowana przez Miki i inni [132]. Porównanie wyżej cytowanych podejść znaleźć można w pracy [43], a także [14, 51, 82], gdzie znajdują się opisy hybrydowych równoległych algorytmów symulowanego wyżarzania.

7.2.1 Algorytm równoległy pSA

Schemat dwóch algorytmów równoległych symulowanego wyżarzania został zaprojektowany dla maszyny MIMD z procesorami bez pamięci współdzielonej oraz przy założeniu, że czas komunikacji pomiędzy procesorami jest o wiele dłuższy od czasu komunikacji wewnątrz pojedynczego procesora. Zastosowano model scentralizowany z procesorem centralnym magazynującym dane oraz procesorami podrzędnymi kooperującymi, równolegle przeglądającymi przestrzeń rozwiązań.

Niech $\pi \in \Pi$ będzie dowolną permutacją (startową), N_π jej otoczeniem, a π^* najlepszym do tej pory znalezionym rozwiązaniem (na początek przyjmujemy za π^* permutację π). Przez $\varphi(t)$ oznaczmy schemat chłodzenia (t – temperatura) oraz przez $\Psi(\pi,\varphi(t))$ funkcję akceptacji.

Algorytm pSA

```
parfor j=1,2,...,p
    repeat
        while i ⩽ L do
        begin
            Wylosuj permutację δ∈ N_π;
            if F(δ)< F(π*) then
            begin
                π*←δ ;
                prześlij π* do procesora centralnego i pobierz aktualną
                wartość F(π*) oraz π*;
            end;
            if (i mod K = 0) then
                wymień informację o π* z procesorem centralnym;
            if F(δ)< F(π) then π←δ ;
            else
                if Ψ_t(π,β) > random[0,1) then π ← δ ;
            i := i+1;
        end;
        i := 0;
        Zmień parametr kontrolny (temperaturę) t;
    until Warunek_Końca
end. {parfor}
```

Ze względu na powolną komunikacją pomiędzy procesorami, częstość komunikowania się procesorów jest bardzo istotna w przypadku badania wydajności algorytmów dla modelu maszyn równoległych bez pamięci współdzielonej. W powyższej implementacji zastosowano dwa rozwiązania.

1. Procesor podrzędny otrzymuje nową permutację π^* od procesora centralnego jedynie wtedy, gdy chce rozesłać jego własną permutację π^*. Sytuacja taka zdarza się względnie rzadko, co pozwala procesorom na dużą niezależność przeprowadzania procesów poszukiwań. W tabeli 7.3 model ten oznaczono symbolem BF.

2. Procesor podrzędny komunikuje się z procesorem centralnym celem wymiany danych (wartości $F(\pi^*)$ oraz permutacji π^*) co ustaloną ilość iteracji K. Przy testowaniu algorytmu zastosowano wartości K = 1,10,100,1000 oraz dla niezależnych przebiegów procesu poszukiwań INF (brak komunikacji, INF $> L$ gdzie L jest maksymalną liczbą iteracji).

Poniżej przedstawię algorytmy równoległe, oparte na metodzie symulowanego wyżarzania, dla dwóch reprezentatywnych problemów szeregowania zadań:

1. przepływowego z minimalizacją czasu zakończenia ($F^*\|C_{max}$),
2. jednomaszynowego z sumokosztową funkcją celu ($1\|\Sigma w_i T_i$).

Oba należą do klasy problemów silnie *NP*-zupełnych. W kolejnych rozdziałach będzie badany wpływ zmiany częstotliwości komunikowania się procesorów na czas obliczeń (przyspieszenie) oraz wartość funkcji celu (jakość rozwiązań).

7.2.2 Problem przepływowy. Algorytm pSA

Rozważany jest problem $F^*||C_{max}$ zgodny z opisem zawartym w Rozdz. 2.2 przy kryterium $\gamma = C_{max}$. Stąd poszukiwana jest permutacja $\pi^* \in \Pi$ taka, że $C_{max}(\pi^*) = \min_{\pi\in\Pi} C_{max}(\pi)$, gdzie $C_{max}(\pi) = C_{mn}$.

Otoczenie i schemat chłodzenia

W implementacji sekwencyjnego oraz równoległego algorytmu dla podanego problemu wykorzystano własności blokowe, opisane już w rozdziale 7.1.2. Otoczenie $\mathcal{N}_\pi$ permutacji π zawiera permutacje wygenerowana z π przez wstawienie (ruch typu *insert*) pewnego zadania z bloku wewnętrznego (B_k^f lub B_k^l) przed pierwsze lub za ostatnie zadanie bloku. Zastosowano funkcję akceptacji Bolzmanna wyrażającą się wzorem

$$\Psi_t(\pi,\delta)=\exp((F(\delta)-F(\pi^*))/t)$$

oraz geometryczny schemat schładzania $t_{i+1}=at_i$. Jeśli po wykonaniu *T_iter* iteracji algorytm nie znalazł lepszego rozwiązania od aktualnie posiadanego π^*, parametr kontrolny był modyfikowany: $t_{i+1} = t_0$. Algorytm zatrzymywał się po wykonaniu *Max_iter* iteracji.

Eksperymenty obliczeniowe

Cel eksperymentów. Eksperymenty komputerowe przeprowadzono dla zbadania zmiany jakości otrzymywanych rozwiązań wraz ze zmianą częstotliwości komunikacji pomiędzy równoległymi wątkami poszukiwań.

Implementacja. Algorytm równoległy symulowanego wyżarzania został zaprogramowane w języku Ada95 na 4-procesorowym komputerze Sun Enterprise 4x400MHz. Rozwiązania startowe otrzymano za pomocą algorytmu NEH (Navaz, Enscore, Ham [139]). Zastosowano następujące wartości parametrów:

t_0	=	60	–	początkowa wartość temperatury,
a	=	0.98	–	stała geometrycznego schematu schładzania,
L	=	n	–	ilość iteracji przy stałej temperaturze t,
T_iter	=	10	–	ilość iteracji bez poprawy najlepszego znanego rozwiązania po której parametrowi t przywracana jest wartość t_0.

Dla obu wersji algorytmu – sekwencyjnej oraz równoległej – przyjęto jednakową ilość iteracji Max_iter równą $200n$. W wersji 4-procesorowej każdy z procesorów wykonywał $50n$ iteracji.

Przykłady testowe. Algorytm był testowany na wielu przykładach o różnych rozmiarach i poziomie trudności. Sposób generowania danych został wzięty z pracy Taillarda [177]. Wykorzystano przykłady oraz najlepsze znane rozwiązania zamieszczone na stronie OR-Library [145].

Wyniki obliczeń W tabeli 7.3 przedstawione są wyniki eksperymentów komputerowych. Mierzono procentowy błąd względny, zdefiniowany wzorem (3.5), rozwiązań x^A generowanych przez badany algorytm, przyjmując za rozwiązania referencyjne x^* najlepsze znane wyniki pobrane z OR-Library [145].

Analiza ilości iteracji pomiędzy procesami komunikacji jednoznacznie wskazuje, że zbyt duża jej częstotliwość ($K = 1$) powoduje uzyskiwanie gorszych wyników niż przy mniejszej częstotliwości komunikacji ($K = 10$), zwiększając oczywiście także koszt wykonania algorytmu równoległego związany ze wzrostem czasu przeznaczonego na komunikację . Dalsze zmniejszania częstotliwości komunikacji ($K = 100, 1000$) powoduje jednak ponowne pogorszenie jakości rozwiązań. Pewnym kompromisem może tu być zastosowanie komunikacji w wypadku znalezienia nowego najlepszego rozwiązania (model BF). Średnia procentowa poprawa jakości rozwiązań otrzymywanych przez algorytm równoległy z komunikacją BF w stosunku do rozwiązań otrzymywanych przez algorytm sekwencyjny wyniosła 32%, przy tej samej liczbie iteracji.

n × m	1 proc.	4 procesory					
		częstotliwość komunikacji (w iteracjach)					
		INF	1	10	100	1000	BF
20 × 5	1,23%	1,28%	1,11%	0,85%	1,24%	1,04%	0,72%
20 × 10	2,08%	1,65%	1,91%	1,77%	1,99%	1,87%	1,63%
20 × 20	1,97%	1,74%	1,84%	1,55%	1,68%	1,77%	1,77%
50 × 5	0,27%	0,12%	0,15%	0,06%	0,13%	0,14%	0,10%
50 × 10	2,04%	1,03%	0,68%	1,26%	0,88%	1,08%	1,00%
średnia	**1,52%**	**1,16%**	**1,14%**	**1,10%**	**1,18%**	**1,18%**	**1,04%**

Tablica 7.3: Średni procentowy błąd względny równoległego algorytmu SA w odniesieniu do najlepszych znanych rozwiązań dla problemu przepływowego.

7.2.3 Problem jednomaszynowy. Algorytm pSA

Rozważany jest problem $1||\sum w_iT_i$ zgodny z opisem zawartym w Rozdz. 2.1 przy kryterium $\gamma = \sum w_iT_i$. Dla zadania i $(i = 1, 2, ..., n)$, niech p_i, w_i, d_i będą odpowiednio: *czasem wykonywania*, *wagą funkcji kosztów* oraz *linią krytyczną*. Jeżeli ustalona jest kolejność wykonywania zadań oraz C_i jest terminem zakończenia wykonywania zadania i, to wielkość $T_i = \max\{0, C_i - d_i\}$ nazywana jest *opóźnieniem*. Należy wyznaczyć taką kolejności wykonywania zadań, aby suma kosztów opóźnień $\sum w_iT_i$ była minimalna.

Istnieją algorytmy optymalne rozwiązywania rozpatrywanego problemu oparte na metodzie podziału i ograniczeń Potts i Van Wassenhove [154], Adrabiński i inni [5], a także na metodzie programowania dynamicznego Schrage i Baker [165], Lawler [111]. Algorytmy te pozwalają rozwiązywać (w rozsądnym czasie) przykłady, w których liczba zadań jest nie większa niż 50. Ze względu na ich małą efektywność, w praktycznych zastosowaniach, stosuje się algorytmy konstrukcyjne, (służące głównie do wyznaczania rozwiązań początkowych dla algorytmów popraw) oraz metaheurystyczne. Najlepsze obecnie wyniki otrzymano stosując algorytm oparty na metodzie tabu search, Crauwels i inni [55] oraz metodzie *dynasearch* [46].

Otoczenie i schemat schładzania

W implementacji sekwencyjnego oraz równoległego algorytmu dla podanego problemu wykorzystano otoczenie N_π permutacji π zawierające wszystkie permutacje wygenerowane przez zamianę miejscami dwóch dowolnych ele-

mentów w π (otoczenie NPI, patrz Rozdz. 3.3.1). Podobnie jak w algorytmie symulowanego wyżarzania dla problemu przepływowego zastosowano geometryczny schemat schładzania $t_{i+1}{=}at_i$. Jeśli po wykonaniu *T_iter* iteracji algorytm nie znalazł lepszego rozwiązania od aktualnie posiadanego π^*, wartość parametru kontrolnego była modyfikowana: $t_{i+1} = t_0$. Algorytm zatrzymywał się po wykonaniu *Max_iter* iteracji.

Eksperymenty obliczeniowe

Cel eksperymentów. Eksperymenty obliczeniowe przeprowadzono w celu zbadania, jak zmienia się jakość otrzymywanych rozwiązań wraz ze zmianą częstotliwości komunikacji pomiędzy równoległymi wątkami poszukiwań.

Implementacja. Algorytm równoległy zostały zaprogramowane w języku Ada95 na 4 - procesorowym komputerze Sun Enterprise 4 x 400MHz. Zastosowano następujące wartości parametrów:

t_0	=	60	–	początkowa wartość temperatury,
a	=	0.98	–	stała geometrycznego schematu schładzania,
L	=	n	–	ilość iteracji przy stałej temperaturze t,
T_iter	=	10	–	ilość iteracji bez poprawy najlepszego znanego rozwiązania po której parametrowi t przywracana jest wartość t_0.

Dla obu wersji algorytmu – sekwencyjnej oraz równoległej – przyjęto jednakową ilość iteracji *Max_iter* równą $400n$. Oznacza to, że w wersji 4-procesorowej każdy z procesorów wykonywał $100n$ iteracji.

Rozwiązanie startowe otrzymano za pomocą algorytmu META wyznaczającego najlepsze z rozwiązań czterech klasycznych heurystyk: SWPT, EDD, AU i COVERT.

Przykłady testowe. Algorytmy były testowany na przykładach, których sposób generowania został przedstawiony w pracy Potts, Van Wassenhove [154]. Przykłady te oraz ich najlepsze obecnie znane rozwiązania są zamieszczone na stronie OR-Library [145].

Wyniki obliczeń Wyniki obliczeniowe przeprowadzonych eksperymentów obliczeniowych przedstawione są w tabeli 7.4. Mierzono procentowy błąd względny, zdefiniowany wzorem (3.5), rozwiązań x^A generowanych przez badany algorytm, przyjmując za rozwiązania referencyjne x^* najlepsze znane wyniki pobrane z OR-Library [145].

n	1 proc.	4 procesory					
		częstotliwość komunikacji (w iteracjach)					
		INF	1	10	100	1000	BF
40	2,21%	1,42%	0,88%	0,79%	1,10%	1,02%	1,20%
50	1,91%	0,90%	0,89%	0,63%	1,15%	1,63%	0,86%
100	3,25%	2,44%	2,34%	1,93%	2,88%	2,31%	1,78%
średnia	**2,46%**	**1,59%**	**1,37%**	**1,12%**	**1,71%**	**1,65%**	**1,28%**

Tablica 7.4: Średni procentowy błąd względny równoległego algorytmu SA w odniesieniu do najlepszych rozwiązań dla problemu jednomaszynowego.

Podobnie jak w przypadku algorytmu równoległego dla problemu przepływowego, największą poprawę jakości otrzymywanych rozwiązań udało się uzyskać dla częstotliwości komunikacji K=10 (co 10 iteracji). Równoległy algorytm symulowanego wyżarzania bazujący na takim modelu uzyskał średnią poprawę rozwiązań w stosunku do algorytmu sekwencyjnego wynoszącą aż 54%, przy tej samej ilości iteracji (liczonej w przypadku algorytmu równoległego jako suma iteracji wykonanych przez każdy z procesorów).

7.2.4 Wnioski i uwagi

W Rozdz. 7.2.2 oraz 7.2.3 przedstawiono algorytmy równoległe oparte na metodzie symulowanego wyżarzania dla rozwiązywania dwóch permutacyjnych problemów szeregowania zadań: jedno oraz wielomaszynowego. Zbadano wpływ różnych sposobów komunikowania się procesorów na czas i jakość wyznaczanych rozwiązań. Na podstawie eksperymentów obliczeniowych można stwierdzić jednoznacznie, że użycie wielu procesorów z odpowiednio dobraną częstotliwością komunikacji w dużym stopniu poprawiło efektywność obliczeniową algorytmów oraz skróciło czas ich działania. Wartości przyśpieszeń (*speedup*) algorytmów równoległych dla obu problemów były ponadliniowe – oznacza to, że przy tej samej, co w algorytmach sekwencyjnych sumarycznej liczbie iteracji na wszystkich procesorach algorytmy równoległe znajdowały rozwiązanie lepsze niż odpowiednie algorytmy sekwencyjne. Zatem, otrzymanie podobnych lub nawet lepszych wyników na maszynie 4 procesorowej wymagało czasu czterokrotnie krótszego w stosunku do czasu obliczeń algorytmu sekwencyjnego.

Rysunek 7.1: Równoległy algorytm genetyczny, podejście globalne.

7.3 Równoległe metody genetyczne

Algorytm genetyczny już z swej natury jest w wysokim stopniu algorytmem równoległym. Sama idea symultanicznego badania nie jednego – ale wielu punktów optymalizowanej przestrzeni rozwiązań, leżąca u podstaw działania algorytmu genetycznego, jest w istocie ideą przetwarzania równoległego. Isnieją trzy podstawowe modele zrównoleglania algorytmu genetycznego:

- globalny (*single-walk model*),
- rozproszony (*diffusion model*),
- wyspowy (*island model*).

W dalszej części rozdziału zaprezentujemy podstawowe założenia powyżej wymienionych modeli. Przedstawione zostaną istniejące implementacje wykorzystujące poszczególne modele, oraz omówione będą ich zalety i ograniczenia.

7.3.1 Model globalny

Model *globalny* zakłada dostępność pracujących równolegle procesorów o mocy obliczeniowej wystarczającej do współbieżnego wyznaczania wartości funkcji przystosowania dla poszczególnych osobników. Algorytm genetyczny oparty na tym modelu bazuje na jednej populacji przechowywanej przez przez procesor centralny, który odpowiada za selekcję, kojarzenie oraz mutację. Procesory podrzędne obliczają wskaźniki przystosowania. Trajektoria poruszania się po przestrzeni rozwiązań takiego algorytmu jest identyczna jak trajektoria algorytmu sekwencyjnego. Stosując tego typu strategię zrównoleglania algorytmu uzyskać można prawie liniowe przyspieszenie, jeśli tylko czas obliczeń wartości funkcji przystosowania (funkcji celu) jest większy niż czas pochłaniany przez pozostałe elementy algorytmu (np. operatory genetyczne). Użycie tego rodzaju strategii znaleźć można w pracy Chalermwat i inni [42] zaimplementowanej na 50 – węzłowym klastrze Beowulf procesorów Pentium Pro 200 MHz połączonych siecią o topologii drzewiastej. Abramson i Abela [3] zastosowali globalny schemat algorytmu genetycznego w oparciu o model *master – slave* do problemu układania planu lekcji na komputerze Encore-Max z 16 procesorami i pamięcią współdzieloną. Podobny model obliczeń stosował też Grefenstette [83].

Rysunek 7.2: Współbieżny rozproszony algorytm genetyczny.

Poza obliczaniem wartości funkcji przystosowania, procesory równoległe mogą wykonywać procedury lokalnego poszukiwania (np. poszukiwanie zstępujące, tabu, symulowane wyżarzanie) dodatkowo poprawiając jakość rozwiązań (podejście *hybrydowe*, Mühlenbein [137], Reeves i Yamada [156]).

7.3.2 Model rozproszony

Algorytm genetyczny bazujący na asynchronicznym modelu *rozproszonym* wykorzystuje identyczne procesory wykonujące niezależnie zarówno operacje genetyczne, jak i obliczenia wskaźników przystosowania. Model ten przeznaczony jest dla dużych komputerów posiadających szybką komunikację pomiędzy procesorami i dużą liczbę procesorów. Każdemu procesorowi przydzielana jest pewna mała podpoulacja, zwykle – jeden osobnik. Selekcja i krzyżowanie możliwa jest w małym otoczeniu ograniczonym topologią sieci połączeń pomiędzy procesorami. Wykonanie selekcji i krzyżowania wymusza rozproszenie nowych osobników i przydzielenie ich do procesorów [44]. Talbi i Muntean [180] zaproponowali drobnoziarnisty rozproszony algorytm genetyczny dla transputera T-800. Kohlmorgern i inni [103] zaprezentowali drobnoziarniste algorytmy genetyczne w wersji rozproszonej oraz wyspowej miedzy innymi dla problemu komiwojażera (*TSP*) oraz problemu przepływowego (*flow shop*). Populacja startowa dzielona była na 1,4,16, 64, 256 i 1024 podpopulacje przydzielane osobno do każdego procesora komputera MasPar MP-1 z topologią połączeń dwuwymiarowej kraty.

7.3.3 Model wyspowy

Model *wyspowy* algorytmu genetycznego opiera się na założeniu, że każdy z procesorów wykonuje autonomiczny sekwencyjny algorytm genetyczny oparty na niezależnej, własnej populacji. Komunikacja pomiędzy procesorami może być wykorzystana na przykład do rozsyłania najlepszych osobników, lub części populacji (pewnej ilości najlepszych osobników) [33,34,148]. W porównaniu z wcześniej wymienionymi modelami, strategia wyspowa charakteryzuje się znaczną redukcją czasu poświęconego na komunikację (nie jest wymagana pamięć współdzielona) i może być wykorzystana w aplikacjach gruboziarnistych. W swej podstawowej formie model wyspowy może być scharakteryzowany przez trzy parametry:

1. *wielkość* subpopulacji znajdującej się na każdym z procesorów,

Rysunek 7.3: Równoległy algorytm genetyczny, model wyspowy.

2. *częstotliwość migracji* pewnej ilości „dobrych" osobników pomiędzy subpopulacjami,
3. *ilość* przesyłanych (migrujących) osobników.

Pewne biblioteki oparte na tej strategii były już wcześniej cytowane w rozdziale 5.2.2 – Bubak i Sowa [31] użyli modelu wyspowego do implementacji równoległego algorytmu genetycznego dla problemu komiwojażera. Belding [17] rozszerzył poprzednią pracę dotyczącą rozproszenia algorytmu genetycznego [183] skupiając się na częstotliwości migracji oraz ich ilościowych proporcji migrujących podpopulacji. Levine [117] zaimplementował równoległy algorytm genetyczny bazujący na modelu wyspowym do rozwiązania problemu rozbicia zbioru (*set partitioning problem*). Testy przeprowadził na 128 węzłach komputera IBM SP-1. Andre i Koza [12] opisali gruboziarnisty równoległy algorytm genetyczny oparty na języku C z biblioteką PVM zaimplementowanym na sieci transputerów, udało się im uzyskać przyspieszenie ponadliniowe (*super-linear speedup*) a najlepsze efekty uzyskiwali dla wielkości migracji równej 8% populacji. Falco i inni [62] zaimplementowali równoległy wyspowy algorytm genetyczny na komputerze Convex Meta Series z użyciem C i PVM.

7.3.4 Modele hybrydowe

W literaturze spotkać można także podejścia hybrydowe do zagadnienia zrównoleglenia algorytmu genetycznego. Oussaidène i inni [147] opisali hybrydową implementację równoległego algorytmu genetycznego, w której każda z podpopulacji jest przydzielona do pary *master – server.* Podpopulacje są związane z procesem *master*, zaś *server* służy do obliczania wartości funkcji przystosowania. Procesy *master* mogą wymieniać ze sobą części swoich podpopulacji. Do implementacji algorytmu użyto języka Java pod kontrolą systemu operacyjnego Condor [75].

Poza wymienionymi pracami istnieje wiele opisów badań równoległego algorytmu genetycznego związanych z badaniem takich jego parametrów jak liczba procesów, rozmiar populacji, współczynnik mutacji, liczba iteracji pomiędzy mutacjami itp., między innymi [9, 10, 35, 36, 167].

7.3.5 Problemu przepływowy. Algorytm pGA

W permutacyjnym problemie przepływowym (*permutation flow shop*) każde z zadań należy wykonać kolejno na wszystkich maszynach, przy czym

kolejność wykonywania zadań na każdej maszynie musi być taka sama. Optymalizacja polega na wyznaczeniu kolejności wykonywania zadań, która minimalizuje sumaryczny czas ich zakończenia. W literaturze problem ten jest oznaczany przez $F||\sum C_i$.

W ostatnich latach opublikowano niewiele algorytmów rozwiązywani omawianego problemu, skupiając się głównie na problemie przepływowym z kryterium maksymalnego a nie sumarycznego czasu zakończenia zadań. Problem $F||C_{max}$ jest uznawany za prostszy w rozwiązywaniu z uwagi na pewne szczególne własności, tzw. własności blokowe [20,79]. Dla problemu z kryterium C_{sum} własności tych niestety nie można zastosować, co znacznie zwiększa złożoność obliczeniową algorytmów jego rozwiązywania, przenosząc uwagę na metody programowania równoległego cechujące się o wiele większymi możliwościami w tej dziedzinie.

W literaturze, z wyżej podanych powodów, znaleźć można niewiele prac dotyczących omawianego problemu. Znane są pewne algorytmy konstrukcyjne (LIT i SPD [187], także NSPD w pracy Liu [122]) które cechuje niestety słaba efektywność. Reeves i Yamada [156] podali hybrydowy algorytm genetyczny, posiadający elementy algorytmu tabu i symulowanego wyżarzania, a także technikę ścieżek łączących (*path relinking*) za pomocą którego, wykonując bardzo dużą ilość iteracji, wyznaczyli najlepsze znane obecnie rozwiązania badanego problemu.

Przestawiony poniżej algorytm równoległy stanowi kontynuację badań własnych autora rozprawy nad konstrukcjami efektywnych równoległych algorytmów rozwiązywania bardzo trudnych problemów kombinatorycznych (np. [24–28]). W dalszej części przedstawiony zostanie algorytm równoległy, oparty na metodzie algorytmu genetycznego, którego wersja równoległa nie tylko przyspiesza działanie algorytmu, ale także poprawia jakość wyznaczanych rozwiązań

Eksperymenty obliczeniowe

Cel eksperymentów. Eksperymenty obliczeniowe przeprowadzono w celu zbadania, jaki wpływ na jakość otrzymywanych rozwiązań ma ilość podpopulacji (wysp), częstotliwości komunikacji pomiędzy nimi oraz zastosowane operatory genetyczne.

Implementacja. Konstrukcja właściwego równoległego algorytmu genetycznego poprzedzona została testami przeprowadzonymi na sekwencyjnym algorytmie genetycznym, a dotyczącymi efektywności wybranych klasycznych operatorów krzyżowania i mutacji dla problemu przepływowego z kryterium C_{sum}. Badane operatory zostały opisane w Dodatku A. Po testach

wstępnych do dalszych badań wybrano operatory krzyżowania PMX, CX i SX oraz operator mutacji I polegający na inwersji (zamianie) dwóch losowych elementów w permutacji. Wyniki testów przedstawione są w tabeli 7.5. Jako miarę efektywności algorytmu mierzono procentowy błąd względny do najlepszych znanych rozwiązań podanych w pracy Reeves i Yamada [156].

operator krzyżowania	bez mutacji	mutacja 0,05	mutacja 0,5
PMX	2,89%	2,39%	4,55%
CX	5,96%	4,23%	5,02%
SX	5,49%	3,17%	3,30%

Tablica 7.5: Efektywność różnych operatorów mutacji i krzyżowania dla problemu przepływowego z kryterium C_{sum}, 500 iteracji, wielkość populacji 100 osobników.

Algorytmu równoległy został zaprojektowany dla maszyny MIMD z procesorami bez pamięci współdzielonej oraz przy założeniu, że czas komunikacji pomiędzy procesorami jest o wiele dłuższy od czasu dostępu procesora do jego pamięci lokalnej. Zastosowano model wyspowy z procesorem centralnym pośredniczącym w komunikacji i magazynującym dane o najlepszych osobnikach oraz procesorami podrzędnymi przechowującymi podpopulacje, z kooperacją polegającą na migracji pewnej grupy najlepszych osobników pomiędzy podpopulacjami (wyspami).

Równoległy algorytm genetyczny zaimplementowany według modelu wyspowego charakteryzowały w powyższej implementacji następujące parametry:

- *wielkość* subpopulacji znajdującej się na każdym z procesorów wynosiła 100 osobników,
- *częstotliwość migracji* osobników pomiędzy subpopulacjami wynosiła 35 iteracji, ale dopiero po wykonaniu wstępnych 80 iteracji bez komunikacji (dla uniknięcia zbytniego upodobnienia się podpopulacji do siebie i lepszego rozproszenia procesu poszukiwań po przestrzeni rozwiązań),
- *ilość* przesyłanych (migrujących) osobników wynosiła 20% wielkości populacji.

Równoległy algorytm genetyczny w naszej implementacji przeszukiwał przestrzeń rozwiązań przy pomocy p równoległych wątków poszukiwań, każ-

dy wątek związanych z jednym procesorem oparty był na swej podpopulacji. Pomiędzy podpopulacjami wykonywana była operacja (quasi-operator) migracji. Zbadano efektywność algorytmu równoległego, który przy ustalonej sumarycznej liczbie iteracji równej 1000, uruchamiany był z różnymi kombinacjami trzech strategii: tej samej bądź różnych populacji początkowych, niezależnych lub kooperujących wątków poszukiwań oraz tych samych lub różnych operatorów genetycznych wykonywanych przez procesory na ich podpopulacjiach.

W tabeli 7.6 przestawione są następujące strategie:

- *ta sama populacja startowa, kooperacja (migracja), ten sam operator genetyczny PMX,*
- *ta sama populacja startowa, kooperacja, różne operatory genetyczne (PMX, CX, SX) na każdym z procesorów równoległych,*
- *ta sama populacja startowa, niezależne wątki poszukiwań (bez migracji), ten sam operator genetyczny PMX,*
- *ta sama populacja startowa, niezależne wątki poszukiwań, różne operatory genetyczne (PMX, CX, SX) na każdym z procesorów równoległych.*

w tabeli 7.7 przedstawiono przypadki odpowiadające algorytmom opartych na następujących strategiach:

- *różne populacje startowe, kooperacja, ten sam operator genetyczny PMX,*
- *różne populacje startowe, kooperacja, różne operatory genetyczne (CX, SX, PMX) na każdym z procesorów równoległych,*
- *różne populacje startowe, niezależne wątki poszukiwań, ten sam operator genetyczny PMX,*
- *różne populacje startowe, niezależne wątki poszukiwań, różne operatory genetyczne (PMX, CX, SX) na każdym z procesorów równoległych,*

Implementacja algorytmu równoległego zostały wykonana w równoległym języku programowania Ada95 na 4-procesorowym komputerze Sun Enterprise 4x400MHz pod kontrolą systemu operacyjnego Solaris 7.

Przykłady testowe. Algorytm był testowany na 50 przykładach o 5 rozmiarach z 100,...,500 operacjami ($n \times m$=20×5, 20×10, 20×20, 50×5, 50×10) opublikowanych przez Taillarda [177], (OR-Library [145]) z najlepszymi znanymi rozwiązaniami wziętymi z pracy Reeves, Yamada [156]. Testy dla większych rozmiarów problemów także były wykonywane, jednak ze względu na brak w literaturze jakichkolwiek rozwiązań referencyjnych nie można było nic powiedzieć o ich jakości. Aby zniwelować różnice wyników wynikające z probabilistycznych elementów algorytmu (losowanie populacji, losowanie pozycji krzyżowania i mutacji) każdy z 50 przykładów testowych wykonywany był 6 – krotnie a uzyskane wyniki (procentowe błędy względne zdefiniowany wzorem (3.5), rozwiązań x^A generowanych przez badany algorytm, przyjmujące za rozwiązania referencyjne x^* najlepsze znane wyniki) były uśredniane. Obliczane było także odchylenie standardowe wyników uzyskanych dla każdego przykładu jako miara dyspersji (rozrzutu), a więc stabilności działania algorytmu.

n × m	1 procesor	4 procesory			
		niezależnie		z kooperacją	
		te same operatory	różne operatory	te same operatory	różne operatory
20x5	1,00%	0,81%	0,73%	0,66%	0,52%
20x10	1,10%	1,00%	0,97%	0,81%	0,79%
20x20	0,93%	0,75%	0,74%	0,65%	0,64%
50x5	2,96%	3,70%	3,44%	3,43%	3,10%
50x10	4,48%	4,97%	4,70%	4,79%	4,64%
średnia	**2,13%**	**2,25%**	**2,11%**	**2,07%**	**1,98%**
odch. std.	0,20%	0,15%	0,12%	0,16%	0,12%

Tablica 7.6: Porównanie równoległych algorytmów genetycznych, różne populacje startowe na każdym z procesorów, 1000 iteracji.

Wyniki obliczeń. Strategia rozpoczynania obliczeń z różnych populacji początkowych na każdym procesorze (tabela 7.6) okazała się znacznie lepsza od strategii rozpoczynania obliczeń z tej samej populacji (tabela 7.7). Także strategia kooperacji dała lepsze wyniki obliczeniowe od strategii z niezależnym przeszukiwaniem przestrzeni rozwiązań bez komunikacji (bez migracji). Okazało się, że bardziej efektywna jest dywersyfikacja procesu poszukiwań poprzez zróżnicowanie operatorów genetycznych wykony-

n × m	1 procesor	4 procesory			
		niezależnie		z kooperacją	
		te same operatory	różne operatory	te same operatory	różne operatory
20x5	1,00%	0,87%	0,78%	0,65%	0,55%
20x10	1,10%	1,11%	0,97%	0,86%	0,84%
20x20	0,93%	0,80%	0,80%	0,76%	0,69%
50x5	2,96%	3,86%	3,62%	3,92%	3,54%
50x10	4,48%	5,38%	5,12%	5,24%	4,99%
średnia	**2,13%**	**2,40%**	**2,26%**	**2,29%**	**2,12%**
odch. std.	0,20%	0,11%	0,16%	0,13%	0,13%

Tablica 7.7: Porównanie równoległych algorytmów genetycznych, te same populacje startowe na wszystkich procesorach, 1000 iteracji.

wanych na swoich podpopulacjach przez każdy z procesorów. Najlepsze wyniki uzyskał genetyczny algorytm równoległy rozpoczynający obliczenia z różnych populacji początkowych, z zastosowaniem migracji pomiędzy podpopulacjami, z różnymi operatorami genetycznymi na poszczególnych procesorach – w tym wypadku poprawa błędu wyniosła 7%, przy tej samej liczbie iteracji równej 1000 dla algorytmu sekwencyjnego i 4 razy po 250 iteracji dla równoległej, 4 procesorowej wersji algorytmu. Ponadto, algorytm równoległy cechowała znacznie większa stabilność otrzymywanych rozwiązań – odchylenie standardowe wyników dla tej samej instancji problemu wynosiło średnio 0,12% dla najlepszego algorytmu wobec 0,20% dla algorytmu sekwencyjnego. Poprawa średniej wartości odchylenia standardowego otrzymywanych wyników wyniosła dla algorytmu równoległego aż 40% w stosunku do algorytmu sekwencyjnego.

7.3.6 Wnioski i uwagi

Przedstawiono konstrukcję algorytmu rozwiązywania permutacyjnego problemu przepływowego opartą na równoległym asynchronicznym algorytmie genetycznym w podejściu wyspowym. Przy implementacji algorytmu zastosowano ideę migracji najlepszych osobników pomiędzy procesorami. Eksperymenty obliczeniowe wskazują, że algorytm równoległy jest znacznie efektywniejszy od sekwencyjnego. Otrzymane wyniki (po niewielkiej liczbie iteracji) tylko nieznacznie różnią się od najlepszych obecnie znanych.

8

Równoległy schemat podziału i ograniczeń

W metodzie wyznaczania rozwiązań opisywanej w tym rozdziale stosuje się pewien sposób ograniczonego przeglądu przestrzenie rozwiązań – nie można bowiem (przynajmniej obecnie) uniknąć gwałtownego wzrostu rozmiarów przestrzeni rozwiązań wraz ze zwiększaniem rozmiaru badanego problemu. Zostanie tutaj opisane ogólne sformułowanie takiego przeszukiwania oraz zaprezentowana będzie implementacja metody podziału i ograniczeń dla pewnego typowego problemu szeregowania. Następnie zaproponowana zostanie metoda dostosowania implementacji algorytmu dla systemu obliczeń równoległych.

8.1 Schemat podziału i ograniczeń

Sekwencyjna metoda podziału i ograniczeń (*Branch and Bound, B&B*) jest stosowana do znajdowania rozwiązań optymalnych dla NP-trudnych problemów optymalizacji kombinatorycznej. Schemat określa ogólne podejście oparte na dekompozycji połączonej z przeszukiwaniem zbioru rozwiązań dopuszczalnych problemu, zaś konkretne algorytmy otrzymuje się poprzez uszczegółowienie elementów składowych schematu. Mimo, że schemat B&B jest stosunkowo dobrze poznanym narzędziem, jest w praktyce mało popularny ze względu na znaczną złożoność oraz problemy implementacyjne. Poniżej podamy krótką i zwięzłą charakterystyką schematu pochodzącą z pracy [170].

Niech $\mathcal{X}$ będzie przestrzenią rozwiązań dopuszczalnych problemu optymalizacyjnego 3.1, a $K(x)$, $x \in \mathcal{X}$ funkcją kryterialną. Poszukując wartości minimalnej funkcji kryterialnej $K(x)$, czyli rozwiązania optymalnego pro-

blemu, wykorzystać można następującą własność:

$$\min_{x\in\mathcal{X}} K(x) = \min\{\min_{x\in\mathcal{X}_1} K(x), \min_{x\in\mathcal{X}_2} K(x), \ldots, \min_{x\in\mathcal{X}_{|\mathcal{S}|}} K(x)\} \tag{8.1}$$

gdzie $\mathcal{X}_j$, $j \in \mathcal{S}$ jest rozbiciem zbioru rozwiązań dopuszczalnych $\mathcal{X}$ na podzbiory parami rozłączne i wyczerpujące, tzn.

$$\mathcal{X}_i \cap \mathcal{X}_j = \emptyset, \quad i,j \in \mathcal{S}, \quad i \neq j, \quad \bigcup_{j\in\mathcal{S}} \mathcal{X}_j = \mathcal{X}. \tag{8.2}$$

Zatem aby rozwiązać problem optymalizacyjny (3.1), potrzeba i wystarcza rozwiązać zbiór podproblemów postaci

$$\min_{x\in\mathcal{X}_j} K(x) \tag{8.3}$$

dla $j \in \mathcal{S}$, a następnie użyć zależności 8.1 do otrzymania rozwiązania głównego problemu 3.1. Pełny zbiór podproblemów jest scharakteryzowany poprzez funkcję celu $K(x)$ oraz rozbicie $\mathcal{P} = \{\mathcal{X}_j : j \in \mathcal{S}\}$. Odpowiednie podproblemy mogą być rozwiązywane sekwencyknie (system jednoprocesorowy) lub równolegle (system wieloprocesorowy, podproblemy są względem siebie niezależne). Podana dekompozycja jest racjonalna, jeśli uzyskanie rozwiązania dla 8.3 jest istotnie łatwiejsze niż dla 3.1. Problem 8.3 dla ustalonego $j \in \mathcal{S}$ można rozwiązać stosując następujące podejścia: (1) relaksacja, (2) bezpośrednio, (3) pośrednio, (4) przez podział.

Relaksacja oznacza usunięcie lub osłabienie ograniczeń lub funkcji celu (bądź też obu). *Relaksacja ograniczeń* to usunięcie lub złagodzenie części (lub całości) ograniczeń wyznaczających zbiór rozwiązań dopuszalnych $\mathcal{X}_j$. W sposób oczywisty zachodzi $\mathcal{X}_j \subseteq \mathcal{X}_j^R$, gdzie $\mathcal{X}_j^R$ jest zbiorem rozwiązań dopuszczalnych problemu zrelaksowanego polegającego na wyznaczeniu

$$\min_{x\in\mathcal{X}_j^R} K(x) \tag{8.4}$$

oraz

$$\min_{x\in\mathcal{X}_j} K(x) \geqslant \min_{x\in\mathcal{X}_j^R} K(x) = K(x^{*R}). \tag{8.5}$$

Jeżeli dla wyznaczonego rozwiązania optymalnego x^{*R} problemu zrelaksowanego zachodzi $x^{*R} \in \mathcal{X}_j$, to otrzymane rozwiązanie jest także rozwiązaniem optymalnym problemu bez relaksacji 8.3. W przeciwnym przypadku problem 8.3 nie został rozwiązany, zaś wielkość $K(x^{*R})$ stanowi *dolne ograniczenie* $LB(\mathcal{X}_j)$ wartości funkcji celu dla wszystkich rozwiązań problemu 8.3. *Relaksacja funkcji celu* to zastąpienie funkcji celu $K(x)$ funkcją

$K'(x)$ taką, że $K'(x) \leqslant K(x)$ dla wszystkich $x \in \mathcal{X}$. W sposób oczywisty zachodzi

$$\min_{x \in \mathcal{X}_j} K(x) \geqslant \min_{x \in \mathcal{X}_j} K'(x) = K'(x'). \tag{8.6}$$

Jeżeli dla wyznaczonego rozwiązania optymalnego x' problemu zrelaksowanego $\min_{x \in \mathcal{X}_j} K'(x)$ zachodzi $K(x') = K'(x')$, to otrzymane rozwiązanie x' jest także rozwiązaniem optymalnym problemu bez relaksacji 8.3. W przeciwnym przypadku problem 8.3 nie został rozwiązany, zaś wielkość $K(x')$ stanowi *dolne ograniczenie* $LB'(\mathcal{X}_j)$ wartości funkcji celu dla wszystkich rozwiązań problemu 8.3. Postać relaksacji jest dowolna, chociaż zakłada się domyślnie, że rozwiązanie problemu zrelaksowanego powinno być istotnie łatwiejsze niż rozwiązanie problemu podstawowego, np. może być dokonane w czasie wielomianowym.

Bezpośrednio został rozwiązany problem, jeśli obliczone zostały wartości funkcji celu $K(x)$ dla wszystkich $x \in \mathcal{X}_j$ w celu wyboru wartości minimalnej. W praktyce bezpośrednio rozwiązywany jest problem wtedy, gdy zbiór $\mathcal{X}_j$ jest niewielkiej liczności.

Pośrednio rozwiązany został problem 8.3, dla którego stwierdzono zachodzenie warunku

$$LB(\mathcal{X}_j) \geqslant UB \tag{8.7}$$

gdzie UB jest pewnym *górnym ograniczeniem* wartości funkcji celu dla wszystkich rozwiązań problemu 3.1. Problem taki faktycznie nie jest rozwiązywany, lecz *eliminowany* z rozważań jako nie zawierający rozwiązań o wartości funkcji celu mniejszej niż wartość progowa UB.

Podział stosowany jest do problemu, którego nie można rozwiązać używając podejść 1-3, tj. przez relaksację, bezpośrednio bądź pośrednio. Problem 8.3 odpowiadający wybranemu $\mathcal{X}_j$ jest usuwany ze zbioru rozbicia $\mathcal{P}$, zaś na jego miejsce dodawane są podproblemy otrzymane przez *podział* $\mathcal{X}_j$ na pewną liczbę podproblemów, skojarzonych ze zbiorami rozwiązań dopuszczalnych $\mathcal{Y}_k$, $k = 1, 2, \ldots, r$ takimi, że $\mathcal{Y}_k$, $k = 1, 2, \ldots, r$ jest rozbiciem zbioru $\mathcal{X}_j$ na podzbiory parami rozłączne i wyczerpujące.

8.2 Problem jednomaszynowy

Przedstawiony w dalszej części pracy algorytm (sekwencyjny i równoległy) oblicza rozwiązanie dokładne dla jednomaszynowego problemu minimalizacji ilości wag zadań opóźnionych, w literaturze oznaczanego przez $1||\Sigma w_i U_i$,

Rysunek 8.1: Drzewo $\mathcal{H}$.

Rysunek 8.2: Relacja poprzedzań.

należącego do klasy problemów NP-trudnych. Przedstawiono także implementację algorytmu sekwencyjnego w celu porównania wydajności obu algorytmów w tych samych warunkach działania systemu.

8.2.1 Algorytm sekwencyjny B&B

W konstrukcji algorytmu wykorzystany zostanie fakt, że pojedyncze rozwiązanie problemu $1||\Sigma w_i U_i$ może być reprezentowane przez permutację zbioru zadań $\mathcal{J}$. Dla potrzeb schematu $B\&B$ proces generowania rozwiązania optymalnego zostanie przedstawiony w postaci *drzewa rozwiązań* $\mathcal{H}$ opartego na porządkach częściowych, budowanych począwszy od *ostatniej* pozycji w permutacji. Zasada konstrukcji drzewa rozwiązań $\mathcal{H}$ jest następująca.

Węzeł 0, korzeń drzewa $\mathcal{H}$, na poziomie zerowym, przedstawia wszystkie rozwiązania tj. wszystkie permutacje $\mathcal{J}$. Z węzła 0 wychodzi n gałęzi prowadzących do n węzłów poziomu pierwszego. Każdy z tych węzłów reprezentuje rozwiązania, w których na ostatniej pozycji znajduje się jeden element, różny dla różnych węzłów. Z każdego węzła na poziomie 1 wychodzi $n-1$ krawędzi do $n-1$ nowych węzłów na poziomie 2. Zatem poziom 2 posiada łącznie $n(n-1)$ węzłów. Każdy węzeł na poziomie 2 przedstawia zbiór rozwiązań, w którym na końcu ustalono kolejność dwóch elementów. Proces ten przedstawiono na Rys. 8.1.

Tak więc, węzeł na l -tym poziomie drzewa $\mathcal{H}$ jest charakteryzowany przez permutację częściową zadań $\pi = (\pi(n-l+1), \pi(n-l+2), \ldots, \pi(n))$ oraz odpowiada wszystkim rozwiązaniom problemu reprezentowanych przez permutacje identyczne z π na l ostatnich pozycjach, $l = 0, 1, \ldots, n$. W tym przypadku, zbiór bezpośrednich następników węzła otrzymujemy umieszczając na pozycji $l+1$ nieuszeregowane jeszcze zadania $j \in S = \mathcal{J} \backslash S'$, gdzie $S' = \{\pi(j) : j = 1, 2, \ldots, l\}$ jest zbiorem zadań już uszeregowanych.

Z powyższego wynika, że każdemu węzłowi można przyporządkować relację poprzedzania $\mathcal{R}_\pi$ z jądrem $\mathcal{R}^*_\pi$ postaci, Rys. 8.2

$$\mathcal{R}^*_\pi = \{(\pi(j-1), \pi(j)) : j = n-l+2, n-l+3, \ldots, n\} \cup$$

$$\cup \{(j, \pi(n-l+1))\}, j \in S\}. \tag{8.8}$$

Samą zaś relację $\mathcal{R}_\pi$ można otrzymać z $\mathcal{R}_\pi^*$ dodając do $\mathcal{R}_\pi$ wszystkie par wynikające z przechodniości relacji $\mathcal{R}_\pi$. Relacją $\mathcal{R}_\pi$ jest ustalona w węźle odpowiadającym permutacji częściowej π w tym sensie, że wymagania częściowego porządku muszą być spełnione w każdym następniku π w drzewie rozwiązań. Z budowy drzewa $\mathcal{H}$ wynika natychmiast, że liście (węzły terminalne) reprezentują wszystkie permutacje zbioru $\mathcal{J}$, to znaczy jest ich $n!$.

Dla zadania $s \in \mathcal{J}$ wprowadźmy dodatkowo oznaczenie Δ_s określające następującą zależność: jeżeli permutacja β została wygenerowana z permutacji π przez ustalenie zadania s na pozycji k, to

$$F(\beta) = F(\pi) + \Delta_s. \tag{8.9}$$

Wyrażenie $F(\pi) + \Delta_s$ jest więc wagą permutacji wygenerowanej z π przez ustalenie zadania wolnego s na $k-$tej pozycji. W toku działania algorytmu wybierane będą takie zadania s, których ustalenie wygeneruje permutację – bezpośredniego następnika o możliwie najmniejszej wadze. Aby wybrać takie zadanie s, należy dla wszystkich zadań i ze zbioru zadań wolnych obliczyć ich wartości Δ_i i wybrać zadanie s o najmniejszej wartości $\Delta_s = \min_i \Delta_i$.

Rozwiązaniem początkowym algorytmu (korzeniem drzewa rozwiązań $\mathcal{H}$) jest permutacja π_0, dla której zbiór zadań wolnych $S_{\pi_0} = \mathcal{J}$. Za najlepsze rozwiązanie przyjmujemy $\pi^* \leftarrow \pi_0$ oraz górne ograniczenie $UB{=}F(\pi^*)$. Poziom drzewa $h{=}0$.

Niech π będzie permutacją (wierzchołkiem) na h-tym poziomie drzewa $\mathcal{H}$. Zbiór zadań wolnych $S_\pi = |k|$, przy czym $k = n - l$. Niech K_π związane z każdym wierzchołkiem drzewa $\mathcal{H}$ będzie zbiorem *kandydatów*, tj. zadań które można ustalić na pozycji wolnej.

Algorytm sB&B

Krok 1. {Dolne Ograniczenie (LB)}

if $LB(\pi) \geqslant F(\pi^*)$ **then go to** Krok 4;

Krok 2. {Górne Ograniczenie (UB)}

if $F(\pi) < UB$ **then**

begin

$UB \leftarrow F(\pi)$;

$\pi^* \leftarrow \pi$;

end;

Krok 3. Wyznacz zbiór kandydatów K_π – zadań, które można ustalić na k-tej pozycji;

if $K(\pi) = \emptyset$ **then goto** Krok 4;

Wybierz zadanie $s \in K_\pi$ takie, że

$$\Delta_s = \min_{i \in K(\pi)} \{\Delta_i\}, \tag{8.10}$$

Wygeneruj nową permutację β (wierzchołek w grafie $\mathcal{H}$) przez ustalenie zadania s na k-tej pozycji.

$h \leftarrow h + 1;\ k \leftarrow n - h;$

goto Krok 1.

Krok 4. {Cofanie}

if π jest korzeniem w drzewie H **then** Stop; (π^* jest rozwiązaniem optymalnym)

if permutacja π została wygenerowana z β przez ustalenie zadania $s \in K_\pi$ na pozycji k **then**

begin

$h \leftarrow h - 1;\ k \leftarrow n - h;\ K_\pi \leftarrow K_\pi \setminus \{s\},$

goto Krok 3.

end;

Czas działania algorytmu $B\&B$ zależy od liczby wyeliminowanych, czyli sprawdzonych pośrednio, poddrzew drzewa rozwiązań $\mathcal{H}$, a to z kolei wiąże się z jakością górnego ograniczenia UB - im lepsze, tym czas działania algorytmu będzie krótszy. W przedstawionym powyżej algorytmie $B\&B$ wartość początkowa UB jest wartością funkcji celu rozwiązania początkowego. Poniżej zamieszczony jest algorytm heurystyczny wyznaczający takie rozwiązanie.

Algorytm AH (Rozwiązanie przybliżone).

Ponumerować zadania tak, aby $d_1 \leqslant d_2 \ldots \leqslant d_n$.
$W \leftarrow \emptyset;\ t \leftarrow 0;$
for $i := 1$ **to** n **do**
begin
if $t + p_i \leqslant d_i$ **then**

Wykonać podstawienia: $W \leftarrow W \cup \{i\}$ oraz $t \leftarrow t + p_i$;
else
Wyznaczyć zbiór: $Q = \{l \in W : \sum_{j \in S} p_j - p_l + p_i \leqslant d_i\}$;
if $Q \neq \emptyset$ **then**
Wyznaczyć zbiór: $V = \{j \in Q : w_j > w_i\}$;
if $V \neq \emptyset$ **then**
Wyznaczyć $k \in V$ taki, że w_k=max$\{w_l$: $l \in V\}$ oraz wykonać podstawienia:
$W \leftarrow W \setminus \{k\} \cup \{i\}$; $t \leftarrow t - p_k + p_i$;
end;

Rozwiązanie konstruowane przez algorytm AH należy interpretować w sposób następujący: należy wykonać zadania ze zbioru W (w kolejności ich występowania) a następnie, w dowolnej kolejności, pozostałe zadania. Algorytm AH ma złożoność obliczeniową $O(n\log n)$.

8.2.2 Algorytm równoległy pB&B

Istnieje wiele metod urównoleglenia – a więc przyspieszenia – schematu $B\&B$. Wymienia się tu między innymi metodę scentralizowaną przydziału zadań dla procesorów (wykorzystaną w opisywanym algorytmie), metody rozproszoną, typu *token ring* i inne [106]. W pracy opiszę ideę równoległego algorytmu podziału i ograniczeń bazującego na metodzie scentralizowanej, ze zrównoważonym obciążaniem procesorów *(balanced loads strategy)* . Podobną metodę dla problemu komiwojażera stosują autorzy pracy [61].

Opisywana metoda zakłada istnienie w systemie stałej liczby procesorów – ważne jest, że ich ilość jest ustalona odgórnie i nie jest zależna od wielkości zadania.

Projektowany algorytm równoległy powinien spełniać trzy podstawowe założenia:

1. obciążenie procesorów powinno zostać zrównoważone – to jest żaden z procesorów nie powinien przed zakończeniem obliczeń pozostawać bezczynnym,
2. wystarczająco dobre rozwiązanie (w naszym przypadku – optymalne) powinno zostać znalezione najprędzej jak to możliwe,
3. jak najwięcej podzbiorów rozwiązań powinno zostać wyeliminowanych.

Realizacja punktu (1) odpowiada za zrównoważony przydział zadań do procesorów – w naszym wypadku przydziałem tym będzie kierował wybrany, „sterujący” procesor (stąd model scentralizowany).

Punkt (2) odpowiada za dobrą komunikację pomiędzy procesorami. Procesor, który znalazł nowe najlepsze rozwiązanie powinien jak najprędzej rozesłać jego wartość do innych procesorów. W przypadku opisywanego algorytmu nie jest to klasyczne rozgłaszanie (*broadcasting*), ponieważ do przechowywania najlepszego rozwiązania π^*używany jest procesor nadzorujący – stąd też jedyną drogą takiej komunikacji jest powiadomienie tego procesora przez jeden z procesorów podrzędnych o znalezieniu lepszego rozwiązania od aktualnie najlepszego π^*. Szybkie powiadomienie procesorów zapewnia dobrą efektywność algorytmu – procesory powiadomione w odpowiednim momencie o nowej wartości najlepszego aktualnie rozwiązania nie będą badały poddrzew, w których nie ma rozwiązania lepszego. Dlatego procesory otrzymują aktualną wartość π^* przy otrzymaniu każdego zadania obliczeniowego od procesora nadzorującego oraz, w międzyczasie, co pewną ustaloną liczbę iteracji *broad_iter*.

Punkt (3) – eliminacja w trakcie obliczeń jak największej liczby podzbiorów rozwiązań – jest egzekwowany przez sam algorytm podziału i ograniczeń. Urównoleglenie może jednak ten proces przyspieszyć (w stosunku do algorytmu sekwencyjnego) – wskutek obliczeń równoległych rozwiązanie aktualnie najlepsze π^* może zostać znalezione szybciej niż czyni to algorytm sekwencyjny (to jest bez badania pewnych rozwiązań pośrednich, sprawdzanych przez algorytm sekwencyjny przed zbadaniem π^*).

Schemat algorytmu równoległego został zaprojektowany dla maszyny MIMD z procesorami bez pamięci współdzielonej. Jak już wspomnianio, zastosowano model scentralizowany z procesorem centralnym magazynującym dane oraz procesorami podrzędnymi kooperującymi, równolegle przeglądającymi przestrzeń rozwiązań. Każdy procesor posiada lokalny stos odpowiadający za operacje ustalania zadań oraz operację cofania. Jest to stos priorytetowy, sortowany ze względu na wartości dolnych ograniczeń LB, podobnie jak lista OPEN w algorytmach przeglądu typu najpierw-najlepszy (*best-first,* np. w algorytmie A^*, [57]).

Algorytm pB&B

parfor dla każdego procesora
begin
Heap : lokalny stos;
while *Heap* $\neq \emptyset$
begin
$\pi :=$ Get(*Heap*);
if $LB(\pi) < UB$ **then**
begin
if $F(\pi) < UB$ **then**
begin
$UB \leftarrow F(\pi)$, $\pi^* \leftarrow \pi$;
przekaż π^* **pozostałym procesorom;**
end;
Wyznacz zbiór kandydatów K_π – zadań, które można ustalić na k-tej pozycji;
Wybierz zadanie $s \in K(\pi)$, takie że

$$\Delta_s = \min_{i \in K_\pi} \{\Delta_i\},$$

Wygeneruj nową permutację β (wierzchołek w drzewie $\mathcal{H}$) przez ustalenie zadania s na k-tej pozycji;
$h \leftarrow h + 1$; $k \leftarrow n - h$;
Put(*Heap*, β);
end;
end;
end.

8.2.3 Eksperymenty obliczeniowe

Cel eksperymentów. Eksperymenty obliczeniowe przeprowadzono w celu zbadania, jak zmienia się ilość przeglądniętych węzłów drzewa rozwiązań $\mathcal{H}$, a co za tym idzie ilość iteracji wykonana przez algorytm, wraz ze wzrostem ilości użytych procesorów.

Implementacja. Algorytm został zaimplementowany w języku Ada95 i uruchamiany pod kontrolą systemu Solaris 7 na komputerze Sun Enterprise wyposażonym w 4 procesory o częstotliwości zegara 400MHz każdy. Zadania współbieżne języka Ada były wykonywane równolegle jako wątki systemowe.

Przykłady testowe Przykłady testowe były generowane następująco [108, 160]: dla każdego zadania i całkowitoliczbowy czas wykonania p_i był losowany z rozkładu jednostajnego w przedziale [1,100], a całkowitoliczbowe wagi w_i losowano z przedziału [1,10]. Trudność problemu zależy silnie od dwóch współczynników wykorzystywanych przy generowaniu przykładów: RDD (*range due dates*) oraz TF (*tardiness factor*). Mając sumaryczny czas wykonywania zadań $P = \sum_{i=1}^{n} p_i$ oraz wartości współczynników RDD i TF ze zbioru {0.2, 0.4, 0.6, 0.8, 1.0} całkowite wartości linii krytycznych d_i zadania i losowane były z rozkładu jednostajnego na przedziale [P(1-TF-RDD/2), P(1-TF+RDD/2)]. Dla każdych z 25 par współczynników RDD i TF generowanych było 5 instancji problemu, co daje 125 przykładów testowych dla każdej wielkości n (ilości zadań).

Wyniki obliczeń W tabeli 8.1 przedstawiono średnią liczbę iteracji algorytmów uruchamianych na różnej liczbie procesorów. Jak można zauważyć algorytm równoległy wykonuje mniej iteracji niż algorytm sekwencyjny (1-procesorowy) dla większych rozmiarów problemu ($n > 25$). W równoległej wersji algorytmu ilość wykonanych iteracji była liczona jako suma ilości iteracji wykonanych przez każdy z procesorów.

n	liczba procesorów		
	1	2	4
20	**1542**	2287	2930
25	**5654**	6052	7468
30	22985	18006	**15913**
35	91587	**66321**	79854
40	293122	226892	**224632**

Tablica 8.1: Średnia liczba iteracji wykonana przez algorytm B & B.

Osobną kwestią jest dobranie najlepszej częstotliwości komunikacji pomiędzy procesorami. W Tab. 8.2 znajdują się wyniki symulacji przeprowadzonej dla przykładu o rozmiarze n=12 przy założeniu różnej liczby iteracji (wartość *broad_iter*), a więc i czasów, pomiędzy kolejnymi momentami komunikacji. Jak widać, najkorzystniejszą wartością *broad_iter* było 1000 iteracji, choć wartość *broad_iter* silnie zależy od prędkości maszyny i możliwej do osiągnięcia szybkości komunikacji. Częsta komunikacji – nawet w każdej iteracji (*broad_iter*=1) powodowała znaczne wydłużenie czasu działania programu równoległego.

implementacja dla 4 procesorów		
broad_iter	liczba iteracji	czas (sekund)
1	208118	37
10	200434	14
100	205357	14
500	206889	9
1000	196128	9
5000	207699	9
10 000	205983	11
20 000	206750	11
implementacja dla 1 procesora		
-	247046	29

Tablica 8.2: Porównanie liczby iteracji i czasów działania dla przykładu o rozmiarze n=12 (RDD=1.0, TF=1.0).

8.2.4 Wnioski i uwagi

Przedstawiono konstrukcję algorytmu rozwiązywania jednomaszynowego problemu szeregowania zadań opartą na równoległym algorytmie podziału i ograniczeń. Przy implementacji algorytmu zastosowano model kooperacji polegający na przekazywaniu sobie przez procesory aktualnej wartości dolnego ograniczenia, co pozwoliło uniknąć przeglądania pewnych podzbiorów zbioru rozwiązań. Eksperymenty obliczeniowe wskazują, że algorytm równoległy jest znacznie efektywniejszy od sekwencyjnego, sprawdzając sumarycznie mniej elementów drzewa rozwiązań oraz wykazując przyspieszenie ponadliniowe (*super-linear speedup*).

9

Krajobraz przestrzeni rozwiązań

Krajobraz (*landscape*) przestrzeni rozwiązań jest rozumiany jako związek między wartościami funkcji celu a wzajemną odległością rozwiązań w przestrzeni. Istotną cechą problemów kombinatorycznych jest brak tradycyjnych cech regularności funkcji celu (tak jak różniczkowalność, gładkość, wypukłość), cech regularności przestrzeni rozwiązań (wypukłość), występowanie wieloekstremalności oraz przekleństwo wymiarowości. W tym kontekście proces optymalizacji można porównać do wędrówki człowieka przez Himalaje, z zawiązanymi oczami, w celu poszukiwania najwyższego szczytu, lub szukania zagłębienia ziemnego w wysokiej trawie, patrz trajektorie poszukiwań zilustrowane w pracach [141, 170].

Obserwacja krajobrazu przestrzeni rozwiązań pozwala określić dokładniej, jakie własności powinna mieć potencjalna metoda rozwiązywania oparta na sekwencyjnym lub równoległym przeszukiwaniu tej przestrzeni. W szczególności można sprecyzować jak określić równowagę pomiędzy rozproszeniem (*diversification*) poszukiwań a ich intensyfikacją (*intensification*). Jest to szczególnie użyteczne przy wielowątkowych poszukiwaniach z kooperacją wątków. Ponieważ dyskutowaliśmy już szczegółowo sposób liczenia wartości funkcji celu, kolejnym niezbędnym narzędziem określającym wzajemne odległości pomiędzy rozwiązaniami w przestrzeni rozwiązań są specyficzne miary odległości, dedykowane do określonych przestrzeni.

9.1 Wybrane miary na przestrzeni rozwiązań

Rozwiązania problemów szeregowania zadań mogą być reprezentowane za pomocą różnorodnych obiektów matematycznych, takich jak permutacje,

wektory, ciągi permutacji, podziały zbioru, wektory binarne lub całkowitoliczbowe, itp. Poniżej, przedstawione zostaną miary odległości rozwiązań dla problemów szeregowania rozważanych w pracy, a mianowicie:

- permutacje (problemy $1||\gamma, F^*||\gamma$),
- ciągi permutacji (problem $J||\gamma$),
- podziały zbioru i permutacje (problem $FP||\gamma$).

Omówimy dalej te miary, wraz z ich relacją do otoczeń API, NPI, INS, używanych powszechnie w algorytmach poszukiwań lokalnych.

9.1.1 Przestrzeń permutacji

Niech $\pi = (\pi(1), ..., \pi(n))$ będzie permutacją na zbiorze n-elementowym [121]. Przez permutację odwrotną do π oznaczamy permutację π^{-1} taką, że $\pi^{-1}(\pi(i)) = i$. Permutacja o (przekształcenie identycznościowe) odpowiada permutacji naturalnej, to jest $o(i) = i$. Złożenie (mnożenie) permutacji α oraz β, zapisywane $\alpha \circ \beta$, określamy jako przekształcenie takie, że $(\alpha \circ \beta)(i) = \alpha(\beta(i))$. Odpowiednio złożenia $\pi \circ \pi^{-1} = o$ oraz $\pi^{-1} \circ \pi = o$.

Z poniżej wymienionych miar tylko trzy mają związek z typowymi otoczeniami używanymi w metodach lokalnych poszukiwań działających na przestrzeni permutacji. Miara tau Kendalla odpowiada liczbie ruchów typu *wymiana par przyległych* (*adjacent swaps*), miara Cayleya odpowiada liczbie ruchów typu *wymiana par dowolnych* (*non-adjacent swaps*) zaś miara Ulama ma związek z liczbą ruchów typu *wcięcia* (inserts), co zostanie wykazane w dalszej częsci pracy.

W literaturze wymienia się kilka typowych miar używanych do analizy permutacji [63]:

- ***Miara stopowa (Footrule):***
  $D(\pi,\sigma) = \sum |\pi(i) - \sigma(i)|$
- ***Korelacja rang Spearmana (Spearman's rank correlation):***
  $S^2(\pi, \sigma) = \sum (\pi(i) - \sigma(i))^2$
- ***Odległość Hamminga (Hamming distance):***
  $H(\pi,\sigma) = n-$ilość i takich, że $\pi(i) \neq \sigma(i)$
- ***Miara tau Kenadalla (Kendall's tau):***
  $I(\pi,\sigma)$ = minimalna ilość zamian par przyległych przeprowadzająca permutację π^{-1} w σ^{-1}.

- ***Miara Cayleya (Cayley's distance):***

  $T(\pi,\sigma)$ = minimalna ilość transpozycji przeprowadzająca permutację π w permutację σ .

- ***Miara Ulama (Ulam's distance):***

  $L(\pi,\sigma) = n-$długość najdłuższego rosnącego podciągu w permutacji $\sigma \circ \pi^{-1}$.

miara	konstrukcja	średnia	wariancja	złożoność
$D(\pi,\sigma)$	$\sum \lvert\pi(i) - \sigma(i)\rvert$	$\frac{n^2-1}{3}$	$\frac{(n+1)(2n^2+7)}{45}$	$O(n)$
$S^2(\pi,\sigma)$	$\sum\{\pi(i) - \sigma(i)\}^2$	$\frac{n^3-n}{6}$	$\frac{n^2(n-1)(n+1)^2}{36}$	$O(n)$
$H(\pi,\sigma)$	$n - \#\{i : \pi(i) \neq \sigma(i)\}$	$n-1$	1	$O(n)$
$I(\pi,\sigma)$	min # wymian par przyległych przeksz--tałcających π^{-1} w σ^{-1}	$\frac{n(n-1)}{4}$	$\frac{n(n-1)(2n+5)}{72}$	$O(n^2)$
$T(\pi,\sigma)$	$n -$ # cykli w $(\pi \circ \sigma^{-1})$	$n - \log n^{*)}$	$\log n^{*)}$	$O(n)$
$L(\pi,\sigma)$	$n-$ długość najdłuż--szego rosnącego podciągu w $\sigma \circ \pi^{-1}$	$n - 2\sqrt{n}$ $^{*)}$	$\sqrt[3]{n}$ $^{*)}$	$O(n \log n)$

$^{*)}$ – asymptotycznie

Tablica 9.1: Zestawienie klasycznych miar na przestrzeni permutacji.

Zestawienie miar na przestrzeni permutacji przedstawiono w Tabl. 9.1. Wśród ważnych własności miary wymienia się jej lewo- i prawo-stronną niezmienniczość (inwariantność), wartość średnią i wariancję dla całej populacji permutacji. Ze względu na zastosowania interesować nas będą jeszcze dwie inne cechy: złożoność obliczeniowa wyznaczenia wartości miary oraz jej interpretacja dla metod lokalnych poszukiwań.

Miara tau Kendalla to minimalna ilość zamian par przyległych przeprowadzających permutację π^{-1} w σ^{-1}. Definicja miary przy użyciu permutacji odwrotnych zapewnia prawostronną inwariantność. W zastosowaniach algorytmicznych zwykle bazuje się jednak na permutacjach π i σ mierząc ilość ruchów potrzebnych do przejścia pomiędzy nimi, co odpowiada wartości miary tau Kendalla na permutacjach π^{-1} i σ^{-1}czyli $I(\pi^{-1},\sigma^{-1})$. W celu policzenia wartości miary tworzymy dwu-wierszową tablicę zawierającą permutacje π oraz σ. Następnie tablice sortujemy kolumnami według pierwszego wiersza oraz liczymy w wierszu drugim liczbę par ustawionych

niezgodnie, tzn. element większy przed mniejszym. Złożoność obliczeniowa policzenia tej miary jest rzędu $O(n^2)$.Wartość średnia miary wynosi $\frac{n(n-1)}{4}$, wariancja $\frac{n(n-1)(2n+5)}{72}$.

Miara Cayleya *T(π,σ)* to ilość ruchów polegających na wymianie dowolnych par przeprowadzających permutację π w σ. Posiada prostą własność: $T(\pi,\sigma) = n$ – ilość cykli w $(\pi \circ \sigma^{-1})$. Liczbę cykli można wyznaczyć w czasie $O(n)$ metodą z pracy [121]. Wartość średnia miary wynosi asymptotycznie $n - \log n$, wariancja asymptotycznie $\log n$.

Miara Ulama. Prostym algorytmem pozwalającym na wyznaczenie wartości miary Ulama jest tzw. *gra Floyda,* patrz np. [63].Wyobraźmy sobie talię zawierającą n kart indeksowanych 1,2,...,n. Karty tasujemy, kartę będącą na samej górze nazywamy $\pi(1)$, następną $\pi(2)$, itd. Zaczynamy układać pasjans (odkrywać karty z talii po jednej) w myśl zasady: można położyć niższą kartę na wyższej. Jeśli odkryta karta jest wyższa niż ta na szczycie stosu kart, należy nią rozpocząć następny stos. Stosów powinno być jak najmniej. Najlepszą strategią jest w tej sytuacji umieszczanie niższej karty na najmniejszej karcie wyższej od niej. Ilość stosów jest równa długości najdłuższego rosnącego podciągu w π. Ilość ruchów gry jest rzędu $O(n \log n)$, taka też jest złożoność obliczeniowa wyznaczenia miary Ulama. Wartość średnia miary wynosi asymptotycznie $n - 2n^{1/2}$, wariancja asymptotycznie $n^{1/3}$.

Pokażemy dalej, że ilość wcięć (ruchów typu *insert*) potrzebnych do przeprowadzenia permutacji π w permutację σ odpowiada wartości miary Ulama na permutacjach odwrotnych do π i σ. Oznaczmy tą liczbę ruchów przez *Ins(π,σ)*. Zachodzą następujące własności.

Własność 1 $Ins(\pi,\sigma) = Ins(\sigma,\pi)$.

Jeśli przeprowadzenie permutacji π w permutację σ wymaga pewnej ilości ruchów typu *insert* to, aby przeprowadzić σ w π należy wykonać tyle samo ruchów odwrotnych oraz w odwrotnej kolejności.

Własność 2 $Ins(\pi,\sigma) = Ins(\gamma \circ \pi,\gamma \circ \sigma)$.

Lewostronne mnożenie permutacji π oraz σ przez permutację γ interpretować można jako zmianę nazw symboli występujących w permutacjach π oraz σ (przenumerowanie). Zmiana nazw symboli w obu permutacjach równocześnie nie ma żadnego wpływu na wartość miary.

Własność 3 *Ins(π,o)= n – długość najdłuższego rosnącego podciągu w π.*

Celem naszym jest przeprowadzenie permutacji π w o (uporządkowanie π). Niech $\pi(i_1) < \pi(i_2) < ... < \pi(i_k)$ będzie najdłuższym rosnącym podciągiem w π. Będziemy wykonywać ruchy typu *insert* polegające na usuwaniu elementów j nie należących do tego podciągu i wstawianie ich w dowolne miejsca poprawne względem tego podciągu. Technika ta przypomina proces sortowania alfabetycznego książek na półce metodą „wyjmij i wstaw". Przesuwanie nie zmienia wzajemnej relacji pomiędzy $\pi(i_1) < \pi(i_2) < ... < \pi(i_k)$, ale może zwiększyć długość podciągu. Każdy wykonany ruch zwiększa długość najdłuższego rosnącego podciągu co najwyżej o 1. Zatem co najwyżej $n - k$ ruchów potrzeba, aby uzyskać podciąg o długości n. Wyprowadzone ograniczenie górne jest obcisłe.

Własność 4 *Ins(*π*,*σ*) = L(*π^{-1}*,*σ^{-1}*).*

Korzystając kolejno z Własności 2, definicji permutacji o, Własności 1 i 3 otrzymujemy następujący ciąg równości: $Ins(\pi,\sigma) = \mathrm{Ins}(\sigma^{-1}\circ\pi,\sigma^{-1}\circ\sigma) = Ins(\sigma^{-1}\circ\pi,\ o) = Ins(o,\sigma^{-1}\circ\pi) =$ (*ilość ruchów „insert" potrzebnych aby przeprowadzić* $\sigma^{-1}\circ\pi$ *w* o) = (n – *długość najdłuższego podciągu rosnącego w* $\sigma^{-1}\circ\pi$) = $L(\pi^{-1},\sigma^{-1})$.

9.1.2 Przestrzeń ciągów permutacji

Niech $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$ będzie uporządkowaną m –ką permutacji $\pi_k = (\pi_k(1), \pi_k(2), \ldots, \pi_k(m_k))$, $k = 1, 2, \ldots, m$. Kolejności wykonywania zadań określone przez π lub σ mogą być zastosowane jako reprezentacje rozwiązania w niektórych problemach szeregowania zadań, na przykład w problemie gniazdowym, modelu kombinatorycznym, patrz też Rozdz. 2.4, w którym zbiór operacji $\mathcal{O}$ może być zdekomponowany na podzbiory operacji wykonywanych na jednej, ustalonej maszynie $k \in M$, $M_k = \{i \in \mathcal{O} : v_i = k\}$, $m_k = |M_k|$. Kolejność wykonywania operacji na maszynie k jest określona permutacją $\pi_k = (\pi_k(1), \pi_k(2), \ldots, \pi_k(m_k))$ zbioru M_k, $k \in M$, gdzie $\pi_k(i)$ oznaczna ten element z M_k, który jest na pozycji i w π_k.

Niech $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ także będzie uporządkowaną m –ką permutacji $\sigma_k = (\sigma_k(1), \sigma_k(2), \ldots, \sigma_k(m_k))$, $k = 1, 2, \ldots, m$. Wartości m_k długości poszczególnych permutacji σ_k są takie same jak długości odpowiednich permutacji π_k (czyli π oraz σ reprezentują rozwiązanie tego samego problemu gniazdowego). Definiujemy odległość pomiędzy ciągami permutacji π oraz σ jako

$$D_p^Z(\pi,\sigma) = \left(\sum_{k=1}^{m}\left(D_k^Z(\pi_k,\sigma_k)\right)^p\right)^{\frac{1}{p}} \tag{9.1}$$

gdzie p jest liczbą naturalną, a $D_k^Z(\pi_k, \sigma_k)$, $k = 1, 2, \ldots, m$, jest jedną z miar odległości pomiędzy permutacjami π_k i σ_k, na przykład miarą tau Kendalla $I(\pi_k, \sigma_k)$, miarą Cayleya $T(\pi_k, \sigma_k)$ lub Ulama $L(\pi_k^{-1}, \sigma_k^{-1}) = Ins(\pi_k, \sigma_k)$. Ponieważ miary $I(\alpha, \beta)$, $T(\alpha, \beta)$ oraz $Ins(\alpha, \beta)$ są równe odpowiednio liczbie ruchów typu *zamiana par przyległych*, *zamiana par dowolnych* oraz liczbie *wcięć*, potrzebnych do przeprowadzenia permutacji α w β, więc miara

$$D_1^Z(\pi, \sigma) = \sum_{k=1}^{m} D_k^Z(\pi_k, \sigma_k) \tag{9.2}$$

także posiada interpretację (sumarycznej) liczby ruchów odpowiedniego typu potrzebnych do przeprowadzenia π w σ. Z kolei dla $p = 2$ otrzymujemy miarę

$$D_2^Z(\pi, \sigma) = \sqrt[2]{\sum_{k=1}^{m} \left(D_k^Z(\pi_k, \sigma_k)\right)^2}, \tag{9.3}$$

natomiast przy $n \to \infty$

$$D_\infty^Z(\pi, \sigma) = \max_{1 \leqslant k \leqslant n} D_k^Z(\pi_k, \sigma_k). \tag{9.4}$$

9.1.3 Przestrzeń podziałów zbioru i permutacji

Reprezentacja rozwiązania problemu szeregowania zadań za pomocą podziałów zbioru i permutacji jest ściśle związana z hybrydowym problemem przepływowym, toteż do opisu tej przestrzeni użyte zostaną pojęcia operacji, zadania, gniazda i wsadu, zdefiniowane wcześniej w Rozdz. 2.3.

Niech $\alpha \in \Pi$ będzie kolejnością wykonywania zadań, a $n_{li}^\alpha = |\mathcal{J}_{li}^\alpha|$, $l \in M$, $i \in M_l$, ilością elementów i -tego wsadu związaną z kolejnością wykonywania zadań α, przyporządkowanego do gniazda l. Definiujemy zbiór par operacji *zgodnych* z kolejnością wykonywania zadań α jako

$$Y(\alpha) = \bigcup_{l \in M} \bigcup_{i \in M_l} \{((l, \alpha_{li}(k')), (l, \alpha_{li}(k''))) : 1 \leqslant k' < k'' \leqslant n_{li}^\alpha\}. \tag{9.5}$$

Odległość pomiędzy dwoma kolejnościami wykonywania zadań $\alpha, \beta \in \Pi$ może być zdefiniowana jako

$$\rho(\alpha, \beta) = A\rho_1(\alpha, \beta) + \rho_2(\alpha, \beta) \tag{9.6}$$

gdzie

$$\rho_1(\alpha, \beta) = \sum_{l \in M} |\mathcal{J}_{l\sigma(i)}^\alpha \setminus \mathcal{J}_{li}^\beta|, \tag{9.7}$$

$$\rho_2(\alpha, \beta) = |Y(\alpha) \setminus Y(\beta)|, \tag{9.8}$$

i stała A jest pewną dużą liczbą naturalną.

9.1.4 Wnioski i uwagi

Otrzymane wyniki pozwalają na wykorzystanie miar do analizowania i kontrolowania zachowania się równoległych algorytmów poszukiwań lokalnych. Istnieje co najmniej kilka sposobów wykorzystania tej informacji:

1. wizualizacja krajobrazu przestrzeni rozwiązań,
2. badanie charakteru przestrzeni rozwiązań,
3. badanie rozkładu ekstremów lokalnych w przestrzeni,
4. detekcja dolin (valleys) zawierających „dobre" ekstrema,
5. określanie trajektorii poszukiwań wykonywanych przez algorytmy,
6. celowe kierowanie poszukiwań,
7. badanie rozłączności trajektorii.

9.2 Badanie rozkładu ekstremów lokalnych

Szereg autorów zajmujących się problemami szeregowania wykazuje, że mamy do czynienia z typowo wieloekstremalnym problemem optymalizacyjnym, z przypadkowo rozrzuconymi lokalnymi ekstremami (powierzchnia typu „szczotka"). Celem naszym jest określenie stopnia skupienia „dobrych" ekstremów lokalnych w celu wskazania najbardziej obiecującego regionu (najbardziej obiecujących regionów) poszukiwań. Do tego celu wykorzystuje się w literaturze kilka typów eksperymentalnych badań, opartych na „próbkowaniu" przestrzeni rozwiązań. Wybrane reprezentatywne próbki zwykle dostarczają rozwiązań startowych dla metody szukania zstępującego (DS), wykrywającej ekstrema lokalne. Następnie badany jest empiryczny rozkład ekstremów lokalnych w jednej z kategorii: (a) związek pomiędzy odległością od najlepszego ekstremum lokalnego a wartością funkcji celu, patrz np. Rys. 9.2, (b) związek pomiędzy średnią odległości w przestrzeni rozwiązań a wartości funkcji celu tych rozwiązań, patrz np. Rys. 9.3. Oczywiście kluczem do badań tego typu jest adekwatnie określona miara w przestrzeni rozwiązań. Zwykle celem nadrzędnym jest wykrycie „statystycznych" cech regularności przestrzeni sugerujących występowanie tzw. dużej doliny (*big valley*), zawierającej skupione „dobre" ekstrema lokalne. Wtedy też możliwe jest zastosowanie zasady poszukiwacza grzybów „blisko obok ekstremum lokalnego istnieje drugie ekstremum być może lepsze". Nie wszystkie jednak problemy posiadają „dużą dolinę", co więcej prawdopodobieństwo

znalezienia tam ekstrem globalnego ma charakter statystyczny. Dodatkowo zakładając, że dolina ta zostanie zlokalizowana oraz, że zawiera ekstremum globalne, bezwzględna liczba rozwiązań tam istniejących wyklucza możliwość jej przeszukania w sposób wyczerpujący. Stąd, poruszanie się algorytmu w obrębie wielkiej doliny nie gwarantuje w pełni otrzymanie dobrego rozwiązania, choć pozwala generować rozwiązania bardzo bliskie optymalnemu przy niskim koszcie przeszukania. Lokalizacja obiecujących obszarów przestrzeni rozwiązań pozwala celowo kierować procesami intensyfikacji i dywersyfikacji, dążąc szybko do otrzymania rozwiązania bliskiego optymalnemu.

9.3 Wizualizacja przestrzeni rozwiązań

Celem naszym jest przedstawienie n-wymiarowej przestrzeni rozwiązań w postaci wygodnej dla człowieka, tzn. w formie obrazu 2D lub 3D. Spodziewamy się, że obserwacje poczynione w tak otrzymanym obrazie pomogą nam zaprojektować odpowiedni algorytm (sekwencyjny lub równoległy) oraz ułatwią analizę jego zachowania się w tej przestrzeni.

Rozpoczniemy od pewnych dość obrazowych, lecz zaskakujących porównań. Rozważmy problem szeregowania ze zbiorem rozwiązań opartym na permutacjach (np. problem jednomaszynowy, przepływowy permutacyjny). Rozważmy przykład problemu tego typu dla liczby zadań 50, co stanowi sensowną dolną granicę praktycznej stosowalności (testy Taillard'a zawierają przykłady o n od 20 do 500). Liczba wszystkich rozwiązań problemu dla n=50 jest równa w przybliżeniu $3{\cdot}10^{64}$. Każde rozwiązanie transformujemy wzajemnie jednoznacznie w punkt na płaszczyźnie XY, ustalając jego wysokość Z równą wartości przyjętej funkcji celu. Pominiemy szczegóły przyjętej transformacji zakładając tylko, że zachowuje ona odległość z przestrzeni permutacji. Chcemy uzyskać powierzchnię 3D obrazującą całą przestrzeń rozwiązań. Zakładając, że wysokość punktu kodujemy barwą oraz, że drukujemy barwną mapę używając najlepszej znanej technologii 2400 dpi (pojedynczy pixel ma wymiary 0.01 na 0.01 mm), otrzymamy całkowicie zadrukowaną powierzchnię o wielkości $3{\cdot}10^{48}$ km^2. Dla porównania Jowisz, największa planeta Układu Słonecznego, ma powierzchnię $6.4{\cdot}10^{10}$ km^2. Zauważmy, że najlepsze znane obecnie algorytmy poszukiwań lokalnych sprawdzają 10^3 - 10^9 rozwiązań w celu uzyskania rozwiązań z błędem względnym 1-3%. W skali tej przestrzeni stanowi to obszar poszukiwania o wielkości „zaledwie" 0.1 m^2.

Posługując się analogią astronomiczną można stwierdzić, iż dobierając odpowiednio rozproszone punkty startowe, generowanie trajektorii cyklicz-

nych (z powtórzeniami) powinno być prawie niemożliwe. Stąd fenomen cyklu, obserwowany często w metodzie tabu, należy traktować jako „błąd w sztuce" projektowania algorytmu TS. Można przypuszczać, że zobrazowanie całej przestrzeni rozwiązań w formie 3D wprowadzi tyle zniekształceń spowodowanych redukcją wymiaru, że uzyskane analogie przestaną być przydatne. Stąd bardziej interesujące jest badanie zachowania się algorytmu w lokalnych regionach zbioru rozwiązań, takich jak tzw. „wielka dolina" z przyległościami. Przydomek „wielka" jest tu nie na miejscu, bowiem w skali całej przestrzeni jest ona faktycznie mikroskopijna, co sugeruje, że trudno do niej trafić. Zauważmy także, że tradycyjne „krokowe" poruszanie się po powierzchni może być nieadekwatne do przyjętej technologii wykonywania ruchów. Przykładowo, dla n=50 dowolny punkt w tej przestrzeni (odpowiednio na płaszczyźnie) może być osiągnięty po wykonaniu nie więcej niż 49 ruchów INS, 49 ruchów NPI, 1225 ruchów API.

Idąc za koncepcją wizualizacji lokalnego obszaru przestrzeni rozwiązań, zaproponowana została metoda punktów referencyjnych (punktów odniesienia). Celem naszym jest uzyskanie transformacji T wybranych rozwiązań przestrzeni permutacji P z miarą odległości d w punkty euklidesowej ciągłej przestrzeni k-wymiarowej E^k z odległością e. W praktyce będziemy przyjmowali k=1,2,3. Oczekujemy, że T nie będzie zbyt kosztowne obliczeniowo oraz „w miarę możliwości" zachowa odległość d przekształcając rozwiązania bliskie w P w punkty bliskie w E^k zaś odległe w P - w odległe w E^k.

Podejście jest dwu-etapowe, wyznaczane są odpowiednio dwie transformacje T oraz U. W pierwszym etapie wybieramy zestaw r rozwiązań referencyjnych π_1, π_2, ..., π_r z P, które odwzorowujemy za pomocą T w punkty $p_1, p_2, ..., p_r$ przestrzeni E^k. Położenie tych punktów nie ulegnie już zmianie. W drugim etapie, każde inne rozwiązania z P odwzorowujemy za pomocą U w pojedynczy punkt w E^k nie zmieniając żadnego ustalonego wcześniej punktu oraz korzystając z ustalonych wcześniej rozwiązań i punktów referencyjnych. W każdym z etapów wyznaczamy odpowiednią transformację rozwiązując pewne zadanie optymalizacji. Punkt $p_j = (x_{1j}, x_{2j}, \ldots, x_{kj})$ z E^k otrzymamy jako obraz rozwiązania referencyjnego π_j. Aby zachować odległość, w sensie średnio-kwadratowego błędu, dla wszystkich k rozwiązań referencyjnych, położenie punktów możemy otrzymać przez minimalizację wyrażenia

$$\sum_{i=1}^{r}\sum_{j=1}^{r}\left(e(p_i,p_j)-d(\pi_i,\pi_j)\right)^2 \rightarrow \min_{p_1,\ldots,p_r}, \tag{9.9}$$

gdzie e jest odległością euklidesową punktów na płaszczyźnie, zaś d jest wybraną miarą odległości rozwiązań (permutacji). Dla $r = 3$ oraz $k = 2$ otrzy-

Rysunek 9.1: Wizualizacja przestrzeni rozwiązań.

many problem zawierający 6 zmiennych. Ustalając zakotwiczenie jednego z punktów (aby zapobiec obrotom obrazów) można znaleźć rozwiązanie analityczne problemu dające rozkład punktów referencyjnych w wierzchołkach trójkąta o bokach równych $d(\pi_1, \pi_2)$, $d(\pi_2, \pi_3)$, $d(\pi_3, \pi_1)$. W ogólnym przypadku wymagane jest rozwiązanie odpowiedniego problemu optymalizacyjnego.

Druga transformacja przekształca rozwiązanie π w punkt p. Przyjmując średnio-kwadratową ocenę zniekształcenia odległości, współrzędne punktu otrzymamy poprzez minimalizację wyrażenia

$$\sum_{i=1}^{r} (e(p_i, p) - d(\pi_i, \pi))^2 \rightarrow \min_{p} \tag{9.10}$$

gdzie π_i, $i = 1, 2, \ldots, r$ oraz p_i, $i = 1, 2, \ldots, r$ są ustalone.

Przykład zastosowania tak opisanych transformacji jest przedstawiony na rysunku 9.1. Obraz zawiera zestaw losowych permutacji 20-elementowych reprezentujących rozwiązania losowej instancji problemu $1||\sum w_i T_i$. Jako miara odległości w P została użyta miara tau Kendalla. Miara ta posiada dość słabo występujący efekt „zlepiania się" punktów (kilka elementów przestrzeni rozwiązań jest rzutowanych na jeden punkt przestrzeni dwuwymiarowej), niekorzystny przy redukcji wymiaru przestrzeni. Wartość funkcji celu została odwzorowana w kolory punktów, punktom czerwonym odpowiada wysoka wartość funkcji celu, niebieskim – niska, szare reprezentują wartości pośrednie. Wyróżniono punkty referencyjne oraz optimum (dokładniej, najlepsze wśród analizowanych). Można zauważyć charakterystyczny chaotyczny rozkład ekstremów lokalnych oraz „dolinę" zawierającą rozwiązania o podobnych, bliskich optymalnej wartościach funkcji celu, górna część rysunku. Rozkład ekstremów lokalnych dla innej instancji problemu $1||\sum w_i T_i$ dla $n = 20$ został przedstawiony na Rys. 9.2, 9.3. Istnienie dolin w przestrzeniach rozwiązań problemów kombinatorycznych potwierdzają także inne najnowsze prace, [21, 22, 48, 71, 158]. W analizowanym problemie, występowanie cech charakterystycznych krajobrazu przestrzeni rozwiązań (chaotyczność, doliny) nie zależy od wyboru instancji problemu ani jego rozmiaru. W ogólnym przypadku rozkład dolin i ekstremów lokalnych zmienia istotnie swój charakter wraz ze zmianą klasy problemu. Co więcej, obserwowane są przypadki, w których rozkład ten zależy także od instancji.

W Tab. 9.2 zamieszczono porównanie wartości współczynników korelacji

Rysunek 9.2: Korelacja pomiędzy odległością od optimum globalnego a wartością funkcji celu.

Rysunek 9.3: Korelacja pomiędzy średnią odległością od innych ekstremów lokalnych a wartością funkcji celu.

liniowej Pearsona obliczonej dla opisanej powyżej metody rzutowania. Do prób wylosowano 100 punktów przestrzeni rozwiązań problemu $1||\sum w_i T_i$ dla $n = 20$ i rzutowano je na płaszczyznę, posługując się formułą (9.9) dla 3,4,5 oraz 6 punktów referencyjnych. Odległości pomiędzy punktami przestrzeni rozwiązań (w badanym problemie – permutacjami) obliczano posługując się miarą tau Kendalla. Uzyskane wyniki potwierdzają tezę mówiącą, że zwiększenie ilości punktów referencyjnych wpływa na wzrost korelacji pomiędzy odległością w przestrzeni rozwiązań a odległością ich rzutu na płaszczyźnie.

	ilość punktów referencyjnych			
n	3	4	5	6
10	0,54	0,56	**0,62**	0,60
20	0,44	**0,63**	0,56	0,54
30	0,41	0,52	0,51	**0,55**
40	0,52	0,47	0,55	**0,58**
średnia	0,48	0,55	0,56	**0,57**

Tablica 9.2: Współczynniki korelacji dla różnych metod rzutowania losowo wybranych 100 punktów przestrzeni rozwiązań problemu $1||\sum w_i T_i$ na płaszczyznę.

9.4 Wizualizacja trajektorii

Celem jest przedstawienie trajektorii poszukiwań, wykonywanej przez wybrany algorytm, na tle lokalnego obszaru przestrzeni rozwiązań. Faktycznie, trajektoria ta składa się z punktów przestrzeni odwzorowanych w 2D (odpowiednio do wybranej metody wizualizacji przestrzeni rozwiązań) i generowanych w trakcie działania algorytmu. Należy się spodziewać, że charakter trajektorii odzwierciedli immanentne cechy użytej metody. Na Rys.

Rysunek 9.4: Błądzenie losowe (otoczenie NPI).

Rysunek 9.5: Błądzenie losowe (otoczenie API).

9.4 – 9.7 przedstawiono trajektorie generowane przez metody RANDOM (poszukiwania losowego), SA (symulowanego wyżarzania) i TS (poszukiwania z zabronieniami) dla losowego przykładu problemu jednomaszynowego $1||\sum w_i T_i$ przy $n = 20$. Za najkorzystniejszą należy uznać metodę, która dociera do najbardziej obiecującego obszaru poszukiwań najszybciej.

Na rysunkach 9.8, 9.9, 9.10 oraz 9.11 zaprezentowano trajektorie poszukiwań wielościeżkowego algorytmu tabu dla problemu $F^*||C_{max}$, przykładu Taillarda 20/5/1 [177] wykonującego równolegle 1,2,3 oraz 4 wątki poszukiwań. Zauważyć można, że zwiększenie ilości równoległych procesów poszukiwań pozwala w większym stopniu „pokryć" przestrzeń rozwiązań, równocześnie nie badając tych samych regionów przestrzeni, przy czym trajektorie skupiają się na części przestrzeni posiadającej największą ilość minimów lokalnych.

9.5 Badanie rozłączności trajektorii

Znajomość miar wyznaczających odległość pomiędzy rozwiązaniami podczas równoległego przeglądania przestrzeni rozwiązań pozwala na skuteczne zabezpieczenie algorytmu równoległego przed redundancją poszukiwań (czyli powtarzaniem przeglądania rozwiązań już wcześniej zbadanych przez inny wątek). Strategia taka daje możliwość większego rozproszenia procesów poszukiwań po przestrzeni rozwiązań, co przekłada się bezpośrednio na „elastyczność" algorytmu – w przypadku algorytmów lokalnego poszukiwania zabezpiecza przed wpadnięciem w jeden region, ekstremum lokalne, z którego trudno się jest wydostać.

Pierwsza możliwość wykorzystania miar odległości do zabezpieczenia się prze redundancją poszukiwań polega na badaniu podczas działania algorytmu wzajemnych miar odległości pomiędzy rozwiązaniami bieżąco badanymi przez każdy z równoległych wątków. Jeśli któraś z takich odległości zmniejszy się poniżej pewnego ustalonego progu, nastąpić powinna celowa zmiana

Rysunek 9.6: Trajektoria poszukiwań algorytmu SA.

Rysunek 9.7: Trajektoria poszukiwań algorytmu TS.

Rysunek 9.8: Algorytm TS. Jeden wątek poszukiwań.

kierunku poszukiwań jednego – bądź obu – wątków, nie dopuszczająca do połączenie, bądź skrzyżowania się trajektorii. Rozwiązania takie, proste w implementacji, posiada dwie wady - wymusza dość częstą komunikację, aby informacje dotyczące wzajemnych odległości były aktualne, oraz wymaga wykonania $\frac{p(p-1)}{2} = O(p^2)$ porównań odległości, gdzie p jest liczbą równoległych wątków poszukiwań – bowiem tyle właśnie jest par trajektorii, które potencjalnie mogą się skrzyżować lub połączyć. Wobec tego, rozwiązanie to można zastosować przy implementacji wielościeżkowych algorytmów równoległych rozwijających względnie niedużą liczbę wątków poszukiwań, rzędu kilku- kilkunastu. Przy liczbie wątków rzędu dziesiątek bądź setek zysk wynikający z niedopuszczenia do redundancji poszukiwań kosztem nakładów na wzajemną komunikację – kwadratowo pochodnym liczbie wątków, a więc rzędu setek czy tysięcy operacji komunikacyjnych – wydaje się być nieopłacalny.

Drugie rozwiązanie, bardziej wyrafinowane, polega na skorzystaniu z tablicy haszującej, przechowywanej w pamięci współdzielonej, adresowanej w dość nietypowy sposób. Mianowicie, jeśli do adresowania tablicy zostaną wykorzystane punkty (x, y) rzutu trajektorii na płaszczyznę skonstruowane tak, jak to opisano w Rozdz. 9.3, to adresowanie takie znacząco zwiększy prawdopodobieństwo znalezienia się punktów bliskich w przestrzeni pod tym samym adresem tablicy haszującej. Oczywiście zastosować należy pewną kwantyzację, transformując ciągłą powierzchnię rzutu w kratę – czy raczej szachownicę. Proces ten spowoduje „zlepianie" się tych punktów na płaszczyźnie, które są blisko siebie w przestrzeni rozwiązań, co może być pożyteczne z uwagi na fakt, że punkty te zostaną zapisane pod tym samym adresem w tablicy haszującej. Ostatecznie, dysponować będziemy narzędziem służącym do szybkiego testowania sytuacji zbliżania się do siebie trajektorii poszukiwań, wykorzystującym informacje o wzajemnym przybliżonym położeniu rzutu tych trajektorii na płaszczyźnie. Porównując to podejście do metody wykorzystującej tradycyjną tablicę haszującą, adresowaną na przykład permutacjami, metoda stosująca adresowanie położeniem rzutu trajek-

Rysunek 9.9: Równoległy algorytm TS. Dwa wątki poszukiwań.

Rysunek 9.10: Równoległy algorytm TS. Trzy wątki poszukiwań.

Rysunek 9.11: Równoległy algorytm TS. Cztery wątki poszukiwań.

torii na płaszczyźnie wykorzystuje pewną dodatkową informację, pomijaną przy tradycyjnym adresowaniu tablicy haszującej, zwiększającą prawdopodobieństwo znalezienia się punktów bliskich w przestrzeń rozwiązań na tej samej liście przyporządkowanej pewnemu adresowi w tablicy haszującej. Jeśli pod adresem tym znajduje się lista rozwiązań – można ją przeglądnąć porównując rozwiązania znajdujące się na liście z rozwiązaniem które chcemy dopisać pod danym adresem. Jeśli natomiast pod tym adresem nie są zapisane żadne rozwiązania, można przyjąć, że badane trajektorie nie znajdują się w bezpośrednim sąsiedztwie w przestrzeni rozwiązań.

9.6 Badanie widma trajektorii

Widmem trajektorii nazywamy rozkład wartości funkcji celu rozwiązań generowanych w trakcie pracy algorytmu generującego trajektorię poszukiwań. Na Rys. 9.12 pokazana widmo trajektorii metody SA na tle widma algorytmu poszukiwania losowego RANDOM, zbadane na 10,000 - elementowej próbie dla problemu $F^*||C_{max}$, przykładu Taillarda 20/5/1 [177]. Każde przesunięcie widma w kierunku malejących wartości funkcji kryterialnych należy uważać za korzystne. Porównanie kilku algorytmów poszukiwań lokalnych pozwala wyselekcjonować ten z najlepszymi cechami.

Z kolei na Rys. 9.13 oraz 9.14 zaprezentowano, dla tego samego problemu oraz przykładu jak powyżej, rozkład odległości 10,000 punktów trajektorii metody SA od ekstremum globalnego na tle rozkładu odległości trajektorii algorytmu RANDOM od tego ekstremum, przy czym punkty trajektorii algorytmu SA podzielono na dwie grupy: po 10,000 iteracjach (Rys. 9.13) oraz po 40,000 iteracjach działania algorytmu (Rys. 9.14). Wyraźnie widoczna jest tendencja zbieżności algorytmu (w sensie skracania odległości) do ekstremum; algorytm RANDOM takiej cechy nie wykazuje.

Badania związane z analizą przestrzeni rozwiązań, zawarte w niniejszym rozdziale, wykorzystywane były przy projektowaniu opisanych wcze-

Rysunek 9.12: Widmo trajektorii metody SA na tle widma trajektorii algorytmu RANDOM.

Rysunek 9.13: Rozkład odległości punktów trajektorii metody SA (po wykonaniu 10,000 iteracji) oraz RANDOM od ekstremum globalnego.

Rysunek 9.14: Rozkład odległości punktów trajektorii metody SA (po wykonaniu 40,000 iteracji) oraz RANDOM od ekstremum globalnego.

śniej algorytmów – szczególnie wielowątkowych. Omówione metody analizy trajektorii poszukiwań pozwoliły w prosty sposób na szybkie dostrojenie parametrów działania algorytmu równoległego w taki sposób, by uniknąć redundancji poszukiwań oraz zapewnić praktyczną cechę ogarnięcia poszukiwaniami różnych, rozległych fragmentów przestrzeni rozwiązań.

10

Wnioski końcowe

Odnosząc się to postawionych tez pracy, na podstawie przeprowadzonych badań, można sformułować następujące wnioski. Rozwiązywanie problemów szeregowania można przyspieszyć w środowisku obliczeń równoległych, jednak właściwe zaprojektowanie algorytmu w celu efektywnego wykorzystania mocy obliczeniowej systemu komputerowego jest zadaniem nietrywialnym. Odmiennie od algorytmów sekwencyjnych, które oceniane są zasadniczo w dwóch kategoriach (złożoność obliczeniowa, dokładność), algorytmy równoległe mają co najmniej cztery kategorie oceny (złożoność obliczeniowa, przyspieszenie, koszt, efektywność) będące funkcją liczby zaangażowanych procesorów. Końcowe własności algorytmu równoległego zależą znacząco od wyboru ogólnego, strategicznego podejścia (np. poszukiwania jednowątkowe, wielowątkowe, równoległe schematy B&B), zaś w zakresie ustalonego ogólnego podejścia od właściwego wyboru sub-podejścia oraz doboru jego elementów składowych.

I tak, dla analizowanej strategii poszukiwań wielowątkowych istnieje co najmniej kilka odmiennych sub-podejść budowy algorytmu równoległego: równoległa metoda tabu, równoległe symulowane wyżarzanie, równoległe algorytmy genetyczne (wątki migracyjne, wyspowe). W pracy pokazano, że równoległość realizacji wątków poszukiwań, przy odpowiedniej kooperacji wątków, pozwala uzyskiwać ponad-liniowe przyspieszenie metody równoległej (większe od liczby zaangażowanych procesorów). Bezpośrednią implikacją tej własności jest możliwość uzyskania w czasie T na p procesorach rozwiązania lepszego niż w czasie Tp na jednym procesorze.

Z kolei dla strategii poszukiwań jednowątkowych, w pracy uzyskano odpowiedź na kilka interesujących pytań o charakterze teoretycznym i aplikacyjnym: (1) jakie podejścia mogą być stosowane do projektowania algorytmów równoległych, w kontekście potrzeb wynikających z różnorod-

nych technik poszukiwań lokalnych, (2) które warianty algorytmów równoległych są kosztowo optymalne, (3) jakie elementy są kluczowe do projektowania algorytmów równoległych w kontekście poszukiwań lokalnych i rozproszonych, (4) jakich granicznych wartości przyspieszeń można się spodziewać przy teoretycznie nieograniczonej liczbie procesorów. Ostatnie pytanie ma *aktualnie* charakter czysto teoretyczny, choć przejście od komputerów półprzewodnikowych do biologicznych może uczynić pytanie zdecydowanie praktycznym. Uzyskane teoretyczne wyniki, sformułowane w postaci twierdzeń i własności, zawierają w dowodach oryginalne konstrukcje algorytmiczne, których wszechstronną charakterystykę przedstawiono i przedyskutowano w odpowiednich wnioskach. Dla prostych problemów szeregowania (problemy jednomaszynowe), kluczem do realizacji efektywnej implementacji równoległej jest rezygnacja z rekurencyjnych mechanizmów obliczania wartości funkcji celu, powszechnie stosowana w algorytmach sekwencyjnych. Zabieg ten jest na tyle skuteczny na ile istnieją odpowiednie nierekurencyjne substytuty wzorów rekurencyjnych. Dla regularnych sieci obliczeń (problem przepływowy) metody wykorzystujące planarność sieci są zdecydowanie efektywniejsze niż metody grafowe oparte na algorytmie Floyda-Warshalla, a także efektywniejsze niż metody oparte na modelu kombinatorycznym. Te ostatnie jednak pozwalają uzyskać większe przyspieszenia, jednak kosztem niskiej efektywności. Dla obliczeń z siecią nieregularną (problem gniazdowy, hybrydowy) właściwie pozostają tylko metody oparte o analizę grafu. Dla jednowątkowych poszukiwań lokalnych wykorzystujących skupione lub rozproszone populacje rozwiązań zaproponowano szereg dedykowanych metod, w tym tzw. równoległe akceleratory (będące odpowiednikiem akceleratorów sekwencyjnych). Niezależnie od przyjętej struktury populacji rozwiązań, naturalną granicą tego podejścia jest złożoność obliczeniowa równoległego wyliczania wartości funkcji celu pojedynczego rozwiązania. Równoległe akceleratory mają przy tym wpływ wyłącznie na efektywność wykorzystania środowiska obliczeń

W pracy zaproponowano wiele różnych podejść do projektowania wielowątkowych algorytmów równoległych rozwiązywania problemów szeregowania zadań. Algorytmy te projektowano w oparciu o odmienne strukturalnie techniki. Zbadana została poprawa jakości otrzymywanych rozwiązań, czasu obliczeń i przyspieszenia dla równoległych implementacji algorytmów szeregowania w porównaniu z ich sekwencyjnymi wersjami, przy czym eksperymenty obliczeniowe zostały przeprowadzone dla różnorodnych reprezentatywnych trudnych przykładów z literatury. Oceniono jakość i efektywność proponowanych algorytmów. Część wyników badań została opublikowana w czasopismach krajowych [20–22, 24, 25, 27, 28] i międzynarodo-

wych [23, 26].

Kierunki dalszych badań

Zdaniem autora, niniejsza praca może stanowić inspirację do dalszych badań dotyczących teoretycznych i praktycznych aspektów szeregowania zadań produkcyjnych przy pomocy równoległych systemów komputerowych. Mnogość modeli obliczeń równoległych, a także różnorodność rzeczywistych komputerów równoległych, pozwala na przeprowadzanie dalszych badań nad równoległymi algorytmami dokładnymi i przybliżonymi, ze szczególnym uwzględnieniem tych ostatnich ze względu na ich efektywność. Równoległe wersje algorytmów metaheurystycznych wydają się być jeszcze potężniejszym narzędziem wyznaczania bardzo dobrych jakościowo rozwiązań trudnych problemów kombinatorycznych, takich jak problemy szeregowania zadań, w krótkim czasie, co pozwala na ich naturalne zastosowanie na przykład w systemach działających w czasie rzeczywistym. Odmiennym kierunkiem przyszłych badań może być rozszerzenie własności teoretycznych równoległych metod szeregowania na inne niż przedstawiono w pracy problemy szeregowania zadań produkcyjnych.

Dodatek A

Operatory genetyczne

Operacje krzyżowania i mutacji w algorytmie genetycznym realizowane są poprzez zastosowanie operatorów genetycznych, zmieniających kodowanie rozwiązania w chromosomie. Najlepsze wyniki uzyskuje się stosując dla problemów przepływowych (takich jak $F^*||C_{sum}$ oraz $F^*||C_{max}$) operatory genetyczne bazujące nie na ciągu binarnym, jak to proponowali twórcy metody, lecz na permutacji.

Podstawowymi operatorami genetycznymi działającymi na jednym rodzicu o chromosomie zakodowanym w postaci permutacji π, są:

- *operator inwersji* (I, *inversion*) określony regułą $\pi : \pi(s) \leftrightarrow \pi(t)$ polegającą na zamianie miejscami w permutacji elementów $\pi(s)$ z $\pi(t)$,

- *operator inwersji podciągu* (SI, *subsequence inversion*) określony regułą

$$\pi : \pi(s+j) \leftrightarrow \pi(t-j),\ j = 0, 1..., (t-s-1)/2, \tag{A.1}$$

gdzie wartości $s, t \in \{1, 2, ..., n\}$ wybrano losowo. Operatory jednoargumentowe stosowanymi są głównie do realizacji procesu mutacji.

Klasyczne operatory dwuargumentowe, wykonywane przy operacji krzyżowania to:

- *jednopunktowy operator krzyżowania* (SX) – dla dwóch rodziców π, σ oraz losowego punktu krzyżowania $s \in \{1, 2, ..., n\}$ tworzy, poprzez wymianę odcinka chromosomu, dwóch potomków:

$$\alpha = (\pi(1), \pi(2), \ldots, \pi(s), \sigma(s+1), \sigma(s+2), \ldots, \sigma(n)),$$

oraz

$$\beta = (\sigma(1), \sigma(2), \ldots, \sigma(s), \pi(s+1), \pi(s+2), \ldots, \pi(n)),$$

Każdy potomek jest następnie korygowany dla przywrócenia mu własności permutacji. W tym celu elementy potomka są przeglądane a każdy

element, który pojawia się dwukrotnie jest zastępowany najmniejszym elementem zbioru J elementów nie występujących w potomku.

- *dwupunktowy operator krzyżowania* (PMX, *partially matched crossover*) – dla dwóch rodziców π, σ oraz sekcji dopasowania określonej losową parą punktów $s, t \in \{1, 2, ..., n\}$, $s < t$ tworzy dwie permutacje potomne dopasowując π do σ poprzez wymianę pomiędzy chromosomami odcinka *s,...,t,* to znaczy $\pi(s) \leftrightarrow \sigma(t)$, *j=s,...,t* oraz dopasowując σ do π poprzez wymianę $\sigma(s) \leftrightarrow \pi(t)$, *j=s,...,t.* Każdy potomek jest następnie korygowany w celu przywrócenia mu własności permutacji według zasady: każdy element, który pojawia się dwukrotnie jest zastępowany najmniejszym elementem zbioru J elementów nie występujących w potomku.

- *operator krzyżowania porządkowego* (OX, *order crossover*) – jego działanie jest podobne jak operatora PMX z tym, że po pobraniu z sekcji dopasowania elementów od jednego z rodziców, reszta permutacji potomka jest dobierana od drugiego rodzica w kolejności ich występowania w chromosomie (drugiego rodzica) tak, aby uzyskać rozwiązanie dopuszczalne.

- *operator krzyżowania pozycyjnego* (PBX, *position-based crossover*) – działanie podobne jak operatora OX z tym, że zamiast sekcji dopasowania geny (pozycje w permutacji) podlegające wymianie wybierane są u rodziców losowo, z równomiernym rozkładem prawdopodobieństwa. Geny te są kopiowane do potomka z zachowaniem ich położenia w chromosomie rodzica, zaś brakujące geny są uzupełniane od drugiego rodzica w kolejności ich występowania.

- *operator krzyżowania porządkowego* (OBX, *order-based crossover*) jest modyfikacją operatora PBX, w której kolejność genów występujących na pozycjach wybranych u jednego z rodziców i przekazywanych potomkowi jest dopasowywana do kolejności ich występowania u drugiego z rodziców.

- *operator krzyżowania cyklicznego* (CX, *cycle crossover*) dla dwóch rodziców π, σ tworzy dwóch potomków, w których pozycja zajmowana przez każdy element w permutacji potomka pochodzi od jednego z rodziców. Rozważając przypadek dopasowania π do σ utworzona zostaje permutacja potomna γ w następujący sposób (dopasowanie odwrotne jest symetryczne): niech pozycja k=1; następnie powtarzamy ciąg podstawień $\gamma\ (k) = \pi(k)$, $k = \bar{\pi}(\sigma(k))$ (gdzie $\bar{\pi}$ jest permutacją odwrotną do permutacji π , czyli spełnia zależność $\bar{\pi}(\pi(i)) = i$) aż do chwili zamknięcia cyklu, to znaczy osiągnięcia stanu $k = \pi(1)$. Pozostałe wolne pozycje w permutacji potomnej γ zapełniane są niewykorzystanymi elementami z permutacji σ w kolejności ich występowania.

- *operator krzyżowania z porządkiem liniowym* (LOX, *linear order crossover*) jest modyfikacją operatora OX. Dla dwóch rodziców π, σ oraz sekcji

dopasowania określonej losową parą punktów $s, t \in \{1, 2, ..., n\}$, $s < t$ tworzy dwie permutacje potomne w dwóch etapach. Rozważając przypadek dopasowania π do σ utworzona zostaje permutacja potomna γ w następujący sposób (dopasowanie odwrotne jest symetryczne): najpierw wstawiamy do γ tylko te elementy z π które nie występują w σ; wstawienia te dokonujemy na te same pozycje, na których elementy te występowały w π (pozostawiając pomiędzy nimi dziury – nie zapełnione pozycje). Następnie przeprowadzamy w γ proces redukcji dziur analizując pozycje w kolejności cyklicznej t+1, t+2,...,n,1,2,...,$s-1$. Dziurę na pozycji k redukujemy przesuwając cyklicznie elementy γ(k+1), γ(k+2),..., $\gamma(n)$, $\gamma(1)$, $\gamma(2)$,..., $\gamma(t)$ w lewo. Ostatecznie otrzymujemy permutację zawierającą dziury tylko w sekcji dopasowania na pozycjach od s do t, które zapełniamy elementami $\sigma(s)$, $\sigma(s+1)$,..., $\sigma(t)$.

Oprócz wymienionych wyżej klasycznych operatorów genetycznych istnieją także inne, bardziej efektywne (lecz także znacznie bardziej złożone obliczeniowo) quasi-operatory krzyżowania i mutacji jak na przykład operator *wielokrokowej fuzji* (MSXF) czy *wielokrokowej mutacji* (MSMF) [156].

Dodatek B

Generowanie podciągów

Lemat 1 *Wszystkie podciągi* $(j_0, j_1, \ldots, j_m)$ *spełniające warunek* $1 = j_0 \leqslant j_1 \leqslant \ldots \leqslant j_m = n$ *można wygenerować w czasie* $O(m)$ *za pomocą* $\binom{n+m-2}{m-1}$ *procesorowej maszyny CREW PRAM.*

Dowód Liczba wszystkich omawianych podciągów odpowiada liczbie kombinacji $m-1$ elementowych z powtórzeniami ze zbioru $n-2$ elementowego, i wynosi $\binom{n+m-2}{m-1}$. Wszystkie podciągi można wygenerować na maszynie CREW PRAM w czasie rzędu długości podciągu, czyli $O(m)$, za pomocą liczby procesorów równej liczbie podciągów, wykorzystując drzewiasty schemat generowania. Najpierw n procesorów generuje n podciągów długości 2: $(1,1),(1,2),\ldots,(1,n)$. Następnie, w oparciu o wygenerowane już podciągi, n procesorów generuje podciągi długości 3 postaci: $(1,1,1),(1,1,2),\ldots,(1,1,n)$; $n-1$ procesorów generuje podciągi długości 3 postaci: $(1,2,2),(1,2,3),\ldots,(1,2,n)$; $n-2$ procesorów generuje podciągi długości 3 rozpoczynające się od (1,3), itd. Na tym etapie (w drugiej iteracji) wykorzystanych zostanie $\binom{n+3-2}{3-1} = \binom{n+1}{2} = \frac{n(n+1)}{2}$ procesorów. Ciągi pamiętane są jako listy, co oznacza że poszczególne elementy ciągów połączone są wskaźnikami – generowanie następnego elementu ciągu nie wymaga kopiowania poprzednich elementów, ale tylko wygenerowania następnego elementu ciągu i wskaźnika do elementu poprzedzającego, w czasie $O(1)$. Postępując w ten sposób, po $m-1$ iteracjach wygenerowane zostaną $\binom{n+m-2}{m-1}$ podciągi długości m i przydzielone będą do generujących je $\binom{n+m-2}{m-1}$ procesorów (każdy z procesorów związany będzie z ostatnim elementem listy, posiadającym wskaźniki do poprzednich elementów). W $m+1$ kroku każdy z procesorów wygeneruje ostatni element ciągu, dla wszystkich ciągów identyczny, $j_m = n$.■

Dodatek C

Tabele zestawień

Zastosowano następujące oznaczenia:

p	–	liczba procesorów,
$T_{A_p,M}(p)$	–	złożoność obliczeniowa algorytmu równoległego używającego p procesorów,
T_{A_s}	–	złożoność obliczeniowa algorytmu sekwencyjnego,
$s_{A_p,M}(p)$	–	przyspieszenie,
$\eta_{A_p,M}(p)$	–	efektywność.

problem	p	$T_{A_p,M}(p)$	T_{A_s}	Tw.
$1\|\|\sum f_i$	$O(\frac{n}{\log n})$	$O(\log\ n)$	$O(n)$	1*
$1\|r_j\|\sum f_i$	$O(\frac{n^2}{\log n})$	$O(\log\ n)$	$O(n^2)$	2
$F^*\|\|\sum f_i$	m	$O(n+m)$	$O(nm)$	3
$F^*\|\|\sum f_i$	$O(\frac{nm}{n+m})$	$O(n+m)$	$O(nm)$	4*
$F^*\|\|f_{max}$	$O(\frac{nm}{n+m})$	$O(n+m)$	$O(nm)$	5*
$F^*\|\|C_{max}$	$O(\frac{(n+m)^{n-1}}{m(n-1)!})$	$O(m+\log n)$	$O(nm)$	6
$F^*\|\|\sum f_i$	$O(\frac{(nm)^3}{\log(nm)})$	$O(\log(n+m)\ \log\ (nm))$	$O(nm)$	7
$F^*\|\|f_{max}$	$O(\frac{(nm)^3}{\log(nm)})$	$O(\log(n+m)\ \log\ (nm))$	$O(nm)$	8
$J\|\|\sum f_i$	$O(\frac{o^3}{\log o})$	$O(\log^2\ o)$	$O(o)$	9
$J\|\|C_{max}$	$O(\frac{o^3}{\log o})$	$O(\log o \log \Lambda)$	$O(o)$	10
$FP\|\|\sum f_i$	$O(\frac{(nm)^3}{\log(nm)})$	$O(\log^2(nm))$	$O(nm)$	11
$FP\|\|C_{max}$	$O(\frac{(nm)^3}{\log(nm)})$	$O(\log(nm) \log \Lambda)$	$O(nm)$	12

* zaznaczono metody kosztowo optymalne

Tablica C.1: Metody równoległe wyznaczania pojedynczej wartości funkcji kryterialnej. Porównanie złożoności obliczeniowej.

problem	p	$s_{A_p,M}(p)$	$\eta_{A_p,M}(p)$	Tw.
$1\|\|\sum f_i$	$O(\frac{n}{\log n})$	$O(\frac{n}{\log n})$	$O(1)$	1*
$1\|r_j\|\sum f_i$	$O(\frac{n^2}{\log n})$	$O(\frac{n}{\log n})$	$O(\frac{1}{n})$	2
$F^*\|\|\sum f_i$	m	$O(\frac{nm}{n+m})$	$O(\frac{n}{n+m})$	3
$F^*\|\|\sum f_i$	$O(\frac{nm}{n+m})$	$O(\frac{nm}{n+m})$	$O(1)$	4*
$F^*\|\|f_{max}$	$O(\frac{nm}{n+m})$	$O(\frac{nm}{n+m})$	$O(1)$	5*
$F^*\|\|C_{max}$	$O(\frac{(n+m)^{n-1}}{m(n-1)!})$	$O(\frac{mn}{m+\log n})$	$O(\frac{m^2n!}{(m+\log n)(n+m)^{n-1}})$	6
$F^*\|\|\sum f_i$	$O(\frac{(nm)^3}{\log(nm)})$	$O(\frac{nm}{\log(n+m)\log(nm)})$	$O(\frac{1}{(nm)^2\log(n+m)})$	7
$F^*\|\|f_{max}$	$O(\frac{(nm)^3}{\log(nm)})$	$O(\frac{nm}{\log(n+m)\log(nm)})$	$O(\frac{1}{(nm)^2\log(n+m)})$	8
$J\|\|\sum f_i$	$O(\frac{o^3}{\log o})$	$O(\frac{o}{\log^2 o})$	$O(\frac{1}{o^2\log o})$	9
$J\|\|C_{max}$	$O(\frac{o^3}{\log o})$	$O(\frac{o}{\log o\log\Lambda})$	$O(\frac{1}{o^2\log\Lambda})$	10
$FP\|\|\sum f_i$	$O(\frac{(nm)^3}{\log(nm)})$	$O(\frac{mn}{\log^2 mn})$	$O(\frac{1}{(mn)^2\log(mn)})$	11
$FP\|\|C_{max}$	$O(\frac{(nm)^3}{\log(nm)})$	$O(\frac{mn}{\log mn\log\Lambda})$	$O(\frac{1}{(mn)^2\log\Lambda})$	12

* zaznaczono metody kosztowo optymalne

Tablica C.2: Metody równoległe wyznaczania pojedynczej wartości funkcji kryterialnej. Porównanie przyspieszenia i efektywności.

Otoczenie	p	$T_{A_p,M}(p)$	T_{A_s}	Tw.
API	$O(n)$	$O(\log n)$	$O(n)$	13
API	$O(n/\log n)$	$O(\log n)$	$O(n)$	14*
INS	$O(n)$	$O(n)$	$O(n^2)$	15*
INS	$O(n^3/\log n)$	$O(\log n)$	$O(n^2)$	16
INS	$O(n^2/\log n)$	$O(\log n)$	$O(n^2)$	17*
NPI	$O(n^3/\log n)$	$O(\log n)$	$O(n^3)$	18*

* zaznaczono metody kosztowo optymalne

Tablica C.3: Przeglądanie otoczenia w problemie $1||\sum f_i$. Złożoności obliczeniowe.

Otoczenie	p	$s_{A_p,M}(p)$	$\eta_{A_p,M}(p)$	Tw.
API	$O(n)$	$O(n)$	$O(1/\log n)$	13
API	$O(n/\log n)$	$O(n/\log n)$	$O(1)$	14*
INS	$O(n)$	$O(n)$	$O(1)$	15*
INS	$O(n^3/\log n)$	$O(n^2/\log n)$	$O(1/n)$	16
INS	$O(n^2/\log n)$	$O(n^2/\log n)$	$O(1)$	17*
NPI	$O(n^3/\log n)$	$O(n^3/\log n)$	$O(1)$	18*

* zaznaczono metody kosztowo optymalne

Tablica C.4: Przeglądanie otoczenia w problemie $1||\sum f_i$. Przyspieszenia i efektywność.

Otoczenie	p	$T_{A_p,M}(p)$	T_{A_s}	Tw.
API	$O(\frac{n^2m}{n+m})$	$O(n+m)$	$O(nm)$	19
API	$O(\frac{nm}{n+m})$	$O(n+m)$	$O(nm)$	20*
API	$O(\frac{n^4m^3}{\log(nm)})$	$O(\log(n+m)(\log(nm)))$	$O(nm)$	21
API	$O(\frac{(nm)^3}{\log(nm)})$	$O(\log(n+m)(\log(nm)))$	$O(nm)$	22
INS	$O(\frac{n^2m}{n+m})$	$O(n+m)$	$O(n^2m)$	23*
INS	$O(\frac{n^5m^3}{\log(nm)})$	$O(\log(n+m)(log(nm)))$	$O(n^2m)$	24
INS	$O(\frac{(nm)^3}{\log(nm)})$	$O(m+\log(n+m)(\log(nm)))$	$O(n^2m)$	25
INS	$O(\frac{(nm)^3}{\log(nm)})$	$O(\log(n+m)(\log(nm)))$	$O(n^2m)$	26
NPI	$O(n^2m)$	$O(nm)$	$O(n^2m^2)$	27
NPI	$O(\frac{(nm)^3}{\log(nm)})$	$O(m\log(n+m)(\log(nm)))$	$O(n^2m^2)$	28
NPI	$O(\frac{n^2m^2}{n+m})$	$O(n+m)$	$O(n^2m^2)$	29*
NPI	$O(\frac{(nm)^3}{\log(nm)})$	$O(\log(n+m)(\log(nm)))$	$O(n^2m^2)$	30

* zaznaczono metody kosztowo optymalne

Tablica C.5: Złożoność obliczeniowa różnych metod równoległego przeglądania otoczeń w problemie $F^*||C_{max}$.

Otocz.	p	$s_{A_p,M}(p)$	$\eta_{A_p,M}(p)$	Tw.
API	$O(\frac{n^2m}{n+m})$	$O(\frac{nm}{n+m})$	$O(\frac{1}{n})$	19
API	$O(\frac{nm}{n+m})$	$O(\frac{nm}{n+m})$	$O(1)$	20*
API	$O(\frac{n^4m^3}{\log(nm)})$	$O(\frac{nm}{\log(n+m)\log(nm)})$	$O(\frac{1}{n^3m^2\log(n+m)})$	21
API	$O(\frac{(nm)^3}{\log(nm)})$	$O(\frac{nm}{\log(n+m)\log(nm)})$	$O(\frac{1}{(nm)^2\log(n+m)})$	22
INS	$O(\frac{n^2m}{n+m})$	$O(\frac{n^2m}{n+m})$	$O(1)$	23*
INS	$O(\frac{n^5m^3}{\log(nm)})$	$O(\frac{n^2m}{\log(n+m)\log(nm)})$	$O(\frac{1}{n^3m^2\log(n+m)})$	24
INS	$O(\frac{(nm)^3}{\log(nm)})$	$O(\frac{n^2m}{m+\log(n+m)\log(nm)})$	$O(\frac{1}{nm^2}\cdot \frac{\log(nm)}{m+\log(n+m)\log(nm)})$	25
INS	$O(\frac{(nm)^3}{\log(nm)})$	$O(\frac{n^2m}{\log(n+m)\log(nm)})$	$O(\frac{1}{nm^2\log(n+m)})$	26
NPI	$O(n^2m)$	$O(nm)$	$O(\frac{1}{n})$	27
NPI	$O(\frac{(nm)^3}{\log(nm)})$	$O(\frac{n^2m^2}{m+\log(n+m)\log(nm)})$	$O(\frac{1}{nm}\cdot \frac{\log(nm)}{m+\log(n+m)\log(nm)})$	28
NPI	$O(\frac{n^2m^2}{n+m})$	$O(\frac{n^2m^2}{n+m})$	$O(1)$	29*
NPI	$O(\frac{(nm)^3}{\log(nm)})$	$O(\frac{n^2m^2}{\log(n+m)\log(nm)})$	$O(\frac{1}{nm\log(n+m)})$	30

* zaznaczono metody kosztowo optymalne

Tablica C.6: Przyspieszenie i efektywność różnych metod równoległego przeglądania otoczeń w problemie $F^*||C_{max}$.

n	m	otoczenie		
		API	INS	NPI
		nm	n^2m	n^2m^2
20	5	100	2 000	10 000
20	10	200	4 000	40 000
20	20	400	8 000	160 000
50	5	250	12 500	62 500
50	10	500	25 000	250 000
50	20	1000	50 000	1 000 000
100	5	500	50 000	250 000
100	10	1000	100 000	1 000 000
100	20	2000	200 000	4 000 000
500	5	2500	1 250 000	6 250 000
500	10	5000	2 500 000	25 000 000
500	20	10000	5 000 000	100 000 000

Tablica C.7: Porównanie szybkości wzrostu funkcji złożoności obliczeniowej różnych metod *sekwencyjnego* przeglądania otoczeń problemu $F^*||C_{max}$.

n	m	otoczenie			
		NPI	API INS NPI	API INS NPI	API INS NPI
		nm	$n+m$	$\lceil \log(n+m)\log(nm) \rceil$	$\lceil m+\log(n+m)\log(nm) \rceil$
20	5	100	25	31	36
20	10	200	30	38	48
20	20	400	40	47	67
50	5	250	55	47	52
50	10	500	60	53	63
50	20	1000	70	62	82
100	5	500	105	61	66
100	10	1000	110	68	78
100	20	2000	120	76	96
500	5	2500	505	102	107
500	10	5000	510	111	121
500	20	10000	520	120	140

Tablica C.8: Porównanie szybkości wzrostu funkcji złożoności obliczeniowej różnych metod *równoległego* przeglądania otoczeń problemu $F^*||C_{max}$.

n	m	otoczenie			
		API	INS	NPI	API INS NPI
		$\left\lceil \frac{nm}{n+m} \right\rceil$	$\left\lceil \frac{n^2m}{n+m} \right\rceil$	$\left\lceil \frac{n^2m^2}{n+m} \right\rceil$	$\left\lceil \frac{(nm)^3}{\log(nm)} \right\rceil$
20	5	5	100	500	150 515
20	10	7	140	1 400	1 046 593
20	20	11	220	4 400	7 404 103
50	5	5	250	1 250	1 961 515
50	10	9	450	4 500	$1,4 \cdot 10^7$
50	20	15	750	15 000	$1,0 \cdot 10^8$
100	5	5	500	2 500	$1,4 \cdot 10^7$
100	10	10	1 000	10 000	$1,0 \cdot 10^8$
100	20	17	1 700	34 000	$7,3 \cdot 10^8$
500	5	5	2 500	12 500	$1,4 \cdot 10^9$
500	10	10	5 000	50 000	$1,0 \cdot 10^{10}$
500	20	20	10 000	200 000	$7,5 \cdot 10^{10}$

Tablica C.9: Porównanie szybkości wzrostu liczby procesorów różnych metod przeglądania otoczeń problemu $F^*||C_{max}$.

Literatura

[1] Aarts E.H.L., de Bont F.M.J., Habers J.H.A., van Laarhoven P.J.M., Parallel implementations of the statistical cooling algorithm, Integration 4 (1986), 209–238.

[2] Aarts E.H.L., Verhoeven M., Local search, in Annotated bibliographies in Combinatorial Optimization (Dell'Amico M., Maffioli F., Martello S., eds.), 163–180, Wiley, 1997.

[3] Abramson D., Abela J., A parallel genetic algorithm for solving the school timetabling problem, Proceedings of the 15th Australian Computer Science Conference, 1–11, 1992.

[4] Adleman L., Molecular computation of solutions to combinatorial problems. Science (266) 1994, 1021-1024.

[5] Adrabiński A., Grabowski J., Wodecki M., Algorytm rozwiązywania zagadnienia kolejnościowego postaci $1\|\sum w_iT_i$, Archiwum Automatyki i Telemechaniki, Tom XXXIII (1988), 623-636.

[6] Aiex R.M., Martins S.L., Ribeiro C.C., Rodriguez N.R., Cooperative multithread parallel tabu search with an application to circuit partitioning, Lecture Notes in Computer Science 1457 (1998), 310–331.

[7] Aiex R.M., Resende M.G.C., Pardalos P.M., Toraldo G., GRASP with path-relinking for the threeindex assignment problem, w druku, 2000.

[8] Alaoui S.M., Frieder O., ElGhazawi T., A parallel genetic algorithm for task mapping on parallel machines, Lecture Notes in Computer Science 1586 (1999), 201–209.

[9] Alba E., Troya J.M., Influence of the migration policy in parallel distributed GAs with structured and panmictic populations, Applied Intelligence 12 (2000), 163–181.

[10] Alba E., Troya J.M., Analyzing synchronous and asynchronous parallel distributed genetic algorithms, Future Generation Computer Systems 17 (2001), 451–465.

[11] Allwright J.R., Carpenter D.B., A distributed implementation of simulated annealing for the travelling salesman problem, Parallel Computing 10 (1989), 335–338.

[12] Andre D., Koza J.R., Parallel genetic programming: A scalable implementation using the transputer network architecture, in Advances in Genetic Programming 2 (Angeline P.J., Kinnear K.E. Jr., eds.), 317–338, MIT Press, 1996.

[13] Argonne National Laboratory, PGAPack Parallel Genetic Algorithm Library, dokument on-line, http://wwwfp.mcs.anl.gov/ CCST/ research/ reports/ pre1998/ comp bio/ stalk/ pgapack.html,

[14] Azencott R., Simulated annealing: Parallelization techniques, Wiley, 1992.

[15] Badeau P., Guertin F., Gendreau M., Potvin J.Y., Taillard E., A parallel tabu search heuristic for the vehicle routing problem with time windows, Transportation ResearchC 5 (1997), 109–122.

[16] Bastos M.P., Ribeiro C.C., Reactive tabu search with path relinking for the Steiner problem in graphs, in Essays and surveys in metaheuristics (Ribeiro C.C., Hansen P., eds.), Kluwer, 2001

[17] Belding T.C., The distributed genetic algorithm revisited, Proceedings of the Sixth International Conference on Genetic Algorithms (Eschelman L., ed.), 114–121, Morgan Kaufmann, 1995.

[18] Błażewicz J., Złożoność obliczeniowa problemów kombinatorycznych, WNT Warszawa 1988

[19] Bolc J., Cytowski S., Algorytmy przeszukiwania heurystycznego, Warszawa 1980

[20] Bożejko W., Grabowski J., Wodecki M. Stabilność metaheurystyk dla wybranych problemów szeregowania zadań, Komputerowo Zintegrowane Zarządzanie, Tom I, WNT Warszawa 2001, 95 - 103

[21] Bożejko W., Smutnicki C., Metody przeszukiwań dyskretnych przestrzeni rozwiązań. Automatyka z. 131, 25 - 35

[22] Bożejko W., Smutnicki C., Własności algorytmów przeszukiwań lokalnych Automatyka, Tom 5 Zeszyt 1/2, Kraków 2001, 95 - 102

[23] Bożejko W., Wodecki M., Solving the flow shop problem by parallel simulated annealing, Lecture Notes in Computer Science 2328, Springer Verlag 2002, 236-247

[24] Bożejko W., Wodecki M., Algorytm równoległy szeregowania zadań oparty na metodzie podziału i ograniczeń, Materiały II Krajowej Konferencji Metody i systemy komputerowe w badaniach naukowych i projektowaniu inżynierskim, Kraków 1999, 591 - 596

[25] Bożejko W., Wodecki M., Algorytmy równoległe dla permutacyjnych problemów szeregowania zadań, Materiały III Krajowej Konferencji Metody i systemy komputerowe w badaniach naukowych i projektowaniu inżynierskim, Kraków 2002, 421-425

[26] Bożejko W., Wodecki M., Solving the flow shop problem by parallel tabu search, IEEE Computer Society PR01730 ISBN 0-7695-1730-7 2002, 189-194

[27] Bożejko W., Wodecki M., Parallel algorithm for some single machine scheduling problems Automatyka z. 134, 81 - 90

[28] Bożejko W., Wodecki M., Permutacyjny problem przepływowy. Algorytmy równoległe symulowanego wyżarzania Automatyka z. 134, 90-101

[29] Bożejko W., Wodecki M., Genetyczny algorytm równoległy dla pewnego permutacyjnego problemu przepływowego, Automatyka, 2003 (praca przyjęta do druku)

[30] Brunner R.K., Kale L.V., Adapting to load on workstation clusters, Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation, 106–112, IEEE Computer Society Press.

[31] Bubak M., Sowa K., Objectoriented implementation of parallel genetic algorithms, in High Performance Cluster Computing: Programming and Applications (Buyya R., ed.), vol. 2, 331–349, Prentice Hall, 1999.

[32] Butenhof D., Programming with POSIX Threads, Addison Wesley, 1997.

[33] Cantú-Paz E., A survey of parallel genetic algorithms, Calculateurs Paralleles, Reseaux et Systemes Repartis 10 (1998), 141–171.

[34] Cantú-Paz E., Implementing fast and flexible parallel genetic algorithms, in Practical handbook of genetic algorithms (Chambers L.D. , ed.), volume III, 65–84, CRC Press, 1999.

[35] Cantú-Paz E., Effcient and Accurate Parallel Genetic Algorithms, Kluwer, 2000.

[36] Cantú-Paz E., Goldberg D.E., Parallel genetic algorithms: Theory and Practice, Computer Methods in Applied Mechanics and Engineering 186 (2000), 221–238.

[37] Canuto S.A., Resende M.G.C., Ribeiro C.C., Local search with perturbations for the prizecollecting Steiner tree problem in graphs, Networks, w druku.

[38] Carriero N., Gelernter D., How to write parallel programs: A guide to the perplexed, ACM Computing Surveys 21 (1989), 323–357.

[39] Carriero N., Gelernter D., Linda in context, Communications of the ACM 32 (1989), 444–458.

[40] Chakrapani J., SkorinKapov J., Massively parallel tabu search for the quadratic assignment problem, Annals of Operations Research 41 (1993), 327–341.

[41] Chakrapani J., SkorinKapov J., Connection Machine implementation of a tabu search algorihm for the traveling salesman problem, Journal of Computing and Information Technology 1 (1993), 29–63.

[42] Chalermwat P., ElGhazawi T., LeMoigne J., 2phase GAbased image registration on parallel clusters, Future Generation Computer Systems 17 (2001), 467–476.

[43] Chandy J.A., Kim S., Ramkumar B., Parkes S., Banerjee P., An evaluation of parallel simulated annealing strategies with applications to standard cell placement, IEEE Transactions on Computer Aided Design 16 (1997), 398–410.

[44] Chipperfield A., Fleming P., Parallel genetic algorithms, Parallel and Distributed Computing Handbook (Zomaya A.Y., ed.), 1118–1143, McGrawHill, 1996.

[45] Colorni A., Dorigo M., Maniezzo V., Distributed optimization by ant colonies, Proceedings of the European Conference on Artificial Life, 134–142, Elsevier, 1991.

[46] Congram R.K., Potts C.N., van de Velde S.L., An Iterative Dynasearch Algorithm for Single-Machine Total Weighted Tardiness Scheduling Problems, Technical Report, 1998.

[47] Cook S., Dwork C., Reischuk R., Upper and lower time bounds for parallel random access machines without simultaneous writes. SIAM Journal on Computing, 15(1), 1986, 87 – 97.

[48] Corne D., Drigo M., Glover F., New Ideas in Optimization. McGraw-Hill, London, 1999

[49] da Costa U.S., D'eharbe D.B., Moreira A.M., Variable orderings of BDDs with parallel genetic algorithms, Proceedings of the International Conference on Parallel and Distributes Processing Techniques and Applications, 1181–1186, CSREA Press, 2000.

[50] Crainic T.G., Gendreau M., Cooperative parallel tabu search for capacitated network design, Journal of Heuristics, w druku

[51] Crainic T.G., Toulouse M., Parallel metaheuristics, in Fleet management and logistics (T.G. Crainic and G. Laporte, eds.), 205–251, Kluwer, 1998.

[52] Crainic T.G., Toulouse M., Gendreau M., Parallel asynchronous tabu search in multicommodity locationallocation with balancing requirements, Annals of Operations Research 63 (1995), 277–299.

[53] Crainic T.G., Toulouse M., Gendreau M., Synchronous tabu search parallelization strategies for multicommodity locationallocation with balancing requirements, OR Spektrum 17 (1995), 113–123.

[54] Crainic T.G., Toulouse M., Gendreau M., Towards a taxonomy of parallel tabu search algorithms, INFORMS Journal of Computing 9 (1997), 61–72.

[55] Crauwels H.A.J., Potts C.N., Van Wassenhove L.N., Local search heuristics for the single machine total weighted tardiness scheduling problem, Informs Journal on Computing 10 (1998), 341-350.

[56] Cormen T.H., Leiserson C.E., Rivest R. L., Wprowadzenie do algorytmów, WNT Warszawa 1997

[57] Cung V.D., Cun L., An efficient implementation of parallel A*, LNCS, Parallel and Distributed computing: Theory and Practice, Nr 805 (1994) 153 - 168.

[58] Czech Z., Three parallel algorithms for simulated annealing, Lecture Notes in Computer Science 2328, Springer Verlag 2002, 210–217.

[59] Czech Z., Programowanie współbierzne. Wybrane zagadnienia, praca zbiorowa, skrypt Politechniki Śląskiej, Gliwice 1991.

[60] Dagum L., Menon R., OpenMP: An industrystandard API for shared-memory programming, IEEE Computational Science and Engineering 5 (1998), 46–55.

[61] Dudek-Dyduch E., Leigh D., The Travelling Salesman Problem - Parallel Algorithms for Distributed Systems, Automatyka 1997.

[62] De Falco I., Del Balio R., Della Cioppa A., Tarantino E., A parallel genetic algorithm for transonic airfoil, Proceedings of the IEEE International Conference on Evolutionary Computing, 429–434, University of Western Australia, 1995.

[63] Diaconis P.: Group Representations in Probability and Statistics. Lecture Notes - Mono-graph Series Vol. 11, Institute of Mathematical Statistics, Harvard University 1988.

[64] Fachat A., Hoffman K.H., Implementation of ensemblebased simulated annealing with dynamic load balancing under MPI, Computer Physics Communications 107 (1997), 49–53.

[65] Felten E., Karlin S.C., Otto S.W., The traveling salesman problem on a hypercubic, MIMD computer, Proceedings of the 1985 International Conference on Parallel Processing, 6–10, 1985.

[66] Fernandez F., Sanchez J.M., Tomassini M., Gomez J.A., A parallel genetic programming tool based on PVM, Lecture Notes in Computer Science 1697, Springer Verlag 1999, 241–248.

[67] Fiechter C.N., A parallel tabu search algorithm for large traveling salesman problems, Discrete Applied Mathematics 51 (1994), 243–267.

[68] Flynn M.J., Very highspeed computing systems, Proceedings of the IEEE 54 (1966), 1901– 1909.

[69] Foster I., Designing and building parallel programs: Concepts and tools for parallel software engineering, AddisonWesley, 1995.

[70] Frost R., Ensemble Based Simulated Annealing, dokument on-line, http://wwwrohan.sdsu.edu/frostr/Ebsa/Ebsa.html, 2001.

[71] Fonlupt C., Robilliard D.: Fitness Landscapes nad Performance of Meta-Heuristics. 2nd Int. Conf. on Metaheuristics (MIC97) 1997, 1–13.

[72] Garcia B.L., Potvin J.Y., Rousseau J.M., A parallel implementation of the tabu search heuristic for vehicle routing problems with time windows constraints, Computers and Operations Research 21 (1994), 1025–1033.

[73] Garey M.R., Johnson D.S., Seti R., The complexity of flowshop and jobshop schedul-ing, Mathematics of Operations Research, 1, (1976), 117–129.

[74] Gendreau M., Guertin F., Potvin J.Y., Taillard E., Parallel tabu search for realtime vehicle routing and dispatching, Transportation Science 33 (1999), 381–390.

[75] Globus A., Lawton J. and Wipke T., Automatic molecular design using evolutionary techniques, Nanotechnology 10 (1999), 290–299.

[76] Glover F., Tabu search Part I, ORSA Journal on Computing 1 (1989), 190–206.

[77] Glover F., Tabu search Part II, ORSA Journal on Computing 2 (1990), 4–32.

[78] Glover F., Laguna M., Tabu Search, Kluwer, 1997.

[79] Grabowski J., A new algorithm of solving the flow-shop problem, Operations Research in Progress, D. Reidel Publishing Company, (1982), 57–75.

[80] Grabowski J., Pempera J., New block properties for the permutation flow-shop problem with application in TS, Journal of Operational Research Society 52, (2001), 210–220.

[81] Graham R.L., Lawler E.L., Lenstra J.K, Rinnooy Kan A.H.G., Optimization and approximation in deterministic sequencing and scheduling, Annals of Discrete Mathematics 5, 1979, 287–326.

[82] Greening D.R., Parallel simulated annealing techniques, Physica D 42 (1990), 293–306.

[83] Grefenstette J.J., Parallel adaptive algorithms for function optimizations, Technical report CS8119, Vanderbilt University, 1981.

[84] Gropp W., Huss-Lederman S., Lumsdaine A., Lusk E., Nitzberg B., Saphir W., Snir M., MPI: The complete reference, Volume 2 The MPI extensions, MIT Press, 1998.

[85] Gropp W., Lusk E., Skjellum A., Using MPI: Portable parallel programming with the Message PassingInterface, MIT Press, 1994.

[86] Gueist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V., PVM: Parallel Virtual Machine A user's guide and tutorial for networked parallel computing, MIT Press, 1994 (http://www.netlib.org/pvm3/book/pvmbook.html).

[87] Gupta S.K., Kyparisis J., Single machine scheduling research, OMEGA International Journal of Management Science 15, 1987, 207–227.

[88] Hajek B., Cooling schedules for optimal annealing, Mathematics of Operations Research, 1988, 311 – 329.

[89] High Performance FORTRAN Forum, http://dacnet.rice.edu/Depts/CRPC/HPFF/.

[90] Holland J.H., Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, artificial intelligence, University of Michigan Press, 1975.

[91] Holland J.H., Genetic algorithms, Scientific American 267 (1992), 44–50.

[92] Ignall E., L.E. Schrage, Application of the branch-and-bound technique to some flow-shop scheduling problems, Operations Research, (1965), 13/3, 400–412.

[93] Ingber L., Lester Ingber's Archive, dokument on-line, http://www.ingber.com/

[94] Institute of Electric and Electronic Engineering, Information Technology Portable Operating Systems Interface (POSIX) Part 1 Ammendment 2: Threads Extension, 1995.

[95] Ishibuchi H., S. Misaki, H. Tanaka, Modified Simulated Annealing Algorithms for the Flow Shop Sequencing Problem, European Journal of Operational Research 81, (1995), 388–398.

[96] Kale L.V., Bhandarkar M., Brunner R., Yelon J., Multiparadigm, multilingual interoperability: Experience with Converse, Lecture Notes in Computer Science 1388, Springer Verlag 1998, 111–112.

[97] Kale L.V., Brunner R., Phillips J., Varadarajan K., Application performance of a Linux cluster using Converse, Lecture Notes in Computer Science 1586, Springer Verlag 1999, 483–495.

[98] Kale L.V., Krishnan S., Charm++: Parallel programming with messagedriven objects, in Parallel Programming using C++ (G.V. Wilson and P. Lu, eds.) 175–213, MIT Press, 1996.

[99] Kirkpatrick S., Gelatt C.D., Vecchi M.P., Optimisation by simulated annealing, Science 220 (1983), 671–680.

[100] Kliewer G., Klohs K., Tschoke S., Parallel simulated annealing library (parSA): User manual, Technical report, Computer Science Department, University of Paderborn, 1999.

[101] Knies A., O'Keefe M., MacDonald T., High Performance FORTRAN: A practical analysis, Scientific Programming 3 (1994), 187–199.

[102] Koelbel C., Loveman D., Schreiber R., Steele G. Jr., Zosel M., The High Performance FORTRAN handbook, The MIT Press, 1994.

[103] Kohlmorgen U., Schmeck H., Haase K., Experiences with finegrained parallel genetic algorithms, Annals of Operations Research 90 (1999), 203–220.

[104] Kravitz S.A., Rutenbar R.A., Placement by simulated annealing on a multiprocessor, IEEE Transactions on Computer Aided Design 6 (1987), 534–549.

[105] Kubale M., Introduction to Computational Complexity and Algorithmic Graph Coloring, Wydawnictwo Gdańskiego Towarzystwa Naukowego, Gdańsk 1998.

[106] Kumar V., Grama A., Gupta A., Karypis G., Introduction to parallel computing design and analysis of parallel algorithms, Benjaming/Cummings, 1994.

[107] Kurtz S.A., Mahaney S.R., Royer J.S., Simon S., Biological Computing, Complexity Theory retrospective II (ed. Hemaspaandra L.A., Selman A.L.). Springer Verlag 1997, 179-195.

[108] Lageweg B.J., J.K., Lenstra, A.H.G. Rinnooy Kan, A General Bouding Scheme for the Permutation Flow-Schop Problem, Opns. Res. 26, (1978), 53-67.

[109] Lai T.H., Sahni S., Anomalies in parallel branchandbound algorithms, Communications of the ACM 27 (1984), 594–602.

[110] Lai T.H., Sprague A., A note on anomalies in parallel branchand-bound algorithms with onetoone bounding functions, Information Processing Letters 23 (1986), 119–122.

[111] Lawler E.L., Efficient Implementation of Dynamic Programming Algorithms for Sequencing Problems, Report BW106, Mathematisch Centrum, Amsterdam (1979).

[112] Lawrence S., Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques. Technical Report. Graduate School of Industrial Ad-ministration, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1984

[113] Lee K.G., Lee S.Y., Efficient parallelization of simulated annealing using multiple Markov chains: An application to graph partitioning, Proceedings of the International Conference on Parallel Processing, volume III, 177–180, St. Charles, 1992.

[114] Lee K.G., Lee S.Y., Synchronous and asynchronous parallel simulated annealing with multiple Markov chains, IEEE Transactions on Parallel and Distributed Systems, 7 (1996), 993–1008.

[115] Lee K.G., Lee S.Y., Asynchronous communication of multiple Markov chains in parallel simulated annealing, Proceedings of the International Conference on Parallel Processing, volume III, 169–176, St. Charles, 1992.

[116] Lenstra J.K., Sequencing by Enumeration Methods, Mathematical Centre Tract 69, Mathematisch Centrum, Amsterdam, 1977.

[117] Levine D., A parallel genetic algorithm for the set partitioning problem, Technical report ANL9423, Argonne National Laboratory, 1994.

[118] Lewis B., Multithreaded programming with Java technology, Prentice Hall, 2000.

[119] Li Y., Pardalos P.M., Resende M.G.C., A greedy randomized adaptive search procedure for quadratic assignment problem, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 16 (1994), 237–261.

[120] Li G.J., Wah B.W., Coping with anomalies in parallel branchand-bound algorithms, IEEE Transactions on Computers C35 (1986), 568–573.

[121] Lipski W., Kombinatoryka dla programistów, WT 1982.

[122] Liu J., A new heuristic algorithm for csum flowshop scheduling problems, Personal Communication, 1997.

[123] Lundqvist K., Wall G., Using object oriented methods in Ada95 to implement Linda. A.Strohmeier (ed.) Reliable Software Technologies - Ada-Europe'96, Lecture Notes on Computer Science 1088, Springer Veralg 1996.

[124] Mans B., Roucairol C., Performances of parallel branch and bound algorithms with bestfirst search, Discrete Applied Mathematics 66 (1996), 57–76.

[125] Marco N., Lanteri S., A two level parallelization strategy for genetic algorithms applied to optimum shape design, Parallel Computing 26 (2000), 377–397.

[126] Martins S.L., Resende M.G.C., Ribeiro C.C., Pardalos P., A parallel GRASP for the Steiner tree problem in graphs using a hybrid local search strategy, Journal of Global Optimization 17 (2000), 267–283.

[127] Martins S.L., Ribeiro C.C., Rodriguez N.R., Parallel computing environments, Handbook of Combinatorial Optimization, Oxford, w druku

[128] Martins S.L., Ribeiro C.C., Souza M.C., A parallel GRASP for the Steiner problem in graphs, Lecture Notes in Computer Science 1457 (1998), 285–297.

[129] Mattson T.G. , Scientific computation, in Parallel and distributed computing handbook (A.Y. Zomaya, ed.), 981–1002, McGrawHill, 1996.

[130] Merlin J., Baden S., Fink S., Chapman B., Multiple data parallelism with HPF and KeLP, Future Generation Computer Systems 15 (1999), 393–405.

[131] Merlin J., Hey A., An introduction to High Performance FORTRAN, Scientific Programming 4 (1995), 87–113.

[132] Miki M., Hiroyasu T., Kasai M., Application of the temperature parallel simulated annealing to continous optimization problems, IPSL Transaction 41 (2000), 1607–1616.

[133] Morse H.S., Practical parallel computing, AP Professional, 1994.

[134] MPI Forum, MPI: A messagepassing interface standard, The International Journal of Supercomputing Applications and High Performance Computing 8 (1994), 159–416.

[135] MPI Forum, A messagepassing interface standard, dokument on-line, http://www.mpiforum.org/docs/docs.html, 2001.

[136] MPI Forum, MPI2: Extensions to the MessagePassing Interface, dokument on-line, http://www.mpiforum.org/docs/docs.html, 2001.

[137] Muhlenbein H., Parallel genetic algorithms in combinatorial optimization, in Computer Science and Operations Research: New Developments in their Interfaces(Balci O., Sharda R., Zenios S., eds.), 441–456, Pergamon Press, 1992.

[138] Muth J.F., Thompson G.L., Winters P.R., Industrial Scheduling, Englewood Cliffs, NJ: Prentice Hall, 1963

[139] Nawaz M., Enscore E.E.Jr, Ham I., A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem, OMEGA 11/1 (1983) 91–95.

[140] Nowicki E.: Metoda tabu w problemach szeregowania zadań produkcyjnych, Prace Naukowe ICT PWr, Seria: Monografie, 1999.

[141] Nowicki E., Smutnicki C., A fast tabu search algorithm for the permutation flow-shop problem, European Journal of Operational Research 91, (1996), 160-175.

[142] Oak S., Wong H., Java Threads, O'Reilly, 1997.

[143] Ochi L.S., Drummond L.M., Victor A.O., Vianna D.S., A parallel evolutionary algorithm for solving the vehicle routing problem with heterogeneous fleet, Future Generation Computer Systems Journal, 14 (1998), 285–292.

[144] Ogbu F., Smith D., The Application of the Simulated Annealing Algorithm to the Solu-tion of the n/m/Cmax Flowshop Problem, Computers and Operations Research, 17(3), (1990), 243–253.

[145] OR-Library: http://mscmga.ms.ic.uk/info.html

[146] Osman I., Potts C., Simulated Annealing for Permutation Flow-Shop Scheduling, OMEGA 17(6), (1989), 551–557.

[147] Oussaidène M., Chopard B., Pictel O.V., Tomassini M., Parallel genetic programming and its application to trading model induction, Parallel Computing 23 (1997), 1183–1198.

[148] Pettey C.C., Leuze M.R., Grefenstette J., A parallel genetic algorithm, Proceedings of the Second International Conference on Genetic Algorithms (Grefenstette J., ed.), 155–161, Lawrence Erlbraum Associates, 1987.

[149] Planquelle B., Méhaut J.F., Revol N., Multicluster approach with PM2, Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications, 779–785, Las Vegas, 1999.

[150] Porto S.C., Ribeiro C.C., Parallel tabu search messagepassing synchronous strategies for task scheduling under precedence constraints, Journal of Heuristics 1 (1995), 207– 223.

[151] Porto S.C., Ribeiro C.C., A tabu search approach to task scheduling on heterogeneous processors under precedence constraints, International Journal of High Speed Computing 7 (1995), 45–71.

[152] Porto S.C., Ribeiro C.C., A case study on parallel synchronous implementations of tabu search based on neighborhood decomposition, Investigaci'on Operativa 5 (1996), 233–259.

[153] Porto S.C., Kitajima J.P., Ribeiro C.C., Performance evaluation of a parallel tabu search task scheduling algorithm, Parallel Computing 26 (2000), 73–90.

[154] Potts C.N., L.N. Van Wassenhove, A Branch and Bound Algorithm for the Total Weighted Tardiness Problem, Operations Research, 33 (1985), 177–181.

[155] Quinn M.J., Parallel computing: Theory and practice, McGrawHill, 1994.

[156] Reeves C., Improving the Efficiency of Tabu Search for Machine Sequencing Problems, Journal of Operational Research Society 44(4), (1993), 375-382.

[157] Reeves C., A Genetic Algorithm for Flowshop Sequencing, Computers and Operations Research 22(1), (1995), 5-13.

[158] Reeves C., Yamada T.: Genetic Algorithms, path relinking and the flowshop sequencing problem. Evolutionary Computation 1988, 6, 45-60.

[159] Reeves C. R., Yamada T., Solving the Csum Permutation Flowshop Scheduling Problem by Genetic Local Search, IEEE International Conference on Evolutionary Computation 1998, 230-234

[160] Rinnoy Kan A.H.G, Lageweg B.J., Lenstra J.K., Minimizing total costs in one-machine scheduling, Operations Research, 25 (1975), 908-927

[161] Rinnoy Kan A.H.G, Machine Scheduling Problems: Classification, Complexity and Computations, Nijhoff, The Hague, 1976.

[162] RiveraGallego W., A genetic algorithm for circulant Euclidean distance matrices, Journal of Applied Mathematics and Computation 97 (1998), 197–208.

[163] Rutenbar R.A., Kravitz S.A., Layout by annealing in a parallel environment, Proceedings of the IEEE International Conference on Computer Design, 434–437, Port Chester, 1986.

[164] Salhi A., Parallel implementation of a geneticprogramming based tool for symbolic regression, Information Processing Letters 66 (1998), 299–307.

[165] Schrage L., Baker K.R., Dynamic Programming solution of Sequencing Prob-lems with Precedence Constraints, Operational Research, 26 (1978), 444–449.

[166] Scientific Computing Associates, Linda's user's guide and reference manual, version 4.0.1 SP2/POE, 1995.

[167] Sena G.A., Megherbi D., Isern G., Implementation of a parallel genetic algorithm on a cluster of workstations: Traveling salesman problem, a case study, Future Generation Computer Systems 17 (2001), 477–488.

[168] Steinhöfel K., Albrecht A., Wong C.K., Fast parallel heuristics for the job shop scheduling problem, Computers and Operations Research 29 (2002), 151-169

[169] Storer R.H., Wu S.D., Vaccari R., New search spaces for sequencing problems with application to job shop scheduling. Management Science 1992;38, 1495-1509

[170] Smutnicki C., Algorytmy szeregowania, Akademicka Oficyna Wydawnicza Exit, Warszawa 2002.

[171] Smutnicki C., Optimization and control in just-in time manufacturing systems, Prace naukowe Instytutu Cybernetyki Technicznej Pol. Wrocł., Seria Monografie, 1997.

[172] Supercomputing Technologies Group, CILK 5.3.1 Reference Manual, MIT Laboratory for Computer Science, http://supertech.lcs.mit.edu/cilk

[173] Taft S.T., Duff R. A. (ed.), Ada95 Reference Manual: Language and Standard Libraries, Lecture Notes on Computer Science 1246, Springer Verlag 1997, International Standard ISO/IEC8652:1995(E)

[174] Taillard E., Some efficient heuristic methods for the flow shop sequencing problem, European Journal of Operational Research 47(1), (1990), 65–74.

[175] Taillard E., Robust taboo search for the quadratic assignment problem, Parallel Computing 7 (1991), 443–455.

[176] Taillard E., Parallel taboo search techniques for the job shop scheduling problem, ORSA Journal on Computing 6 (1994), 108–117.

[177] Taillard E., Benchmarks for basic scheduling problems, European Journal of Opera-tional Research 64, (1993), 278-285.

[178] Talbi E.G., Hafidi Z., Geib J.M., A parallel adaptive tabu search approach, Parallel Computing 24 (1998), 2003–2019.

[179] Talbi E.G., Hafidi Z., Geib J.M., Parallel adaptive tabu search for large optimization problems, in Metaheuristics: Advances and Trends in Local Search Paradigms for Optimization (Voss S., Martello S., Osman I.H., Roucairol C., eds.), 255266, Kluwer, 1999.

[180] Talbi E.G., Muntean T., Hillclimbing, simulated annealing and genetic algorithms: A comparative study, Proceedings of the International Conference on Systems Sciences: Task scheduling in parallel and distributed systems (ElRewini H., Lewis T., eds.), 565–573, IEEE Computer Society Press, 1993.

[181] Talbi E.G., Roux O., Fonlupt C., Robillard D., Parallel ant colonies for combinatorial optimization problems, Lecture Notes in Computer Science 1586, Springer Verlag 1999, 239–247.

[182] Talbi E.G., Roux O., Fonlupt C., Robillard D., Parallel ant colonies for the quadratic assignment problem, Future Generation Computer Systems 17 (2001), 441– 449.

[183] Tanese R., Distribued genetic algorithms, Proceedings of the Third International Conference on Genetic Algorithms (Schaffer J.D., ed.), 434–439, Morgan Kaufmann, 1989.

[184] Verhoeven M.G.A., Aarts E.H.L., Parallel Local Search, Journal of Heuristics 1 (1995), 43–65.

[185] Voss S., Tabu search: Applications and prospects, in Network Optimization Problems (Du D.Z., Pardalos P.M., eds.), 333–353, World Scientific, 1993.

[186] Walker D.W., The design of a standard message passing interface for distributed memory concurrent computers, Parallel Computing 20 (1994), 657–673.

[187] Wang C. Chu C. Proth J., Heuristic approaches for n/m/F/SCi scheduling problems, European Journal of Operational Research (1997), 636-644.

[188] Witte E.E., Chamberlain R.D., Franklin M.A., Parallel simulated annealing using speculative computation, IEEE Transactions on Parallel and Distributed Systems 2 (1991), 483–494.

[189] Yamada T., Nakano R., A genetic algorithm applicable to large-scale job shop prob-lems. Manner R., Manderick B. (ed.) Parallel problem solving from nature II. Amsterdam: North-Holland, 1992, 281-290

Spis tabel

Spis rysunków

Indeks