Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 26, Num. 1 (2019) 11-25

RESEARCH ARTICLE

Evolutionary Models applied to Multiprocessor Task Scheduling: Serial and Multipopulation Genetic Algorithm

Modelos Evolutivos aplicados ao Escalonamento de Tarefas em Sistemas Multiprocessados: Algoritmo Genético Serial e Multipopulação

Bruno W. Dantas Morais, Gina M. Barbosa de Oliveira, Tiago Ismailer de Carvalho

Resumo: This work presents the development of a multipopulation genetic algorithm for the task scheduling problem with communication costs, aiming to compare its performance with the serial genetic algorithm. For this purpose, a set of instances was developed and different approaches for genetic operations were compared. Experiments were conducted varying the number of populations and the number of processors available for scheduling. Solution quality and execution time were analyzed, and results show that the AGMP with adjusted parameters generally produces better solutions while requiring less execution time.
Keywords: multipopulation genetic algorithm — multiprocessor task scheduling

Resumo: Neste trabalho foi desenvolvido um Algoritmo Genético multipopulação para o problema de escalonamento de tarefas com custos de comunicação, com o objetivo de comparar seu desempenho com o Algoritmo Genético serial. Para isto, um conjunto de instâncias do problema foi elaborado e abordagens de operações genéticas foram comparadas. Experimentos foram conduzidos com variação do parâmetro de número de populações e do número de processadores utilizados no escalonamento. Foram avaliados a qualidade das soluções produzidas e o tempo de execução, e concluiu-se que o AGMP com seus parâmetros ajustados em geral obtém soluções melhores demandando um menor tempo de execução.
Palavras-Chave: algoritmo genético multipopulação — escalonamento de tarefas em multiprocessadores

Faculdade de Computação, Universidade Federal de Uberlândia, Brasil
*__Corresponding author__: brunowelldm@gmail.com
DOI: https://doi.org/10.22456/2175-2745.82412 • **Received:** 30/04/2018 • **Accepted:** 11/01/2019

1. Introdução

A crescente utilização de sistemas multiprocessados e o desenvolvimento de programas paralelos e distribuídos implicam no aumento da relevância de métodos que buscam a eficiência em sua execução. Entre eles, o problema de escalonamento envolve alocar as subtarefas de um programa paralelo em cada processador de uma arquitetura multiprocessada, de modo a reduzir o tempo de processamento total. Porém, o espaço de busca relacionado ao problema do escalonamento geralmente apresenta uma alta cardinalidade, dificultando a obtenção da solução ótima. Assim, têm sido utilizadas heurísticas e aproximações para se obter soluções aceitáveis para esse problema. [1]

Algoritmo Genético (AG) é uma técnica bio-inspirada e baseada em processos estocásticos que é utilizada para busca e otimização em problemas que envolvem espaços de busca de alta cardinalidade, incluindo o escalonamento [2]. O AG mantém um conjunto de soluções que são evoluídas iterativamente, a fim de explorar melhor o espaço de busca. Isso é feito por meio de operações que geram novas soluções a partir das soluções já exploradas, e as integram no conjunto de maneira semelhante ao processo da seleção natural: boas soluções tem maior probabilidade de ”sobreviver”e ”reproduzir”. No AG serial, estas operações são executadas sequencialmente.

Neste trabalho foi desenvolvido um Algoritmo Genético Multipopulação (AGMP) para o problema de escalonamento estático de tarefas com custos de comunicação. Tal técnica estende o modelo do AG serial mantendo subconjuntos de soluções que são evoluídos em processos paralelos de maneira semi-independente, com “migrações” periódicas. Essa abordagem é útil por explorar o paralelismo de sistemas multiprocessados, demandando um menor tempo de processamento sem que haja degradação das soluções encontradas [3].

O objetivo deste trabalho é comparar os resultados dos dois algoritmos em termos de qualidade das soluções obtidas e do tempo de execução. É esperado que ambos produzam soluções satisfatórias e que o AGMP apresente um ganho no tempo de processamento em relação ao AG serial, de acordo com o número de processos e processadores utilizados.

Este texto é organizado da seguinte forma. A seção 2 apresenta uma revisão de conceitos de escalonamento, AG, e AGMP. A seção 3 apresenta abordagens exploradas em trabalhos correlatos. A seção 3 contém descrições do conjunto de grafos de *benchmark* e do desenvolvimento e implementação do AG e do AGMP. A seção 5 descreve os parâmetros utilizados, as configurações avaliadas, o modelo de experimentos, e apresenta as comparações feitas entre o AG e o AGMP. A seção 6 descreve as conclusões e trabalhos futuros.

2. Revisão Bibliográfica

Esta seção descreve os principais conceitos utilizados e as abordagens de trabalhos correlatos.

2.1 Escalonamento de tarefas

Escalonamento de tarefas é um problema clássico de otimização combinatória considerado computacionalmente intratável [1]. Ele consiste no mapeamento de um conjunto de tarefas a um conjunto de processadores de modo a satisfazer um critério de avaliação. Assim, algoritmos baseados em heurísticas, aproximações e meta-heurísticas foram desenvolvidos para abordar o problema [4].

O problema tratado neste trabalho apresenta as seguintes propriedades [1].

- Escalonamento estático e determinístico: os tempos de execução, comunicação e as relações de precedência entre tarefas são conhecidos e não se alteram.
- Processadores não-preemptivos: um processador executa uma tarefa até que ela se encerre, sem que haja trocas ou interrupções.
- Os processadores são idênticos para fins de eficiência relativa entre processadores. Ou seja, todo processador leva x unidades de tempo para executar tarefas de custo x.
- Os processadores são totalmente conectados, i.e., se comunicam diretamente com cada um dos demais processadores.
- Um escalonamento não possui duplicação da execução de uma tarefa, i.e., cada tarefa é executada exatamente uma vez por um único processador.

Uma instância do problema de escalonamento é representada por uma tupla (G,P), em que P é o conjunto de processadores que executarão as tarefas, $G = \{V,E\}$ é um grafo direcionado acíclico cujos vértices de V representam um conjunto de tarefas, e arestas de E representam relações de precedência entre as tarefas. Para cada tarefa $T_i \in V$ existe um peso w_i que indica o tempo de processamento necessário para executar a tarefa correspondente. Para cada aresta $e_{ij} \in E$ existe um custo c_{ij} que indica o tempo necessário de comunicação entre o término da execução de T_i e o começo da execução de T_j, no caso dessas tarefas serem executadas em processadores diferentes. Caso sejam executadas no mesmo processador, não há custo de comunicação. Na Figura 1A é apresentado o grafo de tarefas laplace9, que representa o algoritmo para resolução de equações de Laplace, com os custos de comunicação omitidos, uma vez que são iguais a 40 para todas as arestas.

Uma tarefa T_i só pode ser escalonada por um processador P_j após o término da execução de todas as tarefas que precedem T_i, levando em conta os custos de comunicação para predecessoras executadas em processadores diferentes de P_j. Por exemplo, considerando o grafo da Figura 1A, o início do processamento da tarefa $T4$ deve ser posterior ao término de ambas as tarefas $T1$ e $T2$.

Dado um conjunto de p processadores, um escalonamento pode ser representado por p sequências de tarefas, cada uma associada a um processador. Cada sequência é parcialmente ordenada, obedecendo as precedências do grafo. A sequência de tarefas alocada a um processador indica as tarefas que ele executará e a ordem.

Um processador fica ocioso caso a tarefa a ser executada dependa de um resultado de outra tarefa que ainda não foi terminada ou comunicada por outro processador.

Dada a representação de um escalonamento, é possível calcular uma linha do tempo de execução, com pontos de início e término de cada tarefa. Essa linha do tempo pode ser ilustrada por um diagrama de Gantt. A Figura 1B apresenta o diagrama de Gantt para uma possível solução de escalonamento do grafo laplace9 em uma arquitetura de 4 processadores, e que demanda um tempo total de execução igual a 540.

O tempo do escalonamento ou *makespan* corresponde ao ponto da linha do tempo em que a execução de todas as tarefas foi terminada. Isto corresponde à duração total do escalonamento. Um escalonamento ótimo é aquele que apresenta um *makespan* mínimo.

2.2 Algoritmo Genético

Algoritmo Genético (AG) é uma meta-heurística proposta por John Holland que se inspira na evolução das espécies e tem sido usada para tratar diversos tipos de problemas relacionados a busca e otimização, incluindo o problema de escalonamento de tarefas [2].

Um AG opera de maneira estocástica para evoluir um conjunto de soluções com objetivo de encontrar uma solução globalmente satisfatória no espaço de busca de um problema. Para isso, deve ser definida uma forma de representação de uma solução e uma função para avaliação da mesma. Depois é escolhida um modo de inicializar uma população de soluções como ponto inicial da busca, os operadores que modificarão tal população iterativamente, e parâmetros numéricos que controlam seu comportamento. Tais componentes são detalhados a seguir.

Representação genética – Dado um problema a ser resolvido pelo Algoritmo Genético, deve ser definida uma estrutura de dados que codifique uma solução pertencente ao espaço de busca do problema. Uma instância dessa estrutura de dados

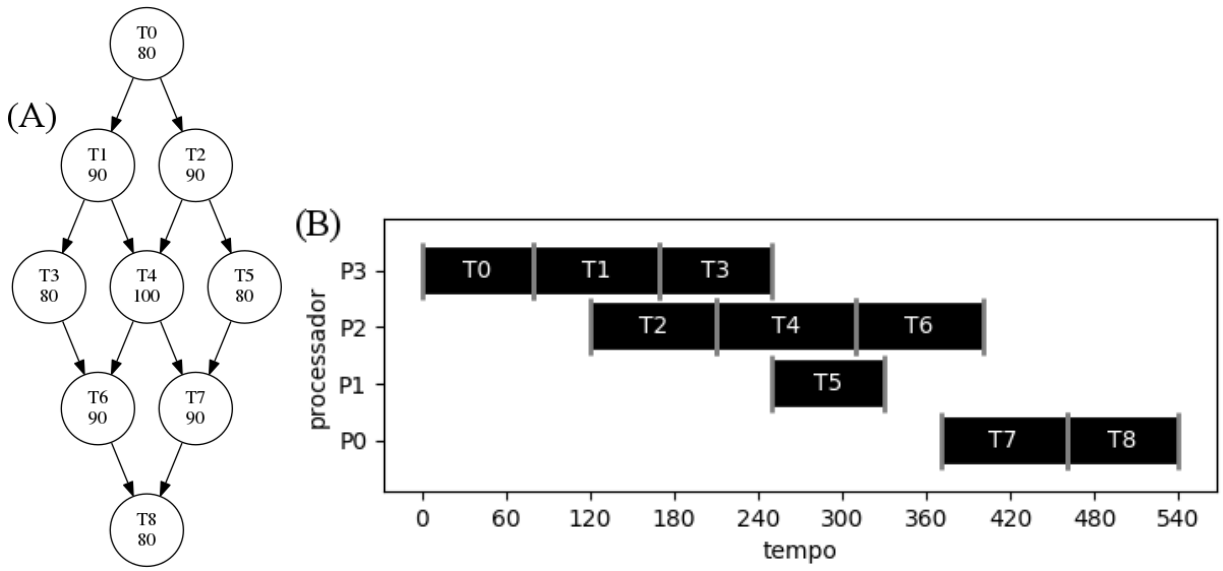

Figura 1. (A): grafo laplace9 com custos de comunicação omitidos, que são iguais a 40 para todas as arestas. **(B)**: diagrama de Gantt que representa um possível escalonamento para 4 processadores com makespan de 540 unidades de tempo.

constitui um indivíduo (ou "cromossomo") que fará parte da população do AG. A representação genética clássica é um vetor binário. [2].

Fitness – Uma função de *fitness* (aptidão) é escolhida para a avaliação da qualidade de um indivíduo. Em geral, todo indivíduo explorado pelo AG é avaliado.

População inicial – Gerar uma população inicial de indivíduos constitui a etapa de inicialização do AG. Dado um parâmetro N_{pop} que indica o tamanho da população, são gerados N_{pop} indivíduos, de forma estocástica, que representam soluções válidas e compõem a população a ser evoluída pelo algoritmo. Geralmente, a população de um AG é uma lista de indivíduos ordenada pelo *fitness*.

Seleção – A cada iteração (ou geração) do AG, pares de indivíduos são selecionados da população para serem recombinados, gerando novas soluções (filhos). Em geral, indivíduos são selecionados de acordo com seu *fitness*, tal que indivíduos melhores terão maior probabilidade de se reproduzir. O número de pares a serem selecionados é obtido a partir da taxa de *crossover* (T_{cross}), que é dada como parâmetro do AG. O método clássico para seleção é a roleta, que dá a cada indivíduo uma probabilidade de ser selecionado que é diretamente proporcional ao seu *fitness* relativo aos demais indivíduos da população. [2]

Crossover – O operador de *crossover* é o método para gerar novos indivíduos a partir de um par de indivíduos pais. Seu objetivo é que os filhos possuam características de ambos os pais combinando suas representações. Por exemplo, *crossover* de um ou mais pontos consiste em dividir dois indivíduos (representados por vetores) em várias secções delimitadas por pontos que são alternadamente permutadas para gerar dois filhos. [2]

Mutação – Após o processo de *crossover*, alguns dos indivíduos filhos podem sofrer uma modificação pelo operador de mutação. A probabilidade de haver mutação em um filho definida pela taxa de mutação (T_{mut}), a qual é dada como parâmetro do AG. Quando necessário, correções e adaptações são feitas para que o novo indivíduo represente uma solução válida, pertencente ao espaço de busca. Para a representação genética clássica de vetor binário, o método de mutação consiste em negar um de seus bits. [2]

Reinserção – Dada uma lista de indivíduos pais e filhos, a operação de reinserção produz uma lista de indivíduos que constituirá a população na iteração seguinte combinando a população atual e a lista de filhos. [2]

2.3 Algoritmo Genético Multipopulação

Um Algoritmo Genético paralelo é um modelo de AG para execução em sistemas multiprocessados visando obter um tempo de processamento menor em relação a execução serial. Segundo Cantú-Paz[5], os vários tipos de AGs paralelos podem ser categorizados em:

- Paralelização global ou mestre-escravo: utiliza uma população única controlada por um programa mestre enquanto avaliações e/ou operadores genéticos são feitas por programas escravos.
- Paralelização de fina granularidade: voltada para sistemas massivamente paralelos. Utiliza uma população única cujos indivíduos competem e reproduzem apenas com uma vizinhança limitada.
- Paralelização multipopulação: utiliza várias populações semi-isoladas que se comunicam com uma certa frequência.

Alba e Troya[3] consideram AGs paralelos como uma classe distinta de meta-heurísticas por apresentarem a característica da interação local em subgrupos da população (com exceção da abordagem de paralelismo global), o que causa um impacto na qualidade das soluções. São citados casos em que AGs paralelos encontram soluções melhores do que as de AGs seriais. Os autores argumentam que a abordagem paralela é benéfica mesmo em sistemas que não possuam múltiplos processadores.

Algoritmo Genético multipopulação é a forma mais popular de AG paralelo [5]. Um AGMP pode ser encarado como uma extensão da implementação de AG, em que vários AGs independentes são executados, trocando alguns indivíduos entre si ocasionalmente. Assim, é pode-se converter um AG serial em AGMP adicionando o procedimento de migração, e é possível executá-lo em vários tipos de sistemas diferentes, seja uma máquina com um número qualquer de processadores ou uma rede de computadores [5].

AGMP é também conhecido como: AG distribuído, por sua abordagem relacionada a paralelismo MIMD (múltiplas instruções, múltiplos dados); AG paralelo de grossa granularidade, por sua alta taxa de computação por comunicação; e AG baseado em ilhas, devido a sua semelhança com o modelo de ilhas em genética populacional. O AGMP se caracteriza por usar poucas populações de tamanho relativamente grande, em contraste com as pequenas e numerosas vizinhanças mantidas na abordagem de fina granularidade [5].

O funcionamento do AGMP também requer parâmetros adicionais como número de populações, frequência de migração (F_{mig}) e taxa de migração (T_{mig}). A seguir, são descritos os conceitos adicionais que caracterizem o AGMP.

Topologia de comunicação – Estabelece as ligações entre populações, definindo as populações de destino a serem consideradas por cada população durante migrações. Topologias são geralmente especificadas *a priori* e imutáveis ao longo da execução. [5]

Frequência de migração – Define a periodicidade de trocas de indivíduos entre populações. Migrações podem ocorrer de modo síncrono num intervalo predeterminado de gerações, ou assíncrono sob satisfação de uma condição [5]. Para o caso especial em que a frequência de migração é igual a zero, o AGMP é equivalente a uma sequência de execuções independentes de AG serial com população reduzida. Tal abordagem é desencorajada por CantÚ-Paz et al.[6] por, geralmente, produzir soluções de baixa qualidade.

Seleção de migrantes – O critério de escolha de quais indivíduos serão removidos de sua população de origem para serem inseridos em outra. Em geral, imigrantes substituem indivíduos emigrantes ou os de pior *fitness*. [5]

3. Trabalhos correlatos

Trabalhos correlatos podem ser divididos entre aplicações de AG para o problema do escalonamento de tarefas e abordagens para desenvolvimento de AGMP de modo geral.

3.1 Algoritmo Genético aplicado a escalonamento de tarefas

A maior parte dos trabalhos consultados formula suas representações considerando a ordenação topológica do grafo direcionado acíclico dado pelo problema de escalonamento, o que permite a disposição dos vértices em uma lista ordenada pela precedência. Assim, para toda aresta e_{ij}, a posição da tarefa T_i na lista é anterior à posição de T_j.

A representação genética de Hou, Ansari e Ren[1], Correa, Ferreira e Rebreyend[7], e Kaur et al.[8] mantém, para cada processador, uma lista de tarefas ordenada pela precedência. Wang et al.[9], Omara e Arafa[10], Chitra, Rajaram e Venkatesh[11], e Morady e Dal[12] empregam dois vetores de inteiros: um representa uma ordenação topológica de tarefas e o outro representa o processador correspondente a cada tarefa. Kwok e Ahmad[13] utilizam uma única lista de tarefas ordenada por precedência, cujos processadores são atribuídos durante a computação do *fitness*. Hwang, Gen e Katayama[14] codificam a sequência de tarefas usando uma lista de prioridades numéricas, na qual os processadores associados correspondem ao valor da prioridade módulo o número de processadores. Xu et al.[15] fazem uso de uma fila de prioridade que indica a ordem de execução da tarefas.

Como função de *fitness*, Hou, Ansari e Ren[1], Wang et al.[9], Correa, Ferreira e Rebreyend[7], Hwang, Gen e Katayama[14], Omara e Arafa[10], Morady e Dal[12] avaliam o *makespan*. Kwok e Ahmad[13] usam o *makespan* normalizado entre 0 e 1. Kaur et al.[8] consideram o *makespan* e o "*flowtime*", que foi definido como soma dos tempos de término de cada tarefa. Chitra, Rajaram e Venkatesh[11] fazem uma soma ponderada entre *makespan* e confiabilidade, obtida ao contabilizar possíveis cenários de falha de processadores. Xu et al.[15] calculam o *fitness* subtraindo o *makespan* de um indivíduo do maior *makespan* presente na população.

As populações iniciais de Hou, Ansari e Ren[1], Correa, Ferreira e Rebreyend[7], Hwang, Gen e Katayama[14], Chitra, Rajaram e Venkatesh[11], Morady e Dal[12] são formadas por indivíduos gerados aleatoriamente. Wang et al.[9], Kwok e Ahmad[13], Kaur et al.[8], Omara e Arafa[10], Xu et al.[15] geram uma parte da população aleatoriamente e obtém a outra por heurísticas de ranqueamento de prioridade das tarefas.

Na literatura existem vários métodos para seleção. Hou, Ansari e Ren[1], Wang et al.[9], Hwang, Gen e Katayama[14], Kaur et al.[8], Chitra, Rajaram e Venkatesh[11], Xu et al.[15] utilizam o método da roleta, que associa a cada indivíduo uma probabilidade de ser selecionado diretamente relacionada a seu *fitness*. Kwok e Ahmad[13] avaliam o *fitness* de um indivíduo em relação ao *fitness* médio da população para definir o número de vezes que cada indivíduo se reproduzirá. Omara e Arafa[10] utilizam o método do torneio binário em que o melhor entre dois indivíduos selecionados aleatoriamente é escolhido. Morady e Dal[12] fazem parte da seleção com torneio e parte com a escolha direta de uma porção dos melhores indivíduos da população.

Para *crossover*, Wang et al.[9], Kwok e Ahmad[13], Chi-

tra, Rajaram e Venkatesh[11], Kaur et al.[8], Omara e Arafa[10], Xu et al.[15] se baseiam em *crossover* de um ponto, com estratégias para reordenar ou recombinar metade do cromossomo conforme necessário. Hou, Ansari e Ren[1] escolhem pontos de *crossover* durante iterações sobre as listas de tarefas de cada processador. Correa, Ferreira e Rebreyend[7] aplicam *crossover* uniforme: mantém-se as tarefas que coincidem nos pais e sorteia-se uma das alternativas caso contrário. Hwang, Gen e Katayama[14] desenvolvem um método baseado em *crossover* de dois pontos. Morady e Dal[12] combinam *crossover* de dois pontos e PMX, um algoritmo utilizado para vetores que envolvem permutação, que mantém uma subsequência do vetor inalterada e elimina elementos repetidos.

Swap é o operador de mutação mais usado na literatura [1, 9, 4, 14, 11, 15], o qual consiste em trocar duas tarefas de lugar na sequência de execução, podendo implicar uma mudança de processador. O operador implementado por Omara e Arafa[10] muda o processador em que uma tarefa é executada. Correa, Ferreira e Rebreyend[7] fazem a remoção de uma tarefa e sua reinserção num ponto diferente. Kaur et al.[8] e Morady e Dal[12] usam duas operações: *swap* sem mudança de processador ou apenas mudança de processador.

A reinserção nos trabalhos de Hou, Ansari e Ren[1], Wang et al.[9], Kwok e Ahmad[13], Correa, Ferreira e Rebreyend[7], Hwang, Gen e Katayama[14], Kaur et al.[8], Chitra, Rajaram e Venkatesh[11], Xu et al.[15], e Morady e Dal[12] é feita por elitismo: uma porção dos melhores indivíduos permanece na população e o restante é preenchido pelos filhos gerados.

3.2 Abordagens em AGMP

Os trabalhos [13, 12] apresentam aplicações de AGMP para escalonamento de tarefas. Os demais trabalhos consultados usam o AGMP para outras aplicações: escalonamento job-shop [16], *data mining* [17], carregamento de contêiners [18], otimização de funções [19, 20], e otimização combinatória [21].

A respeito de topologia de comunicação, as populações de Kwok e Ahmad[13], Qi, Burns e Harrison[16], e Srinivasa, Venugopal e Patnaik[17] são totalmente conectadas. Gehring e Bortfeldt[18] e Morady e Dal[12] usam topologia em anel, na qual cada população possui uma única população de destino, descrevendo um anel conectado. MÜhlenbein, Schomisch e Born[19] dispõem uma estrutura de "escada circular", com múltiplas conexões em cada população. Han et al.[21] aplicam, uma estrutura de agrupamento voltada para a redução da carga de comunicação entre processadores. Yao, Kharma e Grogono[20] utilizam uma topologia dinâmica em que populações podem se unir ou dividir durante a execução.

A frequência de migração é um parâmetro fixo nos trabalhos [19, 16, 21, 18, 17, 12], enquanto Kwok e Ahmad[13] empregam uma frequência exponencialmente crescente ao longo da execução.

Em [19, 13, 16, 21, 18, 17, 12], os melhores indivíduos são selecionados para migração. Migrações são determinadas por meio de uma análise de agrupamento de indivíduos em [20].

4. Desenvolvimento

O trabalho desenvolvido consistiu em quatro etapas: selecionar um conjunto de grafos para *benchmark*, a ser utilizado nas avaliações comparativas dos algoritmos implementados; implementar o AG e suas diferentes configurações de *crossover*; implementar o AGMP; comparar as melhores configurações do AG e do AGMP.

4.1 Grafos para *benchmark*

Para fins de comparação, foi elaborado um conjunto de grafos referentes ao problema de escalonamento com custos de comunicação em algoritmos reais.

Compondo esse conjunto, foram utilizados grafos obtidos diretamente de outros trabalhos e também famílias de grafos baseadas em instâncias usadas na literatura.

4.1.1 Grafos relacionados a problemas paralelos reais

Com o objetivo de incrementar o conjunto com grafos relevantes e com espaços de busca de alta cardinalidade, foram implementados três scripts em linguagem Python que geram grafos pertencentes a "famílias" correspondentes a algoritmos paralelos mencionados na literatura: eliminação gaussiana, algoritmo de Laplace e transformada rápida de Fourier [22]. Assim, um grafo pertencente a uma dessas famílias corresponde à execução do algoritmo para um dado tamanho de entrada. A metodologia adotada na geração desses grafos foi inspirada no trabalho descrito em [22].

Os scripts implementados geram representações de grafos em arquivos de texto e também suas visualizações com a ferramenta Graphviz. Quatro grafos de cada família foram selecionados para compor o conjunto de *benchmark*, num total de 12 grafos baseados em programas paralelos reais.

A família "gauss" é definida pelo algoritmo da Decomposição LU, uma forma matricial do método da eliminação gaussiana. Esse algoritmo que realiza fatoração de matrizes quadradas em matrizes triangulares, o que se reflete na estrutura desses grafos. Seu número de tarefas depende diretamente do tamanho da matriz de entrada, e seus pesos e custos de comunicação obedecem um padrão regular [23]. A Figura 2 mostra duas instâncias desta família, nas quais as arestas contínuas representam custo de comunicação igual a 12 e as arestas tracejadas, custo igual a 8 [24].

A família "laplace" é dada pelo algoritmo para resolução de equações de Laplace, composta por uma estrutura de tarefas que reflete diretamente o seu tamanho de entrada, que é uma matriz quadrada. Nesta família, ilustrada na Figura 3, as dependências entre tarefas são dadas pela maneira em que a matriz é percorrida no algoritmo [25]. Nesta família, os pesos das tarefas obedecem um padrão regular e todos os custos de comunicação são iguais a 40. Os pesos de cada tarefa correspondem à posição do elemento correspondente na matriz, sendo 80 para as tarefas nas 4 extremidades do grafo (acima, abaixo, à direita e à esquerda), 90 para as tarefas nas laterais, e 100 para as tarefas mais internas.

A família "fft" é baseada no algoritmo de Cooley–Tukey

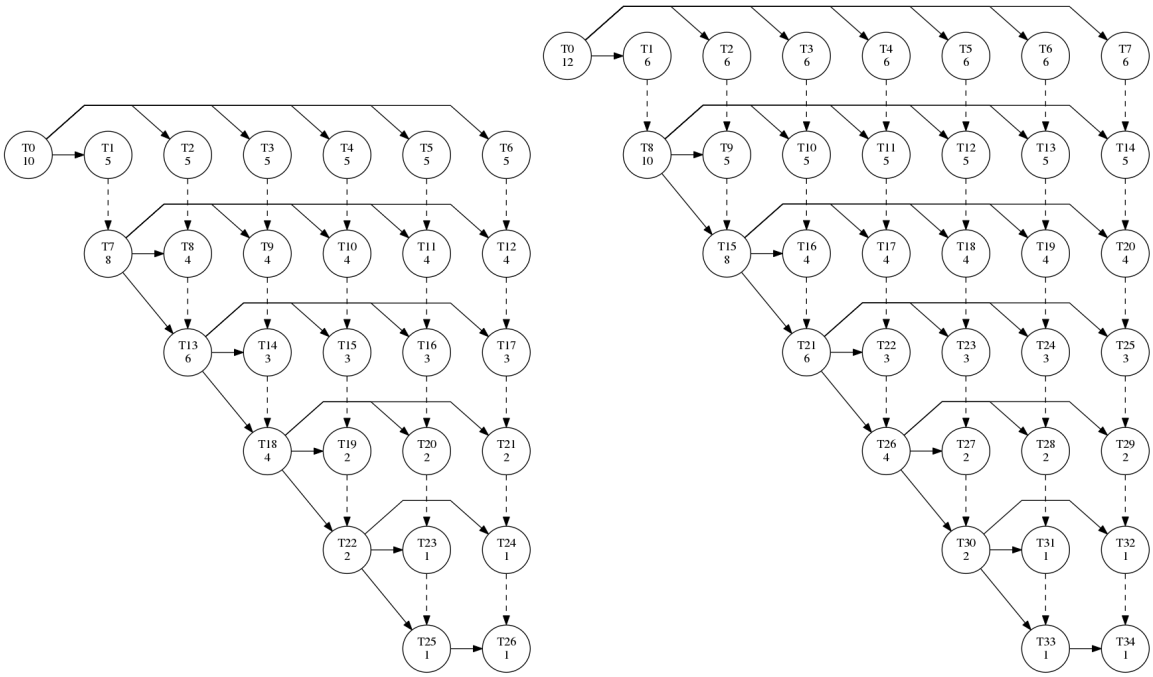

Figura 2. Grafos gauss27 e gauss35. Arestas contínuas representam custo de comunicação igual a 12 e arestas tracejadas, custo igual a 8. Fonte: elaborada pelos autores.

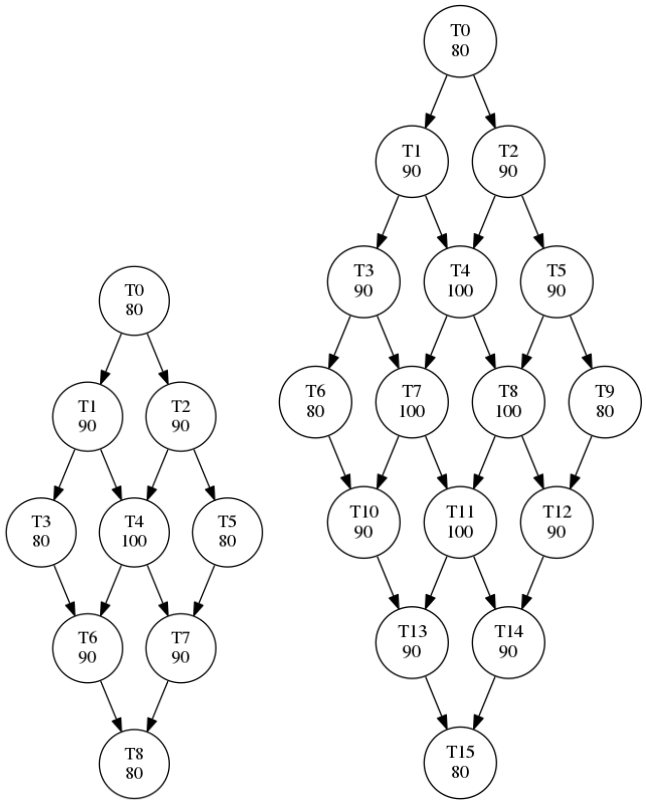

Figura 3. Grafos laplace9 e laplace16. Todos os custos de comunicação nesta família são iguais a 40. Fonte: elaborada pelos autores.

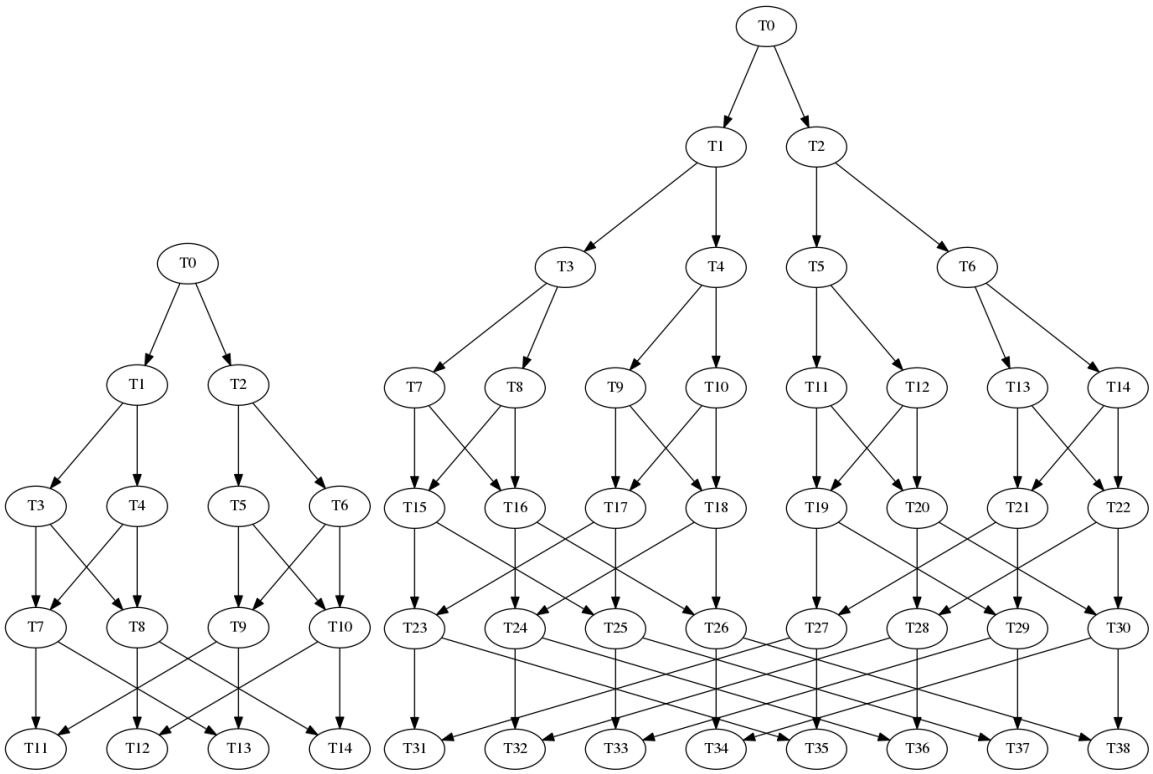

Figura 4. Grafos fft15 e fft39. Nesta família, os vértices possuem pesos iguais a 60 e os custos de comunicação são iguais a 80. Fonte: elaborada pelos autores.

para a transformada rápida de Fourier. Esta família de grafos assume como entrada uma lista de tamanho igual a uma potência de 2. Seus grafos assumem uma estrutura com uma etapa recursiva e uma etapa de operação "borboleta" característica do algoritmo [26]. Duas instâncias são mostradas na Figura 4. Os pesos e custos de comunicação nesta família são iguais a 60 e 80, respectivamente.

4.1.2 Outros grafos

Em [12] foram disponibilizados onze grafos com número de vértices entre 4 e 18, obtidos de diversos trabalhos da literatura. Aqui, eles são referenciados com prefixo "TG". Os autores não mencionam correspondência entre esses grafos e algoritmos paralelos reais.

Em [27] foi utilizada uma ferramenta para criação de três grafos aleatórios com 30, 40 e 50 vértices, identificados pelo prefixo "random".

No total, o conjunto do *benchmark* é composto por 26 grafos, sendo 12 gerados a partir de informações de programas paralelos reais e 14 extraídos da literatura sem conexão com problemas reais.

4.2 Algoritmo Genético Serial

O primeiro algoritmo evolutivo construído foi um AG serial para o problema do escalonamento de tarefas. Nessa etapa, algumas configurações relevantes do AG foram definidas e comparadas a fim de obter uma boa opção de operadores e parâmetros. O propósito deste trabalho foi explorar abordagens de AG "puro", em contraste com a tendência recente de desenvolver AGs híbridos incorporando diversas técnicas e heurísticas específicas do problema [8, 9, 10, 15, 13].

A representação do indivíduo utilizada é a mesma de [9], [10], [11], e [12]: dois vetores de inteiros S, P tal que S é uma sequência de tarefas que obedece às restrições de ordenação topológica do grafo dado e P é um vetor que mapeia os vínculos tarefa/processador de forma que $P[tarefa] = processador$.

A Figura 5B ilustra um indivíduo válido para o grafo laplace9 (Figura 5A), considerando-se alocação para 4 processadores. Para representar seu escalonamento correspondente, ilustrado na Figura 5C, o vetor $S = [0,2,5,1,3,4,6,7,8]$ é particionado em uma sequência para cada processador, obedecendo as associações dadas pelo vetor $P = [3,3,2,3,2,1,2,0,0]$. Por exemplo, as tarefas executadas pelo processador P3 são T0, T1 e T3, pois $P[0] = P[1] = P[3] = 3$. O processamento de cada tarefa é iniciado após a conclusão de todas as suas predecessoras, somado ao custo de comunicação em caso de serem executadas em processadores diferentes. Por exemplo, a tarefa T5 é iniciada após a conclusão e comunicação de T2. Com isso, T5 é executada após o início de T4, embora T5 seja anterior a T4 no vetor S.

A função de *fitness* utilizada é dada pelo *makespan* associado ao escalonamento. Por exemplo, o indivíduo representado na Figura 5B tem *fitness* igual a 540, que corresponde ao

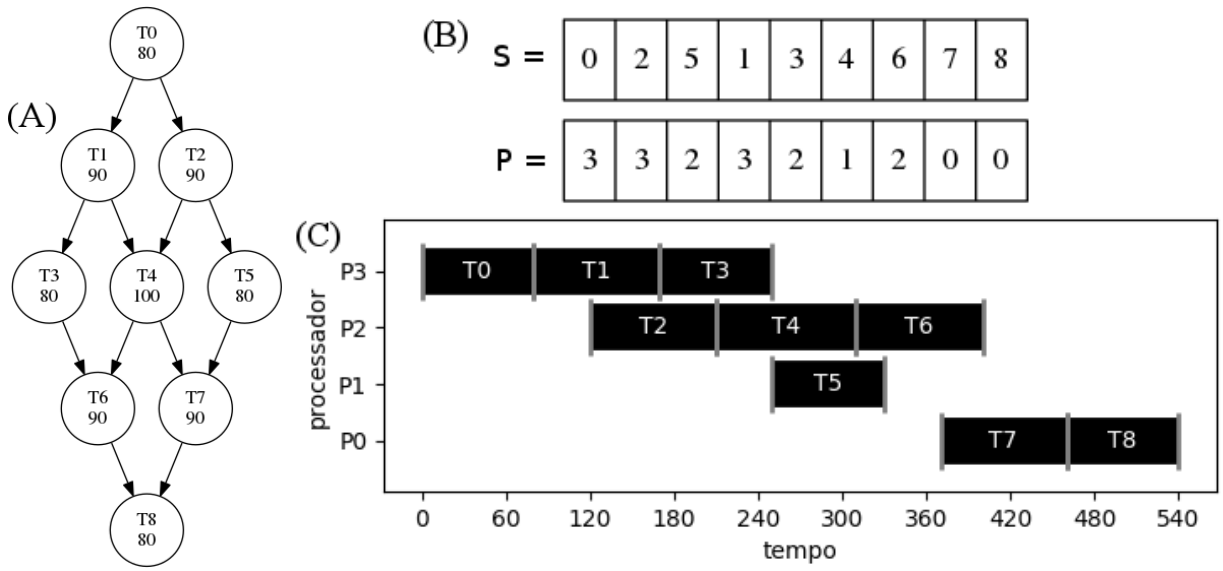

Figura 5. **(A)**: grafo laplace9 com custos de comunicação omitidos, que são iguais a 40 para todas as arestas. **(B)**: exemplo de indivíduo para o grafo laplace9 e 4 processadores. **(C)**: diagrama de Gantt que representa o escalonamento correspondente ao indivíduo de (B), com *makespan* de 540 unidades de tempo.

tempo de finalização do processador P0, o mais tardio na Figura 5C. A população inicial é gerada aleatoriamente com indivíduos válidos: vetor *S* sem repetição das tarefas e respeitando as restrições de ordenação topológica. A seleção dos pais para o *crossover* é feita com torneio binário. A reinserção da população no final de cada geração se dá por elitismo. A mutação é uma operação pontual em um dos dois vetores do indivíduo, mudando o processador de uma tarefa ou trocando duas tarefas de lugar na sequência caso não haja dependência entre elas. Essas duas operações de mutação são referidas aqui como mutação "proc" e mutação "seq", respectivamente.

Como a representação adotada é formada por duas partes distintas (*S* e *P*), existe uma variedade de opções diferentes para a aplicação do operador de *crossover*. Três abordagens foram observadas na bibliografia, aqui nomeadas como *carry*, *choose* e *combine*:

- *Carry*: o *crossover* é aplicado no vetor de tarefas "carregando" o vínculo de cada tarefa com o processador definido no pai correspondente. Assim, tarefas do indivíduo filho vindas do pai1 possuem o mesmo processador do pai1. Do mesmo modo, tarefas vindas do pai2 são executadas nos processadores definidos no pai2. Esta abordagem é análoga a métodos usados em trabalhos com representação de listas de tarefas por processador [1], [7], [8].

- *Choose*: é definida uma operação de *crossover* para o vetor de tarefas e uma para o vetor de processadores. Durante a reprodução de indivíduos, um dos dois tipos de *crossover* é sorteado e aplicado. O outro vetor é copiado do primeiro pai para o primeiro filho e vice-versa [10].

- *Combine*: é definido um *crossover* para o vetor de tarefas e um para o vetor de processadores. Durante a reprodução de indivíduos, ambos *crossovers* são aplicados [12].

Para avaliar estes métodos, foram implementados os seguintes algoritmos clássicos de *crossover*, que foram adaptados para evitar duplicação de tarefas e obedecer as restrições do vetor de tarefas (*S*):

- 1-ponto: sorteia-se um ponto de corte na sequência. Copia-se a primeira parte do pai1 e a segunda parte é preenchida, sem repetição, com tarefas seguindo a ordem em que se elas estão dispostas no pai2 [10].

- 2-pontos: análogo ao anterior; a sequência é dividida em 3 partes. A primeira é uma cópia do pai1, a segunda é preenchida com tarefas na ordem do pai2, e a terceira é preenchida com tarefas restantes do pai1, também obedecendo sua ordem.

- Uniforme: a sequência filha é composta percorrendo os pais e escolhendo aleatoriamente de qual pai será herdada a tarefa seguinte, ignorando repetições.

- Cíclico: uma posição p é sorteada e os p-ésimos elementos das sequências são trocados. Enquanto houver uma duplicata na primeira sequência, o elemento repetido é trocado pelo elemento na mesma posição na outra sequência. É necessária validação posterior para garantir as restrições de precedência.

- PMX: mantém-se uma subsequência do pai1, troca-se o restante com elementos do pai2 e faz-se trocas até eliminar duplicatas. Também é necessária validação do filho.

- OX: mantém-se uma subsequência do pai1 e preenche-se o restante seguindo a ordem do pai2. A versão implementada gera um filho válido por garantir que todas as tarefas predecessoras da subsequência estejam à esquerda dela.

Alguns desses métodos de *crossover* são exemplificados para o grafo laplace9 na Figura 6, em que filhos $F1$, $F2$, $F3$ e $F4$ foram produzidos respectivamente por *crossover* 1-ponto, 2-pontos, uniforme, OX, a partir dos pais $P1$ e $P2$. Além deles, foram implementados *crossovers* 1-ponto, 2-pontos e uniforme para o vetor de processadores, que são análogos aos descritos e sem restrição sobre repetição e ordenação. A implementação foi feita em linguagem C e compilada e otimizada com GCC. Os resultados dos experimentos com o algoritmo evolutivo serial serão apresentados na Seção 5.

4.3 Algoritmo Genético Multipopulação

O AGMP desenvolvido nesse trabalho emprega topologia em anel unidirecional para comunicação entre as populações, que é um modelo de simples implementação e que produziu resultados de boa qualidade em trabalhos correlatos [5]. No anel unidirecional, toda população tem uma vizinha para onde envia alguns indivíduos, e uma segunda vizinha da qual recebe outros indivíduos em um processo conhecido por migração. Os melhores indivíduos da população são os selecionados para a migração. Os parâmetros de migração foram definidos experimentalmente: $T_{mig} = 10\%$ e $F_{mig} = 5$. Demais parâmetros são dados pela *config6* (Tabela 1), escolhida para o AG serial.

Note que a quantidade de avaliações de *fitness* realizadas pelo AGMP é igual à do AG, pois o número total de indivíduos é distribuído igualmente entre cada população. Por exemplo, para 400 indivíduos e 4 populações, o tamanho de cada uma é igual a 100.

A taxa de migração é relativa ao tamanho das populações. No exemplo de 4 populações com 100 indivíduos cada, os parâmetros de migração ditam que os 10 melhores indivíduos (10%) de cada população migrarão.

A implementação foi feita estendendo o AG com procedimentos de inicialização das populações e migração, que foram feitos em memória distribuída com a biblioteca OpenMPI, que implementa o padrão *Message Passing Interface* para execução paralela de programas com múltiplos processos. Desse modo, cada população é executada em um processo diferente que se comunica com populações vizinhas de acordo com a frequência de migração. Por fim, os melhores indivíduos de cada população são enviados para uma mesma população, e o melhor deles é retornado pelo programa. Os demais operadores (seleção, *crossover*, mutação e reinserção) foram implementados da mesma forma descrita para o AG Serial na seção anterior. Uma execução do AG serial é equivalente a executar o AGMP com número de populações igual a 1.

5. Experimentos e Resultados

Nesta seção são definidos os parâmetros utilizados nos algoritmos evolutivos e o método adotado nos experimentos. Em seguida, o AG e o AGMP são comparados em termos de qualidade das soluções produzidas e tempo de execução.

5.1 Definições iniciais de parâmetros e configurações

Para escolha do método de aplicação de *crossover*, foram elaboradas execuções do AG serial com configurações baseadas na literatura, apresentadas na Tabela 1, que contém duplas de operações de *crossover* para vetor de tarefas e de processadores em casos do tipo *choose* ou *combine* e apenas um tipo de *crossover* para o vetor de tarefas no caso *carry*.

Config0 é uma configuração sugerida nas fases iniciais deste trabalho a partir de experiências anteriores do grupo com o problema. *Config1* é baseada na descrição de Omara e Arafa[10]. *Config2* é igual à anterior, com a adição de mutação no vetor de tarefas (mutação "seq"). *Config3* é baseada em [7], com a representação genética de dois vetores aqui descrita em vez de listas de tarefas para cada processador. *Config4* é igual à anterior com seleção via torneio em vez de roleta. *Config12* segue o trabalho de Morady e Dal[12]. As restantes são algumas variações das configurações anteriores.

Tabela 1. Configurações de AG

config	seleção	método	*crossover*	mutação
0	torneio	*carry*	cíclico	seq/proc
1	torneio	*choose*	1-point, 1-point	proc
2	torneio	*choose*	1-point, 1-point	seq/proc
3	roleta	*carry*	uniforme	seq/proc
4	torneio	*carry*	uniforme	seq/proc
5	torneio	*choose*	2-point, 2-point	seq/proc
6	torneio	*combine*	1-point, 1-point	seq/proc
7	torneio	*combine*	uniforme, uniforme	seq/proc
8	torneio	*combine*	2-point, 2-point	seq/proc
9	torneio	*choose*	uniforme, uniforme	seq/proc
10	torneio	*carry*	pmx	seq/proc
11	torneio	*carry*	ox	seq/proc
12	torneio	*combine*	pmx, 2-point	seq/proc
13	torneio	*combine*	cíclico, 2-point	seq/proc
14	torneio	*combine*	ox, 2-point	seq/proc

O número de avaliações de *fitness* feitas em uma execução do AG representa o número aproximado de pontos explorados no espaço de busca. Esse número decorre dos parâmetros adotados no AG, e pode conter imprecisão devido à possibilidade da avaliação de um indivíduo repetido. Os parâmetros usados por Morady e Dal[12] foram tomados como ponto de partida neste trabalho: $N_{pop} = 200$, $N_{ger} = 400$, $T_{cross} = 65\%$ e $T_{mut} = 35\%$. Desse modo, os parâmetros iniciais totalizam 52.200 avaliações de *fitness*, de acordo com o cálculo a seguir, baseado na atual implementação:

$$avaliacoes = N_{pop} + N_{ger} * N_{pop} * T_{cross}$$

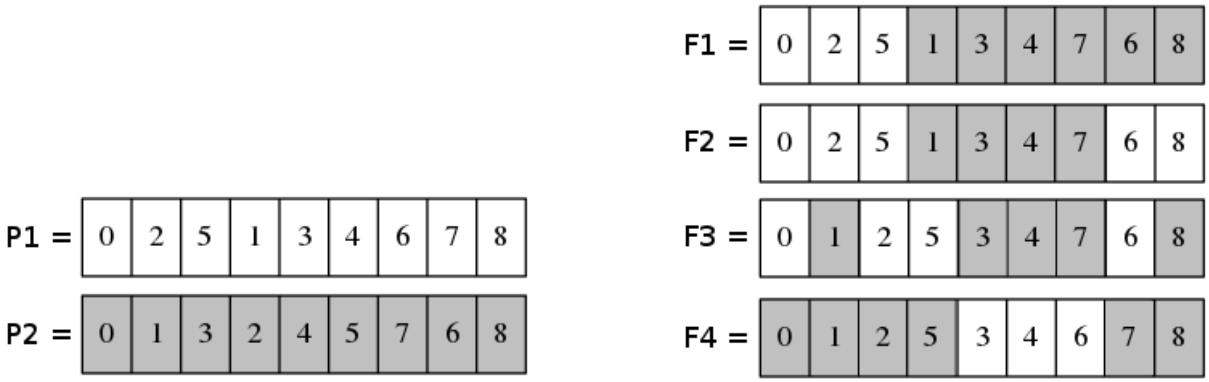

Figura 6. Exemplos de *crossover* sobre vetores de tarefa para o grafo laplace9, omitindo vetores de processadores (P) dos indivíduos. À esquerda, as tarefas dos indivíduos pais. À direita, tarefas de filhos gerados por *crossovers* 1-point ($F1$), 2-point ($F2$), uniforme ($F3$) e OX ($F4$).

$$avaliacoes = 200 + 400 * 200 * 0,65 = 52.200$$

Porém, foi observado que as configurações implementadas produziam resultados melhores se a população fosse maior em relação ao número de gerações (N_{ger}). Então, os parâmetros adotados foram: $N_{pop} = 400$, $N_{ger} = 198$, $T_{cross} = 65\%$ e $T_{mut} = 35\%$, totalizando 51.880 avaliações de *fitness*, ou seja, aproximadamente o mesmo número de avaliações adotado em [12].

5.2 Metodologia

Para avaliar a qualidade das soluções de cada configuração e levando-se em conta o comportamento estocástico dos AGs, os experimentos consistem em 100 execuções para cada grafo do conjunto de *benchmark* em cada uma das configurações da Tabela 1. Os dados avaliados incluem: convergência (número de vezes em que foi encontrada uma solução ótima); melhor *fitness* obtido; *fitness* médio; pior *fitness*.

Entretanto, as soluções ótimas não são conhecidas para a maioria dos grafos do *benchmark* elaborado, que foi descrito na Seção 4.1. Desse modo, os melhores *fitness* encontrados nos experimentos foram tomados como valores pseudo-ótimos, que são suficientes para realizar os experimentos comparativos. Estes valores são apresentados na Tabela 2.

Para analisar e comparar os resultados das configurações, foram usadas quatro métricas: média das convergências e média dos desvios do melhor, médio e pior *fitness* em relação ao *fitness* ótimo, tal que o desvio de um valor de *fitness* x é dado por [1]:

$$desvio(x) = \frac{x - otimo}{otimo}$$

Assim, uma boa configuração é a que obtém alta convergência e baixos desvios, o que significa que suas soluções são próximas da solução ótima.

5.3 Experimentos preliminares com o AG serial

Conforme relatado na seção anteriores, os experimentos com AGs no escalonamento de tarefas encontrados na literatura

Tabela 2. Valores pseudo-ótimos de *makespan* para 4 processadores.

grafo	*makespan*
fft15	560
fft39	820
fft95	1620
fft223	3540
gauss27	67
gauss35	91
gauss44	116
gauss65	175
laplace9	520
laplace16	760
laplace25	1000
laplace36	1290
random30	751
random40	551
random50	496
TG4	34
TG9	21
TG9b	16
TG10	83
TG11	20
TG11b	49
TG11c	12
TG12	27
TG14	35
TG17	37
TG18	440

variam sensivelmente em relação aos métodos de seleção, *crossover* e mutação empregados. Assim, os primeiros experimentos com o AG serial com parâmetros especificados para uma operação mais próxima de aplicações reais, ou seja, com $T_{pop} = 400$ e $N_{ger} = 198$, foram realizados com o objetivo de avaliar qual das configurações apresentadas na Tabela 1 seria a mais adequada para os algoritmos evolutivos. O número de processadores foi fixado em 4 nesses experimentos preliminares, sendo este um ponto de partida que deve possibilitar uma melhor generalização dos resultados para diferentes quantidades de processadores, se comparado com o caso mais trivial de 2 processadores.

A Figura 7 representa graficamente os resultados obtidos de acordo com cada métrica avaliada (convergência média para o ótimo estimado na Tabela 2 e os desvios médios do melhor, médio e pior *fitness*/*makespan*) para cada configuração. É possível verificar que *config6* obteve o melhor resultado em 3 das 4 métricas. Essa configuração utiliza torneio binário na seleção dos pais, método *combine* no *crossover*, com *crossover* 1-ponto em ambos os vetores, ambas mutações "seq" e "proc" e reinserção por elitismo. Além disso, mesmo na única métrica que não foi a melhor, a config6 retornou resultados próximos da melhor configuração (*config10*). Assim, concluise que *config6* produziu boas soluções de maneira mais consistente que as outras no caso geral. A configuração *config6* foi adotada nos experimentos com o AG serial que se seguiram e também com nos experimentos com o AGMP. Ou seja, os ambientes evolutivos empregam: (i) método de seleção por torneio binário, (ii) *crossover* com abordagem *combine* que realiza 2 recombinações independentes para cada vetor que compõe o indivíduo (processadores e tarefas), sendo que em cada vetor foi aplicado o *crossover* de 1 ponto e (iii) mutação mista que aplicam alterações pontuais tanto na sequência dentro de cada processador, quanto nos processadores onde as tarefas estão alocadas.

5.4 Experimentos comparativos entre o AG Serial e o AGMP

Após a realização dos experimentos preliminares com o AG serial, que nos permitiram estabelecer os principais parâmetros e a melhor configuração dos operadores de seleção, *crossover* e mutação, iniciamos os experimentos com o AGMP utilizando os resultados do AG serial como referência. Desse modo, a configuração *config6* foi também aplicada ao AGMP. Inicialmente, foram realizados experimentos para estabelecer qual o melhor valor para o parâmetro número de populações do AGMP. Além disso, foram realizadas comparações entre o AG serial e o AGMP em relação à qualidade das soluções encontradas e ao tempo de execução.

5.4.1 Análise da qualidade das soluções

Seguindo o modelo de experimentos definido para o AG serial, foram feitos testes comparativos variando-se o número de populações do AGMP de 1 a 10, sendo que as execuções com uma população são equivalentes ao AG serial.

Os resultados da Figura 8 mostram que a qualidade das soluções do AG serial foi superada em todas as métricas por pelo menos uma configuração do AGMP. A configuração do AGMP com 7 populações foi a melhor em convergência e obteve o desvio mais baixo de *fitness* médio, mas com um alto desvio de melhor *fitness*. Por outro lado, o AGMP com 8 populações obteve valores baixos e consistentes de desvio, possuindo o menor desvio de pior *fitness*, e alta convergência (a segunda melhor). Por isso, o AGMP com 8 populações foi escolhido como a melhor configuração, e adotada como padrão nos experimentos seguintes.

Em alguns dos experimentos, também nota-se que o AGMP encontrou soluções melhores do que as do AG com a configuração 6, como nos grafos fft39, fft95, laplace36 e random40, que são apresentados na Tabela 3.

Tabela 3. Melhor *fitness* encontrado pelo AG e pelo AGMP com número de populações indicado, para 4 processadores.

grafo	AG	AGMP	populações
fft39	840	820	8
fft95	1620	1600	4
laplace36	1300	1270	8
random40	555	551	6

A Tabela 4 mostra os resultados do teste de hipótese Wilcoxon Rank-Sum com significância de 95%, que compara as medianas dos resultados obtidos pelo AGMP de 8 populações e pelo AG. Cada amostra é composta por 1000 execuções. Foram considerados todos os grafos e, para avaliar se o desempenho relativo dos algoritmos se mantém para instâncias do problema com número diferente de processadores, foram consideradas as arquiteturas com 2, 4, 8 e 16 processadores, totalizando 104 cenários. Nota-se que o AGMP foi mais consistente para a minimização do *fitness* em 44 cenários, se destacando para os grafos da família gauss, laplace16, laplace25, TG14, TG17 e TG18. O AG teve os melhores resultados em 19 cenários, com destaque para os grafos fft39, fft95, fft223 e random40. De modo geral, também pode-se observar que o desempenho relativo dos algoritmos se mantém para as instâncias do problema com número diferente de processadores.

Ao observar que o AGMP superou ou se equiparou ao AG em 85 dos 104 cenários, consideramos que os resultados suportam a hipótese de que o AGMP produz soluções melhores ou equivalentes às do AG no caso geral.

5.4.2 Tempo de execução

Para avaliação de tempo de execução, foram usados os tempos gastos pelos experimentos da seção 5.4.1. Os algoritmos evolutivos, implementados em linguagem C e otimizados pelo GCC, foram executados em um laptop com sistema operacional Ubuntu 16.04 e processador Intel Core i7-6500U, que possui 4 núcleos e clock de 2,50GHz. Tal implementação proporcionou os tempos de execução médios dados na Tabela 5, de acordo com o número de populações. Assim, foi calculado o *Speedup* médio das configurações de AGMP, apresentado na Figura 9, que mensura os desempenhos relativos

Figura 7. Resultados agregados dos testes de configurações do AG para escalonamento de todos os grafos em 4 processadores. Os melhores resultados estão em destaque.

Figura 8. Resultados agregados dos experimentos variando número de populações do AGMP para escalonamento de todos os grafos em 4 processadores. Os melhores resultados estão em destaque.

Tabela 4. Testes de hipótese para as medianas dos resultados do AGMP de 8 populações e do AG. Os símbolos $<$, $=$ e $>$ indicam que a mediana do AGMP foi menor, igual ou maior que a do AG, respectivamente.

	processadores			
grafo	**2**	**4**	**8**	**16**
fft15	=	=	<	=
fft39	=	>	>	>
fft95	<	>	>	>
fft223	=	>	>	>
gauss27	<	<	<	<
gauss35	<	<	<	<
gauss44	<	<	<	<
gauss65	<	<	<	<
laplace9	=	=	=	=
laplace16	=	<	<	<
laplace25	=	<	<	<
laplace36	>	<	<	>
random30	>	>	<	=
random40	>	=	>	>
random50	=	<	<	=
TG4	=	=	=	=
TG9	=	=	<	=
TG9b	=	=	=	=
TG10	>	=	=	>
TG11	=	=	=	<
TG11b	=	=	=	=
TG11c	=	=	=	=
TG12	=	>	<	<
TG14	<	<	<	=
TG17	<	<	<	<
TG18	<	<	<	<

ao tempo de execução do AG serial. O gráfico tem pico em 4 populações e degrada em seguida, devido ao número de núcleos do processador utilizado, o que é previsto pela lei de Amdahl.

Considerando que o valor ideal de *Speedup* é igual ao número de processos utilizado por um programa paralelo, nota-se que o *overhead* relacionado ao procedimento de migração do AGMP teve impacto considerável sobre o seu tempo de execução. Por exemplo, o AGMP de 4 populações obteve *Speedup* de apenas $2,6$, sendo que, idealmente, seria 4. Neste ponto, a configuração mais eficiente do AGMP foi a de 2 populações, que obteve o maior *Speedup* por número de processos.

Dado que o AGMP de 8 populações foi a configuração que obteve as soluções de melhor qualidade nos experimentos anteriores, nota-se que há um *trade-off* entre qualidade das soluções e tempo de execução, uma vez que o *Speedup* desta configuração foi inferior ao obtido pelas configurações supracitadas.

É importante observar que os baixos tempos de execução apresentados na Tabela 5 não refletem o espaço de busca

Tabela 5. Tempo de execução médio dos experimentos, em milissegundos.

populações	1	2	3	4	5
tempo (ms)	40,33	20,52	19,93	15,59	23,30
populações	6	7	8	9	10
tempo (ms)	21,93	23,67	23,00	22,89	23,26

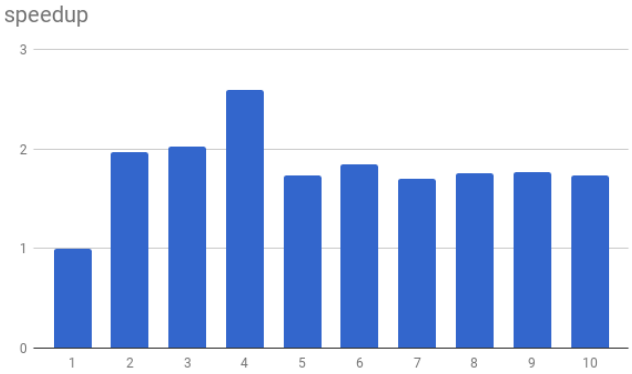

Figura 9. *Speedup* por número de populações.

do problema. Na realidade, esses tempos são resultado da exploração de um número fixo de pontos de busca dado pelos parâmetros dos algoritmos evolutivos, como definido na seção 5.1. Por exemplo, sendo n o número de tarefas de um grafo, a cardinalidade do espaço de busca para o grafo laplace9 e 8 processadores é limitado entre [12]: $P^n = 8^9 = 134.217.728$ e $n!P^n = 9! \cdot 8^9 = 48.704.929.136.640$, ou seja, este limite inferior é 2.571 vezes superior à quantidade de soluções avaliadas pelos algoritmos evolutivos neste trabalho. Para demonstrar experimentalmente a complexidade de se obter uma solução ótima para esta instância relativamente trivial, implementamos o algoritmo *Branch and Bound*, que aplica uma árvore de busca e uma poda baseada no custo das tarefas restantes dividido pelo número de processadores. Ele foi implementado em linguagem Python, por proporcionar maior simplicidade de programação apesar de menor desempenho. Com este programa, após 76 minutos de execução e 414.473.362 avaliações de *makespan*, foi obtido um *makespan* de 520, o mesmo resultado obtido pelos algoritmos evolutivos com apenas 51.880 avaliações. Assim, verifica-se a capacidade destes de obter soluções ótimas mesmo com recursos consideravelmente reduzidos. Também pode-se verificar a relativa complexidade de se obter uma solução ótima mesmo para instâncias menores do problema e a necessidade do uso de métodos aproximados devido ao crescimento exponencial do espaço de busca.

6. Conclusões e Trabalhos Futuros

Neste estudo, o desempenho do AGMP foi comparado com o do AG serial em relação ao problema de escalonamento de tarefas. Ambos foram desenvolvidos com a representação genética de dois vetores, que tem sido utilizada em trabalhos recentes. Foram implementados diversos métodos de

crossover adequados para cada vetor e, para a forma de sua aplicação, foram identificadas na literatura três abordagens diferentes, que foram implementadas e comparadas.

O desempenho do AGMP foi medido variando-se o número de populações e comparado com o desempenho do AG serial em termos de qualidade agregada das soluções e tempo de execução.

Foi composto um conjunto de grafos para *benchmark* dos algoritmos, dividido entre grafos retirados da literatura e famílias de grafos baseadas em algoritmos reais. Para estas famílias, foram implementados scripts que geram grafos correspondentes à execução do algoritmo para diferentes tamanhos de entrada.

Os resultados suportam a hipótese de que AGMP é uma técnica que, dados parâmetros adequados, faz um melhor aproveitamento dos recursos computacionais disponíveis em comparação com o AG serial. Isto se dá de duas formas:

- Produção de soluções de mesma ou melhor qualidade para uma dada quantidade de avaliações de *fitness*.
- Redução do tempo de execução pelo uso de múltiplos núcleos dos processadores atuais.

Como trabalhos futuros, identificamos: realizar experimentos complementares com arquiteturas e grafos diferentes, e com variações do problema de escalonamento. Comparar o desempenho com outras abordagens de AG paralelo e outras meta-heurísticas, e.g. busca tabu [28], GRASP [29] e VNS [30]. Analisar a representatividade das formas de representação genética em relação às soluções alcançáveis no espaço de busca.

7. Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Agradecemos a CAPES, CNPq e FAPEMIG pelo apoio financeiro que possibilitou esta pesquisa.

8. Contribuição dos Autores

- Bruno W. Dantas Morais: desenvolvimento geral do trabalho.
- Gina M. Barbosa de Oliveira: orientação.
- Tiago Ismailer de Carvalho: co-orientação e estudo sobre as famílias de grafos.

Referências

[1] HOU, E. S. H.; ANSARI, N.; REN, H. A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, v. 5, n. 2, p. 113–120, 2 1994.

[2] GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. v. 1.

[3] ALBA, E.; TROYA, J. M. A survey of parallel distributed genetic algorithms. *Complex,*, v. 4, n. 4, p. 31–52, 3 1999.

[4] KWOK, Y.-K.; AHMAD, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput, Surv,*, v. 31, n. 4, p. 406–471, 12 1999.

[5] CANTú-PAZ, E. A survey of parallel genetic algorithms. *Calc. paralleles reseaux syst. repartis*, v. 10, n. 2, p. 141–171, 1998.

[6] CANTÚ-PAZ, E. et al. Are multiple runs of genetic algorithms better than one? In: *Genetic and Evolutionary Computation — GECCO 2003*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 801–812.

[7] CORREA, R. C.; FERREIRA, A.; REBREYEND, P. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Trans. Parallel Distrib. Syst.*, v. 10, n. 8, p. 825–837, 8 1999.

[8] KAUR, K. et al. Heuristics based genetic algorithm for scheduling static tasks in homogeneous parallel system. *Int. J. Comput. Sci. Secur. (IJCSS)*, v. 4, n. 2, p. 183–198, 1999.

[9] WANG, L. et al. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *J. Parallel Distrib. Comput.*, v. 47, n. 1, p. 8 – 22, 1997.

[10] OMARA, F. A.; ARAFA, M. M. Genetic algorithms for task scheduling problem. *J. Parallel Distrib. Comput.*, v. 70, n. 1, p. 13 – 22, 2010.

[11] CHITRA, P.; RAJARAM, R.; VENKATESH, P. Application and comparison of hybrid evolutionary multiobjective optimization algorithms for solving task scheduling problem on heterogeneous systems. *Appl. Soft Comput.*, v. 11, n. 2, p. 2725 – 2734, 2011.

[12] MORADY, R.; DAL, D. A multi-population based parallel genetic algorithm for multiprocessor task scheduling with communication costs. In: . Messina, Italy: IEEE, 2016. (ISCC, '16), p. 766–772.

[13] KWOK, Y.-K.; AHMAD, I. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *J. Parallel Distrib, Comput,*, v. 47, n. 1, p. 58–77, 11 1997.

[14] HWANG, R.; GEN, M.; KATAYAMA, H. A comparison of multiprocessor task scheduling algorithms with communication costs. *Comput. Oper. Res.*, v. 35, n. 3, p. 976 – 993, 2008.

[15] XU, Y. et al. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Inf. Sci.*, v. 270, n. 1, p. 255 – 287, 2014.

[16] QI, J. G.; BURNS, G. R.; HARRISON, D. K. The application of parallel multipopulation genetic algorithms to dynamic job-shop scheduling. *Int. J. Adv. Manuf. Technol.*, v. 16, n. 8, p. 609–615, 7 2000.

[17] SRINIVASA, K. G.; VENUGOPAL, K. R.; PATNAIK, L. M. A self-adaptive migration model genetic algorithm for data mining applications. *Inf, Sci,*, v. 177, n. 20, p. 4295–4313, 10 2007.

[18] GEHRING, H.; BORTFELDT, A. A parallel genetic algorithm for solving the container loading problem. *Int. Trans. Oper. Res.*, v. 9, n. 5-6, p. 497 – 511, 7 2002.

[19] MÜHLENBEIN, H.; SCHOMISCH, M.; BORN, J. The parallel genetic algorithm as function optimizer. *Parallel Comput,*, v. 17, n. 6-7, p. 619–632, 9 1991.

[20] YAO, J.; KHARMA, N.; GROGONO, P. Bi-objective multipopulation genetic algorithm for multimodal function optimization. *IEEE Trans. Evol. Comput.*, v. 14, n. 1, p. 80–102, 2 2010.

[21] HAN, K.-H. et al. Parallel quantum-inspired genetic algorithm for combinatorial optimization problem. In: . Seoul, South Korea: IEEE, 2001. (CEC2001, v. 2), p. 1422–1429 vol. 2.

[22] OLTEANU, A.; MARIN, A. Generation and evaluation of scheduling dags: How to provide similar evaluation conditions. *Comput. Sci. Master Res.*, v. 1, n. 1, p. 57, 2011.

[23] JIANG, Y.; SHAO, Z.; GUO, Y. A dag scheduling scheme on heterogeneous computing systems using tuple-based chemical reaction optimization. *Sci. World J.*, v. 2014, n. 1, p. 404375, 6 2014.

[24] COSNARD, M. et al. Parallel gaussian elimination on an mimd computer. *Parallel Comput.*, v. 6, n. 3, p. 275 – 296, 1988.

[25] WU, M. Y.; GAJSKI, D. D. Hypertool: a programming aid for message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, v. 1, n. 3, p. 330–343, 7 1990.

[26] XU, Y.; LI, K.; HU, J. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Inf. Sci.*, v. 270, n. 1, p. 255–257, 6 2014.

[27] CARNEIRO, M. G. *Abordagens baseadas em autômatos celulares síncronos para o escalonamento estático de tarefas em multiprocessadores*. Tese (Doutorado), Uberlândia, Brasil, 2012.

[28] GLOVER, F.; LAGUNA, M. Tabu search. In: *Handbook of Combinatorial Optimization*. 1. ed. Boston, MA: Springer US, 1999. Volume 1–3, p. 2093–2229.

[29] FEO, T. A.; RESENDE, M. G. C. Greedy randomized adaptive search procedures. *J. Glob. Optim.*, v. 6, n. 2, p. 109–133, 3 1995.

[30] MLADENOVIC, N.; HANSEN, P. Variable neighborhood search. *Comput. Oper. Res.*, v. 24, n. 11, p. 1097 – 1100, 1997.