X Congresso Brasileiro de Inteligência Computacional (CBIC'2011), 8 a 11 de Novembro de 2011, Fortaleza, Ceará

UM MODELO BASEADO EM AUTÔMATOS CELULARES SÍNCRONOS PARA O ESCALONAMENTO DE TAREFAS EM MULTIPROCESSADORES

Murillo G. Carneiro and Gina M. B. Oliveira
Universidade Federal de Uberlândia
murillo@mestrado.ufu.br, gina@facom.ufu.br

Resumo – O Problema de Escalonamento Estático de Tarefas em multiprocessadores (PEET) é NP-Completo e as abordagens propostas para resolvê-lo normalmente utilizam heurísticas ou meta-heurísticas. Entretanto, muitas delas não têm a habilidade de extrair conhecimento do processo de escalonamento de uma aplicação paralela e reusá-lo em outras instâncias. Trabalhos anteriores têm mostrado o uso promissor de Autômatos Celulares (AC) para extração e reuso de conhecimento no PEET, mas o forte paralelismo intrínseco nestas estruturas não tem sido explorado com sucesso. Ao contrário, resultados satisfatórios de escalonamento de tarefas (ótimos ou sub-ótimos) foram obtidos anteriormente com o uso de ACs com modo de atualização de células sequencial, que os tornam estruturas essencialmente sequenciais e não propícias à implementação em *hardware* paralelo. Este artigo apresenta um novo modelo de escalonador baseado em AC com modo de atualização síncrono. Grafos de programa encontrados nas pesquisas relacionadas foram utilizados para avaliar a abordagem. Os experimentos mostram que o modelo é capaz de resolver grafos de programas paralelos que os modelos síncronos da literatura não conseguiram e apresenta resultados superiores aos modelos sequenciais, em termos de convergência para o escalonamento ótimo.

Palavras-chave – Escalonador baseado em AC, atualização síncrona, escalonamento estático de tarefas, sistemas multiprocessadores.

Abstract – The Task Static Scheduling Problem on multiprocessors (TSSP) is NP-Complete and proposed approaches to solve it typically use heuristics or meta-heuristics. However, many of them do not have the ability to extract knowledge of the scheduling process of a parallel application and reuse it in others. Previous works has shown the promising use of Cellular Automata (CA) for extraction and reuse of knowledge in the problem, but the massive parallelism inherent in these structures has not been successfully exploited. On the contrary, satisfactory results of tasks scheduling (optimals or near optimals) were obtained with use of CAs with sequential update mode of cells that make them essentially sequentials and not appropriate to implementation in parallel hardware. This paper presents a new model of CA-based scheduler with synchronous update mode. Program graphs found in related researches were used for evaluate the approach. The experiments show that the model is able to solve graphs of parallel programs that previous models did not resolve and it present superior results to sequential models in terms of convergence to optimal scheduling.

Keywords – CA-based scheduler, synchronous update, task static scheduling problem, multiprocessor systems.

1. INTRODUÇÃO

O uso simultâneo de recursos computacionais tem sido uma das principais alternativas para suprir a demanda computacional de aplicações cada vez mais computacionalmente intensivas. A exploração de ambientes multiprocessadores ou multicomputadores por cientistas e empresas na tentativa de resolver problemas complexos tem sido frequente. Entretanto, para que esses ambientes sejam efetivamente aproveitados é importante que seja possível a divisão da aplicação em tarefas independentes e que essas tarefas sejam alocadas dentre os nós de processamento de forma a tirar o máximo proveito dos mesmos. Assim, o escalonamento eficiente das tarefas tem um papel fundamental nas arquiteturas multiprocessadoras.

De modo geral, o escalonamento é um processo de tomada de decisão, que envolve recursos e tarefas, na busca pela otimização de um ou mais objetivos. De acordo com [1], os recursos podem ser máquinas em uma oficina, unidades de processamento em um ambiente computacional e assim por diante, enquanto que as tarefas podem ser operações em um processo de produção, execuções de um programa de computador, entre outros. Assim, tem-se que o problema de escalonamento pode ser subdividido em vários outros tais como o problema de escalonamento de produção, de empregados e de tarefas computacionais.

O problema do escalonamento de tarefas computacionais em multiprocessadores (PET) consiste em alocar tarefas que compõem um programa paralelo entre os nós de uma arquitetura com múltiplos processadores. No caso do problema de escalonamento estático de tarefas (PEET), todas as informações sobre as tarefas são conhecidas a priori. Uma solução ótima de uma instância do PEET é tal que as restrições de precedência entre as tarefas são atendidas e o tempo total de execução -ou *makespan*- é minimizado. O problema é conhecido por ser NP-Completo em sua forma geral [2] e tem sido um grande desafio para muitos pesquisadores. Diante disso, métodos heurísticos têm sido empregados na tentativa de encontrar boas soluções para o problema. Entretanto, muitos deles não são capazes de extrair conhecimento sobre o processo de escalonamento e reusá-lo em outras instâncias da aplicação.

Autômatos Celulares (AC) têm sido empregados com sucesso nos mais diversos campos de pesquisa da computação: criptografia [3], simulação de sistemas complexos e de vida artificial [4], entre outros. ACs são sistemas dinâmicos discretos (tempo e espaço) e possuem como uma de suas principais características a capacidade de emergir um comportamento global a partir de interações entre unidades locais. Trabalhos anteriores [5], [6] e [7] apontaram o uso promissor de abordagens baseadas em AC para o Problema de Escalonamento Estático de Tarefas (PEET). Um AC é composto de um reticulado e uma regra de transição. A regra de transição é aplicada sobre os estados das células do reticulado por um número finito de passos. O modo de atualização dos estados das células do reticulado mais investigado é o síncrono ou paralelo, uma vez que ele permite explorar o paralelismo intrínseco dos ACs. Entretanto, no caso dos modelos anteriores de ACs para o escalonamento, o modo de atualização sequencial retornou bons resultados, enquanto o modo de atualização síncrono não retornou resultados satisfatórios.

O objetivo deste trabalho é o desenvolvimento de um novo método baseado em AC com atualização síncrona das células capaz de realizar o escalonamento ótimo (ou, pelo menos, sub-ótimo) de tarefas em sistemas multiprocessadores. É desejado que, esse método, além de realizar o escalonamento ótimo de um grafo, esteja apto a extrair conhecimento sobre o processo de escalonamento e reusá-lo no escalonamento de outras instâncias. Para auxiliar nos processos de escalonamento e extração do conhecimento é possível combinar o uso de ACs e Algoritmos Genéticos (AGs) [8] conforme já foi proposto em [9].

O restante do artigo é organizado da seguinte maneira: as Seções 2 e 3 apresentam, respectivamente, o PEET em multiprocessadores e uma introdução sobre os ACs. A Seção 4 contém os principais conceitos sobre escalonamento baseado em AC e a Seção 5 destaca o sistema proposto neste trabalho. A Seção 6 mostra resultados experimentais da aplicação do sistema para o escalonamento de programas paralelos em arquiteturas com dois processadores. Conclusões e trabalhos futuros são apresentados na Seção 7.

2. ESCALONAMENTO EM MULTIPROCESSADORES

No PEET, um programa paralelo pode ser representado por um Grafo Acíclico Direcionado (GAD) definido por uma tupla $G = (V, E, W, C)$, onde $V = \{ t_1, \ldots, t_N \}$ denota o conjunto de N tarefas do grafo; $E = \{ e_{i,j} \mid t_i, t_j \in V \}$ representa o conjunto de arestas de comunicação, também denominadas restrições de precedência; $W = \{ w_1, \ldots, w_n \}$ representa o conjunto de tempos de execução das tarefa, ou seja, para cada tarefa $t \in V$ é associado um peso de computação $w(t) \in W$ referente ao custo de execução da mesma; e $C = \{ c_{i,j} \mid e_{i,j} \in E \}$ denota o conjunto de custos de comunicação de arestas, ou seja, a cada aresta $e_{i,j} \in E$ é associado um custo de comunicação $c_{i,j} \in C$ relacionado ao custo de transferência de dados entre as tarefas t_i e t_j quando são executadas em processadores distintos. Satisfazendo essas condições G é denominado grafo de precedência de tarefas ou, simplesmente, grafo de programa. Na Figura 1 tem-se um grafo de programa com 15 tarefas denominado *outtree15*.

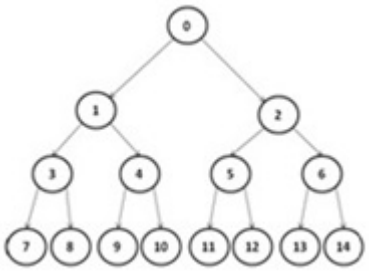

Figura 1: Grafo de programa com 15 tarefas (*outtree15*).

É importante dizer que o conjunto de arestas E define as relações de precedência entre tarefas. Assim, uma tarefa não pode ser executada a menos que todos os seus predecessores tenham completado suas execuções e todos os dados relevantes estejam disponíveis. Preempção de tarefas e execuções redundantes não são permitidas na versão do problema considerado neste artigo.

Um sistema multiprocessador por sua vez, pode ser representado por um grafo não ponderado e não direcionado $G_s = (V_s, E_s)$, denominado grafo de sistema. V_s é o conjunto de P processadores do grafo de sistema e E_s é o conjunto de canais bi-direcionais entre processadores que define a topologia do sistema multiprocessador. Nesse modelo assume-se também que todos os processadores têm o mesmo poder computacional e que as comunicações entre os canais não consomem qualquer tempo adicional do processador além do próprio tempo de comunicação entre as tarefas já especificado no grafo.

Uma política de escalonamento define a ordem de execução das tarefas em cada processador. Assim, enquanto o escalonador distribui as tarefas entre os processadores, a política de escalonamento ordena estas tarefas dentro de cada processador. Neste artigo, foi utilizada a mesma política de escalonamento para todos os testes: a tarefa com maior nível dinâmico deve ser executada primeiro. O nível (b-level) de uma tarefa do grafo de programa é o maior custo entre essa tarefa e uma tarefa de saída do grafo, assim o nível de uma tarefa i pode ser recursivamente calculado por:

$$bl_i = \left\{ \begin{array}{l} w_i, \text{ se } i \text{ é uma tarefa saída;} \\ \max_{j \in successores(i)} (bl_i + c_{i,j}) + w_i, \text{ caso contrário.} \end{array} \right\}$$

O nível da tarefa é dinâmico quando é calculado considerando a alocação das tarefas nos processadores e o custo de comunicação é adicionado quando as tarefas estão alocadas em processadores distintos.

3. AUTÔMATOS CELULARES

Um AC é um sistema composto por espaço celular e uma função de transição de estados. O espaço celular é um reticulado de l células (componentes simples e idênticos que possuem conectividade local e condições de contorno) dispostas em um arranjo *d-dimensional*. Cada célula assume um estado de um conjunto finito de k estados possíveis a cada instante de tempo. A regra ou função de transição f por sua vez, é responsável por determinar o próximo estado de cada célula do AC a partir de seu estado atual e dos estados de suas vizinhas. Assim, a regra determina o comportamento apresentado pelo AC durante a evolução temporal do reticulado. A evolução temporal do AC é o processo de aplicar a regra sobre o reticulado por um número determinado de passos de tempo t. Ainda sobre a evolução temporal de um reticulado, tal processo pode acontecer dos seguintes modos [5]:

- **Paralelo ou Síncrono**: onde todas as células do reticulado atualizam seus estados sincronamente em cada passo de tempo.
- **Sequencial ou Assíncrono**: em que apenas uma célula por vez atualiza o seu estado e esse novo estado é considerado na atualização das outras células. Diz-se sequencial porque a ordem em que cada célula é atualizada é dada pela sua posição no reticulado, da esquerda para a direita.

Para cada célula i, chamada célula central, uma vizinhança de raio R é definida. Assim, o tamanho da vizinhança de cada célula i (que inclui a própria célula) é dado por: $m = 2R + 1$. É importante perceber que se a célula i no tempo t apresenta estado q_i^t, então no tempo $t + 1$ o seu estado q_i^{t+1} dependerá apenas dos estados das células de sua vizinhança no tempo t, ou seja, será dado por: $f(q_{i-R}^t, \ldots, q_{i-1}^t, q_i^t, q_{i+1}^t, \ldots, q_{i+R}^t)$.

Como exemplo, na Figura 2(a) é exibido um autômato celular unidimensional ($d = 1$), binário ($k = 2$), com dez células ($l = 10$) e vizinhança de raio 1 ($m = 3$). Na Figura 2(b) tem-se a regra de transição (f) que apresenta o novo estado da célula central (q_i^{t+1}) para todas as configurações possíveis da vizinhança $(q_{i-R}^t, \ldots, q_i^t, \ldots, q_{i+R}^t)$. Na Figura 2(c) é apresentada a evolução temporal do reticulado por 2 passos de tempo considerando-se o modo de atualização síncrono e com condição de contorno periódica (célula mais à esquerda está conectada à célula mais à direita, formando um anel) por dois passos de tempo. Na Figura 2(d) é apresentada a evolução temporal do reticulado em modo sequencial e com condição de contorno nula (tanto a vizinhança à esquerda da 1^a célula quanto a vizinhança à direita da última célula são consideradas no estado 0) também por dois passos de tempo. Na figura, o primeiro passo de tempo foi representado de forma detalhada, apresentando a atualização sequencial de cada célula até se obter o novo reticulado. No segundo passo de tempo, apenas o resultado final do reticulado (após essa atualização seqüencial) está representado.

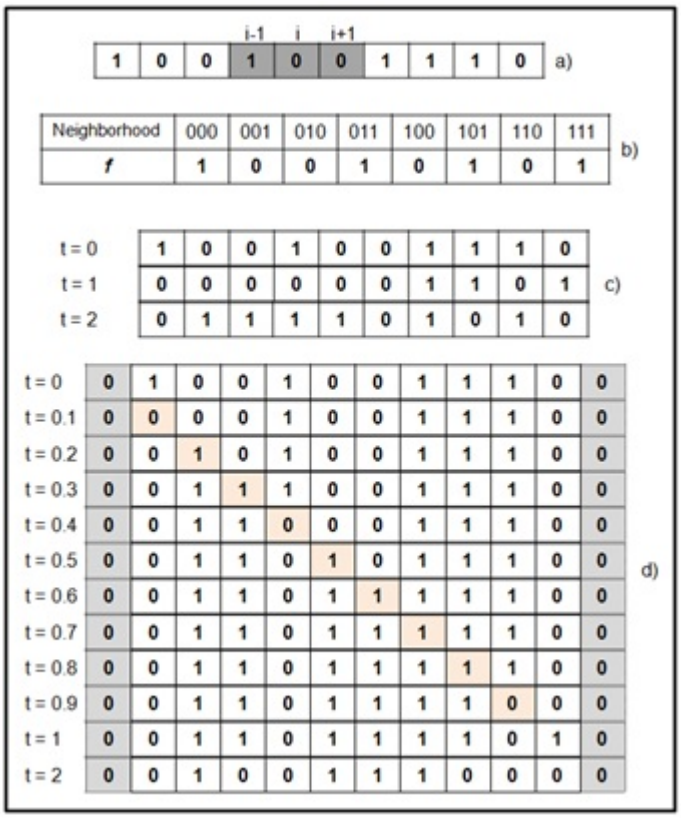

Figura 2: Exemplo de AC: (a) reticulado inicial; (b) regra de transição; (c) modo de atualização síncrono; (d) modo de atualização sequencial;

4. ESCALONADOR BASEADO EM AC: MODELOS ANTERIORES

No modelo de escalonador baseado em AC assume-se que cada célula do reticulado está associada com uma tarefa do grafo de programa. Assim, se um conjunto de tarefas do grafo de programa tem cardinalidade x, o reticulado do AC deve possuir x células. Além disso, considerando-se uma arquitetura composta por P processadores, o AC a ser utilizado terá N possíveis estados. Dessa forma, supondo um sistema com dois processadores (P0 e P1), tem-se que cada célula do AC pode assumir, em um instante de tempo t, o valor 0, indicando que a tarefa correspondente está alocada no processador P0, ou o valor 1, indicando que a tarefa está alocada no processador P1. Por exemplo, um grafo de programa que é composto por 4 tarefas, deve ser representado por um reticulado de 4 células e em uma configuração onde as tarefas 0 e 3 estão alocadas em P0 e as tarefas 1 e 2 em P1, o reticulado será 0110. Para calcular o tempo do escalonamento T desse reticulado é necessário usar uma política de escalonamento que é responsável por definir a ordem das tarefas em cada processador.

No modelo proposto em [5], é apresentado um escalonador baseado em AC que opera em dois modos: aprendizagem e operação. No modo de aprendizagem, o escalonador faz uso de um AG para descobrir regras de ACs capazes de encontrar soluções ótimas (ou sub-ótimas) partindo-se de diferentes alocações aleatórias de um grafo de programa, ou seja, diferentes reticulados iniciais. A população inicial desse AG é composta de possíveis regras de transição geradas aleatoriamente. A cada geração do AG, um conjunto de reticulados iniciais (CR) é sorteado. Assim, o método de avaliação da população (função de *fitness* do AG) é a evolução temporal a partir de cada reticulado inicial desse CR pelos individuos (regras de transição) por t passos de tempo. No tempo t, cada reticulado obtido é escalonado mediante uma política de escalonamento escolhida a priori. Considerando-se todos os reticulados iniciais do CR obtêm-se a média de custo de escalonamento para aquela regra. Vale destacar que o modelo de AG em [5] utiliza uma estratégia elitista, onde as T_{elite} melhores regras são mantidas para a próxima geração, e que apenas a elite é considerada na seleção de pais para o *crossover*. A evolução temporal das células do reticulado, a partir do instante inicial, pode ser feita no modo síncrono ou sequencial sendo este um parâmetro de entrada do escalonador.

No modo de operação normal espera-se que, para uma dada alocação inicial qualquer de tarefas, as regras de AC sejam capazes de minimizar o *Makespam*. É esperado também que a evolução temporal das regras na fase de aprendizagem possa ser utilizada com bom desempenho no escalonamento de outros grafos de programa.

Em [6], tem-se um escalonador baseado em AC que opera em três modos: aprendizagem, operação e reutilização. Os dois primeiros modos são semelhantes aos propostos por [5]. No modo de reutilização, as regras descobertas anteriormente são reusadas, com o apoio de um Sistema Imunológico Artificial (SIA), para resolver novas instâncias do problema. No modelo de SIA, as regras descobertas são consideradas anticorpos enquanto que novas instâncias do problema de escalonamento são tratadas como antígenos. Dessa forma, um dado anticorpo reconhece um antígeno específico se ele pode encontrar um escalonamento ótimo para ele.

Os experimentos apresentados em [5] e [6] alcançaram bons resultados em seus modelos através do modo de atualização sequencial das células do reticulado, porém não conseguiram obter resultados satisfatórios com o modo de atualização síncrono. Outra característica para discernir os trabalhos anteriores [5] e [6] é o modo de vizinhança empregado. Em [5], apesar de utilizar modelos de vizinhança linear como é investigado no presente trabalho, a ênfase é no emprego de um modelo de vizinhança não linear no qual os vizinhos de uma célula não são definidos simplesmente pelo raio. Essa vizinhança chamada de selecionada é mais complexa e define um espaço de regras de alta cardinalidade, tornando o processo de aprendizagem mais lento e de difícil convergência. Em [6], o único modelo de vizinhança investigado é o linear, o mesmo adotado no presente trabalho.

5. ESCALONAMENTO COM AC SÍNCRONO

Localidade de interações celulares, simplicidade de componentes básicos (células) e possibilidade de implementação em *hardware* paralelo estão entre as mais notáveis caracteristicas dos autômatos celulares [4]. Contudo, os modelos na literatura que obtiveram melhores resultados não são capazes de explorar a capacidade intrínseca de paralelismo dos ACs uma vez que utilizam o modo de atualização sequencial (apenas uma célula pode atualizar seu estado por vez) [5]. Nosso trabalho tem por objetivo a construção de um modelo de escalonador baseado em ACs que, através do modo de atualização síncrono, consiga obter resultados satisfatórios de escalonamento, similares aos obtidos nos trabalhos anteriores com o modo sequencial.

A abordagem de escalonador baseado em AC com modo de atualização síncrono apresentada neste trabalho utiliza vizinhança linear por três razões: simplicidade, baixo custo computacional e número ilimitado de processadores. É certo que em trabalhos anteriores [5] as vizinhanças não lineares apresentaram resultados melhores do que as lineares, entretanto elas estão limitadas a uma arquitetura multiprocessadora com dois nós e possuem uma estrutura bastante complexa. Como exemplo, a vizinhança não linear investigada em [5] emprega regras com 250 *bits*, enquanto na vizinhança linear, considerando o tamanho do raio igual a 3, o tamanho das regras é igual a 128 *bits*. No caso da vizinhança não-linear investigada em [5], os modelos de vizinhança só prevêem o uso de 2 processadores no grafo de sistema e uma generalização desses modelos para um número maior de processadores seria bastante complexa e levaria a uma regra de dimensão ainda maior. No caso da vizinhança linear, para um número maior de processadores no grafo de sistema, basta utilizar um número maior de estados por célula.

Outra característica importante diz respeito à condição de contorno utilizada nos modelos anteriores que é nula. Contudo, no novo modelo, as células de contorno à esquerda do reticulado possuem valor 0 e as da direita 1, diferente dos demais modelos que utilizam 0 dos dois lados. Para decidir por esses valores fizemos um estudo acerca da influência da condição de contorno nos resultados encontrados pelas vizinhanças onde concluiu-se que tal condição (0 e 1) oferece maior equilíbrio (uso mais distribuído dos estados de transição das regras) na evolução temporal do reticulado.

Além das mudanças na estrutura do AC, o AG empregado no novo modelo também utiliza uma abordagem diferente dos modelos anteriores por não utilizar estratégia elitista (operadores de seleção e re-inserção utilizam elitismo). Dessa forma, no novo modelo, a seleção é realizada por meio de torneio simples e a re-inserção acontece baseada na aptidão, isto é, a população total (pais e filhos) é ordenada e selecionam-se as melhores regras. O intuito dessas modificações é estimular a competição entre os indivíduos da população de modo a permitir uma busca mais ampla no espaço de possíveis soluções para o problema e consequentemente formar um conjunto de regras mais eficiente em sua totalidade, o que é bastante difícil utilizando estratégia elitista.

Uma descrição sobre o novo modelo é realizada à seguir, para efeito de entendimento considere: G o número de gerações, T_{pop} o tamanho da população, P_{cross} o percentual de indíviduos gerados no *crossover* em cada G e P_{mut} a taxa de mutação do AG, R o raio que define o tamanho das regras e a vizinhança no reticulado, CI o conjunto de reticulados ou configurações iniciais aleatórias e S o número de passos de atualização do AC.

Na Figura 3 é apresentado um *framework* do modelo de escalonador proposto. Chamamos esse escalonador de EACS (Escalonador baseado em Autômato Celular com atualização Síncrona). O escalonador recebe como entrada um grafo de programa e um grafo de sistema. No modo de aprendizagem, assim como em [9], utiliza-se um AG para buscar regras de AC capazes de encontrar o escalonamento ótimo para o grafo de programa. A Figura 4 apresenta o pseudo-código do AG utilizado no modelo.

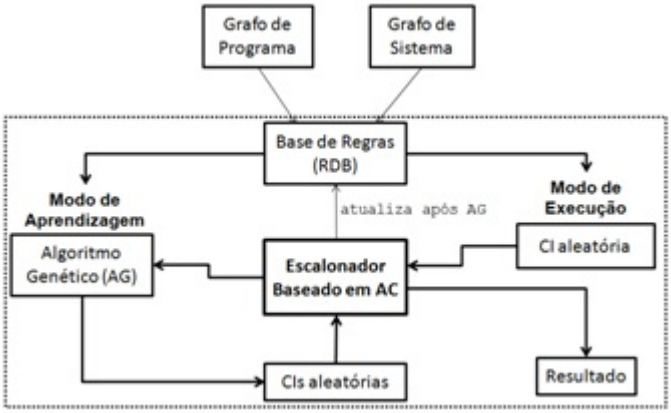

Figura 3: Modelo de escalonador baseado em AC (EACS).

Modo de Aprendizagem

Passo 0 Gere aleatoriamente uma população de T_{pop} regras;

Passo 1 Enquanto (!condição_término) // faça Passos 2-6

Passo 2 Calcule o *fitness* para todas as regras // faça Passos 2.1-2.3

Passo 2.1 Gere aleatoriamente um conjunto de configurações iniciais (CI);

Passo 2.2 Utilizando evolução paralela, aplique as T_{pop} regras sobre cada CI por S passos de tempo e calcule o tempo total de execução T para cada alocação final;

Passo 2.3 Obtenha o *fitness* para cada regra através da soma dos valores de T encontrados para cada CI avaliada;

Passo 3 Selecione pares de regras em T_{pop} utilizando torneio simples;

Passo 4 Aplique o *crossover* de ponto simples para os pares selecionados gerando P_c regras;

Passo 5 Submeta P_{cross} regras à mutação com probabilidade P_{mut};

Passo 6 Ordene T_{pop}+P_{cross} e escolha as T_{pop} melhores regras para a nova geração;

Figura 4: Pseudo-código do AG utilizado no modo de aprendizagem.

No modo de execução, o grafo de programa é carregado em uma CI e o AC é equipado com uma regra da RDB. O AC então atualiza sincronamente o reticulado por S passos de tempo obtendo a alocação final das tarefas que é então submetida a política de escalonamento. Posterior a isso, tem-se o tempo de conclusão encontrado para o escalonamento do grafo de programa.

6. EXPERIMENTOS E RESULTADOS

No novo modelo do escalonador EACS o foco é o modo de atualização síncrono das células e a vizinhança linear do AC. Assim, o objetivo desse trabalho é desenvolver um modelo de escalonador com essas características que seja capaz de encontrar escalonamentos ótimos (ou próximos do ótimo) para vários grafos de programa.

Nos experimentos, assim como em trabalhos anteriores [5], considerou-se um grafo de sistema com 2 processadores. Para avaliar a qualidade do modelo foram considerados grafos de programa disponíveis na literatura, incluindo todos aqueles com possibilidade de reprodução encontrados nos trabalhos anteriores [5], [6], [7]. A Figura 5 mostra quatro grafos de programa utilizados nos experimentos. A Figura 5(a) apresenta o grafo de programa *g18* com custos computacionais (custo de cada vértice) impressos na figura e custos de comunicação (custo de cada aresta) igual a 1 para todas as arestas. Figura 5(b) destaca o grafo de programa *g40* com custos de computação e comunicação iguais, respectivamente, a 4 e 1 para todas as tarefas e arestas. A Figura 5(c) apresenta o grafo de programa conhecido como *gauss18* com custos de comunicação e computação exibidos na figura e a Figura 1 mostra um grafo de programa da árvore binária (*out-tree*) com 15 tarefas denominado *outtree15*. Para avaliar o reuso do conhecimento extraído, também foram utilizados grafos de programa *out-trees* com 31, 63, 127, 255, 511 e 1023 tarefas. Os custos de computação e comunicação de todas as tarefas e arestas nas árvores binárias é igual a 1.

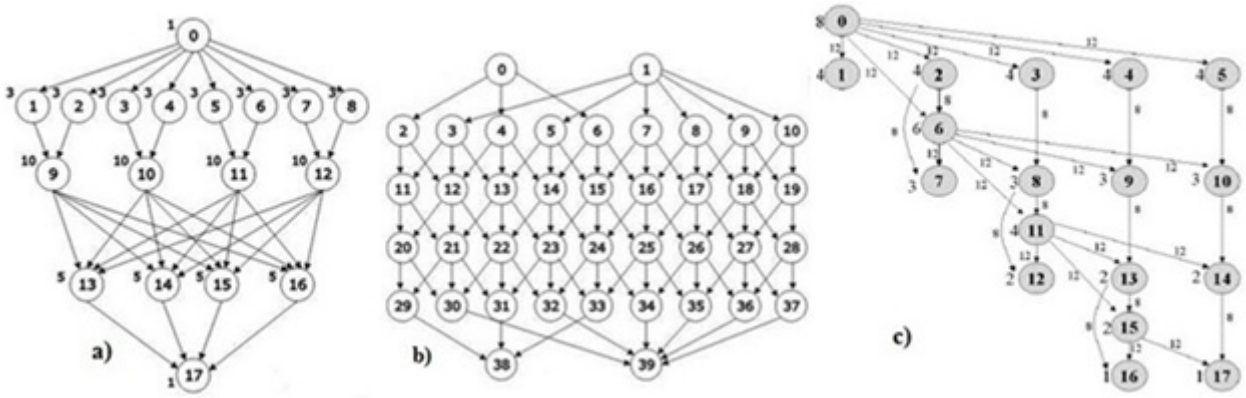

Figura 5: Grafos de programa: (a)*g18*; (b)*g40*; (c)*gauss18*.

As vizinhanças utilizadas para o AC nos experimentos foram raio $R \in 1, 2, 3$. O número de passos do AC S foi definido de acordo com a complexidade do grafo de programa considerado, sendo igual ao número de tarefas para grafos simples como os *out-trees* e igual a 50 para os demais grafos. Outros parâmetros utilizados foram: torneio simples *Tour* $= 3$, taxa de *crossover* $P_{cross} = 100\%$, taxa de mutação $P_{mut} = 3\%$ e conjunto de configurações iniciais aleatórias $CI = 50$. Os parâmetros que apresentaram maior variação de um experimento para outro foram representados através da tupla (tamanho da população T_{pop} e número de gerações G). Foram realizadas 10 execuções para cada experimento e a política de escalonamento adotada em todos eles foi "a tarefa com maior nível dinâmico primeiro". Também foram reproduzidos em todos os experimentos os modelos síncrono e sequencial apresentados em [6] por considerá-los modelos de referência para a vizinhança linear. Assim, nas figuras e tabelas exibidas nesta seção, "Reprod. [6] Seq" está relacionado aos resultados obtidos na reprodução do modelo apresentado em [6] para o modo de atualização sequencial, "Reprod. [6] Sinc" está relacionado aos resultados encontrados para a reprodução do modelo apresentado em [6] utilizando o modo de atualização síncrono e, finalmente, "EACS" está relacionado aos resultados alcançados pelo modelo proposto neste trabalho.

O Experimento 1 foi realizado com o grafo de programa apresentado na Figura 5(a), *g18*. A vizinhança mínima na literatura para resolver este grafo é raio 2 [6] e o tempo ótimo T em um sistema de dois processadores é igual a 46. Para $R = 1$ o modelo EACS encontrou soluções próximas do ótimo, mas utilizando $R = 2$ e ($T_{pop} = G = 100$) o AG rapidamente encontra regras de AC capazes de encontrar T para qualquer CI. Na reprodução de [6], tanto para o modo de atualização sequencial como síncrono o AG também foi capaz de encontrar regras aptas a realizar o escalonamento ótimo, utilizando-se os mesmos parâmetros empregados no EACS. A Figura 6(a) e 7(a) mostram o número de soluções ótimas alcançadas por cada uma das regras no modo de execução do escalonador proposto neste trabalho e dos reproduzidos para 1000 configurações iniciais aleatórias. É importante notar que o modelo sequencial reproduzido a partir de [6] apresenta melhor desempenho que o modelo síncrono também baseado em [6], contudo este obteve desempenho inferior à nova abordagem (EACS).

No Experimento 2, o grafo de programa considerado é o *g40* que pode ser visto na Figura 5(b). A menor vizinhança na literatura para resolver este grafo é raio 2 e o tempo ótimo T em um sistema com dois processadores é igual a 80 [6]. Os parâmetros utilizados foram $R = 2$ e ($T_{pop} = G = 100$). Apesar de utilizarem os mesmos parâmetros, a convergência do modo de aprendizagem para o *g40* é mais demorada que para o *g18*. Entretanto, todos os modelos encontraram regras capazes de realizar o escalonamento ótimo. Assim, avaliou-se o conjunto de regras obtidos através do escalonamento de 1000 CI no modo de execução. O resultado dessa avaliação é apresentado nas Figuras 6(b) e 7(b) que destaca o número de soluções ótimas alcançadas por cada uma das regras de cada modelo. EACS obteve os melhores resultados. Comparando-se os experimentos com os modelos reproduzidos, o modelo sequencial retornou um resultado médio melhor de acordo com os gráfico das Figuras 6(b) e

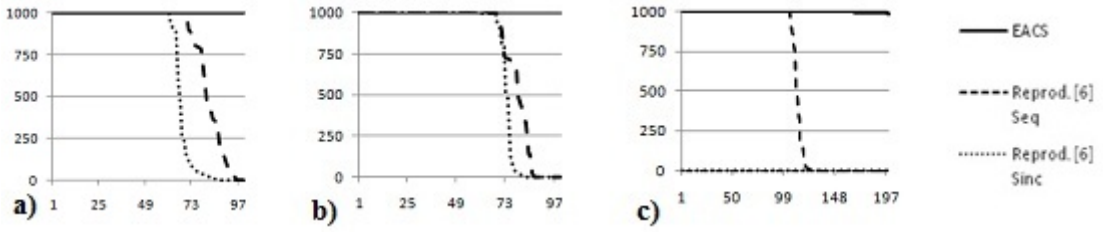

Figura 6: Modo de execução dos escalonadores: (a) *g18*; (b) *g40*; (c)*gauss18*.

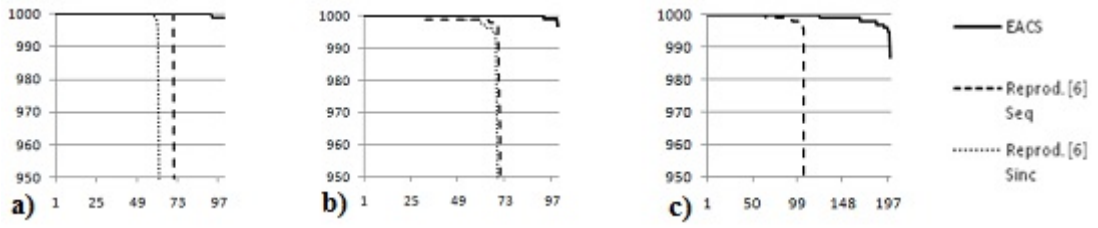

Figura 7: Modo de execução dos escalonadores (detalhamento da escala de 950 a 1000): (a) *g18*; (b) *g40*; (c)*gauss18*.

7(b), por outro lado considerando a convergência do ótimo, o modelo síncrono apresenta um melhor desempenho de acordo com a Tabela 1.

O grafo de programa usado no Experimento 3 é o *gauss18* que é apresentado na Figura 5(c). Ele é considerado mais difícil que os demais para o modo de aprendizagem devido a sua estrutura irregular [6]. O seu tempo ótimo T em um sistema com dois processadores é igual a 44 [5]. Os parâmetros utilizados em "EACS" foram os mesmos dos utilizados em [6] ($R = 3$, $T_{pop} = 200$ e $G = 1000$). Vale destacar também que nenhum dos trabalhos relacionados [5], [6] e [7] foi capaz de encontrar T para este grafo utilizando vizinhança linear e modo de atualização síncrono. Contudo, utilizando modo de atualização sequencial, a vizinhança mínima necessária para alcançar T foi raio 3 [6].

A reprodução de [6] confirma as afirmações do parágrafo anterior, uma vez que na melhor execução, "Reprod. [6] Sinc" encontra 47 para qualquer configuração inicial gerada enquanto que "Reprod. [6] Seq" consegue obter 44. Contudo, diferentemente do modelo síncrono apresentado em [6], "EACS" foi apto a levar qualquer CI gerada para um escalonamento ótimo. As Figuras 6(c) e 7(c) mostram que no modo de execução, "EACS" obteve melhor desempenho que "Reprod. [6] Seq", além de que 61% do seu conjunto de regras foi capaz de encontrar T para as 1000 CI contra $31,5\%$ deste modelo, conforme apresentado na Tabela 1.

A Tabela 1 apresenta um resumo dos resultados alcançados por cada uma das abordagens em cada um dos experimentos realizados. "MA" e "ME" representam os melhores resultados obtidos, respectivamente, no modo de aprendizagem e execução. Entre parênteses, em "ME", é exibido o percentual de regras aptas a escalonar 1000 CI para o tempo ótimo.

Abordagens	Experimento 1 (*g18*)		Experimento 2 (*g40*)		Experimento 3 (*gauss18*)	
	MA	ME	MA	ME	MA	ME
Reprod. [6] Seq	46	46 (70%)	80	80 (31%)	44	44 (31,5%)
Reprod. [6] Sinc	46	46 (58%)	80	80 (49%)	47	47 (0%)
EACS	46	46 (92%)	80	80 (92%)	44	44 (61%)

Tabela 1: Resultados obtidos nos experimentos com os grafos *g18*, *g40* e *gauss18*.

Também foi avaliada a capacidade de reuso de regras em outras instâncias de grafos de programa. O Experimento 4 foi conduzido com o grafo de programa *outtree15*. A vizinhança foi definida por $R = 1$ e os parâmetros utilizados foram ($T_{pop} = G = 30$). Este grafo foi considerado fácil uma vez que o AG rapidamente encontrou regras de AC capazes de encontrar o tempo de resposta ótimo para qualquer CI apresentada. Contudo, o propósito foi escolher aleatoriamente uma regra do conjunto de regras obtido para realizar o escalonamento com os grafos de programa *outtree31*, *outtree63*, *outtree127*, *outtree255*, *outtree511* e *outtree1023*. Não obstante a isso, utilizamos um AG padrão com mesmos parâmetros do AG utilizado no modo de aprendizagem e comparamos seu resultado e tempo computacional ao alcançado por "EACS" no modo de execução. O intuito desta comparação é mostrar que apesar do nosso modelo também fazer uso de um AG, o seu uso ocorre apenas no modo de aprendizagem e seu foco é a busca no espaço de regras do AC enquanto que o AG padrão tem como objetivo encontrar uma solução para determinado grafo de programa. A Tabela 2 apresenta o comparativo para 10 execuções. Assim, "min" e "tempo (ms)" referem-se, respectivamente, ao menor custo de escalonamento encontrado e ao tempo médio, em milisegundos, consumido em cada execução.

Os resultados apresentados na Tabela 2 destacam a principal vantagem do escalonamento baseado em AC, o reuso do conhecimento extraído de um programa paralelo em outros grafos de programa. É importante notar que aumentando o número de

Grafo de Programa	Modo de Aprendizagem		Modo de Execução		AG	
	min	tempo (ms)	min	tempo (ms)	min	tempo (ms)
outtree15	9	17.874	9	0	9	11
outtree31	-	-	17	1	17	43
outtree63	-	-	33	3	33	196
outtree127	-	-	65	11	66	1.084
outtree255	-	-	129	45	140	6.587
outtree511	-	-	257	191	287	53.267
outtree1023	-	-	513	725	602	282.592

Tabela 2: Resultados obtidos pelo EACS e um AG simples para os grafos de programa *outtree15*, *outtree31*, *outtree63*, *outtree127*, *outtree255*, *outtree511* e *outtree1023*.

tarefas, o espaço de busca de possíveis soluções também aumenta e o AG, por não utilizar qualquer conhecimento, precisa realizar novamente para cada aplicação todos os passos evolutivos. Obviamente, estas características são as maiores responsáveis pelo desempenho inferior do AG em relação ao modo de execução do escalonador proposto neste artigo.

7. CONCLUSÕES

Este trabalho apresentou um novo modelo de escalonamento baseado em AC com modo de atualização síncrono para resolver o problema de escalonamento estático de tarefas em multiprocessadores. Esse modelo foi chamado de EACS (Escalonador baseado em Autômato Celular com Atualização Síncrona). O objetivo foi obter um escalonador que explorasse o forte paralelismo inerente aos ACs com atualização síncrona de células e mostrar que esse modo de atualização também pode ser adotado, uma vez que trabalhos anteriores [5], [6] e [7] apesar de destacarem o uso promissor dos ACs com atualização sequencial, desvalorizaram o modo síncrono por apresentar resultados inferiores. Na verdade, uma das principais características dos ACs é a sua adequação à implementação massivamente paralela e um modelo de escalonador que usufrua dessa capacidade deve ser almejado.

Experimentos foram realizados com os grafos de programa encontrados na literatura. Os resultados mostraram que o EACS foi capaz de obter resultados semelhantes aos publicados para modelos de ACs com o modo de atualização sequencial, durante a fase de aprendizagem. Por outro lado, no modo de execução, as regras obtidas pelo EACS apresentaram uma maior convergência para o tempo de escalonamento ótimo. Além disso, o EACS foi capaz de encontrar o tempo ótimo para grafos de programa não resolvidos por modelos síncronos publicados anteriormente. O modo de execução também se mostrou bom no reuso das regras e destacou a maior vantagem do escalonamento baseado em ACs sobre a maioria das outras meta-heurísticas: a capacidade de extração e reuso do conhecimento.

Contudo, apesar dos bons resultados obtidos nos experimentos, é necessário ainda aprimorar o comportamento do nosso modelo sobre outros aspectos tais como o escalonamento de grafos gerados aleatoriamente e o desempenho computacional.

REFERÊNCIAS

[1] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Science, third edition, 2008.

[2] J. D. Ullman. "NP-complete scheduling problems". *Journal of Computer and System Science*, vol. 10, no. 3, pp. 384–393, 1975.

[3] S. Wolfran. "Cryptography with cellular automata". *Advances in Cryptology: Crypto '85 Proceedings*, vol. 218, pp. 429–432, 1986.

[4] M. Sipper. *Evolution of Parallel Cellular Machines, The Cellular Programming Approach*. Springer, 1997.

[5] F. Seredynski and A. Y. Zomaya. "Sequential and Parallel Cellular Automata-Based Scheduling Algorithms". *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 10, pp. 1009–1022, 2002.

[6] A. Swiecicka, F. Seredynski and A. Y. Zomaya. "Multiprocessor Scheduling and Rescheduling with Use of Cellular Automata and Artificial Immune System Support". *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 3, pp. 253–262, 2006.

[7] P. M. Vidica and G. M. B. Oliveira. "Cellular automata-based scheduling: A new approach to improve generalization ability of evolved rules". *Brazilian Symposium on Artificial Neural Networks (SBRN'06)*, 2006.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[9] M. Mitchell, J. P. Crutchfield and R. Das. "Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work". *In Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96)*, 1996.