Publicado nos Anais do XXXII Simpósio Brasileiro de Pesquisa Operacional (XXXII SBPO), 2000, (em CD-ROM), páginas: 295-313, ISBN: 1518-1731

Um Algoritmo GRASP para o Problema de Escalonamento de Tarefas utilizando o modelo LogP

L. S. Vianna, L. M. A. Drummond e Luiz Satoru Ochi
Departamento de Ciência da Computação – Universidade Federal Fluminense
Correspondência: Travessa Capitão Zeferino, 56/808 – Icarai
Niterói – Rio de Janeiro – Brasil – 24220-230
e-mail: leo@pgcc.uff.br, lucia@dcc.ic.uff.br, satoru@dcc.ic.uff.br

Abstract

We propose a metaheuristic GRASP to solve a sheduling multiprocessor tasks model called LogP. In this model is considered the latency and the overhead constraints. Comparisons with heuristic algorithms of the literature attest that our method produces good quality solutions in reasonable times.

Resumo

Propomos neste artigo, um algoritmo baseado em conceitos da metaheurística GRASP para a solução de um modelo de escalonamento de tarefas em múltiplos processadores, denominado LogP. Este modelo considera além da latência, as restrições de sobrecarga. Resultados computacionais mostram a viabilidade de uso do método proposto.

palavras – chave: escalonamento de tarefas, heurísticas, algoritmos adaptativos

1 – Introdução

Metaheurísticas têm se mostrado ferramentas bastante eficientes na solução de vários problemas NP-Árduos. Entre elas, Tabu Search, VNS (*Variable Neighborhood Search*) e GRASP (*Greedy Randomized Adaptive Search Procedure*) têm apresentado bons resultados quando utilizados em problemas de otimização.

Neste trabalho, é proposto um algoritmo GRASP para a solução do problema de escalonamento de tarefas, utilizando a técnica de aglomeração de tarefas. Uma vez que estamos utilizando uma metaheurística para a solução desse problema, o espaço de soluções pode ser explorado mais intensamente e, assim, temos mais chances de escaparmos das armadilhas dos ótimos locais. Como conseqüência, podemos atingir soluções de melhor qualidade do que as obtidas pelas heurísticas convencionais e em tempos de execução satisfatórios.

Na literatura, podemos encontrar vários trabalhos utilizando o modelo de latência [17,18,24] em diversas variações do problema de escalonamento de tarefas. Por outro lado, poucos trabalhos são encontrados utilizando o modelo LogP [3,4,14,25] uma vez que esse modelo foi proposto recentemente. Quanto ao uso de metaheurísticas para a solução desse problema, poucos trabalhos são encontrados e eles geralmente consideram apenas o modelo de latência.

Para avaliar a performance do algoritmo proposto, os resultados obtidos foram comparados com a heurística *HAL*, proposto por Boeres [4], que, atualmente, possui os melhores resultados da literatura.

Publicado nos Anais do XXXII Simpósio Brasileiro de Pesquisa Operacional (XXXII SBPO), 2000, (em CD-ROM), páginas: 295-313, ISBN: 1518-1731

2 – O Problema de Escalonamento de Tarefas

Este trabalho se concentra no problema de escalonamento estático de um conjunto de tarefas não-preemptivas de uma aplicação paralela em um conjunto de processadores (homogêneos e totalmente conectados), utilizando o modelo LogP. Cada um desses processadores possui sua própria memória local e a comunicação entre eles é feita exclusivamente através de troca de mensagens, o que caracteriza um sistema distribuído. Por tarefas não-preemptivas, podemos entender que a execução de cada tarefa não pode ser interrompida por nenhum outro evento, ou seja, uma vez iniciada a execução de uma tarefa, o processador só será liberado ao terminar tal execução.

Dada uma aplicação paralela a ser executada num sistema com p processadores, o objetivo do problema aqui apresentado é encontrar um escalonamento que *minimize* o tempo total de execução (*makespan*) da aplicação. Este escalonamento é feito de maneira que todas as relações de precedência entre as tarefas sejam respeitadas. Além disso, uma tarefa v_i só pode ser escalonada se todos os seus *predecessores imediatos* já tiverem sido e todos os dados necessários para a sua execução já estiverem disponíveis no processador p_j onde v_i será escalonada [7].

Para a solução deste problema, as características da aplicação e da arquitetura utilizada devem ser bem definidas. Tais características são especificadas pelo *modelo de escalonamento* que é composto pelo *modelo de aplicação* e pelo *modelo arquitetural*, descritos a seguir.

2.1 – Modelo de Aplicação

Uma aplicação paralela pode ser representada por um Grafo Acíclico Direcionado (*GAD*) denotado por $G = (V,E,\varepsilon,\omega)$, onde V é o conjunto de n nós do grafo e E o conjunto de arcos. Cada nó $v \in V$ representa uma tarefa com tempo de execução $\varepsilon(v)$ e cada arco $(u,v) \in E$ representa a restrição de precedência entre as tarefas u e v, ou seja, a tarefa u deve completar sua execução antes que a tarefa v comece a sua. Um peso $\omega(u,v)$ pode estar associado ao arco (u,v), representando a quantidade de dados a serem enviados de u para v.

Uma tarefa consiste numa unidade de computação indivisível que pode ser uma instrução, uma subrotina ou um programa inteiro.

O conjunto das tarefas predecessoras imediatas à tarefa $v \in V$ é denotado por $pred\ (v) = \{u \mid (u,v) \in E\}$ enquanto que o conjunto de seus sucessores imediatos é dado por $succ\ (v) = \{z \mid (v,z) \in E\}$.

2.2 – Modelo Arquitetural

O modelo arquitetural define as características da arquitetura paralela a ser considerada. Neste trabalho, considera-se um computador paralelo com memória distribuída, sendo cada processador associado à sua própria memória local. Os processadores se comunicam exclusivamente através de troca de mensagens e estão interconectados conforme alguma topologia. Além disso, considera-se que os processadores estão totalmente conectados entre si e são homogêneos, isto é, possuem as mesmas características.

Atualmente, existe uma grande variedade de máquinas paralelas em uso. Tais máquinas possuem características particulares que as diferem uma das outras, como por exemplo, o tipo de acesso à memória ou o tipo de interconexão entre os processadores. O ideal ao se criar uma aplicação paralela é que o programador não se preocupe com detalhes da máquina e, para isso, é importante a criação de um *modelo de computação paralela* suficientemente abstrato para que detalhes da máquina sejam ignorados, mas que seja, ao mesmo tempo, versátil para permitir que a estrutura computacional do programa possa ser mapeada eficientemente para uma grande variedade de plataformas paralelas.

O primeiro modelo de computação paralela definido foi o *PRAM* [10]. Neste modelo assume-se que o número de processadores é ilimitado e que eles trabalham de forma síncrona. Além disso, a comunicação entre os processadores pode ser considerada nula já que eles se comunicam através de uma memória compartilhada [21]. Outra característica deste modelo é o fato de permitir que vários processadores acessem simultaneamente a memória, sem que esta se torne um gargalo, nem quando o número de processadores que a compartilham for muito elevado. Devido a essas características, o modelo *PRAM* não é considerado realístico e, portanto, algoritmos desenvolvidos para este modelo possuem na maioria dos casos desempenhos ruins quando são assumidas características que representam máquinas paralelas reais [8].

Com o desenvolvimento das máquinas paralelas distribuídas, fez-se necessário a criação de modelos de computação paralela que representassem estas máquinas de forma mais realística e que definissem as características de comunicação com mais precisão.

Neste cenário, surgiu o modelo de latência, onde o único parâmetro de comunicação considerado é a latência (*delay*), que consiste no tempo de transferência de uma unidade de dado entre dois processadores distintos [18].

Um outro modelo, proposto recentemente, é o modelo LogP onde, além da latência, também são considerados outros importantes parâmetros de comunicação que tornam este modelo o mais realístico atualmente.

A seguir, são descritos mais detalhadamente os modelos de latência e LogP, que consistem nos modelos de computação paralela mais considerados na área de escalonamento de tarefas e que são estudados neste trabalho.

→ Modelo de Latência

Neste modelo, o único parâmetro arquitetural associado ao custo de comunicação é a latência, denotada por τ. Assume-se que um processador não gasta tempo preparando o envio nem o recebimento de mensagens permitindo, dessa forma, que comunicação e computação se sobreponham totalmente. Uma vez que um processador não gasta tempo preparando o envio de mensagens, ele pode enviar simultaneamente várias mensagens distintas para vários destinos (*multicast*), como ilustrado na Figura 1(*b*), onde a tarefa v_0 envia, simultaneamente, mensagens para as tarefas v_1 e v_3. Como podemos observar, o envio de tais mensagens se sobrepõe com a execução da tarefa v_2.

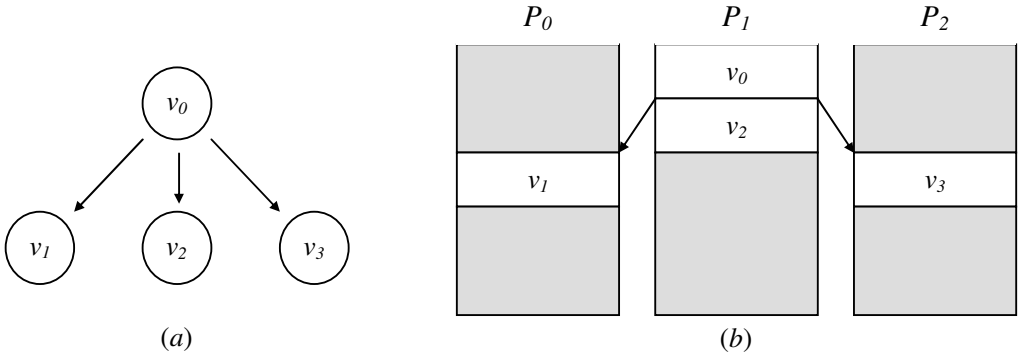

Figura 1: (a) Grafo do tipo fork. (b) Possível escalonamento utilizando o modelo de latência

→ Modelo LogP

O modelo LogP representa o fato de que nas máquinas paralelas atuais, o processador deve tratar ou ao menos iniciar cada comunicação e que, além disso, comunicação e

computação não podem ser sobrepostas totalmente [8]. O próprio nome dado ao modelo já define os parâmetros considerados, ou seja:

- L – latência de comunicação.
- o – sobrecarga (*overhead*), que é o tempo durante o qual o processador permanece preparando o recebimento (*sobrecarga de recebimento*) ou o envio (*sobrecarga de envio*) de uma mensagem, tempo este durante o qual o processador não pode realizar outras operações.
- g – *gap* que é o intervalo mínimo permitido entre dois envios ou dois recebimentos consecutivos em um mesmo processador. Este parâmetro é uma característica da rede e está associado à capacidade do canal de saída do processador.
- P – conjunto dos processadores disponíveis.

Sempre que uma tarefa $v_i \in V$ for escalonada num processador $p_j \in P$ no qual um de seus predecessores imediatos não se encontra, duas tarefas extras deverão ser escalonadas: uma tarefa de envio e uma de recebimento, como ilustrado na Figura 2. Uma tarefa de envio deve ser escalonada imediatamente após cada predecessor de v_i que não se encontra em p_j. Para cada uma dessas tarefas de envio, deve estar associada uma tarefa de recebimento que será escalonada em p_j antes de v_i. Tais tarefas representam a sobrecarga gasta pelo processador para o envio ou o recebimento de uma mensagem. Alguns autores preferem utilizar os termos *tempo de empacotamento* e *tempo de desempacotamento* para se referirem ao tempo de preparação do envio e do recebimento de uma mensagem, respectivamente.

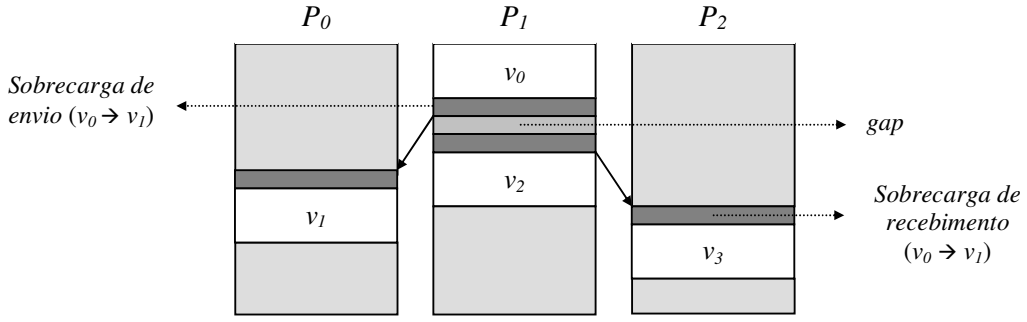

Figura 2: Possível escalonamento para a aplicação da Figura 1(a), utilizando o modelo LogP

Como podemos ver, esse modelo é mais realístico do que o modelo de latência pelo fato de também considerar outros importantes parâmetros de comunicação. Para que os resultados obtidos pelo algoritmo aqui proposto tenham maior precisão, as sobrecargas de envio e de recebimento serão tratadas separadamente como em [5], sendo denotadas, respectivamente, por λ_s e λ_r, e $o = (\lambda_s + \lambda_r)/2$.

Podemos verificar que o modelo de latência consiste numa particularização do modelo LogP quando $o = g = 0$.

2.3 – Heurísticas de Escalonamento

Uma grande variedade de heurísticas de escalonamento foram propostas para vários modelos de aplicação e de arquitetura. Estas heurísticas podem ser classificadas em duas classes: heurísticas de construção como, por exemplo, os algoritmos baseados em *list scheduling* [1], aglomeração de tarefas [20], análise de caminho crítico do grafo [15] e particionamento de grafos [13] que constróem um único escalonamento; e heurísticas de construção e busca (metaheurísticas) onde vários escalonamentos são gerados a procura de uma boa solução [19].

Uma abordagem clássica é a classe de heurísticas do tipo *list scheduling*, onde a idéia básica é atribuir prioridades às tarefas criando, assim, uma lista ordenada. O escalonamento é

feito selecionando a tarefa livre de mais alta prioridade da lista e um processador ocioso para alocar esta tarefa. Uma tarefa é considerada livre se todos os seus predecessores imediatos já foram escalonados. A maior diferença entre os algoritmos baseados nesta estratégia é a prioridade usada para se escolher a próxima tarefa livre a ser escalonada, que pode ser, por exemplo, o nível, o co-nível ou o número de sucessores da tarefa. Um dos problemas principais desta estratégia é que as prioridades nem sempre representam a real importância da tarefa naquele momento [15] e, além disso, as decisões se baseiam nas informações disponíveis em cada iteração. Exemplos de estratégias baseadas em *list scheduling* podem ser encontradas em [1,6,22].

Nos algoritmos baseados em aglomeração de tarefas, uma coleção de tarefas ou *cluster* consiste em um conjunto de tarefas que devem ser executadas em um mesmo processador. As tarefas são aglomeradas em uma coleção visando minimizar o tempo de execução paralelo da aplicação através da minimização do custo de comunicação e do tempo total de computação gasto por um processador. Isto é possível pois a comunicação entre as tarefas pertencentes a uma mesma coleção tem custo nulo. Exemplos de algoritmos de aglomeração podem ser encontrados em [11,17,18,20].

Heurísticas baseadas na análise de caminho crítico tentam diminuir o caminho mais longo do grafo removendo a comunicação existente entre as tarefas deste caminho. Isto é feito alocando estas tarefas em um mesmo processador [13]. Exemplos de heurísticas baseadas nesta abordagem podem ser encontrados em [15].

A abordagem que utiliza particionamento de grafos tem como objetivo particionar o grafo de tarefas em múltiplas partições, de forma que as arestas conectando vértices que pertençam a diferentes partições sejam minimizados. Uma heurística que utiliza esta técnica é encontrada em [13].

Uma metaheurística por outro lado, pode criar um escalonamento distinto a cada iteração do algoritmo e a solução ser melhorada a cada passo. Existem metaheurísticas que criam várias soluções simultâneas que evoluem iterativamente buscando melhorias [19,23]. Metaheurísticas possuem como características estruturas com uma menor rigidez que as dos métodos exatos tradicionais, proporcionando mecanismos que tentam escapar das armadilhas dos ótimos locais. É comum que algoritmos baseados em metaheurísticas usem uma heurística de construção para gerar o primeiro escalonamento e, a partir daí, melhora-se a solução através de busca local.

Neste trabalho, propomos uma metaheurística uma vez que essas ferramentas nos permitem explorar mais intensamente o espaço de busca. A metaheurística utilizada foi do tipo GRASP (*Greedy Randomized Adaptive Search Procedure*) e a heurística de construção adotada utiliza a abordagem de aglomeração de tarefas, visto que esta tem apresentado resultados melhores que as heurísticas baseadas em outras abordagens como, por exemplo, *list scheduling*.

3 – Aglomeração de Tarefas

Heurísticas de aglomeração possuem objetivos a serem alcançados através de uma seqüência de passos. No primeiro passo, assume-se que cada tarefa é uma coleção; depois, a cada passo, executa-se um mecanismo de refinamento que consiste basicamente na junção de coleções de acordo com alguma função de custo; no último passo a aglomeração final é produzida [11]. De uma forma geral, os algoritmos de aglomeração seguem as seguintes etapas propostas por Sarkar [20]:

1) Aglomeração de tarefas em coleções considerando um número ilimitado de processadores.
2) Unificação e escalonamento das coleções de tarefas considerando que o número de processadores pode ser menor que o número de coleções de tarefas produzidas na etapa 1.

É importante ressaltar a existência de um mecanismo vantajoso ao se escalonar tarefas, especialmente quando os custos de comunicação são altos: a *replicação de tarefas*. Para que se

minimize o custo de comunicação, uma tarefa pode ser copiada em vários processadores, sendo essas cópias independentes entre si. Em arquiteturas com alto custo de comunicação, os algoritmos de escalonamento com replicação de tarefas geralmente produzem resultados com tempo de execução menores que os algoritmos que não consideram tal mecanismo [12].

A seguir, apresentamos um exemplo de como a replicação de tarefas pode ser benéfica. Seja o grafo apresentado na Figura 3, com tempo de computação das tarefas e de comunicação unitários, $\lambda_s = \lambda_r = 0$ (modelo de latência) e $\tau = 1$.

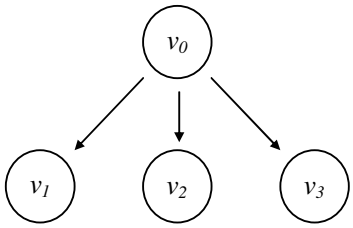

Figura 3: grafo do tipo fork

Escalonando este grafo de tarefas em um conjunto de processadores, observa-se o seguinte: na Figura 4(*a*), o tempo total paralelo (*makespan*) gasto é de 3 unidades de tempo, utilizando um escalonamento sem replicação de tarefas. Por outro lado, na Figura 4(*b*), onde é utilizado o mecanismo de replicação, são gastas apenas 2 unidades de tempo. Isto ocorre pois, no primeiro caso, a tarefa v_3 tem que aguardar até que a mensagem vinda da tarefa v_0 chegue até ela. No segundo caso isto não acontece, pois existe uma cópia de v_0 no mesmo processador onde está a tarefa v_3. Logo, v_3 espera apenas o término de v_0 para começar a executar.

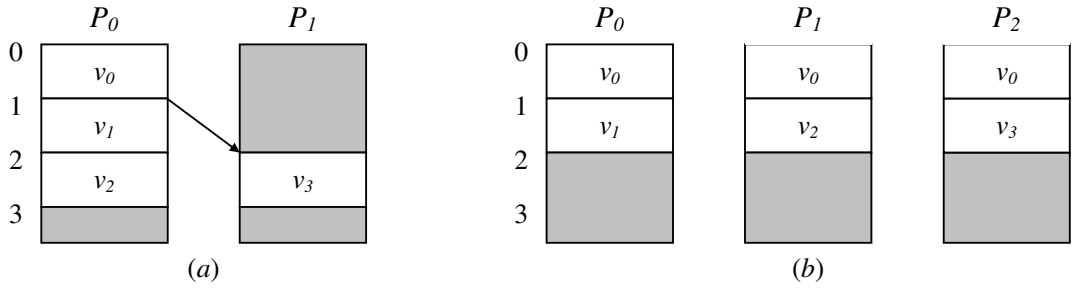

Figura 4: (a) Escalonamento sem replicação. (b) Escalonamento com replicação.

Observa-se que quanto maior o tempo gasto com comunicação, melhor pode vir a ser o escalonamento produzido quando se utiliza replicação de tarefas. Uma desvantagem em relação à replicação é que cada vez que é feita uma cópia de uma tarefa, mais espaço em memória é necessário.

3.1 – Construindo um Algoritmo de Aglomeração

A qualidade dos resultados produzidos por um algoritmo de aglomeração, tanto para o modelo de latência como para o modelo LogP, depende diretamente de como são implementados os seguintes pontos básicos:

1) Cálculo do tempo de execução paralelo.
2) Ordenação das tarefas dentro da coleção.
3) Condição de parada do algoritmo.
4) Escolha da próxima tarefa candidata a entrar em uma coleção.

O primeiro item se refere ao cálculo do tempo de execução paralelo de uma coleção e, para isto, deve ser identificada a função custo a ser usada pelo algoritmo para este cálculo. Uma

maneira de se determinar este tempo é encontrando o maior caminho para se chegar à tarefa dona de uma coleção (*caminho crítico*). Na verdade, ao se calcular o tempo paralelo de uma coleção $C(v)$, existem apenas dois tipos de caminhos a serem considerados: um formado pelos custos de computação das tarefas dentro de $C(v)$, o qual será chamado *caminho-coleção* e o outro tipo formado pelos caminhos existentes de cada predecessor imediato de $C(v)$ (não pertencente a $C(v)$) até v, que será chamado *caminho-predecessor*. O maior caminho dentre os *caminhos-predecessores* será chamado de *caminho-predecessor-crítico*.

A ordem em que as tarefas são executadas dentro de uma coleção tem uma grande influência nos custos dos caminhos descritos anteriormente; por isso, é necessária uma boa ordenação das tarefas presentes numa mesma coleção. Uma abordagem que é utilizada pela maioria das heurísticas de aglomeração [17,24] por produzir resultados satisfatórios é a ordenação de tarefas pela ordem crescente do tempo mais cedo de execução.

Outro ponto importante a ser analisado é a condição de parada de criação de uma coleção $C(v)$. A escolha desta condição é importante para que o algoritmo não pare prematuramente e nem continue executando além do necessário. Normalmente, esta condição pode ser definida como sendo a comparação do tempo paralelo de $C(v)$ antes e depois da inclusão de uma tarefa. Usando este critério de parada, considera-se que uma coleção está completa quando o seu tempo paralelo não puder mais ser reduzido.

Um bom critério de escolha de tarefas candidatas é selecionar a tarefa que pertença ao *caminho crítico*, visando desta forma diminuir o tempo gasto para se chegar até a tarefa dona da coleção. Um problema com este critério é que nem sempre a inclusão da tarefa candidata produzirá uma função decrescente do tempo de execução paralelo, o que faz com que algoritmos que usam como condição de parada a comparação do tempo de execução paralelo de uma coleção $C(v)$ antes e depois da inclusão de uma tarefa candidata possam terminar prematuramente. Esta situação pode ocorrer, por exemplo, ao se incluir uma tarefa u em uma coleção $C(v)$ e um novo arco incidente (w,u) que seja muito custoso, passe a contribuir para o cálculo do novo tempo paralelo de $C(v)$. Assim, se o tempo paralelo for muito maior, o algoritmo vai parar e considerar que não vale a pena que u pertença a $C(v)$. Isto pode ser apenas um mínimo local, pois a inclusão da tarefa u e mais tarde da tarefa w poderia levar a uma solução melhor. Logo, se a tarefa escolhida não é sempre a melhor, são necessárias outras condições de parada que permitam que o algoritmo chegue a resultados bastante razoáveis.

Um problema que deve ser analisado separadamente para os modelos de latência e LogP é o efeito da inclusão de uma tarefa em uma coleção. Analisando este ponto, fica mais fácil de se escolher uma condição de parada para o algoritmo de forma que este não termine prematuramente, como descrito anteriormente.

→ Modelo de Latência

Neste modelo, a inclusão de tarefas tem dois efeitos: o *caminho crítico* pode passar a ser o *caminho-coleção* ou o novo *caminho crítico* pode ser proveniente de um dos arcos incidentes à coleção, ou seja, *caminho-predecessor-crítico.* O tempo paralelo de uma coleção só pode ser reduzido se o custo do *caminho crítico* for diminuído; se o *caminho crítico* é um *caminho-predecessor*, então deve se incluir na coleção o respectivo predecessor imediato. Por outro lado, o custo do *caminho-coleção* nunca diminui, já que a cada passo uma nova tarefa é incluída neste caminho[4].

Considerando a análise acima, o algoritmo pode prosseguir enquanto:

- o *caminho-coleção* não for o caminho crítico (*Caminho-Coleção Não Crítico - CCNC*).
- a inclusão da tarefa candidata não aumentar o tempo paralelo da coleção (*Vale a Pena Inserir Tarefa – VPIT*).

→ Modelo LogP

No modelo LogP, o *caminho-coleção* poderá diminuir. Isto ocorre pois as sobrecargas geradas pela comunicação devem ser tratadas como tarefas, já que elas são custos designados aos processadores, mesmo sendo estes valores associados aos *caminhos-predecessores*. Considerando o exemplo apresentado na Figura 5, ao incluir uma tarefa *u* em uma coleção *C*(*v*), a sobrecarga de recebimento (λ_r) referente à mensagem enviada por *u* antes da sua inclusão será substituída pela própria tarefa *u*. Logo, o custo do caminho-coleção aumentará após a inclusão de *u* se a sobrecarga de recebimento for menor que o tempo de computação de *u* e diminuirá caso contrário.

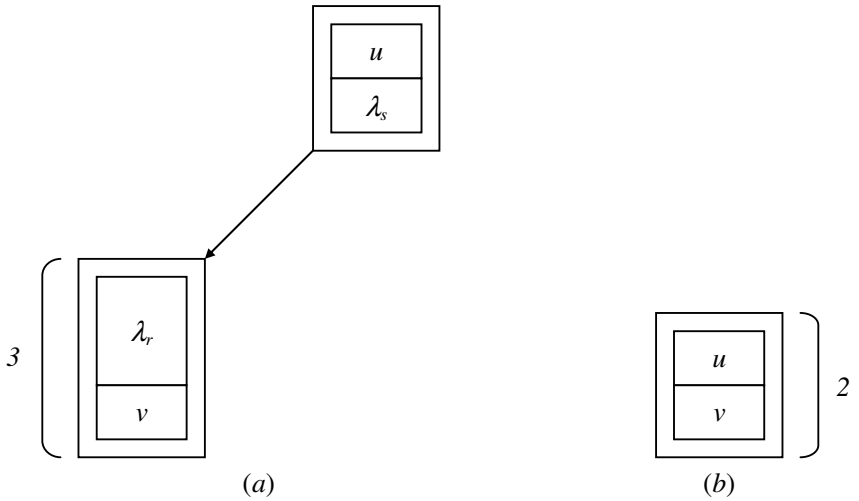

Figura 5: (a) o caminho-coleção de C(v) tem custo 3. (b) após a inclusão de u elimina-se o tempo gasto com a sobrecarga de recebimento e o caminho-coleção passar a ter custo 2. Tempos de comunicação, computação e τ unitários, $\lambda_s = 1$, $\lambda_r = 2$.

Sendo sobrecarga de comunicação uma tarefa designada ao processador, um *caminho-predecessor* pode afetar não somente o *caminho-coleção*, como também outros *caminhos-predecessores* da coleção. Logo, a inclusão de tarefas que pertençam a caminhos não críticos podem diminuir o tempo de execução paralelo de uma coleção, assim como pode ser proveitoso continuar com o algoritmo além da condição que detecta se o *caminho-coleção* é o *caminho-crítico*.

Como se pode observar, os quatro pontos básicos identificados são dependentes entre si. Como uma função de custo influencia cada um dos pontos básicos do algoritmo, um aspecto importante pode ser a escolha de diferentes funções para a implementação destes pontos. A implementação de cada ponto afeta diretamente no desempenho dos outros e, consequentemente, no desempenho geral do algoritmo e na qualidade de suas soluções.

4 – Algoritmo *Proposto*

O algoritmo proposto neste trabalho tenta acoplar o conceito de metaheurísticas ao algoritmo *HAL* (Heurística de Aglomeração para modelos LogP), proposto por Boeres [4], que, atualmente, possui os melhores resultados da literatura. Os resultados obtidos serão comparados com os apresentados pelo algoritmo *HAL* a fim de estudarmos a viabilidade do uso de metaheurísticas em problemas de escalonamento de tarefas uma vez que, na literatura, poucos trabalhos podem ser encontrados utilizando tais ferramentas.

HAL é baseado em aglomeração de tarefas e, ainda, explora a técnica de replicação, visto que esta pode produzir resultados melhores em arquiteturas com alto custo de comunicação. A idéia é propor uma heurística de escalonamento de tarefas onde várias características de comunicação são relevantes, não somente a latência, mas também as

sobrecargas de envio e recebimento de mensagens. As sobrecargas são características muito importantes, pois como foi constatado em [16], a maioria das aplicações paralelas são mais sensíveis a elas do que à latência. Logo, este é um parâmetro que não deve ser ignorado.

Algoritmos de aglomeração têm como objetivo achar um escalonamento ótimo para um *GAD* através da construção de coleções $C(v)$, de tal forma que v possa começar a executar em um tempo mais cedo possível. Nestes algoritmos, uma coleção $C(v)$ é construída para cada tarefa $v \in V$, tal que $C(v)$ conterá a tarefa v (dona da coleção) e cópias de algumas tarefas predecessoras de v. A replicação de tarefas é feita de uma maneira bem simples. Para cada $v \in V$, uma nova coleção $C(v) = \langle u_1,u_2,...,u_n,v\rangle$ é formada e qualquer tarefa $u_i \in C(v)$ é considerada, na verdade, uma cópia. As únicas tarefas que não são consideradas cópias são as tarefas donas das coleções, logo u_i só não será uma cópia na coleção $C(u_i)$.

O algoritmo *HAL* é constituído de dois estágios:

1) Criação de coleções determinando quais tarefas devem ser inseridas e a ordenação entre elas dentro da coleção.
2) Identificação das coleções necessárias para o escalonamento e execução do grafo de entrada, visto que com a replicação de tarefas nem todas as coleções são sempre necessárias.

4.1 – Primeira Fase do Algoritmo

Em uma coleção de tarefas $C(v)$ são agrupadas as tarefas predecessoras de v que permitem que v execute mais cedo. A intenção é encontrar o tempo mais cedo de escalonamento ($e_s(v)$) das tarefas donas das coleções ou, pelo menos, uma aproximação $s(v)$ deste valor.

Para se formar a coleção $C(v)$ para cada tarefa $v \in V$, as tarefas predecessoras de v são visitadas. Cada visita decide se uma tarefa w_i predecessora de v deve ser incluída em $C(v)$ ou não. Em princípio, dada uma perfeita implementação dos quatro pontos básicos para a construção de um algoritmo de aglomeração, uma tarefa w_i só deve ser incluída em $C(v)$ se isto permitir que v possa começar a executar mais cedo; caso contrário, decide-se que w_i não deve fazer parte de $C(v)$.

Os quatro pontos básicos implementados pelo algoritmo *HAL* são descritos a seguir e a estrutura do *HAL* é apresentada no Algoritmo 1.

4.1.1 – Cálculo do Tempo de Execução Paralelo de uma Coleção

O tempo de execução paralelo de uma coleção $C(v)$ é definido por $s(v) + \varepsilon(v)$; o valor $s(v)$ de uma tarefa v será o custo máximo entre o *caminho-predecessor-crítico* e o *caminho-coleção* (linha 12 do Algoritmo 1). O custo associado ao *caminho-coleção* representa todo o custo de computação relacionado a uma coleção $C(v) = \langle u_1,u_2,...,u_n,v\rangle$ e é calculado somando todo o tempo de computação que pode ser gasto pelo processador: os custos de computação das tarefas até a tarefa dona da coleção ($\varepsilon(u_1)$ até $\varepsilon(u_n)$) e todas as sobrecargas de recebimento de mensagens (linha 9). Uma restrição presente no algoritmo é que apenas tarefas donas de coleção podem enviar mensagens para outras coleções, minimizando, assim, atrasos que possam comprometer o valor $s(v)$.

4.1.2 – Ordenação das Tarefas dentro de uma Coleção

Uma boa ordenação de tarefas dentro de uma coleção é crucial pois ela contribui diretamente no tempo de escalonamento produzido pelo algoritmo. Na tentativa de melhorar os resultados obtidos, uma nova heurística para a ordenação de tarefas, chamada *ordenação sucessor*, foi adotada.

A maioria das heurísticas adotam a ordenação em função do valor $s(\ )$, ou seja, todas as tarefas u_i contidas em uma coleção estariam ordenadas não decrescentemente pelo seu valor

$s(u_i)$. Os resultados obtidos eram satisfatórios; porém, em alguns tipos de grafos onde o número de tarefas paralelas aglomeradas em uma coleção é grande, observou-se que uma outra ordenação de tarefas pode levar a soluções melhores.

A modificação proposta foi a seguinte: sempre que se inserir uma nova tarefa em $C(v)$, deve-se incluí-la imediatamente antes de seu primeiro sucessor imediato já pertencente à coleção. Com isso, toda vez que se inserir uma nova tarefa u_i, atrasa-se a execução de u_i ao máximo desde que não se aumente o valor do *caminho-coleção* para $C(v)$. Assim, é permitido que u_i seja escalonada em tempo suficiente para receber mensagens, não atrasando desnecessariamente a computação de outras tarefas paralelas que tenham o seu valor s maior do que $s(u_i)$.

4.1.3 – Condição de Parada

Se o custo do *caminho-predecessor-crítico*, *Custo*, for maior que o custo do *caminho-coleção*, *CColeção*, o algoritmo prossegue e então *Tarefa* é inserida em $C(v)$. Esta condição verifica se o *caminho crítico* não é o *caminho-coleção* (condição CCNC) (linha 6 do Algoritmo 1). Para saber se *Tarefa* deve permanecer em $C(v)$ ou não, compara-se o custo do *caminho-predecessor-crítico* com o custo do *caminho-coleção* de $C(v) \cup \{Tarefa\}$ (condição VPIT) (linha 10).

Observa-se que estas condições são menos rígidas do que a comparação para saber se o tempo paralelo de execução de uma coleção sempre diminui. Assim é, permitido que o algoritmo prossiga mesmo que o tempo paralelo aumente. Desta forma, é possível se escapar de alguns mínimos locais enquanto o *caminho-coleção* não for crítico.

4.1.4 – Tarefa Candidata

A tarefa candidata é a tarefa da qual o *caminho-predecessor-crítico* é proveniente. O que se deseja com isso é diminuir o caminho crítico e, assim, reduzir o valor $s(v)$. No caso de existir mais de uma tarefa w_i candidata a ser incluída em $C(v)$, o critério de desempate adotado consiste em priorizar a tarefa com menor *nível* para ser incluída na coleção. O nível de uma tarefa w_i é dado pelo caminho mais longo desde w_i até um nó de saída do grafo. Desta forma, quando as coleções estão sendo formadas, a preferência é dada aos predecessores mais próximos da tarefa dona da coleção.

Em linhas gerais, o Algoritmo 1 implementa a construção de $C(v)$ e o cálculo de $s(v)$, dada uma tarefa $v \in V$.

No modelo LogP, pode ser que o tempo paralelo ainda diminua depois que o *caminho-coleção* se torne crítico. Isto ocorre com a inclusão de alguns predecessores da coleção e, como conseqüência, a retirada de sobrecargas de recebimento relativas às mensagens provenientes destes predecessores. Exemplos de predecessores candidatos para os quais isto seria viável seriam as tarefas que possuem tempo de computação menor que λ_r, sendo estas tarefas origens ou tarefas cujos seus predecessores já estejam na coleção. Para contornar este problemas foi criado um procedimento chamado *Otimiza_Coleção* (linha 15). Tal procedimento não precisa ser usado para o modelo de latência visto que, neste caso, o valor do *caminho-coleção* nunca diminui.

Algoritmo 1: *calcula s(v);*

```
1   se pred(v) = ∅ então s(v) ← 0;
2   senão
3       Calcula_Caminho_Pred_Crítico (C(v), Tarefa, Custo);
4       CColeção ← 0;
5       s(v) ← max {CColeção, Custo};
6       Enquanto (CColeção < Custo) faça
7           C(v) ← C(v) ∪ {Tarefa};
8           CColeção' ← CColeção;
9           CColeção ← Custo_Coleção (C(v));
10          se (CColeção ≤ Custo) então
11              Calcula_Caminho_Pred_Crítico (C(v), Tarefa, Custo);
12              s(v) ← max {CColeção, Custo};
            senão
13              s(v) ← max {CColeção', Custo};
14              C(v) ← C(v) - {Tarefa};
        fim-Enquanto
15      Se LogP então (s(v),C(v)) ← Otimiza_Coleção (C(v));
    fim-se
```

4.2 – Segunda Fase do Algoritmo

Ao final da primeira fase do *HAL*, para cada tarefa $v \in V$, uma coleção $C(v)$ foi produzida. No entanto, devido à replicação de tarefas, nem todas as coleções são realmente necessárias. Dessa forma, na segunda fase, são determinadas quais das coleções geradas farão parte do escalonamento final. A seleção de coleções é feita da seguinte maneira: inicialmente, são selecionadas todas as coleções $C(v)$ onde v é tarefa destino do grafo. A seguir, para toda tarefa u contida em uma coleção $C(v)$ já selecionada, verifica se os seus predecessores w também estão contidos em $C(v)$. Caso não estejam contidos, a coleção cuja a tarefa w é dona também é selecionada.

5 – GRASP

A metaheurística *Greedy Randomized Adaptive Search Procedures* (GRASP) [9] é uma técnica iterativa de amostragem randômica, onde em cada iteração obtém-se uma solução para o problema. Cada iteração consiste de duas fases: construção e busca local. Na primeira, uma solução viável é construída, cuja vizinhança será explorada na fase de busca local. A melhor solução entre todas as iterações é guardada como resultado final.

Na fase de construção, uma solução viável é construída elemento a elemento. Cada elemento a ser incluído na solução é escolhido utilizando-se uma função gulosa adaptativa, que estima o benefício de cada elemento e cria uma lista restrita de candidatos. Seleciona-se então, aleatoriamente, um elemento da lista. A função é adaptativa em razão da lista restrita de candidatos ser recriada a cada iteração da fase de construção, a fim de refletir as mudanças associadas à escolha do elemento anterior. A componente probabilística do GRASP é, portanto, caracterizada pela escolha randômica de um dos melhores candidatos da lista, que não é necessariamente o melhor deles. Desta forma, é possível obter uma grande diversidade de soluções viáveis ao longo das iterações.

A solução gerada pelo GRASP na fase de construção não é necessariamente um ótimo local, mesmo levando-se em consideração definições simples de vizinhança. Em conseqüência, é quase sempre vantajoso aplicar um procedimento de busca local a fim de melhorar cada solução encontrada na fase de construção. Algoritmos de busca local funcionam de maneira iterativa, trocando sucessivamente a solução corrente por uma solução melhor, encontrada ao se explorar sua vizinhança. A busca local termina ao atingir um ótimo local, quando não encontra melhores soluções na vizinhança da solução corrente. O sucesso do algoritmo de busca depende da escolha de uma vizinhança adequada, das técnicas de busca em vizinhança utilizadas e da solução inicial. Com respeito a este último ponto, a fase de construção do GRASP possui um papel importante, uma vez que produz boas soluções iniciais para o procedimento de busca local.

Assim, GRASP pode ser visto como uma metaheurística que se utiliza de boas características tanto de algoritmos gulosos (soluções de qualidade) como de procedimentos construtivos randomizados (diversificação). O Algoritmo 2 ilustra uma implementação GRASP genérica. Como parâmetros de entrada, GRASP utiliza o tamanho máximo da lista restrita de candidatos, o número de iterações e a semente para o gerador de números aleatórios. Após ler os dados de entrada na linha 1 e fazer as inicializações na linha 2, as iterações do GRASP são realizadas no *loop* que vai da linha 3 à linha 7. Cada iteração consiste da fase de construção (linha 4), da fase de busca local (linha 5) e, se houver necessidade, da atualização da melhor solução encontrada (linha 6).

Algoritmo 2: *GRASP* (*TamanhoLista, NumIterações, Semente*);

```
1  Ler dados de entrada
2  Inicializar o custo da melhor solução, Custo (S*) ← ∞;
3  Para i = 1, 2, ..., NumIterações faça
4      Construir uma solução S de forma randomicamente gulosa
5      Aplicar um procedimento de busca local em S retornando S'
6      se Custo (S') < Custo (S*) então S* ← S'
7  fim-Para
8  Retornar a melhor solução S*
```

6 – Algoritmo Proposto

Os resultados obtidos pelo algoritmo *HAL* não são necessariamente ótimos, visto que estamos lidando com uma heurística. Em vista disso, propomos, neste trabalho, um algoritmo GRASP acoplado ao *HAL* mas sem utilizar as suas fases de refinamento a fim de avaliarmos a viabilidade do uso de metaheurísticas na solução do problema de escalonamento de tarefas.

Como descrito na seção 3.1, são quatro os pontos básicos na construção de um algoritmo de aglomeração: cálculo do tempo de execução paralelo de uma coleção, ordenação das tarefas dentro da coleção, condição de parada e escolha da próxima tarefa candidata a entrar em uma coleção.

O tempo de execução paralelo de uma coleção é calculado baseado na função de custo $s(\,)$ dos predecessores imediatos desta coleção (*caminhos-predecessores*) e das tarefas da coleção (*caminho-coleção*), como no algoritmo *HAL*.

Inicialmente, ordenamos as tarefas dentro de uma coleção $C(v) = \langle u_1,u_2,...,u_n,v\rangle$ crescentemente de acordo com o valor $s(u_i)$, como na maioria dos algoritmos de aglomeração. Uma alternativa a esta abordagem foi a utilização da *ordenação sucessor*, como descrita em *HAL*.

O critério de parada está baseado nas condições CCNC (*Caminho-Coleção Não Crítico*) e VPIT (*Vale a Pena Inserir Tarefa*) resultantes da comparação entre o *caminho-predecessor-crítico* e o *caminho-coleção*:

- calcula-se $s(v)$ considerando que $w_i \notin C(v)$ (*caminho-predecessor-crítico*)
- calcula-se $s(v)$ considerando que $w_i \in C(v)$ (*caminho-coleção*)

Ao definir a tarefa w_i candidata a fazer parte da coleção $C(v)$, o algoritmo não escolhe necessariamente aquela da qual o *caminho-predecessor-crítico* é proveniente. Ao invés disso, é criada uma lista (*lista restrita de candidatos*) contendo as k tarefas que mais atrasam a execução da tarefa v, ou seja, passam a ser armazenados k *caminhos-predecessores-críticos* ao invés de apenas um. Tal lista é ordenada segundo o custo destes caminhos, de forma decrescente e, então, uma tarefa da lista é escolhida randomicamente. A cada tarefa escolhida a lista deve ser atualizada, uma vez que novos predecessores de $C(v)$ podem surgir. O algoritmo permanece escolhendo tarefas da lista restrita de candidatos até que algum dos critérios de parada seja atingido. Neste ponto, o algoritmo terá construído a coleção $C(v)$ e este processo deve ser repetido para toda tarefa $v \in V$.

Na seção 5, foi visto que cada iteração GRASP é composta de duas fases: construção de uma solução viável e busca na vizinhança de tal solução. Até o momento, foi descrito como uma solução viável é construída no algoritmo proposto, ou seja, foi apresentada a fase de construção do GRASP. O processo de busca local consiste em constantes trocas da solução atual por uma melhor solução presente em sua vizinhança. Nos poucos algoritmos metaheurísticos desenvolvidos para o problema de escalonamento de tarefas, a busca local geralmente consiste na troca de posições de duas ou mais tarefas, como ilustrado na Figura 6.

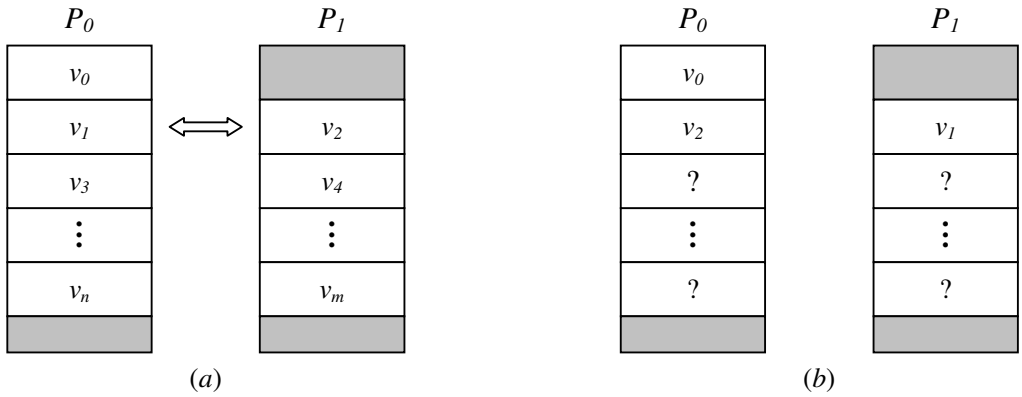

Figura 6: (*a*) *busca local através da troca das tarefas* v_1 *e* v_2. (*b*) *após a troca, as tarefas que inicialmente estavam escalonadas abaixo de* v_1 *e* v_2 *deverão ser reescalonadas.*

Como podemos observar, a busca local no problema de escalonamento de tarefas se mostra bastante custosa uma vez que parte do escalonamento deverá ser refeito a cada troca de tarefas. Provavelmente, este fato é o responsável pelo número tão pequeno de algoritmos metaheurísticos desenvolvidos para a solução deste problema. Em vista dessa dificuldade, optamos por desenvolver um algoritmo GRASP não-convencional composto apenas da fase de construção que é implementada de forma que a solução gerada tenda a ser um ótimo local. O que diferencia o algoritmo proposto das heurísticas convencionais de construção, é justamente as características do GRASP embutidas na fase de construção, como a escolha elemento a elemento através de uma lista restrita de candidatos e a repetição de uma iteração GRASP um determinado número de vezes, ou seja o nosso algoritmo gera ao invés de um, k soluções, onde k é o número de iterações GRASP.

7 – Resultados Computacionais e Conclusões Parciais

O algoritmo proposto foi submetido à mesma bateria de testes que o algoritmo *HAL*, composta de grafos diamantes, árvores binárias simples e invertidas. O prefixo no nome do arquivo indica o tipo de *GAD*, isto é, *DI* para grafos diamantes, *OUT* para árvores binárias (*out-trees*) e *IN* para árvores binárias invertidas (*in-trees*). Seguido do prefixo, o número de tarefas do *GAD* é mostrado. A grande maioria dos grafos de entrada utilizados possuem custos de computação de tarefas e pesos comunicação unitários.

Tais testes foram realizados tanto para o modelo de latência quanto para o modelo LogP. No modelo de latência, variamos o parâmetro τ para avaliarmos o desempenho do algoritmo para grafos de granularidade fina e grossa. Para o modelo LogP, variamos, além da latência, as sobrecargas de envio e de recebimento.

→ Modelo de Latência

Na Tabela 1, apresentamos os resultados obtidos para $\tau = 1$. Nesse caso, o tempo gasto durante a comunicação entre dois processadores não é alto quando comparado com o tempo de execução das tarefas (*granularidade grossa*).

Como podemos observar nesta tabela, o algoritmo proposto apresentou as mesmas soluções obtidas pela heurística *HAL* em todos os testes realizados.

Grafos de Entrada	**Algoritmos considerados**	
	HAL	**Algoritmo Proposto**
IN_9	5	5
IN_{15}	7	7
IN_{31}	9	9
IN_{63}	11	11
IN_{127}	13	13
IN_{255}	15	15
IN_{511}	17	17
OUT_7	3	3
OUT_{12}	4	4
OUT_{15}	4	4
OUT_{31}	5	5
OUT_{63}	6	6
OUT_{127}	7	7
OUT_{255}	8	8
OUT_{511}	9	9
DI_{16}	10	10
DI_{25}	13	13
DI_{36}	16	16
DI_{64}	22	22
DI_{100}	28	28
DI_{144}	34	34
DI_{225}	43	43
DI_{256}	46	46
DI_{400}	58	58
DI_{1024}	64	64

Tabela 1: *Resultados para o modelo de latência* ($\tau = 1$)

Já na Tabela 2, é considerado um tempo de latência mais alto ($\tau = 5$), fazendo com que o tempo de comunicação entre dois processadores seja muito superior ao tempo de execução das tarefas (*granularidade fina*).

Analisando a tabela 2, vemos que em 20% dos testes realizados, a heurística *HAL* obteve melhores resultados dos que os apresentados pelo algoritmo aqui proposto. Porém, também podemos observar que o erro máximo encontrado foi de 5,77% para o grafo DI_{400}.

Grafos de Entrada	**Algoritmos considerados**	
	HAL	**Algoritmo Proposto**
IN_9	*8*	*8*
IN_{15}	*10*	*10*
IN_{31}	*14*	*14*
IN_{63}	*17*	*17*
IN_{127}	*21*	*21*
IN_{255}	*24*	*24*
IN_{511}	*28*	*28*
OUT_7	*3*	*3*
OUT_{12}	*4*	*4*
OUT_{15}	*4*	*4*
OUT_{31}	*5*	*5*
OUT_{63}	*6*	*6*
OUT_{127}	*7*	*7*
OUT_{255}	*8*	*8*
OUT_{511}	*9*	*9*
DI_{16}	*14*	*14*
DI_{25}	*19*	*20*
DI_{36}	*25*	*25*
DI_{64}	*36*	*36*
DI_{100}	***48***	*50*
DI_{144}	*59*	*59*
DI_{225}	***76***	*78*
DI_{256}	***82***	*85*
DI_{400}	***104***	*110*
DI_{1024}	***172***	*180*

Tabela 2**:** *Resultados para o modelo de latência* ($\tau = 5$)

→ Modelo LogP

Os testes realizados considerando o modelo LogP foram divididos segundo o tipo de grafo utilizado e são apresentados nas Tabelas 3 (árvores) e 4 (grafos diamantes).

A Tabela 3, apresentada a seguir, é composta dos resultados obtidos nos testes realizados sobre árvores completas. Foram escolhidos alguns dos grafos utilizados em testes para o modelo de latência: IN_{127}, IN_{511}, OUT_{127} e OUT_{511}.

Para as árvores do tipo *out-trees*, os resultados obtidos foram os mesmos apresentados pela heurística *HAL*. Por outro lado, os resultados obtidos para as árvores *in-trees* apresentaram erro máximo de 12,24% (grafo IN_{511} com $\lambda_s = 4$, $\lambda_r = 2$ e $\tau = 4$).

Grafos de Entrada	λ_s	λ_r	τ	Algoritmos considerados	
				HAL	**Algoritmo Proposto**
IN_{127}	*1*	*1*	*1*	*23*	*23*
IN_{127}	*1*	*1*	*4*	***25***	*28*
IN_{127}	*2*	*2*	*1*	*31*	*31*
IN_{127}	*2*	*4*	*1*	***39***	*43*
IN_{127}	*4*	*2*	*2*	*32*	*32*
IN_{127}	*4*	*2*	*4*	***35***	*37*
IN_{511}	*1*	*1*	*1*	*31*	*31*
IN_{511}	*1*	*1*	*4*	***34***	*38*
IN_{511}	*2*	*2*	*1*	*43*	*43*
IN_{511}	*2*	*4*	*1*	***55***	*61*
IN_{511}	*4*	*2*	*2*	*64*	*64*
IN_{511}	*4*	*2*	*4*	***49***	*55*
OUT_{127}	*1*	*1*	*1*	*7*	*7*
OUT_{127}	*1*	*1*	*4*	*7*	*7*
OUT_{127}	*2*	*2*	*1*	*7*	*7*
OUT_{127}	*2*	*4*	*1*	*7*	*7*
OUT_{127}	*4*	*2*	*2*	*7*	*7*
OUT_{127}	*4*	*2*	*4*	*7*	*7*
OUT_{511}	*1*	*1*	*1*	*9*	*9*
OUT_{511}	*1*	*1*	*4*	*9*	*9*
OUT_{511}	*2*	*2*	*1*	*9*	*9*
OUT_{511}	*2*	*4*	*1*	*9*	*9*
OUT_{511}	*4*	*2*	*2*	*9*	*9*
OUT_{511}	*4*	*2*	*4*	*9*	*9*

Tabela 3**:** *Resultados para o modelo LogP considerando árvores (in-trees e out-trees)*

Os resultados obtidos com os testes realizados sobre grafos diamantes são apresentados na Tabela 4 onde observamos que o maior erro cometido (comparado com os resultados obtidos pela heurística *HAL*) foi de 9,68% (grafo DI_{36} com $\lambda_s = 2$, $\lambda_r = 2$ *e* $\tau = 1$).

Grafos de Entrada	λ_s	λ_r	τ	Algoritmos considerados	
				HAL	**Algoritmo Proposto**
DI_{36}	*1*	*1*	*1*	***25***	*27*
DI_{36}	*1*	*1*	*4*	***30***	*32*
DI_{36}	*2*	*2*	*1*	***31***	*34*
DI_{36}	*2*	*4*	*1*	***36***	*38*
DI_{36}	*4*	*2*	*2*	***34***	*35*
DI_{36}	*4*	*2*	*4*	*34*	*34*
DI_{64}	*1*	*1*	*1*	***36***	*38*
DI_{64}	*1*	*1*	*4*	***44***	*46*
DI_{64}	*2*	*2*	*1*	***46***	*50*
DI_{64}	*2*	*4*	*1*	***58***	*61*
DI_{64}	*4*	*2*	*2*	***52***	*53*
DI_{64}	*4*	*2*	*4*	***53***	*55*

DI_{100}	1	1	1	47	47
DI_{100}	1	1	4	**58**	63
DI_{100}	2	2	1	**61**	63
DI_{100}	2	4	1	**85**	90
DI_{100}	4	2	2	70	70
DI_{100}	4	2	4	**72**	76
DI_{144}	1	1	1	**58**	63
DI_{144}	1	1	4	**72**	78
DI_{144}	2	2	1	**76**	80
DI_{144}	2	4	1	**105**	110
DI_{144}	4	2	2	88	88
DI_{144}	4	2	4	**90**	96

Tabela 4**:** *Resultados para o modelo LogP considerando grafos diamantes*

Com os resultados obtidos, concluímos que o uso de metaheurísticas na solução do problema de escalonamento de tarefas, qualquer que seja o modelo de computação paralela adotado, é viável e merece uma atenção especial por parte da comunidade científica. Os resultados, embora, em determinados casos, se mostraram inferiores, em qualidade, aos obtidos pelo algoritmo *HAL*, nos incentivam a continuar trabalhando com tal problema, a fim de obtermos melhores soluções. Esta afirmação decorre do fato do *HAL* embora seja uma heurística de construção, possuir fases de refinamento que com certeza a tornam um algoritmo mais custoso que o GRASP aqui proposto, já que o nosso algoritmo inclui etapas do HAL mas sem as fases de refinamento deste, que justamente são a parte mais cara em termos de tempo computacional exigido. Testes comparando os tempos de execução não foram possíveis devido a inexistência dos tempos para o algoritmo *HAL*.

8- Trabalhos Futuros:

Uma modificação a ser feita no algoritmo é incluir a otimização realizada ao final do algoritmo *HAL* para o modelo LogP na tentativa de eliminar, de uma coleção $C(v)$, sobrecargas de recebimento cujo tempo de execução seja superior ao tempo de execução de um dos predecessores da coleção.

Propomos também a utilização de outras metaheurísticas para a solução deste problemas, como *VNS* e *Tabu Search*, bem como metaheurísticas híbridas que utilizem características de mais de uma metaheurística, como, por exemplo, *VNS + GRASP*.

Quando tivermos de posse de um algoritmo que apresente soluções de boa qualidade, será desenvolvida uma versão paralela deste, utilizando o ambiente de programação distribuída *MPI* (*Message-Passing Interface*), com o intuito de obtermos tempos de execução razoáveis, principalmente para as maiores instâncias.

Neste trabalho, consideramos um número infinito de processadores homogêneos. Desejamos também desenvolver versões (seqüenciais e paralelas), considerando um número finito de processadores, sejam estes homogêneos ou heterogêneos.

9- Referências Bibliográficas:

[1] T. Adam, K. Chandy e J. Dickson. *A comparison of list schedulers for parallel processing systems*. Comm. ACM, 17(12), pp:685-690, 1974.

[2] C. Boeres. *Versatile communication cost modeling for multicomputer task scheduling heuristics*. Tese de Doutorado, Departamento de Ciência da Computação, Universidade de Edinburgh, 1997.

[3] C. Boeres e V.E.F. Rebello. *A versatile cost modeling approach for multicomputer task scheduling*. Parallel Computing, 25(11), pp:63-68, 1999.

[4] C. Boeres, A. Nascimento e V.E.F. Rebello. *Scheduling arbitrary task graphs on LogP machines*. Proceedings of the 5th International Euro-Par Conference on Parallel Processing (Euro-Par'99), Toulouse, France, LNCS 1685, pp:340-349, 1999.

[5] G. Chochia, C. Boeres, M. Norman e P. Thanisch. *Analysis of multicomputer schedules in a cost and latency model of communication*. Proceedings of the 3rd Workshop on Abstract Machine Models for Parallel and Distributed Computing, Leeds, UK, 1996.

[6] E.G. Coffman Jr. *Computer and job shop scheduling theory*. John Wiley, Nova Iorque, 1976.

[7] R.C. Corrêa, A. Ferreira e P. Rebreyend. *Scheduling multiprocessor tasks with genetic algorithms*. Para constar no IEEE Transactions on Parallel and Distributed Systems, 1996.

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian e T. von Eicken. *LogP: Towards a realistic model on parallel computation*. Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, 1993.

[9] T.A. Feo e M.G.C. Resende. *Greedy Randomized Adaptive Search Procedures*. Journal of Global Optimization 6, pp:109-133, 1995.

[10] S. Fortune e J. Wyllie. *Parallelism in random access machines*. Proceedings of the 10th Annual ACM Symposium on Theory of Computing, ACM Press, pp:114-118, 1978.

[11] Gerasoulis a T. Yang. *A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors*. Journal of Parallel and Distributed Computing, 16, pp:276-291, 1992.

[12] H. Jung, L. Kirousis e P. Spirakis. *Lower bounds and efficient algorithms for multiprocessor scheduling of DAGs with communication delays.* In Proc. ACM Symposium on Parallel Algorithms and Architectures, pp:254-264, 1989.

[13] A.A. Khan, C.L. McCreary e M.S. Jones. *Comparison of multiprocessor scheduling heuristics*. Proc. of the Eighth International Parallel Processing, volume II, pp:243-250, Cancun, México, 1994, IEEE Computer Society Press – ACM SIGARCH.

[14] Kort e D. Trystram. *Scheduling fork graphs under LogP with an unbounded number of processors*. Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par'98), Southampton, UK, LNCS 1470, pp:940-943, 1998.

[15] Y.K. Kwok e I. Ahmad. *Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors*. IEEE Transactions on Parallel and Distributed Systems, 7(5), pp:505-521, 1996.

[16] R.P. Martin, A.M. Vahdat, D.E. Culler e T.E. Anderson. *Effects of communication latency, overhead, and bandwith in a cluster architecture*. In Proceedings of the 24th Annual International Symposium on Computer Architecture, pp:85-97, 1997.

[17] M.A. Palis, J.-C. Liou e D.S.L. Wei. *Task clustering and scheduling for distributed memory parallel architectures*. IEEE Transactions on Parallel and Distributed Systems, 7(1), pp:46-55, 1996.

[18] C.H. Papadimitriou e M. Yannakakis. *Towards an architecture-independent analysis of parallel algorithms*. SIAM J. Comput., 19, pp:322-328, 1990.

[19] Colin R. Reeves. *Modern heuristic techniques for combinatorial problems*. Blackwell Scientific Publications, 1993.

[20] V. Sakar. *Partitioning and scheduling parallel programs for execution on multiprocessors*. MIT Press, 1989.

[21] G. Selim. *Parallel computations: models and methods.* Prentice Hall, 1997.

[22] G.C. Sih e Lee E.A.. *A compile-time scheduling heuristic for interconnection-constraned heterogeneous processor architectures.* IEEE Transactions on Parallel and Distributed Systems, 4(2), pp:175-187, 1993.

[23] S. Voss, S. Martello, I. Osman e C. Roucairol. *Metaheuristics: advances and trends in local search paradigms for optimization*. Kluwer, 1999.

[24] T. Yang e A. Gerasoulis. *DSC: Scheduling parallel tasks on an unbounded number of processors*. IEEE Transactions on Parallel and Distributed Systems, 5(9), pp:951-967, 1994.

[25] W. Zimmerman, M. Middendorf e W. Lowe. *On optimal k-linear scheduling of tree-like task graph on LogP-machines*. Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par'98), Southampton, UK, LNCS 1470, pp:328-336, 1998.