Coloração de Grafo: Um Algoritmo Paralelo Escalável

Juliano Henrique Foleiss, Anderson Faustino da Silva
Laboratório de Linguagens, Compiladores e Programação Paralela
Departamento de Informática
Universidade Estadual de Maringá
Maringá, Brasil
Email: julianofoleiss@gmail.com, anderson@din.uem.br

***Resumo*—A solução de problemas complexos utilizando apenas um processador pode tornar-se rapidamente inviável com o aumento de sua entrada. Portanto, torna-se importante o desenvolvimento de estratégias de paralelização, como também a investigação de modelos de programação distribuída, com o objetivo de encontrar mecanismos de programação que maximizem a expressividade e o desempenho de programas parelelos. Este trabalho propõe um algoritmo paralelo para a solução heurística de um problema NP-Completo: o problema da satisfação de restrições aplicado a coloração de grafos. Além disto, este trabalho avalia dois modelos de programação paralela: `MPI` e `HLRC`. Os resultados obtidos demonstram que a estratégia utilizada na paralelização possui um bom desempenho para os dois modelos.**

***Palavras Chave*-Problema da Satisfação de Restrições; Coloração de Grafo; Paralelismo; Cluster;**

I. Introdução

O modelo de programação sequencial, concebido com a arquitetura de Von Neumann [1] vem mostrando sinais de envelhecimento. Devido aos avanços recentes em diversas as áreas da computação, principalmente as áreas de arquitetura de computadores e redes, a computação paralela vem ganhando cada vez mais importância [2].

A base da computação paralela [3] é a capacidade de múltiplos processadores trabalharem sinergeticamente em uma mesma tarefa. Idealmente, a computação paralela deve contribuir para o desempenho da aplicação em ordem linear, ou seja, a diminuição do tempo de execução deve diminuir de maneira inversamente proporcional ao número de processadores.

Existem basicamente duas especializações da computação paralela, a saber: computação *multicore* e computação distribuída. A computação *multicore* utiliza múltiplos processadores em uma mesma `CPU`, normalmente compartilhando uma memória principal, podendo ou não compartilhar partes de suas memórias *cache* [4].

A computação distribuída, por outro lado, foca na utilização de múltiplos processadores localizados em `CPU`s distintas [5]. Estes por sua vez, preferencialmente interligados por uma rede de alta velocidade e cada um com sua própria memória principal.

Um ambiente comum utilizado para computação distribuída é denominado *cluster* [1]. Nesse ambiente, micro-computadores são ligados por meio de uma rede de alta velocidade, cada um com uma instância diferente de sistema operacional. A escalabilidade nesse tipo de ambiente é muito boa, pois para aumentar o número de processadores basta adicionar máquinas na rede com o *software* necessário para a gerência dos protocolos de comunicação.

O desenvolvimento de programas para tais arquiteturas envolvem basicamente a utilização de dois modelos de programação paralela, a saber: passagem de mensagem [6] e memória compartilhada distribuída [7]. Na passagem de mensagem [8] os processadores comunicam entre si por meio de mensagens, que são normalmente entregues por meio dos protocolos `UDP` e/ou `TCP` [9]. Existem diversos modelos baseados em passagem de mensagem, sendo os mais comuns `PVM` (*Parallel Virtual Machine*) [10] e `MPI` (*Message Passing Interface*) [8].

A idéia da memória compartilhada distribuída é prover virtualmente aos programas um espaço de endereçamento global entre os elementos processadores [11]. Portanto, neste modelo os processadores comunicam entre si lendo e escrevendo em um espaço de endereçamento comum. Entre os protocolos que se destacam estão: `ThreadMarks` [12] e `HLRC` (*Home-Based Lazy Release Consistency*) [13].

Problemas NP-completo [14] são largamente utilizados tanto em computação científica como não científica. Um problema NP-completo com diversas aplicações é o problema de coloração de grafo [15], [16], [17], que é uma instância do problema da satisfação de restrições [18] (um *framework* que permite a solução de diversos problemas complexos). Problemas NP-completo geralmente são tratados heuristicamente, devido ao alto custo computacional.

Uma estratégia para melhorar o desempenho de tais problemas é desenvolver algoritmos paralelos para arquiteturas com diversos processadores.

O objetivo deste trabalho é descrever um algoritmo para a resolução paralela do problema de satisfação de restrições, mais especificamente para o problema de coloração de grafos. Além desta contribuição, dois modelos de programação paralela serão avaliados. Assim, será possível determinar qual modelo se encaixa melhor as características do problema.

Este trabalho está organizado como segue. A Seção II

descreve alguns trabalhos relacionados. A Seção III descreve o problema de satisfação de restrições e como este pode ser mapeado no problema de coloração de grafo. A Seção IV apresenta os modelos de programação paralela utilizados no desenvolvimento do algoritmo paralelo. A Seção V apresenta o algoritmo paralelo proposto. A Seção VI apresenta uma análise preliminar de desempenho. E, finalmente, a Seção VII apresenta as conclusões e os trabalhos futuros.

II. Trabalhos Relacionados

Alguns trabalhos anteriores [19], [20] também propuseram algoritmos para arquiteturas com memória distribuída. Estes trabalhos descrevem algoritmos paralelos iterativos cujo objetivo é a coloração de grafos independentes. Assim como nossa proposta, os vértices que desconectam o grafo são pré-coloridos em uma fase inicial e a computação segue a partir deste ponto de forma independente.

Jones e Plassmann [21] descrevem uma heurística para ser aplicada a coloração paralela de grafo composta de duas fases. A primeira é uma fase heurística aleatória executada em paralelo, cujo objetivo é determinar os máximos conjuntos independentes. Após estes conjuntos serem determinados, na próxima fase cada processador colore seu subgrafo utilizando uma heurística sequencial estabelecida.

O trabalho de Jones e Plassmann se aproxima bem do nosso trabalho pelo fato de descrever uma heurística que minimiza a necessidade de sincronização entre os processadores. No entanto, a estratégia adotada para a verificação da consistência de atribuições intermediárias não utiliza as nossas otimizações propostas em relação à comparação eficiente dos domínios de cores disponíveis. Além disto, a metodologia utilizada na avaliação segue um caminho diferente, não mostrando nem tempo de execução nem o *speedup* obtido pela heurística proposta.

O trabalho de Boman *et al.* [22] propõe um algoritmo guloso iterativo para a coloração paralela de grafo. Inicialmente, o processador mestre particiona o grafo e distribui os subgrafos pelos processadores que compõem a computação. Assim que cada processador escravo recebe um subgrafo, este tenta colorir o subgrafo e então envia o resultado para o mestre. Na ocorrência de um conflito, o mestre redistribui parte do trabalho entre os processadores, até que não existam mais conflitos.

Um problema decorrente desta estratégia é o fato desta acarretar muito tráfego na rede, podendo consequentemente ocasionar perda de desempenho. Isto é bem diferente da nossa estratégia, que não possui comunicação entre os processadores durante a fase da computação da coloração de grafos.

III. O Problema de Satisfação de Restrições

Matematicamente, o problema de satisfação de restrições (PSR) [18] é definido como um conjunto de variáveis – $\{X_1, X_2, ..., X_n\}$ – e um conjunto de restrições – $\{C_1, C_2, ..., C_n\}$. Cada variável X_i possui um domínio D_i constituído de seus possíveis valores. Cada restrição C_i envolve um subconjnuto dos valores e especifica todas as combinações permitidas de valores para aquele conjunto. Um estado do problema é definido como uma atribuição de valores a algumas ou a todas as variáveis, $\{X_i = v_i, X_j = v_j\}$. Uma atribuição que não viola nenhuma restrição é denominada consistente ou válida. Uma atribuição completa é uma na qual toda variável é mencionada e uma solução a um PSR é uma atribuição completa que satisfaz todas as restrições.

Um PSR pode ser representado por meio de um grafo de restrições. Nesse grafo, os vértices são as variáveis e as arestas indicam restrições entre os vértices. Os rótulos dos vértices indicam o domínio da variável sendo representada, enquanto os rótulos das arestas indicam a restrição em questão.

O problema principal na resolução de um PSR é escolher um valor para uma variável tal que nenhuma restrição seja violada. O algoritmo AC-3 [23] permite a verificação da consistência dos valores atribuídos em relação às restrições do problema, para um PSR representado como um grafo de restrição. O Algoritmo 1 mostra o funcionamento de um PSR qualquer.

Se algum domínio for vazio depois da aplicação do algoritmo AC-3, o problema possivelmente não pode ser verificado 3-consistente e um algoritmo de mais alta ordem deve ser utilizado para a consistência de arcos [18]. O algoritmo para uma verificação de consistência de ordem arbitrária existe, no entanto foge do escopo deste trabalho.

Mackworth mostra em [24] que o desempenho deste algoritmo no pior caso é O(ed^3), onde e é o número de arcos e d é o tamanho do maior domínio do PSR.

O problema da coloração de grafos é um PSR. Para isso, basta considerar os vértices do grafo como as variáveis do problema e as arestas como restrições que impedem que os vértices ligados a ela sejam coloridos da mesma cor. As cores disponíveis são elementos dos domínios de cada variável, todos inicialmente contendo todas as cores disponíveis. Assim, o objetivo é colorir o grafo de forma que dois vértices adjacentes não tenham a mesma cor, como mostrado na Figura 1.

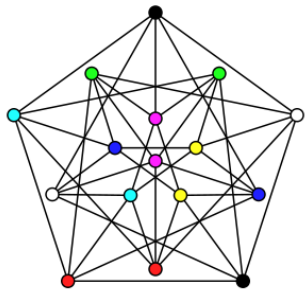

Figura 1. Um grafo após a execução do algoritmo de coloração.

Algoritmo 1 Consistência de Arcos AC-3

```
00  AC-3(PSR)
01      ENTRADA:
02         PSR - um PSR binário com variáveis X1 até Xn
03      VARIÁVEL LOCAL:
04         FILA - uma fila de arcos inicializada com todos os arcos do PSR
05      RETORNO:
06         PSR com domínios possivelmente reduzidos

08      ENQUANTO a FILA é não-vazia FAÇA
09          (Xi, Xj) = PRIMEIRO(FILA)
10          SE REMOVER_VALORES_INCONSISTENTES(Xi, Xj) ENTÃO
11              PARA CADA Xk em VIZINHOS(Xi) FAÇA
12                  Coloque (Xk, Xi) na FILA

14  REMOVER_VALORES_INCONSISTENTES(Xi, Xj)
15      ENTRADA:
16         Variáveis Xi e Xj
17      RETORNO:
18        VERDADEIRO
19             se e somente se um valor for removido do domínio de Xi

21      removido = FALSO
22      PARA CADA x em DOMÍNIO(Xi) FAÇA
23          SE nenhum valor y em DOMÍNIO(Xj) permitir que (x, y) satisfaça
24              a restrição entre Xi e Xj ENTÃO
25              remova x do DOMÍNIO(Xi)
26              removido = VERDADEIRO
27      RETORNE removido
```

Algoritmo 2 Coloração de Grafo

```
01  COLORACAO_GRAFO(GRAFO, CORES)
02     RETORNO:
03        Atribuição de cores aos vértices tal que não
04        existam vértices adjacentes coloridos com a mesma cor

06     PARA TODO v em VERTICES(GRAFO) FAÇA
07        DOMÍNIO(V) = CORES

09     PARA TODO v em VERTICES(GRAFO) FAÇA
10        SE DOMÍNIO(V) != VAZIO ENTÃO
11            DOMÍNIO(V) = COR_DISPONÍVEL(DOMÍNIO(V))
12            AC-3(GRAFO)
13        SENÃO
14            Não é possível colorir este grafo com #CORES!
```

O Algoritmo 2 mostra o código utilizado para resolver o problema de coloração de grafo.

Levando em consideração que as restrições são todas *"x diferente de y"* e que o domínio inicial de cada vértice é o mesmo, o algoritmo AC-3 pode ser simplificado. Considerando o domínio como um conjunto de variáveis lógicas, onde cada cor é uma variável e cada cor pode ser escolhida ou não, pode-se armazenar as variáveis como inteiros com `k` bits, onde `k` é o número de cores disponíveis. Com isso, a função REMOVER_VALORES_INCONSISTENTES na linha 14 do Algoritmo 1 pode ser executada em O(1), ao invés de O(d^2). Além disso, as linhas 11 e 12 (deste mesmo algoritmo) tornam-se desnecessárias. Segue que, o limitante do desempenho desta versão do algoritmo AC-3 agora torna-se sua inicialização, que é $\Omega(n^2)$ utilizando uma matriz de adjacência. Portanto, o Algoritmo 2 executa em $\Theta(n^3)$.

Embora esse algoritmo seja de ordem polinomial, para entradas muito grandes seu desempenho pode ser consideravelmente ruim, acarretando em um tempo elevado de execução. Para melhorar o desempenho de tal algoritmo, é proposto o uso de programação paralela, o que permite a utilização de diversos processadores.

IV. Modelos de Programação Distribuída

Existem dois modelos básicos de programação distribuída, a saber: passagem de mensagem e memória compartilhada distribuida [2], [7]. A principal diferença entre estes modelos está na abstração de como a comunicação é realizada entre os processadores. O nível da abstração e sua complexidade levam a diferentes problemas e oferecem diferentes possibilidades ao usuário.

A. Passagem de Mensagem

Na passagem de mensagem, os processadores se comunicam por meio de mensagens enviadas pela rede. Essas mensagens são enviadas e/ou recebidas através de algumas primitivas oferecidas pelo modelo. O padrão mais utilizado deste modelo é o MPI [25].

`MPI` [8] é uma biblioteca de passagem de mensagem com o objetivo de estabelecer um padrão portável, eficiente e flexível. `MPI` é a primeira biblioteca aberta de passagem de mensagem a ser padronizada. Além disso, esta biblioteca foi projetada para ser utilizada na implementação de programas com propósito geral, ou seja, independente de domínio de aplicação.

Como um modelo de programação, `MPI` pode ser utilizado para implementar outros modelos de programação distribuída. De fato, `MPI` é comumente utilizado na implementação de alguns modelos de memória compartilhada distribuída [7]. Além disto, este padrão pode ser utilizado em uma ampla gama de plataformas distribuídas, a saber: memória compartilhada, memória distribuída e arquiteturas híbridas.

Uma restrição original, hoje já vencida pela versão 2 da especificação, é que o número de tarefas executando em paralelo é estático, ou seja, novas tarefas não podiam ser criadas dinâmicamente em tempo de execução.

Neste padrão todo paralelismo é explícito, ou seja, em `MPI` o programador é responsável pela identificação do paralelismo e pela implementação dos algoritmos paralelos com a utilização das primitivas oferecidas, que podem ser bloqueantes ou não, dependendo da necessidade da aplicação e da capacidade da rede em uso.

B. Memória Compartilhada Distribuída

Ao contrário da passagem de mensagem, em memória compartilhada distribuída não existe o envio explícito de mensagens pelo programador [7]. A nível de linguagem de programação, esse modelo permite que memórias fisicamente distribuídas possam ser acessadas por todos os processadores como se fossem uma única memória global (compartilhada). Isso permite que programas distribuídos possam ser escritos de maneira mais natural, sem o excesso de primitivas de comunicação comumente necessárias em aplicações com passagem de mensagem.

Como as memórias estão distribuídas fisicamente pelas diversas máquinas que compõem a arquitetura física, um determinado dado pode estar localizado em qualquer máquina. Além disso, múltiplas cópias de um mesmo dado compartilhado podem existir simultaneamente. Isso leva a problemas de consistência, onde uma operação de escrita no dado compartilhado deve ser visto por todos os processadores. Para gerenciar situações como esta, as implementações oferecem primitivas de sincronização.

A sincronização é feita por meio de *locks* e/ou barreiras [4]. *Locks* garantem acesso mutuamente exclusivos. E uma barreira determina um ponto de sincronização entre todos os processadores, ponto no qual todos os dados compartilhados são atualizados. Ambos mecanismos garantem que as memórias fisicamente distribuídas estarão consistentes em acessos sincronizados.

`HLRC` [13], [26] é uma implementação de memória compartilhada distribuída que utiliza uma variação do protocolo de consistência por liberação preguiçosa [27] através de um *software* para detectar escritas e um esquema de propagação de escritas baseado em mecanismo de *diffs*, que são registros contendo todas as modificações realizadas nos dados compartilhados. `HLRC` utiliza um nó residência para cada dado compartilhado, no qual atualizações são coletadas e aplicadas, garantindo que em seu nó residência o dado compartilhado esteja sempre atualizado.

Uma característica importante deste modelo é o fato do ambiente de programação ser simples e conveniente. Além disto, compartilhar memória torna transparente a comunicação entre processos e as aplicações escritas são usualmente mais fáceis de se entender. Além disto, com um espaço de endereçamento global, o programador pode focar

no desenvolvimento do algoritmo ao invés do particionamento dos dados e da comunicação entre processos, o que ocorre no uso do modelo de passagem de mensagem.

V. O Algoritmo Paralelo Proposto

Nossa proposta para uma solução paralela é particionar o grafo de maneira que as atribuições de cor aos vértices do grafo sejam consistentes com as atribuições de todos os processadores.

Uma maneira elegante de particionar o grafo é encontrar suas articulações. Uma articulação é um nó, que quando retirado do grafo, desconecta o mesmo. As articulações são identificadas utilizando-se uma modificação no algoritmo de busca em profundidade que pode ser executado em O(n). A partir das articulações é possível utilizar o seguinte algoritmo para colorir o grafo com K processadores.

Seja $\{A_1, A_2, ..., A_n\}$ o conjunto de articulações do grafo. Primeiramente, escolha uma cor para todos os A_i e execute o algoritmo de consistência de arcos para cada escolha. Retire então, todos as articulações do grafo, e distribua igualmente os subgrafos resultantes G_i entre todos os K processadores. Aplique o Algoritmo 2 para cada um dos grafos $G_{i,k}$. Depois que todos os processadores terminarem, basta coletar as soluções parciais e reinserir todas as articulações, reconectando o grafo.

A estratégia utilizada no algoritmo proposto é capaz de desconectar o grafo em W instâncias em um tempo computacional que não causa *overhead* ao tempo total de execução (como será demonstrado na Seção VI). O objetivo a ser alcançado pela estratégia adotada na decomposição do grafo é desconectá-lo em uma quantidade de subgrafos que gere um balanceamento de carga entre os processadores que participarão da computação. No entanto, dado que a ocorrência de articulações é algo não previsível, não é possível garantir o balanceamento de carga de maneira igualitária entre os processadores.

Por outro lado, caso existam articulações no grafo, a estratégia utilizada gera potencialmente tarefas com granularidade grossa, eliminando desta forma a necessidade de barreiras de sincronização durante a execução do algoritmo de coloração de grafos.

A. A Versão com MPI

Na implementação da versão paralela utilizando `MPI`, o processador com *rank* 0 é responsável pelo cálculo das articulações e pela distribuição do grafo aos demais processadores. Enquanto o processador 0 calcula as articulações, os demais ficam esperando uma mensagem com a especificação do trabalho a ser realizado.

Quando o trabalho chega aos demais processadores, tais processadores aplicam o Algoritmo 2 ao seu subgrafo sem a necessidade de enviar mensagens aos outros processadores. Isto é possível devido à fatoração da aplicação, de forma a permitir o cálculo dos domínios dos vértices de forma independente em relação aos demais processadores. Um ponto importante a ser ressaltado é o fato de que nesta versão, o processador com o *rank* 0 também *recebe* um subgrafo para colorir.

Como o grafo original não é compartilhado, é necessário que todos os processadores carreguem o grafo (gerado sinteticamente) na memória. Portanto, não é necessária a transferência do grafo durante a inicialização. Além disso, é necessário que o processador 0 envie quais são as articulações para os demais, para que esses possam retirá-las do grafo para que o algoritmo opere corretamente.

Um outro dado necessário a ser enviado é o domínio inicial das variáveis. Isso é necessário pois as articulações são coloridas pelo processador 0, e todos os outros processadores devem respeitar a escolha das cores feita, que implica na adequação de seus vértices adjacentes.

Assim, os dados que compõem o trabalho dos processadores são: *o número de cores a colorir o grafo, quais e quantas são as articulações, os subgrafos que os competem* e os *domínios iniciais de cada vértice que não possui o domínio completo no início do processamento*. Todos esses dados são enviados pelo processador 0 aos demais no início do processamento.

Quando cada processador termina seu trabalho, este envia mensagens com as respostas de seus subgrafos ao processador 0, que é responsável por coletar as soluções parciais e reconectar o grafo.

B. A Versão com HLRC

A implementação com `HLRC` difere um pouco. O algoritmo básico utilizando articulações permanece. No entanto, a distribuição de trabalho é realizada de maneira diferente.

No início é necessário que todos os processadores aloquem memória na mesma ordem. Como não é possível que o processador 0 calcule exatamente o tamanho do trabalho e envie aos demais processadores, a quantidade de memória alocada é baseada nos parâmetros de entrada (número de vértices, cores e número de pontes no grafo), que podem oferecer esse tamanho de maneira aproximada.

Feita a alocação, o processador 0 calcula o trabalho de maneira semelhante à feita na versão com `MPI` e os armazena na memória compartilhada. Para que a memória de todas as máquinas estejam sincronizadas, existe um ponto de sincronização antes do inicio da computação útil. Após este ponto de sincronização todos os processadores copiam os seus dados para uma memória local, a fim de eliminar o custo de ler a memória compartilhada.

Uma vez copiado o trabalho, os processadores executam cada uma de suas tarefas e, quando terminam, escrevem em um vetor compartilhado os domínios finais de cada uma de suas variáveis. O processador 0 por sua vez, lê o vetor compartilhado, constrói a solução final e reconecta o grafo.

VI. Avaliação Experimental Preliminar

A solução paralela para o problema da coloração de grafos foi implementada na linguagem de programação `C`, compilada com GCC 4.4.3 e utiliza a biblioteca GMP 5.0.1 para manipulação lógica e aritmética. Os experimentos foram executados em um *cluster* homogêneo, composto de 8 máquinas AMD 64 X2 Dual Core Processor 5000+, com 2GB RAM, 64KB de *cache* L1 separada e 512KB de *cache* L2 unificada.

A implementação escolhida para `MPI` foi a OpenMPI 1.4.2 [8] compilada a partir do código fonte. Para `HLRC`, foi utilizada uma versão modificada da implementação de Rutgers [26] feita no Laboratório de Computação Paralela da COPPE/UFRJ, onde o protocolo foi adaptado para ser utilizado via `TCP`. O *overhead* de comunicação no `MPI` foi medido com a biblioteca `mpiP` [28]. Na versão *HLRC* o *overhead* foi medido pelo próprio ambiente de execução, que gera um relatório ao final de cada execução.

Foram utilizados três grafos com tamanhos diferentes para avaliar o desempenho do algoritmo. O primeiro grafo possui 5000 vértices, o segundo 6000 vértices e o terceiro 7000 vértices. Todos com 50% de cobertura e 75 cores.

Os três grafos foram gerados sinteticamente. O gerador de grafos aleatórios possui parâmetros que permitem a criação de grafos com algumas características: número de vértices, semente de números aleatórios, cobertura e número de pontes (arestas que quando retiradas, desconectam o grafo). Nesse caso, segue que, a criação das pontes implica na criação de duas articulações que podem ser utilizadas para a divisão do trabalho na execução do algoritmo paralelo proposto.

As Figuras 2 e 3 apresentam o *speedup* obtido pelas duas versões implementadas, para as três entradas.

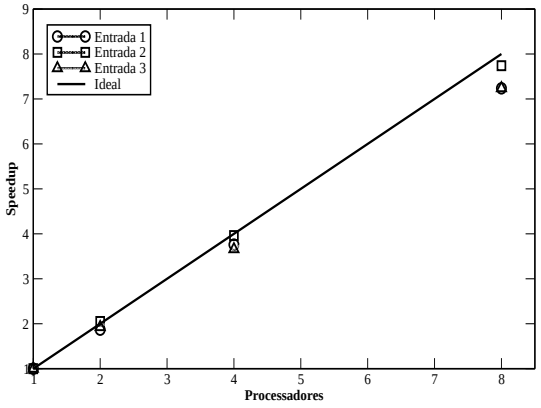

Figura 2. *Speedup* obtido pela versão com `MPI`.

A Figura 2 demonstra que a implementação `MPI` obteve um desempenho próximo ao linear para todas as instâncias do problema, com desvio médio de apenas 5,07% em relação à curva ideal. No `HLRC` o desempenho foi melhorando

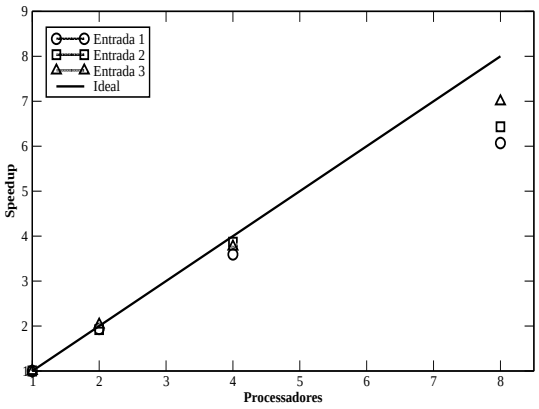

Figura 3. *Speedup* obtido pela versão com `HLRC`.

conforme a entrada foi aumentando, e o desvio médio foi de 9,01%.

Na versão com `MPI` houve um pico de desempenho na segunda entrada. Para tal, o desvio em relação à curva ideal é de apenas 0,58%, sendo a instância com 2 processadores superlinear em 0,02%. Após esse pico, houve uma leve queda para a entrada maior, no entanto mantendo um bom desempenho para todos processadores envolvidos, com apenas 3,21% de desvio padrão.

Na versão com `HLRC` o desempenho foi melhorando consideravelmente com o aumento do tamanho da entrada. Nos experimentos, o melhor desempenho obtido ocorreu com a maior entrada, onde o desvio em relação à curva ideal foi de apenas 5,57%, sendo duas vezes mais próxima que o desvio da primeira entrada em relação à curva ideal. No caso geral, nota-se que entre uma entrada e outra, o desvio em relação à curva reduz linearmente da menor para a maior entrada.

É interessante ressaltar que a granularidade das tarefas é alta e que o máximo paralelismo a ser obtido é a nível de divisão de grafo. Nenhuma cooperação a nível de coloração é feita, devido à natureza sequencial e interligada da operação de coerência de restrições.

Como os subgrafos gerados são de tamanho e complexidade parecidos, todos os processadores terminam aproximadamente ao mesmo tempo. Com isso, a rede permanece ociosa durante a maioria do tempo de processamento, tornando-se carregada ao final da execução, quando todos os processadores estão enviando as soluções parciais. Além disso, o início do processamento também sobrecarrega um pouco a rede, quando as informações sobre os trabalhos são enviadas aos processadores, no entanto em uma escala menor.

As Figuras 4 e 5 apresentam o tempo de execução decomposto em *computação útil* e *overhead*. Este último componente indica a comunicação realizada entre os processadores durante a passagem de mensagem (`MPI`) e gerência do protocolo (`HLRC`). Da esquerda para a direita, as três

primeiras barras representam o tempo de execução para a menor entrada, as três barras seguintes para a entrada média e as últimas para a maior entrada.

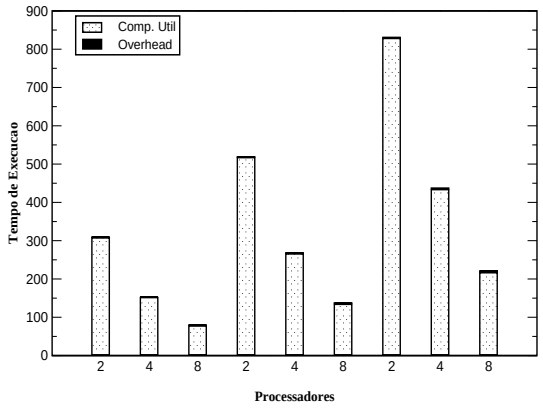

Figura 4. Decomposição do tempo de execução da versão com MPI.

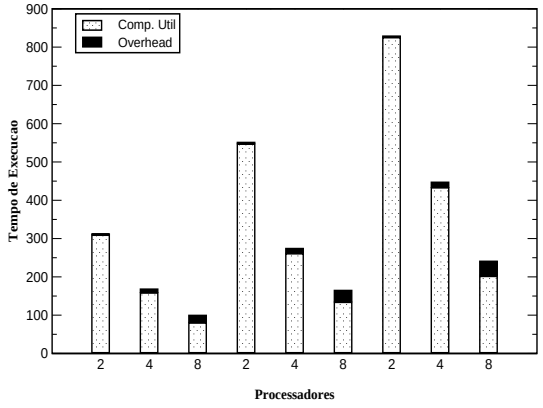

Figura 5. Decomposição do tempo de execução da versão com HLRC.

Em geral, o *overhead* no tempo de execução da versão MPI é menor do que na versão HLRC. Isso provavelmente está relacionado ao nível de abstração dos dois modelos. No MPI, toda comunicação é feita de forma explícita e exata, ou seja, o envio dos dados deve ser cuidadosamente feito de maneira a empacotar corretamente os dados com o objetivo de enviar somente os dados cruciais para o processamento da aplicação. Como os dados deste problema são pequenos e não há transferência do grafo, o *overhead* esperado para MPI é baixo. Além disso, não é necessária a sincronização para que os processadores iniciem seus respectivos trabalhos.

O HLRC, por outro lado, oferece um nível de abstração mais alto, que pode resultar em um envio de mensagens um pouco mais elevado, principalmente para a gerência do protocolo. Outro fator importante a ser considerado é que na implementação da versão com HLRC é necessária a sincronização entre os processadores na distribuição das tarefas.

Na inicialização da versão com HLRC deve existir um ponto de sincronização, pois enquanto o processador 0 escreve o trabalho no vetor compartilhado, não há como os demais processos identificarem quem já teve seu trabalho especificado sem a utilização de uma primitiva de sincronização, o que aumenta ainda mais o tráfego na rede. Isso não é necessário no MPI, pois o aviso de que o trabalho está especificado é implícito na chegada da mensagem no *rank* receptor.

Embora o tempo de *overhead* tenha aumentado conforme a entrada, seu percentual em relação ao tempo total da aplicação diminui em MPI e HLRC. Nessa aplicação, isso é esperado, pois a quantidade de processamento útil deve aumentar em relação ao tempo de *overhead*, devido ao fato de não haver sincronização durante o processamento. Além disso, a quantidade de dados a serem enviados aumenta pouco com o aumento da entrada. Esses fatores levam o tempo de processamento a eventualmente esconder o tempo de *overhead*, desde que o número de processadores não aumente drasticamente, o que pode levar a um aumento excessivo no tráfego na rede, aumentando consideravelmente o *overhead* da comunicação.

Embora o MPI ofereça um controle maior ao programador a nível de tráfego de dados, a abstração oferecida pelo HLRC permite a escrita de programas mais legíveis e naturais. Porém, modelos com níveis de abstração mais elevados geralmente geram uma quantidade maior de *overhead* na comunicação e manutenção do protocolo, podendo levar a certas limitações quanto à flexibilidade no desenvolvimento das aplicações.

VII. Conclusões e Trabalhos Futuros

Neste trabalho foi proposto um algoritmo paralelo para coloração de grafo em um ambiente distribuído. A estratégia proposta alcança um balanceamento de carga potencialmente distribuindo de forma igualitária o trabalho entre os processadores que compõem a computação, desde que o grafo apresente articulações. Outra vantagem do algoritmo proposto é o fato deste evitar a sincronização durante o processamento do problema, além de não diminuir significantemente sua granularidade. Esta última é muito importante, devido ao fato de que matematicamente a ocorrência de articulações não divide o grafo em uma quantidade de subgrafos muito expressiva.

De uma maneira geral, ambas versões distribuídas obtiveram ótimos resultados nos experimentos realizados. Nota-se a vantagem da versão MPI para entradas menores, para qualquer quantidade de processadores. A versão HLRC por outro lado, mostra uma melhoria gradativa no desempenho de acordo com o aumento da entrada.

Trabalhos futuros incluem: uma estratégia de subdivisão sucessiva com o objetivo de provocar divisões que levem a uma quantidade de subgrafos que possam garantir um balanceamento de carga mais justo e eficiente. Além disto,

estamos trabalhando na avaliação do algoritmo com o *benchmark* OR-LIBRARY [29] com o objetivo de facilitar a comparação com trabalhos relacionados. Outro trabalho sendo avaliado é a possibilidade de aplicação deste algoritmo para a solução de outros problemas NP-Completo a partir de reduções polinomiais, como por exemplo, o problema do caixeiro viajante (TSP) e o problema da satisfatibilidade booleana (3SAT).

REFERÊNCIAS

[1] W. Stallings, *Computer Organization and Architecture: Design and Performance*, 8th ed. USA: Prentice Hall, 2009.

[2] C. Lin and L. Snyder, *Principle of Parallel Programming*, 1st ed. California, USA: Addison Wesley, 2009.

[3] D. B. Skillicorn and D. Talia, "Models and Languages for Parallel Computation," *ACM Computing Survey*, vol. 30, no. 2, pp. 123–169, 1998.

[4] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2009.

[5] C. Leopold, *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches*. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[6] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, 1st ed. USA: Addison-Wesley, 2000.

[7] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed Shared Memory: Concepts and Systems," *IEEE Parallel and Distributed Technology*, vol. 4, no. 2, pp. 63–79, Jun. 1997.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings of the European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[9] W. R. Stevens, *UNIX Network Programming*, 2nd ed. USA: Prentice Hall, 1998, vol. 1.

[10] R. J. Manchek, "Design and Implementation of PVM Version 3," Knoxville, TN, USA, Tech. Rep., 1994.

[11] J. Tao, W. Karl, and C. Trinitis, "Trinitis: Implementing an OpenMP Execution Environment on Infiniband Clusters," in *Proceedings of the International Workshop on OpenMP*, 2005.

[12] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *IN PROCEEDINGS OF THE 1994 WINTER USENIX CONFERENCE*, 1994, pp. 115–131.

[13] L. Iftode, "Home-Based Shared Virtual Memory," Ph.D. dissertation, Princeton University, Princeton, USA, Jun. 1998.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[15] A. W. Appel and M. Ginsburg, *Modern Compiler Implementation in C*. Press Syndicate of the University of Cambridge, 1998.

[16] G. J. Chaitin, "Register Allocation & Spilling via Graph Coloring," in *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. New York, NY, USA: ACM, 1982, pp. 98–105.

[17] J.-P. Hamiez and J.-K. Hao, "An Analysis of Solution Properties of the Graph Coloring Problem," pp. 325–345, 2004.

[18] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[19] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin, "A Comparison of Parallel Graph Coloring Algorithms," Technical report SCCS-666, Northeast Parallel Architectures Center at Syracuse University, Tech. Rep., 1995.

[20] R. K. Gjertsen, Jr., M. T. Jones, and P. E. Plassmann, "Parallel Heuristics for Improved, Balanced Graph Colorings," *Journal of Parallel and Distributed Computing*, vol. 37, pp. 171–186, 1996.

[21] M. T. Jones, P. E. Plassmann, and P. Mcs-p, "A Parallel Graph Coloring Heuristic," *SIAM J. Sci. Comput*, vol. 14, pp. 654–669, 1992.

[22] D. Bozdag, A. H. Gebremedhin, F. Manne, E. G. Boman, and U. V. C. Catalyürek, "A Framework for Scalable Greedy Coloring on Distributed-memory Parallel Computers," *Jornal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 515–535, 2008.

[23] A. K. Mackworth, "Consistency in Networks of Relations," Vancouver, BC, Canada, Canada, Tech. Rep., 1975.

[24] A. K. Mackworth and E. C. Freuder, "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," Vancouver, BC, Canada, Canada, Tech. Rep., 1982.

[25] M. Group, "Message Passing Interface Forum," 2010, http://www.mpi-forum.org/; Acesso em 7 de Junho de 2010.

[26] M. Rangarajan and L. Iftode, "Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance," in *Proceedings of The Annual Linux Showcase*, Out. 2000.

[27] P. Keleher, A. Cox, and W. Zwenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proceedings of the 9th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp. 13–21.

[28] J. Vetter and C. Chambreau, "mpiP: Lightweight, Scalable MPI Profiling," 2010, http://mpip.sourceforge.net/; Acesso em 7 de Junho de 2010.

[29] J. E. Beasley, "OR-Library: Distributing Test Problems by Electronic Mail," *Journal of the Operational Research Society*, vol. 41, pp. 1069–1072, 1990.