Experimentos com um Algoritmo Distribuído para Troca Ordenada de Mensagens entre Tarefas que Migram*

N.C. COURI, S.C.S. PORTO
Computação Aplicada e Automação
Universidade Federal Fluminense
Rua Passo da Pátria, 156
24210-240 Niterói - RJ
{couri, stella@caa.uff.br}

V.C. BARBOSA
Programa de Sistemas, COPPE
Universidade Federal do Rio de Janeiro
Caixa Postal 68511
21945-970 Rio de Janeiro - RJ
{valmir@cos.ufrj.br}

Resumo – Para muitos algoritmos que utilizam trocas de mensagens, é fundamental que o sistema de computação garanta a entrega ordenada de mensagens entre processadores. No entanto, mesmo quando esta garantia é oferecida, a migração de tarefas entre processadores pode fazer com que a ordenação seja perdida. Neste trabalho, consideramos o PSP (*Pipe Shortening Protocol*), um algoritmo distribuído assíncrono que garante a entrega ordenada de mensagens entre tarefas que migram. Um ambiente de teste, composto por uma aplicação sintética e diferentes políticas de migração, foi construído para experimentação do algoritmo numa máquina paralela real. Para diferentes aplicações paralelas sintéticas foram considerados dois casos distintos: com e sem o uso do algoritmo PSP. Foram gerados resultados numéricos comparativos do tempo de resposta a fim de delimitar as configurações dos parâmetros sob as quais o algoritmo PSP alcança melhor desempenho.

Abstract – For many message-passing algorithms, the computational system must guarantee an ordered message delivery between processors. However, even when this guarantee is assured, the task migration between processors may lead to the loss of this ordering. We consider in this paper, the *Pipe Shortening Protocol* (PSP), an asynchronous distributed algorithm that ensures the automatic FIFO message delivery between migrating tasks. A testing environment consisting of a synthetic application and different migration policies was developed in order to experiment the PSP in an real parallel machine. Two configurations were considered under different parallel synthetic applications: with and without the PSP algorithm. Comparative numerical results of the response time were measured in order to determine under which parameter settings the PSP algorithm achieves best performance.

*Este trabalho foi possível graças ao apoio financeiro do CNPq e da FAPERJ. Os experimentos computacionais foram realizados nas máquinas IBM SP do LNCC e NCE/UFRJ.

1 Introdução

No contexto deste trabalho, consideram-se aplicações paralelas onde tarefas se comunicam através da troca de mensagens. Este tipo de computação pode tanto ocorrer em máquinas MIMD com memória distribuída quanto em ambientes distribuídos formados por uma rede de estações de trabalho, sobre uma plataforma de comunicação baseada em troca de mensagens, como PVM e MPI. Em ambos os casos, é comum a garantia de entrega ordenada de mensagens entre processadores. Definem-se como *processadores vizinhos* aqueles que se conectam diretamente. As mensagens enviadas entre dois processadores que não são vizinhos devem ser conseqüentemente roteadas por processadores intermediários para uma eventual entrega dessas mensagens ao processador de destino.

Trocas de mensagens neste artigo são não bloqueantes, de tal maneira que se uma tarefa envia uma mensagem à outra, a primeira não espera que a mensagem seja entregue ao seu destino para dar prosseguimento à sua computação. Tarefas devem ser alocadas a processadores e a troca de mensagens entre tarefas alocadas a processadores diferentes é realizada por uma rede de interconexão, representada por canais de comunicação. Vários autores já consideraram a migração de tarefas durante a execução de uma dada aplicação paralela, ou seja, a re-alocação de uma tarefa para outro processador durante sua execução, particularmente no contexto de sistemas operacionais em rede como o V-System [13] e o Sprite [10].

Considera-se aqui, com especial interesse, sistemas com memória distribuída que utilizam uma estrutura de roteamento fixa e cujos canais de comunicação entre processadores são do tipo FIFO, ou seja, mensagens são entregues na mesma ordem que foram enviadas ao longo de cada canal de comunicação. No caso em que não há migração é possível garantir a entrega FIFO de mensagens a nível das tarefas através da manutenção de filas com propriedade FIFO: uma para cada tarefa contendo as mensagens que estas recebem e outras específicas por processador, para as mensagens que enviam. Esta é uma propriedade desejável para muitos algoritmos paralelos distribuídos, como por exemplo n algoritmo para registrar estados globais de Chandy e Lamport [2], cuja simplicidade é perdida caso não haja garantia de entrega de mensagens entre tarefas em ordem FIFO; ou no mecanismo de simulações distribuídas de eventos discretos de Jefferson (*Time Warp*) [7], cujo funcionamento pode ser imensamente simplificado se a ordem FIFO de entrega de mensagens entre tarefas for garantida.

Quando tarefas migram de um processador a outro, não existem garantias de que as mensagens serão entregues na ordem FIFO. É sempre possível considerar a abordagem direta de colocar rótulos junto às mensagens e então usar esses rótulos no processador destino, de maneira a re-ordenar as mensagens antes de sua entrega efetiva às suas respectivas tarefas de destino. No entanto, em muitos casos, esse procedimento não é desejável pois, em princípio, necessita-se de um número de rótulos e de um espaço de armazenamento ilimitados para que o re-seqüenciamento das mensagens esteja livre de *deadlock*.

Embora o problema de roteamento de mensagens entre tarefas que migram já tenha sido estudado anteriormente [11], bem como o tópico relativo à comunicação transparente entre essas tarefas [10, 13], não se tem conhecimento acerca da existência de outras publicações referentes à garantia de entrega em ordem FIFO de mensagens

entre tarefas, a não ser pela proposta descrita em [1] por Barbosa e Porto, considerada no presente trabalho. Naquela oportunidade, descreveu-se um algoritmo distribuído assíncrono para a manutenção da ordem FIFO de entrega de mensagens entre tarefas que migram em um sistema de memória distribuída, chamado de PSP (*Pipe Shortening Protocol*). Os dois requisitos necessários ao funcionamento deste algoritmo são a existência de uma estrutura fixa de roteamento e interligação entre processadores baseada em canais FIFO. Para a migração concorrente de um conjunto de K tarefas, o PSP requer $O(nm_K)$ mensagens entre processadores e tempo de execução $O(n)$, onde n é o número de processadores no sistema e m_K é o número de pares de tarefas que se comunicam (considerando aqueles que envolvem pelo menos uma das K tarefas). Entretanto, para muitas arquiteturas atuais de memória distribuída, esses valores podem ser reduzidos a $O(m_K)$ e $O(1)$, respectivamente [1]. O algoritmo PSP pode também ser aplicado a um grupo de estações de trabalho e até mesmo em redes de computadores, bastando para isso, que esses requisitos sejam atendidos.

Em [1], Barbosa e Porto provaram a corretude do algoritmo PSP então proposto. No entanto, as condições nas quais se baseia o desempenho do algoritmo precisam ser comprovadas experimentalmente. Para tanto, um ambiente de teste, composto por uma aplicação sintética e duas políticas de migração, foi construído para experimentação prática do algoritmo PSP numa máquina paralela real. Para diferentes aplicações paralelas sintéticas foram considerados dois casos distintos: (i) com o uso do algoritmo PSP e (ii) sem o uso do PSP. Foram gerados resultados numéricos do tempo de resposta com o objetivo de delimitar as circunstâncias em que o algoritmo PSP alcança melhor desempenho, de acordo com parâmetros da aplicação sintética e da política de migração.

A descrição do algoritmo PSP encontra-se na próxima seção, assim como suas principais propriedades. Na Seção 3, descreve-se o ambiente de teste construído para avaliação de desempenho deste algoritmo sobre uma máquina paralela real. Na Seção 4 apresentam-se as configurações dos problemas teste usadas e os principais resultados numéricos comparativos até o momento. A Seção 5 finaliza o artigo com algumas considerações e as principais conclusões.

2 O Algoritmo PSP

Considere, inicialmente, que tarefas não possam migrar para processadores nos quais já tenham estado anteriormente durante a execução da aplicação em questão. Sejam u e v duas tarefas que se comunicam e que se encontram no momento nos processadores p e q, respectivamente. Considere a migração de v para o processador q' ($q' \neq q, p$). Enquanto isso, p continua enviando ao processador q todas as mensagens de u destinadas à tarefa v, ao mesmo tempo em que o processador q re-envia ao processador q' todas as mensagens recebidas por ele, destinadas a v. Neste caso, a propriedade FIFO ainda se mantém. Da mesma forma, considere a migração de u para outro processador p'. Toda mensagem enviada por u é roteada primeiramente através de p. Novamente, neste caso, a propriedade FIFO é mantida. Se continuamente essas tarefas migrarem para outros processadores, este esquema de re-envio será suficiente para manter a ordem FIFO das mensagens entre estas duas tarefas.

Não se pode esperar, no entanto, que este esquema suporte uma computação eficiente no caso de uma grande seqüência de migrações de suas tarefas, pois, neste caso, as mensagens tendem a seguir caminhos cada vez mais longos antes de sua eventual entrega à tarefa destino. Porém esta observação serve ao propósito de se mostrar que a presença de uma seqüência de processadores, inicialmente com dois (p e q), aumenta gradativamente através da adição de novos processadores (p' e q'), conforme a migração de u e v. O algoritmo PSP proposto em [1], além de permitir a migração de tarefas para quaisquer processadores (mesmo para aqueles onde elas já residiram anteriormente), realiza o encurtamento dessa seqüência de processadores sempre que uma tarefa migra para outro processador, através da retirada do processador original da seqüência, ou seja, mantendo o tamanho da seqüência limitado.

2.1 Operação do Algoritmo

Seja In_u o conjunto de tarefas das quais a tarefa u recebe mensagens durante sua computação e Out_u o conjunto de tarefas para as quais u envia mensagens duran sua computação. Por questões de simplicidade assume-se que estes conjuntos são fixos porém, em [1], descreve-se como lidar com mudanças dinâmicas nestes conjuntos. Associado a cada par de tarefas u, v que se comunicam está um conjunto de processadores que chamaremos de *pipe* (ou canal) para enfatizar a ordem FIFO seguida pelas mensagens enviadas por esse *pipe* (chamado de $pipe(u, v)$). Este *pipe* é uma seqüência de processadores com a propriedade de armazenar (ou ter armazenado) pelo menos uma das duas tarefas. Além disso, u reside no primeiro processador do *pipe* e v no último. Quando u e v (ou ambos) migram para outro(s) processador(es), alongando o *pipe*, o algoritmo distribuído remove do *pipe* o(s) processador(es) onde a(s) tarefa(s) que migra(m) residia(m). Processadores adjacentes no *pipe* não são necessariamente vizinhos, e inicialmente o *pipe* contém no máximo dois processadores.

Um processador p mantém, para cada tarefa u nele residente e para toda tarefa $v \in Out_u$, uma variável $pipe_p(u, v)$ que armazena sua visão do $pipe(u, v)$. A inicialização dessa variável deve ser feita de acordo com a localização inicial das tarefas nos processadores. Além disso, para cada tarefa v uma variável $proc_p(v)$ é utilizada em p para indicar o processador onde se acredita que a tarefa v esteja localizada. Essa variável também é inicializada com o valor inicial da alocação de tarefas a processadores sendo utilizada como processador destino de mensagens enviadas a v por outras tarefas. Uma relação importante entre as variáveis em p é a seguinte: se $v \in Out_u$ então $pipe_p(u, v) = \langle p, \cdots, q \rangle$ se, e somente se, $proc_p(v) = q$. Mensagens enviadas a $proc_p(v)$ são na realidade enviadas ao $pipe(u, v)$. Essa redundância é mantida no algoritmo por uma questão de clareza.

Inicialmente, o algoritmo é descrito de maneira informal considerando-se um único *pipe* - $pipe(u, v)$ - onde p é o processador onde reside u e q o processador onde reside v. Quando u migra de p para outro processador p', o processador p envia uma mensagem $flush(u, v, p')$ sobre o $pipe_p(u, v)$. Essa mensagem informa ao processador q (ou q' no caso da tarefa já ter migrado) que u reside agora em p', e também "empurra" todas as mensagens que estão em trânsito de u para v através do *pipe*. Quando essas mensagens chegam em q ou q' o pipe está vazio e a nova localização

de u pode ser atualizada. Uma mensagem $flushed(u, v, q)$ (ou $flushed(u, v, q')$) é então enviada diretamente para p', que atualiza a informação sobre a localização de u e sua visão do *pipe* através da alteração do conteúdo do $pipe_{p'}(u, v)$.

Durante todo esse processo, a tarefa u permanece bloqueada e, portanto, não realiza qualquer computação ou migração. O algoritmo pode ter sua operação ativada por q no caso da migração de v para q', e então v deve ser também bloqueada. Nesse caso, uma mensagem $flush\_request(u, v)$ é enviada por q para p, que por sua vez entra no procedimento de esvaziamento (*flushing*), descrito anteriormente, após o bloqueio da tarefa v. Existe também a possibilidade que p e q iniciem o algoritmo concorrentemente. Isso acontece quando u e v migram, para p' e q', respectivamente, de forma concorrente; ou seja, antes que a informação sobre a migração da outra tarefa seja recebidas. Os procedimentos são exatamente os mesmos, com uma única restrição de que um $flush(u, v, p')$ não será enviado novamente após o recebimento de um $flush\_request(u, v)$, caso este já tenha sido enviado anteriormente. Essa última situação é mostrada na Figura 1 na qual "eixos locais de tempo" para cada processador envolvido aparecem (como linhas pontilhadas) junto às mensagens enviadas e recebidas (representadas por linhas contínuas).

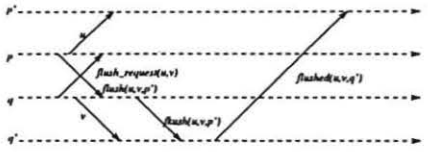

Figura 1: Migração concorrente de u e v.

Quando uma tarefa u migra de p para p', o procedimento que foi descrito é executado concorrentemente para cada $pipe(u, v)$ tal que $v \in Out_u$ e cada $pipe(v, u)$ tal que $v \in In_u$. A tarefa u só pode continuar sua execução em p' (e, então, talvez migrar novamente), depois que todos os *pipes* $pipe(u, v)$ tais que $v \in Out_u$ e $pipe(v, u)$ tal que $v \in In_u$ tenham sido esvaziados ("flushed"), e então é dita ativa (caso contrário esta encontra-se inativa e não pode migrar). A tarefa u pode também tornar-se inativa depois do recebimento de um $flush\_request(u, v)$ quando residindo em p. Nesse caso, somente depois que $pipe_p(u, v)$ for atualizado, a tarefa u torna-se novamente ativa.

A seguir, o algoritmo que será executado é descrito para um processador genérico p. As variáveis empregadas relativas à tarefa u são as seguintes. A variável lógica $active_u$ (inicializada com *true*), é utilizada para indicar se a tarefa u está ativa. Dois contadores $pending\_in_u$ e $pending\_out_u$ são utilizados para registrar o número de pipes que precisam ser esvaziados ("flushed") antes que u possa tornar-se novamente ativa. O primeiro contador se refere ao $pipe(v, u)$ tal que $v \in In_u$ e o outro contador ao $pipe(u, v)$ tal que $v \in Out_u$. Ambos os contadores são inicializados com zero. Para cada $v \in In_u$, a variável lógica $pending\_in_u(v)$ (inicializada com *false*) indica se é necessário realizar um *flush* no $pipe(v, u)$ para que u se torne ativa. Mensagens destinadas a u oriundas de outras tarefas são mantidas em uma fila (em ordem FIFO) chamada de $queue_u$. Constantes e variáveis carregando o índice u em seus nomes podem ser vistas como parte do registro de ativação da tarefa u e para tanto migram juntamente com u sempre que esta migra.

O algoritmo PSP para o processador p pode ser descrito como sendo o seguinte conjunto de seis ações atômicas.

1. Recepção de uma mensagem destinada a tarefa u:

```
if proc_p(u) = p then
   adicione msg a queue_u
else    envie msg a proc_p(u);
```

2. Para a migração da tarefa u para o processador q:

```
active_u := false;
for all v ∈ Out_u do
   begin
      envie flush(u, v, q) a proc_p(v);
      pending_out_u := pending_out_u + 1
   end;
for all v ∈ In_u do
   begin
      envie flush_request(v, u) a proc_p(v);
      pending_in_u := pending_in_u + 1;
      pending_in_u(v) := true
   end;
proc_p(u) := q;
envie u a q;
```

3. Recepção da tarefa u:

```
proc_p(u) := p;
```

4. Recepção de $flush(v, u, q)$:

```
if proc_p(u) = p then
begin
   if proc_p(u) ≠ q then
      proc_p(v) := q;
   envie flushed(v, u, p) a q;
   if pending_in_u(v) then
   begin
      pending_in_u(v) := false;
      pending_in_u := pending_in_u − 1;
      active_u := (pending_in_u = 0) and (pending_out_u = 0)
   end
end
else
   envie flush(v, u, q) a proc_p(u);
```

5. Recepção de um $flush\_request(u, v)$:

```
if proc_p(u) = p then
begin
   active_u := false;
   envie flush(u, v, p) a proc_p(v);
   pending_out_u := pending_out_u + 1
end;
```

6. Recepção de um $flushed(u, v, q)$ se $proc_p(u) = p$, ou quando $proc_p(u)$ se torna igual a p depois da recepção de um $flushed(u, v, q)$:

   $proc_p(v) := q$;
   $pipe_p(u, v) := \langle p, q \rangle$;
   $pending\_out_u := pending\_out_u - 1$;
   $active_u := (pending\_in_u = 0)$ **and** $(pending\_out_u = 0)$;

A maior parte da simbologia utilizada pelo algoritmo é auto-explicativa. Note-se, entretanto, que a condição na qual a ação **6** deve ser executada é mais elaborada que as demais ações atômicas. A razão para isso reside no fato de que uma vez que a mensagem $flushed(u, v, q)$ atinja p antes da tarefa u que migra, esta deva ser armazenada para que seja tratada quando a tarefa chegar, ou seja, até que $proc_p(u)$ se iguale a p através da ação **3**. Isso é de fato possível, haja visto que os caminhos seguidos por uma tarefa u que migra e a mensagem $flushed(u, v, q)$ para alcançar p não são necessariamente os mesmos e, portanto, não há uma implicação da ordem FIFO entre suas chegadas em p.

Propriedades Três propriedades importantes do algoritmo apresentado anteriormente são descritas a seguir. Estão relacionadas à complexidade e à corretude do algoritmo. As provas de tais propriedades podem ser vistas com detalhe em [1].

Teorema 1 Para qualquer par de tarefas u e v que se comuniquem, tais que $v \in Out_u$, mensagens enviadas por u para v são entregues em ordem FIFO.

Lema 2 Para quaisquer pares de tarefas u e v que se comuniquem, tais que $v \in Out_u$, e em todos os estados globais consistentes, $|pipe(u, v)| \leq 4$.

Teorema 3 Para a migração concorrente de um conjunto de K tarefas, o algoritmo requer $O(nm_K)$ mensagens entre processadores e $O(n)$ de tempo, onde n é o número de processadores do sistema e m_K é o número de pares de tarefas que se comunicam (considerando aqueles que envolvem pelo menos uma das K tarefas).

3 O Ambiente para Experimentos Computacionais

O ambiente para experimentação do algoritmo PSP se compõe de dois programas construídos a partir de quatro componentes básicos: (i) uma aplicação paralela sintética parametrizada, (ii) política de migração de tarefas, (iii) esquema de entrega ordenada de mensagens na ausência do algoritmo PSP e (iv) o algoritmo PSP propriamente dito. A seguir os detalhes de implementação de cada um desses componentes são descritos, bem como a interação entre mesmos.

3.1 A Aplicação Paralela Sintética

A aplicação paralela sintética foi elaborada no intuito de se criar uma plataforma sobre a qual atuem a política de migração das tarefas e os algoritmos de entrega de

mensagens, de tal maneira a simular o comportamento de uma aplicação real num ambiente paralelo. Essa aplicação sintética segue um modelo de computação idealizado inicialmente em [3]. Este modelo é de fato uma espécie de formato geral para toda computação baseada em troca de mensagens do tipo reativa, tendo sido projetado para detectar a terminação de computações deste tipo. A escolha deste modelo no contexto do nosso trabalho se deve ao fato de que toda a computação por troca de mensagens pode ser reduzida a este formato reativo. Ao variar os parâmetros envolvidos na descrição desta aplicação sintética sobre um donínio suficientemente amplo de valores, será possível prever diversos aspectos do comportamento de diferentes aplicações paralelas baseadas em trocas de mensagens, inclusive o da própria simulação do tipo "Time Warp", mencionado anteriormente, que apresenta um caráter altamente irregular. O uso de "benchmarks" paralelos já existentes foi descartado por dois motivos essenciais:(i)"benchmarks" não são comprovadamente melhores para esse tipo de estudo, e (ii) os programas usados por estes "benchmarks" são em geral bastante simples e assim a propriedade FIFO pode nem ser importante.

A aplicação aqui implementada, de acordo com seu caráter sintético e parametrizado, não realiza de fato computação real. O objetivo é representar uma aplicação real, de forma controlada, através de eventos inerentes a qualquer processamento paralelo, necessários a avaliação de desempenho do algoritmo em questão, tais como: criação e término de tarefas, simulação do tempo de computação para cada tarefa e comunicação entre tarefas através de troca de mensagens.

Considere-se então, uma aplicação descrita através de um grafo direcionado, eventualmente cíclico. Os arcos deste grafo, no entanto, não representam relações de precedência entre tarefas, determinante de uma ordem parcial de execução. Apesar de direcionado, o grafo representa um conjunto de tarefas (nodos) que se comunicam ao longo do seu tempo de vida, segundo algumas regras pré-estabelecidas. Os arcos direcionados determinam apenas a direção de envio das mensagens entre tarefas. Para cada nodo u_i define-se o conjunto de nodos In_{u_i} composto pelas tarefas que potencialmente enviam mensagens à tarefa u_i. Da mesma forma, o conjunto Out_{u_i} é formado pelas tarefas que potencialmente recebem mensagens de u_i. A tarefa u_0 é diferenciada das demais por possuir $In_{u_0} = 0$ e ter funções especiais durante a execução da aplicação sintética, como por exemplo, determinar o início e o fim da computação como um todo. Os demais nodos (tarefas) são chamados de nodos *internos*, cujo estado no começo da computação é dito *neutro*, pois não simulam ainda qualquer computação.

O início da computação é marcado pela atividade da tarefa u_0 que, aleatoriamente, envia mensagens para uma ou mais das tarefas pertencentes a Out_{u_0}. Ao receber uma mensagem de u_0, os nodos internos saem do estado neutro, passando a simular computação seguida de envio, também aleatório, de novas mensagens aos seus respectivos conjuntos Out_{u_i}. Esse modelo computacional apresenta um caráter reativo, no sentido de ser o recebimento de mensagens que viabiliza a computação e o envio de novas mensagens por parte de qualquer tarefa. Cada tarefa possui, ao longo do seu tempo de vida, um número finito de mensagens que serão enviadas a outras tarefas. Segundo o modelo computacional, existe um momento ao longo da execução onde tarefas não mais terão mensagens a serem enviadas, e isso deve marcar o final da execução da aplicação. Para que o término da aplicação seja reconhecido, utiliza-se um eficiente esquema de sinalização, entre as tarefas, através

de mensagens ditas *de controle*. Assim, as arestas do grafo de tarefas marcam a direção das mensagens ditas *de computação* que não possuem função de controle e que determinam que a tarefa destino realize nova computação (sintética) e envio de novas mensagens de computação a outras tarefas do grafo. As mensagens de controle que têm funções de sinalização trafegam no sentido contrário àquele determinado pelas arestas do grafo. Assume-se que sobre cada aresta haverá, até o final da computação, o mesmo número de mensagens de controle e de computação. Associa-se a cada aresta ao longo da execução da aplicação sintética um número, *expected*, que identifica o número total de mensagens de controle que um dado nodo espera receber. Quando esse valor chega a 0 diz-se que a tarefa alcançou o estado neutro e sua computação chegou ao fim. No momento que o valor *expected* da tarefa u_0 chega a 0, identifica-se o término da aplicação, logo, nenhum nodo realizará qualquer outra computação.

Esse tipo de modelo pode ser formalizado algoritmicamente, como se segue. A aplicação sintética funciona através do uso de dois tipos de mensagens:

forward – mensagem que dispara na tarefa recebedora o processo de simulação de computação, ativando-a.

backward – mensagem de controle que funciona como um *acknowledgement* enviado à tarefa que enviou um **forward** anteriormente. Logo, existe um **backward** associado a cada **forward** enviado.

De acordo com o tipo de mensagem recebida a tarefa realiza diferentes ações:

forward –

- Se $expected > 0$, então envie um **backward** imediatamente.
- Caso contrário, guarde a origem da mensagem em uma variável, *parent*, e faça a computação local, possivelmente enviando novos **forwards**. Ao término desta computação, envie um **backward** a *parent* se $expected = 0$. Note que *expected* é sempre incrementado quando se envia um novo **forward**.

backward –

- Decremente *expected*.
- Se $expected = 0$, então envie **backward** a *parent*.

No caso da tarefa u_0, a chegada do último **backward** sinaliza o término da aplicação e garante que todos os demais nodos internos se encontram novamente em estado neutro e podem ser corretamente finalizados.

3.2 Política de Migração

A política de migração adotada para o ambiente é similar àquela proposta de Suen e Wong em [12]. Esta política baseia-se no processo de demanda de carga (novas tarefas) por parte de processadores considerados abaixo de um limite mínimo de carga. Antes da descrição propriamente dita, é importante compreender o conceito de carga no contexto da aplicação paralela sintética em execução.

Conceito de carga. Cada processador do sistema possui um certo número de tarefas. Cada tarefa u por sua vez possui um vetor de elementos inteiros, chamado de *vetor_de_carga*(u), cujo número de elementos é dado pelo número total de processadores envolvidos na execução da aplicação sintética. Cada processador possui também um vetor de carga, resultante do somatório dos vetores de carga das tarefas correntemente alocadas a ele. Os vetores de carga das tarefas e de cada processador serão atualizados a cada recebimento/envio de mensagens **forward** ou **backward** de acordo com regras pré-estabelecidas. Note-se que a atualização dos vetores de carga durante o envio de mensagens deve ser feita de forma análoga à sua recepção. No sentido de ilustrar essa atualização, considere a chegada de uma mensagem destinada à tarefa u.

forward – A posição do *vetor_de_carga*(u) relativa ao processador de origem da mensagem é incrementada de um. O vetor de carga do processador onde reside u é também atualizado de acordo.

backward – A posição do *vetor_de_carga*(u) relativa ao processador de origem da mensagem é decrementada de um. Também o vetor de carga do processador onde reside u é atualizado de acordo.

Dessa forma, observa-se que a carga de um processador depende do número de tarefas ativas (que não estejam no estado neutro) que ele possui. Neste trabalho, dois esquemas de alocação de tarefas a processadores foram considerados:

- alocação segundo um mecanismo *round-robin* (as tarefas são distribuídas uniformemente através dos processadores); e
- alocação aleatória de tarefas.

Níveis de Carga. Dois parâmetros são utilizados na determinação dos níveis de carga em cada um dos processadores: (i) LIM_INF_CARGA - limite inferior de carga (valor de carga mínima), e (ii) LIM_SUP_CARGA - limite superior de carga (valor de carga máxima). De acordo com esses valores, as variações de carga que ocorrem em um determinado processador alteram seu estado:

1. *disponível*: valores do vetor de carga abaixo de LIM_INF_CARGA.
2. *normal*: valores dentro do intervalo entre LIM_INF_CARGA e LIM_SUP_CARGA.
3. *sobrecarregado*: valores do vetor de carga acima de LIM_SUP_CARGA.

Protocolo de migração. Um processador p, no estado *disponível* em função dos valores do seu vetor de carga, envia uma mensagem, **available**, comunicando esse fato a outros processadores que possivelmente determinarão que uma de suas tarefas migre para o processador p em disponibilidade. Sempre que há variação na carga de um processador p, seu vetor de carga é devidamente analisado para que decisões de migração sejam tomadas. Considere as seguintes condições:

- todos os valores do vetor de carga do processador p são menores que o parâmetro LIM_INF_CARGA;
- o processador p não está participando como nodo *disponível* de nenhum outro protocolo de migração com outros processadores, ou seja, todas as respostas referentes a possíveis mensagens **available** enviadas anteriormente já foram recebidas; e
- o processador p não está participando de qualquer processo de migração para envio de suas tarefas a outros processadores.

Caso todas as condições acima sejam satisfeitas, o processador p pode ser classificado como *disponível*. É nesse momento que p difunde a mensagem **available** a outros processadores, para que se inicie o processo de negociação anterior à migração efetiva de tarefas para p.

Ao receber um **available**, um processador q verifica seu vetor de carga na expectativa de que algum de seus elementos esteja acima do LIM_INF_CARGA. Em caso positivo, o processador passa para a etapa de aplicação do *critério de escolha da tarefa que deve migrar*, onde, se possível, uma tarefa de q será escolhida para ser enviada a p. O processador q neste momento poderá enviar a p: (i) uma mensagem **proposta_de_migração**, com a informação da identidade e do vetor de carga da tarefa escolhida ou (ii) uma mensagem de **nada_a_enviar**, caso não tenha qualquer tarefa que possa ser migrada para q.

A recepção de uma mensagem **proposta_de_migração** de q, faz com que o processador p some ao seu vetor de carga o vetor de carga da tarefa (definida na proposta) enviado por q. Se o vetor resultante apresentar pelo menos um de seus elementos agora na faixa de normalidade de carga (valor entre os parâmetros LIM_INF_CARGA e LIM_SUP_CARGA), p envia a q uma mensagem **ack_proposta**, determinando o seu acordo em receber a tarefa escolhida por q. Depois de ter enviado uma mensagem **ack_proposta**, p não poderá participar de nenhum outro processo de migração até que este seja concluído.

Critério para escolha da tarefa que deve migrar. Dois requisitos são utilizados na escolha da tarefa a ser enviada em uma proposta para um nodo disponível. O primeiro requisito é fundamental e se resume ao fato de que a tarefa escolhida deve estar presente e ativa no processador origem. Ou seja, seu vetor de carga não pode ser nulo. O segundo requisito é opcional e se refere à prioridade dada às tarefas que já tenham migrado anteriormente. O objetivo dessa prioridade é o de estimular o crescimento dos *pipes* de comunicação. Assim sendo, as tarefas escolhidas são aquelas que já migraram pelo menos alguma outra vez.

3.3 Ordenação de mensagens na ausência do PSP

Como já foi mencionado anteriormente, o algoritmo PSP diminui automaticamente o tamanho dos *pipes* de comunicação entre as tarefas que se comunicam. A ordenação de mensagens na ausência do PSP é realizada através da manutenção - sem qualquer encurtamento – desses *pipes* toda vez que as tarefas migram. Logo,

enquanto o tamanho máximo dos *pipes* na aplicação com o PSP é igual a 4, sem o PSP o tamanho máximo do *pipe* fica limitado a $n-1$, onde n é igual ao número de processadores que executam a aplicação sintética. A escolha desse método para manutenção da ordenação das mensagens no caso sem o PSP deve-se à simplicidade de sua implementação e independência de parâmetros.

Uma outra opção para realização dessa implementação seria a parametrização de valores que estariam vinculados ao tempo de re-ordenação de tarefas no processador destino. O problema de uma implementação como essa reside na dificuldade de se parametrizar, de forma justa esses tempos de re-ordenação. Além disso, essa opção também traz à tona as questões críticas de espaço de armazenamento limitado nos processadores de destino, onde se dá a re-ordenação.

4 Resultados Experimentais

Método Estatístico. Para uma mesma configuração inicial de teste, a aleatoriedade do modelo de computação usado na aplicação sintética, bem como a carga total variável imposta à máquina paralela real utilizada, produzem tempos de resposta distintos a cada execução da aplicação. Em função desse não determinismo na medição desses tempos de resposta, foram colhidas diversas amostras e dado um tratamento estatístico aos resultados obtidos dos tempos de resposta para os casos com e sem o PSP. Foram usadas as seguintes medidas para efeitos de comparação: média aritmética do tempo de resposta e desvio padrão percentual considerando todos os valores de cada amostra.

Impacto de Parâmetros Relevantes. Diversos parâmetros caracterizam o ambiente de teste. Os valores desses parâmetros foram determinados após a realização de testes exaustivos, nos quais a sensibilidade do tempo de resposta da aplicação pôde ser avaliada. Vale ressaltar que as configurações de teste foram escolhidas tendo como alvo um maior número de migrações, bem como um aumento no tamanho dos *pipes*, para que a eficiência do algoritmo PSP pudesse ser devidamente avaliada. A seguir, descrevemos a influência de cada um desses parâmetros da execução da aplicação:

número de tarefas – observou-se que um maior número de migrações é alcançado elevando-se o número de tarefas da aplicação sintética.

número de processadores – apesar de não acarretar necessariamente um maior número de migrações, o aumento do número de processadores implica no crescimento dos *pipes* de comunicação das tarefas que migram.

grafo de tarefas – a maior densidade (número de arcos) do grafo de tarefas não altera, com raras exceções, o número total de migrações nem tampouco o tamanho dos *pipes*.

limites de carga – os já mencionados limites de carga (LIM_INF_CARGA e LIM_SUP_CARGA) influenciam diretamente o número total de migrações e o número médio de migrações por tarefa. Isso acontece porque o estado dos nodos (disponível, normal, sobrecarregado) é dependente desses limites.

número de mensagens por tarefa – de forma análoga ao que acontece com os limites de carga, o número de mensagens que as tarefas enviam durante o decorrer da aplicação atua diretamente sobre o tempo de execução das mesmas.

alocação inicial de tarefas a processadores – a alocação inicial de tarefas a processadores pode implicar em um maior desbalanceamento inicial e conseqüentemente em um maior número de migrações durante um estágio inicial de execução da aplicação.

homogeneidade – parâmetros como limites de carga e número de mensagens por tarefa podem ser distribuídos entre os processadores de forma homogênea ou heterogênea. Chamamos de distribuição homogênea quando esses parâmetros recebem valores idênticos em todos os processadores. Para o caso heterogêneo, cada processador possui seu próprio conjunto distinto de parâmetros. Em determinadas situações essa característica pode implicar no crescimento do número de migrações como também do tamanho dos *pipes* de comunicação entre as tarefas.

Resultados numéricos. Os experimentos computacionais foram realizados utilizando-se a máquina paralela IBM SP e os respectivos programas desenvolvidos em C mais MPI. Para este tipo de arquitetura, pode-se considerar que o custo de comunicação interprocessador independe do par de processadores envolvidos na troca de mensagens. Os grafos de tarefas que descrevem a aplicação sintética são conexos e foram gerados aleatoriamente. Diversos conjuntos de testes foram realizados para 3 valores distintos do número de processadores (3, 5 e 6). Os resultados das comparações entre as duas implementações (com e sem o PSP) são descritos nas Tabelas 1 a 5. Cada uma dessas tabelas contém 5 grandezas:

ntarefas – número de tarefas da aplicação;

tempo – tempo médio (em segundos) de duração da aplicação;

migr – número médio de migrações;

tam_pipe – tamanho máximo do pipe de comunicação entre as tarefas; e

desvio% – desvio padrão percentual.

Podemos perceber que quando o número de processadores não é muito elevado (Tabelas 1 e 2), o tempo de resposta da aplicação com o algoritmo PSP é razoavelmente maior que o tempo da aplicação sem o PSP. Isso se deve ao fato de que o *overhead* (custo) de envio e tratamento das mensagens de controle do protocolo é maior que o *overhead* de retransmissão das mensagens na aplicação sem o PSP, considerando que o tamanho dos *pipes* de comunicação, nesse último caso, alcança no máximo um valor igual a 2. Ainda nessas tabelas, observa-se que o número médio de migrações é maior nas implementações com o PSP, porque o *overhead* da troca de mensagens de controle introduzido pelo PSP implica em uma variação gradual da carga nos processadores. Logo, novas situações de migração estão mais propensas a acontecer.

Quando o número de processadores aumenta (Tabela 3), os tempos de resposta tornam-se mais próximos. Nessa tabela, o tamanho máximo dos *pipes* não alcança o valor máximo $(n-1)$ descrito na Seção 3.3, pois o número de migrações total não é muito elevado. O mesmo não acontece na Tabela 4, onde os *pipes* atingem o valor $n-1$ e os tempos de resposta para os dois casos ficam ainda mais parecidos.

Finalmente, quando o número de processadores chega a 6 (Tabela 5), os tempos praticamente se igualam. Pode-se dizer que nesse momento o *overhead* relacionado à inclusão do algoritmo PSP se iguala ao *overhead* de retransmissão de mensagens, para o caso sem o PSP, quando então o tamanho dos diversos *pipes* de comunicação aumenta. Nesse momento o número de migrações também se iguala. É possível vislumbrar, a partir desses experimentos, que com um número ainda maior de processadores, sob um número suficiente de migrações, o protocolo PSP apresente um desempenho ainda mais elevado com relação ao caso sem reordenação automática. Testes com um número maior que 6 de processadores não foram possíveis devido a limitações das máquinas paralelas disponíveis cujos recursos são fortemente compartilhados entre um número elevado de usuários

Pode se observar nas Tabelas 1 a 5, que os casos da aplicação com PSP mostram valores com desvio padrão mais elevado em comparação àqueles vistos para o caso sem o PSP. A inclusão de novas mensagens (de controle) ao longo da execução da aplicação devido à presença do algoritmo PSP altera a ordem de tratamento das mensagens de computação inerentes à aplicação sintética. Logo, a aleatoriedade é ainda maior entre diferentes execuções para configurações de teste idênticas. No caso da aplicação sem o PSP, essa maior aleatoriedade surge apenas quando os *pipes* de comunicação entre tarefas ficam maiores.

	ntarefas	tempo	migr	tam_pipe	desvio%
Com PSP	25	1680	28	2	5.88
Sem PSP	25	1580	19	2	0.10

Tabela 1: Para uma alocação *round-robin* com 3 processadores e uma distribuição homogênea de parâmetros entre os processadores.

	ntarefas	tempo	migr	tam_pipe	desvio%
Com PSP	25	1560	31	2	4.93
Sem PSP	25	1495	23	2	0.11

Tabela 2: Para uma alocação aleatória com 3 processadores e uma distribuição heterogênea de parâmetros entre os processadores.

5 Observações Finais

O artigo descreve o PSP, um algoritmo distribuído assíncrono para a manutenção da ordem FIFO de entrega de mensagens entre tarefas que migram em um sistema

	ntarefas	tempo	migr	tam_pipe	desvio%
Com PSP	25	1682	23	3	2.58
Sem PSP	25	1615	17	3	1.60

Tabela 3: Para uma alocação *round-robin* com 5 processadores e uma distribuição homogênea de parâmetros entre os processadores.

	ntarefas	tempo	migr	tam_pipe	desvio%
Com PSP	50	2413	40	4	2.73
Sem PSP	50	2392	36	4	1.02

Tabela 4: Para uma alocação *round-robin* com 5 processadores e um maior número de tarefas e uma distribuição homogênea de parâmetros entre os processadores.

MIMD com memória distribuída. A idéia essencial por trás do algoritmo é que a comunicação entre tarefas ocorre através da transmissão de mensagens através de "pipes" de processadores, os quais se tornam maiores com as migrações de tarefas. O que o algoritmo faz é encurtar esses "pipes" tão logo haja migração de tarefas, através da remoção do processador origem da tarefa que migra, de todos os "pipes" relacionados a essa migração.

Um ambiente de teste foi proposto e usado para experimentação e avaliação do algoritmo PSP. O ambiente é composto de uma aplicação sintética, diferentes políticas de migração e uma técnica de entrega ordenada através de manutenção de *pipes* longos para o caso onde o PSP não é utilizado. Neste ambiente, são estudados de forma comparativa, sob diferentes configurações de paraâmetros, os casos com e sem o PSP. Os resultados numéricos destes experimentos baseiam-se nos tempos de resposta da aplicação sintética, no intuito de encontrar as condições sob as quais o algoritmo PSP produz execuções mais eficientes. A análise destes resultados nos mostra que o algoritmo PSP apresenta melhor desempenho à medida que os *pipes* de comunicação aumentam em função de diversas migrações consecutivas de uma mesma tarefa.

	ntarefas	tempo	migr	tam_pipe	desvio%
Com PSP	66	2211	43	4	2.13
Sem PSP	66	2201	43	5	2.40

Tabela 5: Para uma alocação *round-robin*, com 6 processadores e uma distribuição homogênea de parâmetros entre os processadores.

Referências

[1] V.C. BARBOSA e S.C.S. PORTO, "An algorithm for FIFO delivery among migrating tasks", *Information Processing Letters* **53** (1995), 261–267.

[2] K.M. CHANDY e L. LAMPORT, "Distributed snapshots: determining global states of distributed systems", *ACM Trans. on Computer Systems* **3** (1985), 63–75.

[3] E.W. DIJKSTRA e C.S. SCHOLTEN, "Termination detection for diffusion computations", *Information Processing Letters* **11** (1980), 1-4.

[4] M.R. ESKICIOĞLU e L.-F. CABRERA, "Process migration: an annotated bibliography", Technical Report RJ 7935 (72918), IBM Research Division, San Jose, CA, 1991.

[5] M. GERLA e L. KLEINROCK, "Flow control protocols", in P. E. Green, Jr. (Ed.), *Computer Network Architectures and Protocols*, Plenum Press, New York, NY, 1982, 361–412.

[6] Intel Supercomputer Systems Division, comunicação interna sobre iPSC/860 and Paragon systems.

[7] D.R. JEFFERSON, "Virtual time", *ACM Trans. on Programming Languages and Systems* **7** (1985), 404–425.

[8] L. LAMPORT e N.A. LYNCH, "Distributed computing: models and methods", in J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Vol. B, The MIT Press, Cambridge, MA, 1990, 1156–1199.

[9] L.M. NI e P.K. MCKINLEY, "A survey of wormhole routing techniques in direct networks", *IEEE Computer* **Jan.** (1993), 62–76.

[10] J.K. OUSTERHOUT, A.R. CHERENSON, F. DOUGLIS, M.N. NELSON, e B.B. WELCH, "The Sprite network operating system", *IEEE Computer* (1988), 23–36.

[11] T.M. RAVI e D. JEFFERSON, "A basic protocol for routing messages to migrating processes", *Proc. Int. Conf. on Parallel Processing*, Saint Charles, IL, 1988, 188–196.

[12] T.T.Y. SUEN e J.S.K.WONG, "Efficient Task Migration Algorithm for Distributed Systems", *IEEE Transactions on Parallel and Distributed* **3 July** (1992), 488–499.

[13] M. THEIMER, K. LANTZ e D. CHERITON, "Preemptable remote execution facilities for the V-System", *Proc. 10th Symp. on Operating System Principles*, 1985, 2–12.