Uma Ferramenta para Atribuição de Relógios Lógicos a Execuções Distribuídas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Vitor Almeida dos Santos
`vitor@lia.ufc.br` e aprovada pela Banca Examinadora.

Fortaleza, 22 de março de 2004.

Ricardo Cordeiro Corrêa (DC/UFC)

Dissertação apresentada ao Mestrado em Ciência da Computação, UFC, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Mestrado em Ciência da Computação

Universidade Federal do Ceará

Uma Ferramenta para Atribuição de Relógios Lógicos a Execuções Distribuídas

Vitor Almeida dos Santos

vitor@lia.ufc.br

Fevereiro de 2004

Banca Examinadora:

- Ricardo Cordeiro Corrêa (DC/UFC)
- Valmir Carneiro Barbosa (COPPE/UFRJ)
- Manoel Bezerra Campêlo Neto (DEMA/UFC)

Agradecimentos

Agradeço ao meu pai e à minha mãe (*in memorian*) pela liberade e apoio e à Carla pelo companheirismo. Agradeço ao Prof. Valmir Barbosa por ter-me recebido e orientado na UFRJ em agosto de 2002 e por estar na banca da defesa desta dissertação. Também ao Prof. Manoel Campêlo, por compor a banca e, especialmente, ao Prof. Ricardo Corrêa pela inestimável orientação sempre acompanhada de interesse e incentivo. Finalmente, agradecimentos à CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - pela bolsa cedida e ao NACAD - Núcleo de Atendimento em Computação de Alto Desempenho, UFRJ - por disponibilizar seu *cluster* na fase inicial de experimentos desta dissertação.

"A mente que se abre a uma nova idéia jamais voltará ao seu tamanho original."
A. Einstein

Sumário

1

Introdução

O objeto de estudo desta dissertação pertence à área da ciência da computação conhecida como *Algoritmos Distribuídos*. Um algoritmo distribuído é um conjunto de algoritmos seqüenciais os quais devem ser executados de forma concorrente. Em um algoritmo distribuído, chamamos cada um destes algoritmos seqüenciais de *tarefa*. As tarefas, no decorrer da execução de um algoritmo distribuído, podem interagir por meio de troca de mensagens. Um algoritmo seqüencial pode ser entendido como um caso particular de algoritmo distribuído que só possui uma tarefa.

Ao ser executado, um programa implementado a partir de um algoritmo distribuído gera um processo para cada tarefa. É desejável, portanto, que um programa distribuído seja executado em um ambiente que possua diversos processadores para que suas tarefas possam, de fato, trabalhar de forma concorrente. Tal ambiente é chamado de *sistema de memória distribuída* e é um conjunto de processadores interconectados por algum tipo de rede de comunicação física. Os processadores, os quais não compartilham memória, são interligados por canais que possibilitam o tráfego de dados de forma bidirecional. Mecanismos de roteamento e controle de fluxo são empregados para tal. Mais ainda, estes canais se comportam como uma fila, isto é, garatem que os dados serão entregues na ordem em que foram enviados (FIFO). No entanto, o tempo de entrega dos dados é

arbitrário.

Alguns sistemas de memória distribuída conhecidos até então são as redes de computadores (Internet e Intranets, por exemplo), redes de processamento paralelo e multiprocessadores de memória distribuída com um único (e virtual) espaço de endereçamento de memória. Utilizamos neste trabalho, redes de processamento paralelo e redes de computadores locais.

Em geral, as motivações para o desenvolvimento de algoritmos distribuídos se devem à possibilidade de várias unidades de processamento estarem trabalhando simultaneamente na resolução de um problema. Para resolver um problema de forma distribuída, é necessário, quase sempre, que as tarefas envolvidas se comuniquem. Vamos considerar que comunicações entre as tarefas se dão por meio de trocas de mensagens. Como exemplos, comunicações entre as tarefas são necessárias para que haja a distribuição da instância do problema, ou mesmo, para manterem atualizadas, durante a execução do algoritmo, variáveis compartilhadas. No entanto, é no fato de as tarefas se comunicarem onde se encontram duas das maiores dificuldades em tornar mais eficiente a execução de um algoritmo distribuído.

- A primeira diz respeito ao contexto físico, pois a comunicação demanda recursos do meio onde é executado o algoritmo. Veja que o acesso direto à memória de um processador no intuito de mudar o valor de uma variável é praticamente instantâneo se compararmos ao tempo que este mesmo processador levaria para comunicar um outro o valor desta variável.

- A segunda diz respeito a uma questão mais teórica e é nesta que vamos direcionar nossos estudos. O fato de várias tarefas estarem envolvidas na resolução de um problema e se comunicarem para tal implica no surgimento de uma infinidade de possibilidades de execução de um mesmo algoritmo. Mais adiante esclareceremos melhor este fato. O importante é saber que, para a maioria dos problemas, algumas execuções são mais eficientes do que outras, pois induzem uma convergência mais

rápida da solução do problema.

O nosso objetivo é criar uma ferramenta que seja capaz de atuar sobre um algoritmo distribuído no intuito de fazê-lo seguir uma boa execução.

A fim de esclarecer melhor o nosso objetivo, apresentaremos agora uma aplicação, sua versão distribuída e como nossa ferramenta atuaria sobre ela. Mostraremos também uma instância resolvida por esta aplicação e os resultados obtidos dependendo das implementações citadas acima.

Uma aplicação simples e facilmente paralelizável seria uma boa aplicação para estes fins. Escolhemos, portanto, um algoritmo iterativo geral. Assim, dado x, um vetor de estados, um algoritmo iterativo geral parte de uma atribuição inicial de estados para x e evolui de forma iterativa até que x alcance um estado, chamado estado final, que depende de um dado coeficiente de erro. Em outras palavras, $x(k+1) = f(x(k))$, até que $e(x) \leq \epsilon$, onde e é a função que deverá determinar o estado final de x de acordo com ϵ, $\epsilon \in \mathbb{R}, k \geq 0$, $f = (f_1, f_2, \ldots f_n)$ é um vetor de operadores sendo cada função f_i dependente do vetor x e $x(k)$ representa a k-ésima atribuição para o vetor x no decorrer das iterações. Segue o algoritmo seqüencial descrito acima:

Algoritmo SEQ
1. **leia** ϵ
2. **leia** valores iniciais para x
3. **faça**
4. **para** $i = 1$ **até** n **faça**
5. $x_i \leftarrow f_i(x)$
6. **enquanto** $e(x) > \epsilon$

A partir de agora, voltaremos nossa atenção para o algoritmo distribuído que resolve este problema. Esse algoritmo distribuído será executado por diversas tarefas. Assim, no momento inicial, será determinada para cada tarefa um subvetor de x pelo qual ela ficará

responsável. Tarefas que se ligam por um canal de comunicação, isto é, que podem trocar mensagens durante a execução do algoritmo são ditas vizinhas. Vamos supor que todas as tarefas são vizinhas.

Antes de mostrarmos a versão distribuída de SEQ precisamos fazer algumas considerações. De acordo com algumas características temporais, podemos ter, na verdade, duas versões distribuídas para SEQ, assim como para a maioria de outros algoritmos seqüenciais. Estas duas versões são chamadas de *síncrona* e *assíncrona*. Mais adiante, definiremos de forma detalhada os conceitos necessários para a formalização dos modelos síncrono e assíncrono de computação distribuída. Apresentaremos aqui apenas as principais diferenças entre estes modelos através dos algoritmos síncrono e assíncrono iterativos, no intuito de alcançar nosso objetivo neste momento que é mostrar o funcionamento da nossa ferramenta.

Durante a execução de um algoritmo síncrono, o tempo é dividido em intervalos. Em cada um desses intervalos, as tarefas envolvidas na execução deste algoritmo realizam uma computação e enviam mensagens para um subconjunto, que pode ser vazio, de vizinhos. Uma característica importante deste algoritmo é que todas as mensagens enviadas em um desses intervalos de tempo chegarão ao seu destino antes do início do intervalo de tempo seguinte. Isso possibilita um algoritmo síncrono se comportar de forma semelhante a um algoritmo seqüencial, com o diferencial de estar sendo executado por mais de um processador. Dessa forma, uma vantagem desse algoritmo se torna bem clara: o fato de poder herdar características do algoritmo seqüencial correspondente. O modelo síncrono, no entanto, precisa ser simulado, pois, na maioria dos casos, redes de processadores não respeitam tais restrições de tempo do modelo. É neste ponto que recai a grande desvantagem deste algoritmo, que é o alto custo computacional da simulação. A seguir, a versão distribuída síncrona de SEQ.

Algoritmo SINC$_t$

1. **leia** ϵ
2. **leia** valores iniciais para $x^t = \{x_i \mid i \in n_t\}$
3. **envie** $x^t = \{x_i \mid i \in n_t\}$ para todas as tarefas
4. **enquanto** $e(x) > \epsilon$
5. **receba** x de todas as tarefas vizinhas
6. **para** $i \in n_t$ **faça**
7. $x_i \leftarrow f_i(x)$
8. **envie** $x^t = \{x_i \mid i \in n_t\}$ para todas as tarefas vizinhas

Em SINC$_t$, n_t é o conjunto tal que a tarefa t é reponsável por x_i se e somente se $i \in n_t$. A linha 5 representa o ponto de sincronização do algoritmo. É o início de cada "tempo", quando cada tarefa espera a chegada das mensagens vindas das demais enviadas no tempo anterior.

Em um algoritmo assíncrono, as ações, isto é, as computações realizadas nas tarefas são disparadas pelo recebimento de mensagens [1]. Nele não existe a noção de intervalos de tempo existente no modelo síncrono. No momento inicial da execução do algoritmo, algumas tarefas realizam uma computação espontânea, enviam mensagens para outras tarefas e, em seguida, cada tarefa ao receber alguma mensagem realiza uma computação de acordo com o algoritmo e envia mensagens para um subconjunto, que pode ser vazio, de tarefas. Uma versão distribuída assíncrona para SEQ é a seguinte:

[1] A definição precisa para o modelo de algoritmo distribuído que adotaremos será um pouco diferente. No entanto, as propriedades discutidas nesta seção também são válidas para este modelo.

Algoritmo ASSINC$_t$

1. **leia** ϵ
2. **leia** valores iniciais para $x^t = \{x_i \mid i \in n_t\}$
3. **envie** $x^t = \{x_i \mid i \in n_t\}$ para cada tarefa
4. **faça**
5. **receba** x^t de alguma tarefa
6. **se** $e(x) > \epsilon$
7. **para** $i \in n_t$ **faça**
8. $x_i \leftarrow f_i(x)$
9. **envie** $x^t = \{x_i \mid i \in n_t\}$ para todas as tarefas
10. **enquanto** condição de terminação global não for verificada

No algoritmo síncrono SINC$_t$, cada tarefa em cada intervalo de tempo tem seu vetor x completamente atualizado, pois recebe os x_i vindos das demais tarefas do intervalo de tempo anterior. Assim, em cada intervalo de tempo, todas as tarefas têm o mesmo x e podem determinar se devem encerrar. Mais ainda, o fato de em cada intervalo de tempo o vetor x estar completamente atualizado em todas as tarefas faz com que sua convergência evolua da mesma forma que no algoritmo seqüencial.

No algoritmo assíncrono, no entanto, essa convergência evolui de forma bem diferente. Isto ocorre, pois, quando uma tarefa vai recalcular os seus x_i, ela não está com os valores de x completamente atualizados em relação à última vez em que ela o fez. Lembre-se que uma tarefa sempre atualiza seus x_i quando recebe uma mensagem, mas esta contém somente valores dos x_i da tarefa que a enviou. Logo, podemos perceber que a convergência de x nas tarefas tende a ser mais lenta, além de ser diferente para cada tarefa.

Vamos exemplificar agora a execução dos algoritmos descritos acima. Para tanto, vamos especificar estes algoritmos como algoritmos iterativos para a resolução de sistemas de equações lineares. O método utilizado será o método de Gauss-Jacobi [2].

Seja um sistema de equações lineares com 6 equações e 6 variáveis definido como segue.

$$\begin{bmatrix} 6 & 1 & 1 & 1 & 1 & 1 \\ 1 & 7 & 1 & 1 & 1 & 1 \\ 1 & 1 & 8 & 1 & 1 & 1 \\ 1 & 1 & 1 & 9 & 1 & 1 \\ 1 & 1 & 1 & 1 & 10 & 1 \\ 1 & 1 & 1 & 1 & 1 & 11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \end{bmatrix}$$

Faremos $x(0) = (1.5, 1.5, 0.5, 0.5, 1.5, 1.5)$ a atribuição inicial ao vetor x que tem como solução $(1, 1, 1, 1, 1, 1)$. Para um coeficiente de erro ϵ igual a 0.1, e uma função de erro $e(x)$ que é igual a média do módulo da diferença entre os valores de x_i das duas últimas iterações, x deverá evoluir da seguinte forma na versão seqüencial do algoritmo:

$x(0) = (1.5, 1.5, 0.5, 0.5, 1.5, 1.5)$

$x(1) = (0.9167, 0.9286, 0.8125, 0.8333, 0.9500, 0.9545)$

$x(2) = (1.0868, 1.0761, 1.0521, 1.0486, 1.0554, 1.0508)$

$x(3) = (0.9528, 0.9580, 0.9603, 0.9643, 0.9685, 0.9710)$

$x(4) = (1.0296, 1.0262, 1.0232, 1.0210, 1.0194, 1.0178)$

Nas versões distribuídas que resolvem esse mesmo problema, vamos supor que temos 3 tarefas, t_1, t_2 e t_3, em que, t_1 é responsável pelas variáveis x_1 e x_2, t_2 é responsável por x_3 e x_4 e t_3 é responsável por x_5 e x_6. No início de cada intervalo de tempo, todas as tarefas possuem a mesma configuração do vetor x. No algoritmo síncrono, a evolução dos valores de x que segue é, portanto, a mesma do algoritmo seqüencial:

$tempo = 1$; $x(0) = (1.5, 1.5, 0.5, 0.5, 1.5, 1.5)$

$tempo = 2$; $x(1) = (0.9167, 0.9286, 0.8125, 0.8333, 0.9500, 0.9545)$

$tempo = 3$; $x(2) = (1.0868, 1.0761, 1.0521, 1.0486, 1.0554, 1.0508)$

$tempo = 4$; $x(3) = (0.9528, 0.9580, 0.9603, 0.9643, 0.9685, 0.9710)$

$tempo = 5$; $x(4) = (1.0296, 1.0262, 1.0232, 1.0210, 1.0194, 1.0178)$

Já na versão assíncrona do algoritmo em questão, cada tarefa realiza o recálculo da sua parte do vetor x somente com a parte do vetor x enviada por alguma outra tarefa. Isso, muitas vezes, faz com que a solução em cada tarefa necessite de mais passos para alcançar o coeficiente de erro. Além disso, a convergência em cada tarefa evolui de forma diferente. Para simplificar o algoritmo, vamos considerar que a tarefa que inicia a execução deverá ser a responsável por determinar a terminação, quando o seu vetor x já tiver convergido o suficiente, segundo o coeficiente de erro. Este será o vetor solução. Seja t_1 a tarefa que iniciou uma execução assíncrona a qual recebeu as mensagens na ordem em que estamos indicando abaixo. Estamos omitindo as demais tarefas, pois vamos supor que t_1 é a tarefa escolhida para retornar a solução do sistema. Após cada mensagem recebida, o vetor x em t_1 está indicado ao lado.

msg = NULO; $x = (1.5, 1.5, 0.5, 0.5, 1.5, 1.5)$

msg de t_3; $x = (0.9167, 0.9286, 0.5000, 0.5000, 0.9500, 0.9545)$

msg de t_2; $x = (1.1945, 1.1684, 0.8125, 0.8333, 0.9500, 0.9545)$

msg de t_2; $x = (1.0469, 1.0365, 0.9152, 0.9269, 0.95, 0.9545)$

msg de t_3; $x = (1.0361, 1.0295, 0.9152, 0.9269, 1.1200, 1.1095)$

msg de t_3; $x = (0.9832, 0.9846, 0.9152, 0.9269, 1.0399, 1.0354)$

msg de t_2; $x = (1.0163, 1.0142, 1.0404, 1.0372, 1.0399, 1.0354)$

Essas execuções síncrona e assíncrona são mencionadas no capítulo seguinte, de forma que as Figuras 2.1 e 2.2, já neste ponto, podem ajudar no seu entendimento. Apesar de parecer pelo número de passos da evolução de x que a versão assíncrona é mais lenta do que

a síncrona, é necessário lembrar que cada passo acima mostrado na execução assíncrona é conseqüência de uma única mensagem recebida. Nos passos da execução síncrona, isto é, em cada intervalo de tempo e em cada tarefa, existe o recebimento das mensagens de todas as demais tarefas.

De forma resumida, temos nos algoritmos síncrono e assíncrono dois contextos diferentes. No primeiro, precisamos simular o modelo para o funcionamento do algoritmo. As tarefas possuem um certa dependência, pois só evoluem nas suas execuções de forma conjunta, isto é, uma tarefa só pode executar um intervalo de tempo quando as demais já tiverem executado o intervalo de tempo anterior. Por conta disso, na aplicação descrita acima, o vetor x tende a evoluir de forma similar à sua evolução no algoritmo seqüencial. No segundo algoritmo, o assíncrono, as tarefas são mais independentes, ao passo que o vetor x tende a convergir mais lentamente para a solução real do sistema.

Intuitivamente, teríamos uma boa situação se pudéssemos controlar a dependência entre as tarefas e a convergência do vetor x. Esta é a proposta da nossa ferramenta: atuar sobre um algoritmo assíncrono através da manipulação da troca de mensagens do mesmo com o objetivo de impor uma boa execução para ele. De forma mais específica, utilizaremos mecanismos de retardo de recebimento das mensagens nas tarefas para atingir esse objetivo. Esta ferramenta atuará sobre a aplicação interceptando, em cada tarefa, as mensagens enviadas e recebidas e impondo-lhes a ordem em que serão executadas pela aplicação, possibilitando inclusive que várias mensagens sejam executadas "ao mesmo tempo" pela aplicação em uma tarefa, como ocorre no algoritmo síncrono. A ordem em que as mensagens serão recebidas também deverá ser fornecida pelo usuário da ferramenta, além, obviamente, da aplicação.

Podemos exemplificar o funcionamento da ferramenta sobre o algoritmo assíncrono de resolução de sistemas lineares da seguinte forma. Uma tarefa só irá recalcular os seus x_i quando ela receber as mensagens de um determinado subconjunto de tarefas vizinhas. Para isso, uma tarefa, ao receber uma mensagem, deve retardar o seu uso até que tenha recebido, de outras tarefas vizinhas, as demais mensagens de que necessita.

Além disso, esse subconjunto de tarefas vizinhas de que uma tarefa depende, pode variar. Um mecanismo similar é aplicado nos algoritmos síncronos, que, no entanto, é bastante mais restrito, já que durante *toda* a execução, as tarefas, em cada tempo, esperam as mensagens das demais tarefas vizinhas do tempo anterior.

Um exemplo da execução deste algoritmo assíncrono da mesma tarefa t_1, agora conduzida pela ferramenta, seria o seguinte:

msg = NULO; $x = (1.5, 1.5, 0.5, 0.5, 1.5, 1.5)$

msgs de t_2 e t_3; $x = (0.9167, 0.9286, 0.8125, 0.8333, 0.9500, 0.9545)$

msg de t_2; $x = (1.0869, 1.0761, 0.9152, 0.9269, 0.9500, 0.9545)$

msgs de t_2 e t_3; $x = (1.0295, 1.0238, 1.0404, 1.0372, 1.0399, 1.0354)$

Veja que esta combinação de mensagens recebidas resultou em uma boa convergência do vetor x.

Neste exemplo utilizado, assim como em qualquer algoritmo assíncrono, saber quais mensagens e qual a ordem ideal para o recebimento das mesmas depende de alguns fatores, dentre os quais, a instância do problema e o número de tarefas. De fato, nossa ferramenta não impõe a "boa execução" para a aplicação sobre a qual ela atua sem a intervenção do seu usuário, mesmo porque uma boa execução para uma aplicação depende completamente dela. Nosso objetivo com esta ferramenta é possibilitar ao seu usuário executar uma aplicação distribuída de uma forma *parcialmente síncrona*, isto é, nem completamente assíncrona nem completamente síncrona, se ele perceber que assim poderá obter melhores resultados.

Para desenvolver esta ferramenta, combinamos idéias referentes a problemas que já vêm sendo estudados, como o problema de atribuição de relógios lógicos [3] [2] [4] e sincronizadores [1].

O principal objetivo desta dissertação, que é a proposta do algoritmo da ferramenta, é seguido pela implementação da mesma, sua validação através de uma aplicação e ex-

perimentos. A aplicação escolhida foi o problema do fluxo máximo, um problema modelado por grafos, e os algoritmos estudados para a sua resolução são baseados em uma abordagem conhecida como pré-fluxo [6]. As implementações seqüenciais conhecidas que resolvem este problema e que se baseiam na abordagem do pré-fluxo já têm se mostrado bastante eficientes. No entanto, este é um problema que ocorre freqüentemente como subproblema de outros, como em problemas de telecomunicações, de forma que obter boas execuções distribuídas para ele é um desafio. Além disso, o problema do fluxo máximo é um problema de difícil paralelização por duas razões. A primeira está na dificuldade de conciliar a quantidade de trabalho realizado pelas tarefas e o custo da comunicação. Nos exemplos mostrados há pouco, vimos que cada computação realizada nas tarefas, de maneira geral, se dá após o recebimento de mensagens e termina com o envio de mensagens. No problema do fluxo máximo, quando tentamos realizar muito trabalho durante essas computações nas tarefas, geramos efeitos colaterais que retardam a execução. Quando pouco trabalho é realizado, o custo da comunicação se torna elevado, pois a execução se estende. A segunda razão é o custo de se processar e, principalmente, de gerar as mensagens que é alto. De fato, não conhecemos resultados de implementações distribuídas do problema de fluxo máximo, de forma que este também é um dos objetivos dessa dissertação.

Os objetivos desta dissertação são alcançados através da apresentação de assuntos que estão organizados da seguinte forma. No segundo capítulo apresentamos um modelo de computação distribuída. Mostramos de que forma podemos tornar mais restrito o conjunto de execuções possíveis de um algoritmo distribuído segundo esse modelo através de uma atribuição de relógios lógicos.

O capítulo seguinte é dedicado, em sua maior parte, aos aspectos algorítmicos da ferramenta. Definimos o problema que a ferramenta se propõe a resolver, bem como a idéia do funcionamento da mesma. Em seguida, apresentamos o seu algoritmo.

Ao final do terceiro capítulo, nosso objetivo principal, isto é, a definição da ferramenta que procura conduzir uma execução distribuída de acordo com uma atribuição de reló-

gios lógicos, já foi alcançado. No quarto capitulo, definimos a aplicação cuja execução distribuída pudemos manipular através da ferramenta. A aplicação, como ja havíamos mencionado, é o problema do fluxo máximo. Implementamos essa aplicação segundo os modelos seqüencial, síncrono, assíncrono e parcialmente síncrono, este último obtido, naturalmente, com o uso da ferramenta.

No quinto capítulo, apresentamos os resultados dos experimentos realizados com as implementações do fluxo máximo.

O últimos capítulo é dedicado às conclusões. Incluímos ainda um apêndice com os códigos fonte das implementações desenvolvidas.

2

Algoritmos e Execuções Distribuídas

Uma execução de um algoritmo distribuído pode ser definida como um conjunto de seqüências de execuções locais de cada tarefa, envolvendo trocas de mensagens entre tarefas. Como mostramos no capítulo inicial, a ordem em que as mensagens são entregues é capaz de gerar diferentes execuções de um mesmo algoritmo. Nosso objetivo neste capítulo é apresentar os aspectos teóricos de um mecanismo para restringir o conjunto de execuções de um algoritmo distribuído.

Na primeira seção definimos o modelo matemático para a representação de algoritmos distribuídos, o modelo de computação distribuída utilizado neste trabalho, bem como a noção de evento. Na seção seguinte, falaremos a respeito de relógios lógicos. Mostraremos a idéia original de Lamport para atribuição de relógios lógicos a execuções de algoritmos distribuídos e mostraremos de que forma generalizamos esta noção.

2.1. Um modelo de execução distribuída

A fim de escrevermos algoritmos distribuídos de forma mais clara, bem como facilitar e padronizar os conceitos envolvidos, representaremos um algoritmo distribuído através de um grafo conexo não orientado $\Gamma = (N, C)$. Os vértices em N representarão as tarefas e as

arestas em C representarão os canais de comunicação entre as tarefas. Faremos $n =\mid N \mid$ e $m =\mid C \mid$. Dado um vértice $t \in N$, o conjunto de vizinhos de t, isto é, vértices que formam aresta com t, será denotado por Viz_t.

Vamos definir agora os conceitos do modelo de computação distribuída utilizado aqui. Este modelo baseia-se no modelo definido em [1], mas possui algumas diferenças para adequá-lo aos propósitos deste trabalho. O Algoritmo TAREFA$_t$ descrito a seguir mostra o comportamento de um algoritmo distribuído segundo o nosso modelo.

Algoritmo TAREFA$_t$
1. Estado inicial σ_t
2. $r_t \leftarrow 0$
3. **Execute** $\Phi'_t \leftarrow$EVENTO$_t(r_t, \emptyset)$
4. **envie** cada mensagem em Φ'_t para a tarefa de Viz_t correspondente
5. **enquanto** σ_t não é estado final **faça**
6. $\quad r_t \leftarrow r_t + 1$
7. $\quad$ **Receba** Φ_t, que pode ser vazio
8. $\quad$ **Execute** $\Phi'_t \leftarrow$EVENTO$_t(r_t, \Phi_t)$
9. $\quad$ **envie** cada mensagem em Φ'_t para a tarefa de Viz_t correspondente

Em um algoritmo distribuído, a partir de um estado inicial, as tarefas realizam alguma computação através da função EVENTO$_t$. De forma geral, estamos considerando que todas as tarefas executam essa função pela primeira vez dessa maneira, mesmo que algumas não realizem trabalho e, portanto, não alterem seu estado inicial. No decorrer do algoritmo, cada tarefa ao executar EVENTO$_t$, gera um conjunto de mensagens Φ'_t - que pode ser vazio - as quais devem ser enviadas às tarefas vizinhas. Cada nova execução da função EVENTO$_t$ é precedida pelo incremento do rótulo r_t e recebimento do conjunto de mensagens Φ_t - que também pode ser vazio - o qual será entrada para EVENTO$_t$.

De acordo com TAREFA$_t$, a execução de um algoritmo distribuído corresponde à execução de um conjunto de eventos. Podemos definir um *evento* ξ como uma sêxtupla $\langle t, r, \Phi, \sigma, \sigma', \Phi' \rangle$, onde

t é a tarefa onde o evento ocorre;

r é o rótulo que indica a ordem em que o evento ocorre em t. O primeiro evento em t recebe o rótulo 0, o segundo, rótulo 1, e assim por diante;

Φ é o conjunto, que pode ser vazio, de mensagens recebidas naquele evento;

σ é o estado local da tarefa antes da ocorrência do evento;

σ' é o estado local da tarefa após a ocorrência do evento;

Φ' é o conjunto de mensagens enviadas por t após a ocorrência do evento, caso haja alguma.

Ao longo do texto, podemos usar as notações $t(\xi)$, $r(\xi)$, $\Phi(\xi)$, $\sigma(\xi)$, $\sigma'(\xi)$ e $\Phi'(\xi)$ para indicar os parâmetros de um evento ξ quando essa precisão for necessária.

De acordo com a definição de evento, algum detalhamento para TAREFA_t pode ser dado aqui. O inteiro r_t indica a ordenação dos eventos nas tarefas. O primeiro evento em cada tarefa recebe o rótulo 0 e cada evento seguinte recebe o rótulo igual ao do evento anterior incrementado de 1. O conjunto Φ_t consiste das mensagens recebidas pela tarefa t após o evento com rótulo $r_t - 1$ e é entrada para EVENTO_t. A tarefa t pode ter seu estado local alterado após a execução de EVENTO_t, que, como vimos, depende das mensagens que recebe.

Como havíamos dito, não é necessário haver mensagens para um evento ocorrer. Vamos chamar de *eventos espontâneos* os eventos ξ tais que $\Phi(\xi) = \emptyset$. Os eventos responsáveis por iniciar uma execução distribuída (no máximo um por tarefa) são, particularmente, eventos espontâneos.

Uma execução distribuída Ξ é caracterizada pelo seu conjunto de eventos, sobre o qual pode-se definir uma relação de *ordem parcial* $\prec^+$. Seja $\prec$ uma relação binária sobre os eventos de Ξ tal que, se ξ_1 e ξ_2 são dois eventos quaisquer da execução, então $\xi_1 \prec \xi_2$ se e somente se:

1. ξ_1 e ξ_2 ocorrem na mesma tarefa e $r(\xi_2) = r(\xi_1) + 1$.

2. ξ_1 e ξ_2 ocorrem em tarefas diferentes e $\Phi'(\xi_1) \cap \Phi(\xi_2) \neq \emptyset$.

A relação $\prec^+$ é o fecho transitivo de $\prec$. É transitiva e não reflexiva e, portanto, é uma ordem parcial.

Algumas noções relacionadas a ordens parciais e que utilizaremos posteriormente são enumeradas a seguir.

Eventos Concorrentes: ξ_1 e ξ_2 são ditos eventos concorrentes quando

$$(\xi_1, \xi_2) \in \Xi \times \Xi - \prec^+ \text{ e } (\xi_2, \xi_1) \in \Xi \times \Xi - \prec^+$$

Passado(ξ): Conjunto de eventos ξ' tais que $\xi' \prec^+ \xi$, com $\xi, \xi' \in \Xi$.

Futuro(ξ): Conjunto de eventos ξ' tais que $\xi \prec^+ \xi'$, com $\xi, \xi' \in \Xi$.

De acordo com o nosso modelo de execução distribuída podemos estabelecer definições para execuções assíncrona e síncrona. Uma execução assíncrona possui um ou mais eventos que a iniciam. Estes serão os seus únicos eventos espontâneos. Depois disso, sempre que uma tarefa recebe uma mensagem ela realiza um evento. Assim, os eventos ξ não espontâneos são tais que $|\Phi(\xi)| = 1$ e ξ é realizado imediatamente após a chegada da mensagem de $\Phi(\xi)$. Em uma execução síncrona, inicialmente, um evento (de rótulo 0) ocorre em cada tarefa. Em seguida, cada evento ξ ocorre quando $t(\xi)$ recebe todas as mensagens enviadas por eventos com rótulos $r(\xi) - 1$. Uma execução síncrona pode, portanto, ter diversos eventos espontâneos.

Um algoritmo termina corretamente se alcança um estado final correto. Em particular, um algoritmo de acordo com TAREFA$_t$ que termina corretamente para toda execução síncrona é um algoritmo síncrono. Se terminar corretamente para todas as execuções, ele é dito assíncrono. Se terminar corretamente para um determinado conjunto de execuções, é dito parcialmente síncrono.

As Figuras 2.1 e 2.2 representam o comportamento das execuções síncrona e assíncrona apresentadas no exemplo do primeiro capítulo. As linhas horizontais representam o tempo. Os números abaixo das linhas tracejadas são os rótulos dos eventos, enquanto as setas representam as mensagens enviadas ao final dos eventos.

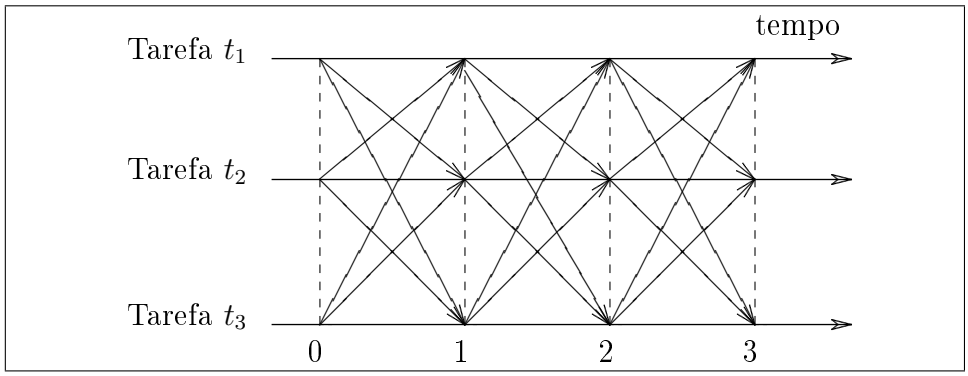


Figura 2.1. *Execução síncrona para a resolução de sistemas lineares segundo o exemplo da introdução. Cada mensagem enviada ao final de um evento com relógio r chega antes do início do evento com relógio $r+1$ da tarefa de destino. Ainda que algumas mensganes não ocorressem, esta seria ainda uma execução síncrona, pois os eventos de tal execução não dependem de mensagens.*

2.2. Relógios lógicos

Conforme definido no algoritmo de Lamport [3], a *atribuição de relógios lógicos* a uma execução distribuída Ξ é a atribuição de um inteiro não-negativo $\ell(\xi)$, chamado de *relógio lógico* de ξ, a cada evento $\xi \in \Xi$ tal que:

1. $\ell(\xi) = 0$, se ξ é um evento que inicia a execução, ou seja, $r(\xi) = 0$.

2. $\ell(\xi) = \ell(\xi') + 1$, onde ξ' é tal que $\ell(\xi') = \max_{\xi'' \in Passado(\xi)} \ell(\xi'')$.

No algoritmo de Lamport, todas as tarefas têm uma variável auxiliar através da qual elas atribuem os relógios aos seus eventos. Essas variáveis começam com o valor 0 e o evento que inicia a execução recebe, portanto, o relógio 0. Quando uma tarefa gera uma

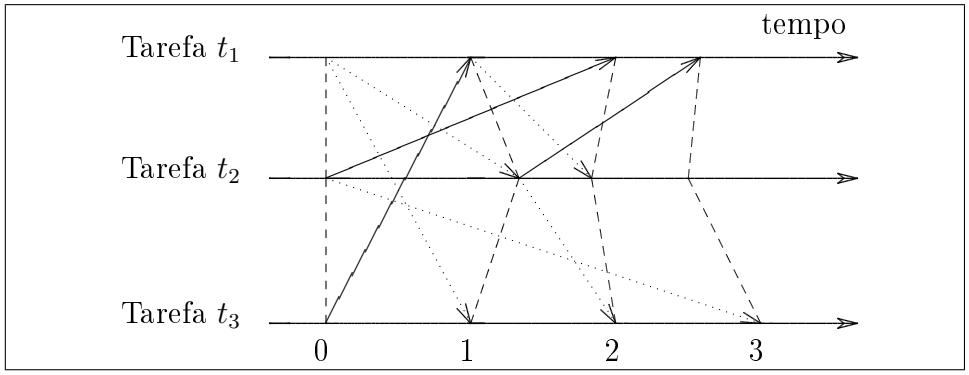


Figura 2.2. *Execução assíncrona para a resolução de sistemas lineares segundo o exemplo da introdução. Naquele exemplo, mostramos somente as mensagens que chegaram à tarefa t_1. As setas não pontilhadas representam algumas dessas mensagens.*

mensagem, anexa a ela o valor do relógio lógico do seu evento correspondente. Cada vez que uma tarefa recebe uma mensagem, o evento que ela dá origem recebe o maior dentre o valor do relógio lógico anexado à mensagem e o valor da sua variável auxiliar incrementado de um. A variável auxiliar, em seguida, recebe o valor desse último relógio lógico. A atribuição de relógios lógicos estabelecida pelo algoritmo Lamport é única para cada execução. A Figura 2.3 apresenta a atribuição de relógios lógicos a uma execução distribuída.

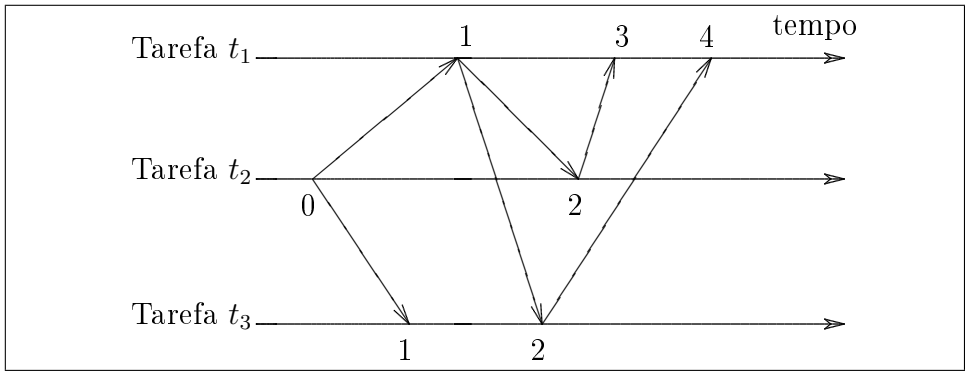


Figura 2.3. *Atribuição de relógios lógicos segundo o algoritmo de Lamport.*

Perceba que relógios lógicos e rótulos são grandezas diferentes. Na Figura 2.3, por exemplo, o evento que recebe o relógio lógico 3 tem rótulo 1.

Nesta dissertação consideraremos a seguinte generalização desta atribuição de relógios lógicos. Dada uma execução distribuída Ξ, para todo $\xi \in \Xi$, uma *$S-$atribuição* de relógios lógicos a Ξ é a atribuição de um inteiro não-negativo $\ell(\xi)$ tal que:

$\ell(\xi) = 0$, se $r(\xi) = 0$.

$\ell(\xi) \geq \ell(\xi') + S_{t(\xi')}(\ell(\xi'), t(\xi))$, onde $\ell(\xi') = \max_{\xi'' \in Passado(\xi)} \ell(\xi'')$.

O parâmetro S, o qual chamaremos *passo*, é um conjunto de funções inteiras positivas $S_i(\ell, t)$, para todas as tarefas i do algoritmo em questão, que valem 1 quando $i = t$. Os parâmetros de $S_{t(\xi')}(\ell(\xi'), t(\xi))$ se justificam, pois o que esta função procura é estabelecer o quão distante as atribuições dadas a eventos de $t(\xi)$ em $Futuro(\xi')$ devem estar de ξ'.

A Figura 2.4 apresenta uma $S-$atribuição de relógios lógicos a uma execução distribuída segundo as funções de S definidas a seguir.

$$S_i(\ell, t) = \begin{cases} 1, & \text{se } t = 2, \text{ ou } t = 1 \text{ e } \ell \text{ é par, ou } t = 3 \text{ e } \ell \text{ é ímpar} \\ 2, & \text{caso contrário.} \end{cases}, 1 \leq i \leq 3.$$

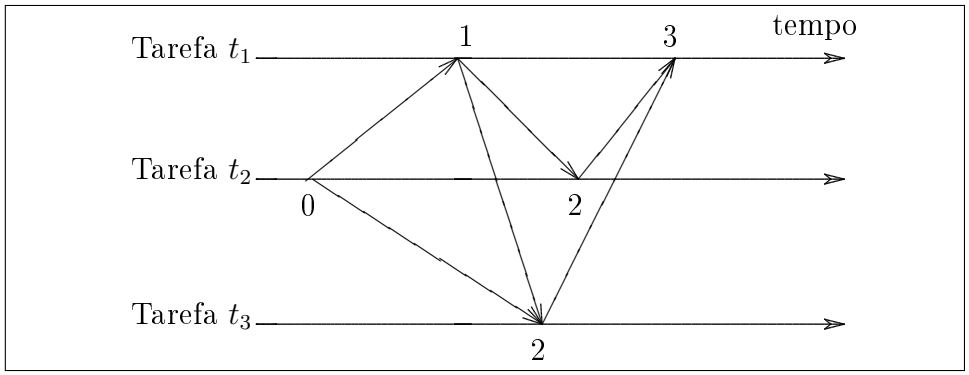


Figura 2.4. *Atribuição de relógios lógicos segundo o passo S.*

Uma $S-$atribuição, ao contrário da atribuição de Lamport, não é única. Teríamos uma outra $S-$atribuição válida para a execução da Figura 2.4 se trocássemos de 2 para

3 o valor dado ao evento da tarefa t_3 e de 3 para 4 o valor dado ao segundo evento da tarefa t_1.

Seja, agora, T um conjunto de funções inteiras positivas $T_i(\ell, t)$, para todas as tarefas i de um determinado algoritmo, o qual chamaremos *prazo*. Dizemos que uma $S-$atribuição a uma execução Ξ deste algoritmo *respeita* o prazo T se, para $\xi, \xi' \in \Xi$, então,

$$\ell(\xi) \leq \ell(\xi') + T_{t(\xi')}(\ell(\xi'), t(\xi)), \text{ onde } \ell(\xi') = \max_{\xi'' \in Passado(\xi)} \ell(\xi'').$$

Uma execução, como vimos, pode admitir várias $S-$atribuições, mas não necessariamente todas elas respeitam um prazo T. Como exemplo, a $S-$atribuição da Figura 2.4 respeita o prazo formado por funções constantes iguais a 2, mas o outro exemplo de $S-$atribuição à mesma execução que demos em seguida não respeita esse mesmo prazo. Dizemos que uma execução Ξ é $(S,T)-$*compatível* quando Ξ admite uma $S-$atribuição que respeita T. A questão de, dado um algoritmo Γ e funções S e T, encontrar uma execução $(S,T)-$compatível é o assunto do próximo capítulo.

A Figura 2.5 mostra uma execução $(S,T)-$compatível e sua $S-$atribuição de relógios lógicos. O conjunto de funções S é o mesmo já definido anteriormente e T é formado pelas seguintes funções constantes.

$$T_i(\ell, t) = 2,\ 1 \leq i \leq 3.$$

Perceba que a execução da Figura 2.4 é $(S,T)-$compatível, considerando a $S-$atribuição da figura. Na Figura 2.5 temos uma execução semelhante à da Figura 2.4. A diferença entre as duas execuções é que a mensagem enviada pela tarefa t_2 a partir do evento com relógio lógico 2 que era recebida pelo evento com relógio lógico 3 na tarefa t_1 foi "atrasada" e recebida pelo evento seguinte, o qual recebeu o relógio lógico 4. Esta execução ainda assim é $(S,T)-$compatível devido ao prazo T.

Podemos neste ponto fornecer definições para execuções assíncronas e síncronas em termos de funções passo e limite.

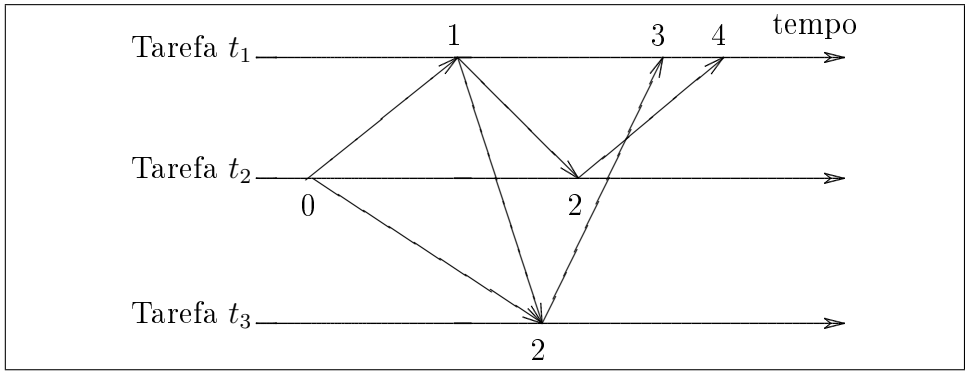


Figura 2.5. *$S-$atribuição a uma execução $(S,T)-$compatível.*

Uma execução é síncrona se for $(S,T)-$compatível para S e T formados por funções constantes iguais a 1.

Uma execução é assíncrona se for $(S,T)-$compatível para o limite S formado por funções constantes iguais a 1 e o prazo T formado por funções que retornem um valor tão grande que qualquer $S-$atribuição sempre consegue respeitá-lo.

No capítulo introdutório, após mostrarmos exemplos de execuções segundo os algoritmos síncrono e assíncrono de resolução de sistemas lineares, mostramos como poderia ser uma execução do algoritmo assíncrono conduzida pela ferramenta que estávamos apresentando. Definidos os conceitos de passo e prazo, podemos neste momento fazer uma relação entre estes e o exemplo mencionado. As funções PASSO_SEL e PRAZO_SEL dadas a seguir são tais que execução do exemplo é (PASSO_SEL, PRAZO_SEL)-compatível. A Figura 2.6 mostra graficamente esta execução.

$$S_i(\ell,t) = 1,\ 1 \leq i \leq 3.$$

$$T_1(\ell,t) = T_2(\ell,t) = 1.$$

$$T_3(\ell,t) = \begin{cases} 3, & \text{se } \ell \text{ é par,} \\ 2, & \text{caso contrário.} \end{cases}$$

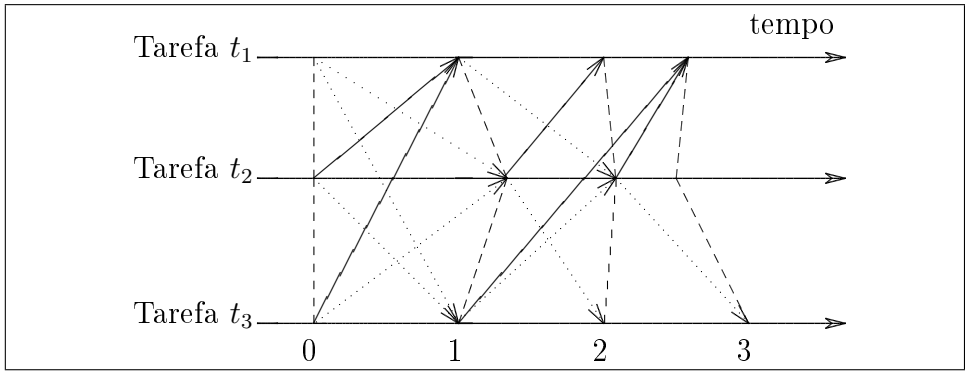


Figura 2.6. *Execução do algoritmo de resolução de sistemas lineares (*PASSO_SEL *e* LIMITE_SEL*)-compatível. Esta execução corresponde ao exemplo da introdução em que mostramos o algoritmo assíncrono conduzido pela ferramenta. Naquele exemplo, mostramos apenas as mensagens que chegaram à tarefa t_1, as quais estão representadas pelas setas não pontilhadas.*

Perceba que a execução mostrada na Figura 2.6 está de acordo com o algoritmo TAREFA$_t$. Os rótulos dos eventos estão mostrados na parte inferior da figura. Todas as mensagens podem ser recebidas, em cada tarefa, pelo seu próximo evento e isto é indicado por $S_i(\ell, t) = 1$. Como $T_1(\ell, t) = T_2(\ell, t) = 1$, as mensagens enviadas pelas tarefas t_1 e t_2 precisam ser recebidas, necessariamente, por eventos com relógios lógicos maiores em uma unidade em relação aos relógios lógicos dos eventos que as enviaram. As mensagens enviadas por t_3 não têm restrição de recebimento tão forte, visto que $T_1(\ell, t) \geq 2$. Esta não é uma execução assíncrona nem síncrona. Apesar disso, ela proporcionou uma boa convergência para o vetor x, solução do sistema de equações lineares do exemplo. De fato, um estudo do método iterativo em questão poderá mostrar que nem sempre uma execução síncrona ou assíncrona levarão à convergência mais rápida do vetor solução. Essa convergência muitas vezes poderá ser alcançada através de uma execução que configure um meio termo entre esses modelos.

3

Atribuição de Relógios Lógicos

Um algoritmo assíncrono, de acordo com a definição dada na capítulo anterior através dos conjuntos passo e prazo, admite qualquer execução dentro do modelo definido por $\textsc{Tarefa}_t$. Além disso, o tempo de chegada das mensagens trocadas em uma execução distribuída é arbitrário. A combinação dessas características torna uma execução assíncrona não determinística no sentido de que não sabemos em que ordem as mensagens que partem de diversas tarefas chegarão aos seus destinos. A mudança na ordem de recebimentos das mensagens pode acarretar uma mudança no decorrer de uma computação distribuída. Nesse contexto, podemos ter execuções de um mesmo algoritmo mais ou menos eficientes de acordo com os recebimentos das mensagens nas tarefas.

Um algoritmo síncrono se comporta de forma bastante diferente, visto que uma execução síncrona evolui acompanhando um relógio comum a todas as tarefas. Em cada unidade de tempo do relógio, a qual podemos chamar de pulso, um evento pode ser disparado em cada tarefa e todas as mensagens enviadas por estes eventos chegarão aos seus destinos antes do início do próximo pulso. Veja que cada evento pode, dessa forma, receber mais de uma mensagem, ou mesmo, nenhuma. Um sistema distribuído, por se tratar de uma rede assíncrona, no entanto, não é capaz de respeitar essas características do modelo síncrono que dizem respeito ao recebimento das mensagens. Para que pos-

samos executar um algoritmo de forma síncrona, precisamos simular o modelo de que ele necessita sobre o sistema distribuído. Em [1], podemos encontrar o *sincronizador Alpha*, um algoritmo assíncrono que tem como objetivo fornecer esta simulação. O sincronizador Alpha deverá receber como parâmetro uma aplicação, que é um algoritmo síncrono, o qual será executado respeitando as restrições impostas sobre a chegada das mensagens, de acordo, naturalmente, com o seu modelo síncrono.

A ferramenta que nos propomos a desenvolver neste trabalho, a qual apresentaremos neste capítulo, tem uma relação estreita com este sincronizador. O sincronizador Alpha procura possibilitar a execução de um algoritmo síncrono, enquanto a nossa ferramenta procura possibilitar a execução de qualquer algoritmo distribuído, isto é, qualquer algoritmo da forma $\textsc{Tarefa}_t$. Particularmente, os algoritmos síncronos poderão ser executados com a ferramenta.

Esta ferramenta, naturalmente, sempre será executada em conjunto com um algoritmo distribuído Γ o qual deverá ser manipulado por ela. Γ será, algumas vezes, chamado de *aplicação*.

De acordo com as noções de relógios lógicos introduzidas no capítulo anterior, a formalização do problema resolvido pela ferramenta é dada a seguir.

3.1. Definição do problema

Problema 1. *Dados*

1. Γ*, um algoritmo assíncrono,*
2. S*, um conjunto de funções passo,*
3. T*, um conjunto de funções prazo,*

determinar uma execução Ξ *de* Γ $(S,T)-$*compatível, ou indicar que esta execução não existe.*

Vamos chamar os conjuntos S e T de *restrições de sincronização*. As funções EVENTO_t das tarefas t de Γ recebem como parâmetro, ao invés de rótulos, como no algoritmo TAREFA_t, relógios lógicos.

Enquanto que o algoritmo do Lamport forneceria uma atribuição de relógios lógicos a uma execução de um algoritmo assíncrono Γ, nossa ferramenta funciona, de certa forma, de maneira inversa. Ela procura fazer com que a execução de Γ seja capaz de receber uma atribuição de relógios lógicos definida pelas restrições de sincronização.

3.2. Idéia do funcionamento da ferramenta

3.2.1. Apresentação por camadas

No intuito de alcançar o objetivo definido no parágrafo anterior, nossa ferramenta intercepta todas as mensagens geradas durante a execução de Γ e, de acordo com as restrições de sincronização, estabelece quando essas mensagens de fato dispararão novos eventos.

A ferramenta estará sendo executada, portanto, diretamente sobre o sistema distribuído, enquanto a aplicação Γ estará sendo executada sobre a ferramenta. Este contexto pode ser refinado e melhor entendido através da sua apresentação por camadas.

Figura 3.1. *Modelo de camadas da execução de uma aplicação sob a interveção da ferramenta.*

Rede Assíncrona: Esta rede é um sistema distribuído formado pelos processadores onde serão executadas as tarefas. Cada processador executa uma tarefa e estão interligados por canais bidirecionais formando um grafo completo.

Ferramenta: É um algoritmo distribuído que controla a execução da aplicação através de uma atribuição de relógios lógicos. Chamaremos a função EVENTO_t, para cada tarefa t deste algoritmo, de SINC_t. Os dados de entrada para a ferramenta serão a aplicação e conjuntos passo S e prazo T. A ferramenta deverá gerar uma execução da aplicação $(S,T)-$compatível.

Aplicação: É um algoritmo assíncrono. Quando uma tarefa i executa EVENTO_i e esta função gera uma mensagem para a aplicação na tarefa j, a ferramenta em i é responsável por encaminhar esta mensagem para EVENTO_j no relógio adequado.

Seguem as considerações gerais sobre as funções que compõem as interfaces entre as camadas.

Rede Assíncrona - Ferramenta: Aqui é definida a função de inicialização da rede assíncrona, bem como as funções de envio e recebimento das mensagens que trafegam na rede assíncrona.

Ferramenta - Aplicação: Aqui é definida a função de inicialização da ferramenta, para a qual são passados como parâmetros, pela aplicação, o grafo de tarefas, a função EVENTO_i, e os conjuntos passo e limite. Aqui também é definida a função de envio de mensagens que é usada por EVENTO_i.

3.2.2. Restrições de sincronização

Como falamos anteriormente, no intuito de gerar uma execução $(S,T)-$compatível da aplicação, dados um passo S e um prazo T, a ferramenta, em cada tarefa, procura estabelecer uma ordem para as mensagens recebidas de forma que, aos eventos que serão disparados por essas mensagens, seja possível atribuir os relógios lógicos definidos pelas

restrições de sincronização. Portanto, para estabelecer quando as mensagens relacionadas a estes eventos deverão de fato ser "entregues" à aplicação para que ela possa gerá-los, a função SINC_i, ℓ_i. Seu valor deverá ser incrementado sempre que um novo evento tiver que acontecer, representando o relógio lógico deste evento.

Cada mensagem estará acompanhada dos valores das funções em passo e prazo relacionadas ao evento que a gerou. Assim, quando um evento ξ_1 ocorre na tarefa i e gera uma mensagem para a tarefa j, a ferramenta em i envia essa mensagem juntamente com os valores $S_i(\ell_i, j)$ e $T_i(\ell_i, j)$. Ao receber a mensagem, a ferramenta na tarefa j saberá que deverá entregar esta mensagem a um evento ξ_2 que acontecerá no relógio ℓ'_j tal que $\ell_i + S_i(\ell_i, j) \leq \ell'_j \leq \ell_i + T_i(\ell_i, j)$. Podemos enumerar algumas propriedades inerentes.

1. Se $S_i(\ell_i, j) = T_i(\ell_i, j)$, então $\ell'_j = \ell_i + S_i(t_i, j)$. Em uma execução síncrona, por exemplo, $S_i(\ell_i, j) = T_i(\ell_i, j) = 1$ para todos os i, t_i e j. Dessa forma, cada mensagem deverá ser entregue ao evento com relógio lógico imediatamente posterior ao relógio lógico do evento que a gerou.

2. Se $S_i(\ell_i, j) > T_i(\ell_i, j)$, então não será possível relacionar a mensagem a nenhum evento, configurando a impossibilidade de a execução da aplicação ser $(S, T)-$compatível. Nesse caso, ela é interrompida.

3. Sejam duas mensagens geradas por SINC_i nos eventos em ℓ_i e $\ell_i + k$, $k > 0$, que deverão ser entregues à tarefa j a fim de gerarem os eventos em ℓ'_j e ℓ''_j, respectivamente. A desigualdade $\ell''_j \geq \ell'_j$ deverá ser válida, pois os canais de comunicação são FIFO. Assim, $S_i(\ell_i + k, j) \geq S_i(\ell_i, j) - k$ e $T_i(\ell_i + k, j) \geq T_i(\ell_i, j) - k$. A Figura 3.2 mostra duas mensagens partindo de eventos em i nos relógios lógicos ℓ_i e $\ell_i + k$, $k > 0$. Elas chegam, respectivamente, nos relógios ℓ_{j_1} e ℓ_{j_2}, mas são entregues a eventos nos relógios, respectivamente, ℓ'_j e ℓ''_j. Quando ambas são recebidas por um mesmo evento em j, o seu relógio é, no mínimo, $\ell'_j = \ell''_j = \ell_i + S_i(\ell_i, j)$, o caso em que $S_i(\ell_i + k, j)$ possui o menor valor possível, que é $S_i(\ell_i, j) - k$. Veja que estamos considerando que $\ell_{j_1} < \ell_i + S_i(\ell_i, j)$ e $\ell_{j_2} < \ell_i + k + S_i(\ell_i + k, j)$. Se $\ell_{j_1} \geq \ell_i + S_i(\ell_i, j)$,

então $\ell'_j > \ell_i + S_i(\ell_i, j)$, mas ainda assim, devemos manter $\ell''_j \geq \ell'_j$. A Figura 3.3 mostra situação semelhante, desta vez para funções prazo. Quando as mensagens vindas de ℓ_i e $\ell_i + k$ são ambas recebidas por um mesmo evento em j, o seu relógio é, no máximo, $\ell'_j = \ell''_j = \ell_i + k + T_i(\ell_i + k, j)$, o caso em que $T_i(\ell_i, j)$ possui maior valor possível, isto é, $T_i(\ell_i, j) = T_i(\ell_i + k, j) + k$.

4. A ferramenta considerará que se j receber uma mensagem vinda de i no relógio ℓ_i, estão $\ell_j \leq \ell_i + T_i(\ell_i, j)$. Do contrário, consideramos que funções dos conjuntos passo e prazo produziram valores errôneos e a execução da aplicação é interrompida.

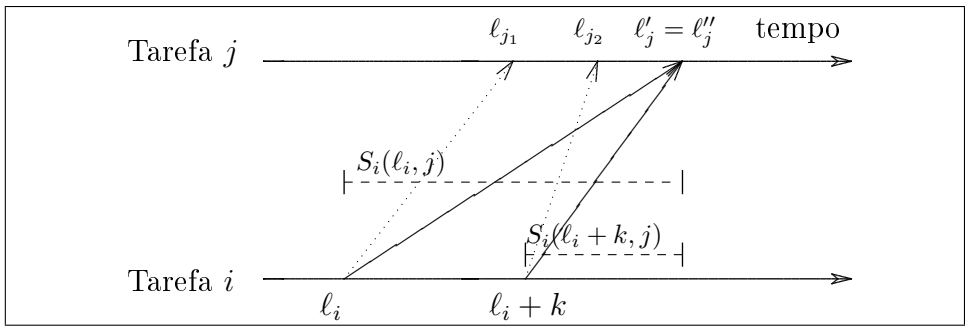


Figura 3.2. *ℓ_{j_1} e ℓ_{j_2} são os valores de relógios lógicos em j quando as mensagens vindas de i pelos eventos em ℓ_i e $\ell_i + k$ chegam. ℓ'_j e ℓ''_j são os valores dos relógios lógicos dos eventos que recebem as mensagens. Se $\ell'_j = \ell''_j$, o menor valor possível de $S_i(\ell_i + k, j)$ ocorre quando $\ell'_j = \ell''_j = \ell_i + S_i(\ell_i, j)$.*

3.2.3. Cálculo das restrições de sincronização

As funções passo e prazo poderiam ser usadas somente após o término de cada evento, para determinar os valores que acompanharão cada mensagem gerada no intuito de estabelecer, quando da sua chegada, o intervalo de relógios dos eventos os quais poderão recebê-la. Dessa forma, somente a tarefa que envia a mensagem poderia determinar quando esta deveria ser entregue à aplicação. No entanto, como o relógio lógico do evento que receberá uma determinada mensagem depende do canal por onde ela trafega, as tarefas que usam esse canal devem interagir para calculá-lo. Dessa forma, os conjuntos passo S e prazo T

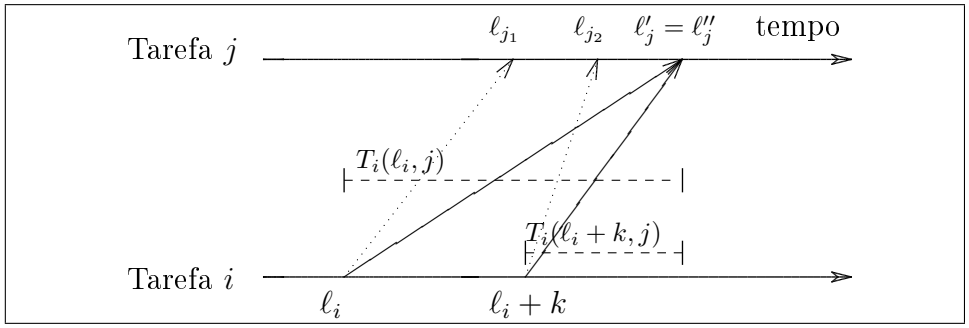


Figura 3.3. *ℓ_{j_1} e ℓ_{j_2} são os valores de relógios lógicos em j quando as mensagens vindas de i pelos eventos em ℓ_i e $\ell_i + k$ chegam. ℓ'_j e ℓ''_j são os valores dos relógios lógicos dos eventos que recebem as mensagens. Se $\ell'_j = \ell''_j$, o maior valor possível de $T_i(\ell_i, j)$ ocorre quando $\ell'_j = \ell''_j = \ell_i + k + T_i(\ell_i + k, j)$.*

serão, agora, quatro, nomeados S^+, S^-, T^+ e T^-, de forma que S^+ e T^+ se relacionam com os eventos que geram a mensagem e S^- e T^-, com os que recebem. Estes conjuntos são definidos de acordo com as propriedades a seguir que devem ser simultaneamente satisfeitas.

1. Uma mensagem gerada pelo evento de relógio ℓ_i na tarefa i enviada para a tarefa j deverá ser entregue a um evento da aplicação cujo relógio ℓ_j é tal que $\ell_i + S_i^+(\ell_i, j) \leq \ell_j \leq \ell_i + T_j^-(\ell_j, i)$.

2. O evento de relógio ℓ_j na tarefa j só poderá receber mensagens geradas por eventos da tarefa i cujos relógios ℓ_i são tais que $\ell_j - T_i^+(\ell_i, i) \leq \ell_i \leq \ell_j - S_j^-(\ell_j, i)$.

De acordo com estas propriedades, temos que

$$\max\{\ell_i + S_i^-(\ell_i, j), \min_{\ell_i \leq \ell'_j - T_j^+(\ell'_j, i)} \ell'_j\} \leq \ell_j \leq \min\{\ell_i + S_i^+(\ell_i, j), \max_{\ell_i \geq \ell'_j - T_j^-(\ell'_j, i)} \ell'_j\}$$

3.3. O Algoritmo da ferramenta

O algoritmo da nossa ferramenta é um algoritmo distribuído e, como tal, segue o modelo de TAREFA$_t$. Chamaremos a sua função de evento de SINC, a qual descreveremos a seguir.

SINC terá como entrada, além da aplicação, na forma das funções APLIC$_i$, as funções S_i^-, T_i^-, S_i^+ e T_i^+, para cada tarefa i. Chamaremos de *tarefa iniciadora* a tarefa que será responsável por ler a instância do problema e determinar o grafo de tarefas. Assim, SINC na tarefa iniciadora deverá receber a instância da aplicação, além do grafo de tarefas, nas variáveis, respectivamente, $inst\_ini$ e $grafo\_tarefas$.

Uma outra informação de entrada, une_msgs, possibilita o usuário da ferramenta determinar a abordagem de execução dos eventos da aplicação em cada tarefa. De acordo com as restrições de sincronização, seria possível que durante a execução de uma aplicação com a ferramenta, diversos eventos (chamadas a APLIC$_i$) consecutivos da aplicação em uma tarefa i pudessem ser executados sem a necessidade da espera por mensagens. Se for possível, de acordo com as características da aplicação, que todos estes eventos sejam executados como um só, isto é, unindo todas as suas entradas em uma única e passando como parâmetro para APLIC$_i$, une_msgs deverá ter o o valor $VERDADEIRO$. Do contrário, une_msgs deverá ser $FALSO$.

SINC$_i$(APLIC$_i$, S_i^-, T_i^-, S_i^+, T_i^+, $inst\_apli$, $tarefas\_sinc$ e une_msgs)

1. $\ell_i \leftarrow 0$
2. $iniciado_i \leftarrow FALSO$
3. $contini_i \leftarrow 0$
4. $msgatraso_i[k] \leftarrow \infty, \forall k \geq 0$
5. $\Phi_i[k] \leftarrow \emptyset, \forall k \geq 0$
6. **se** i é tarefa iniciadora **então**
7. $\quad iniciado_i \leftarrow VERDADE$
8. $\quad$ **envie** INI_MSG($inst\_ini$) para todos os vizinhos
9. **enquanto** não for determinada a terminação global **faça**
10. $\quad$ **receba** msg_i de $j \in Viz_i$
11. $\quad\quad$ **se** $msg_i =$ INI_MSG($inst\_ini$) **então**
12. $\quad\quad\quad$ **se não** $iniciado_i$ **então**
13. $\quad\quad\quad\quad iniciado_i \leftarrow VERDADE$
14. $\quad\quad\quad\quad pai_i \leftarrow j$
15. $\quad\quad\quad\quad$ **determine** Viz_i a partir de $tarefas\_sinc$
16. $\quad\quad\quad\quad ultenv_i[k] \leftarrow -1, \forall k \in Viz_i$
17. $\quad\quad\quad\quad mintrf_i[k] \leftarrow maxtrf_i[k] \leftarrow 0, \forall k \in Viz_i$
18. $\quad\quad\quad\quad \Phi_i[\ell_i] \leftarrow inst\_ini$
19. $\quad\quad\quad\quad$ **envie** INI_MSG($inst\_ini$) para todos os vizinhos menos pai_i
20. $\quad\quad\quad contini_i \leftarrow contini_i + 1$
21. $\quad\quad\quad$ **se** $contini_i =\mid Viz_i \mid$ **então**
22. $\quad\quad\quad\quad \Phi' \leftarrow$ APLIC$_i(\ell_i, \Phi_i[\ell_i])$
23. $\quad\quad\quad\quad$ **envie** INI_MSG($\emptyset$) para pai_i
24. $\quad\quad\quad\quad$ **envie** APLI_MSG($\ell_i, S_i^+(\ell_i, j), T_i^+(\ell_i, j), \Phi'[j]$) para cada vizinho $j \in Viz_i$

25. **senão se** $msg_i = \text{APLI\_MSG}(j\ell, js, jt, aplibuf)$ de $j \in Viz_i$ **então**
    //Esta parte inicial calcula o relógio em que a mensagem deverá ser recebida,
    //bem como atualiza variáveis necessárias para determinar se eventos podem ser realizados.
26. $ultenv_i[j] \leftarrow j\ell$
27. $apmin \leftarrow \max\{\ell_i + 1, j\ell + js, mintrf_i[j]\}$
28. **enquanto** $apmin - j\ell < S_i^-(apmin, j)$ **faça**
29. $apmin \leftarrow apmin + 1$
30. $mintrf_i[j] \leftarrow apmin$
31. $apmax \leftarrow j\ell + jt$
32. **enquanto** $apmax - j\ell > T_i^-(apmax, j)$ **faça**
33. $apmax \leftarrow apmax - 1$
34. $maxtrf_i[j] \leftarrow \max\{apmax, maxtrf_i[j]\}$
35. **se** $apmax < apmin$ **então**
36. **Pare**: execução incorreta
37. $\Phi_i[apmin] \leftarrow \Phi_i[apmin] \cup aplibuf$
38. $msgatraso_i[apmin] \leftarrow \min_{msgatraso_i[apmin], apmax}$

```
    //Este laço determina se eventos seguintes podem ser realizados.
39.   faça
40.     novoℓ ← ℓ_i
41.     auxmsgat ← ∞
42.     execlivre ← VERDADE
43.     enquanto execlivre e novoℓ + 1 ≤ auxmsgat e novoℓ + 1 ≤ max_{j∈Viz_i} mintrf_i[j] faça
44.       novoℓ ← novoℓ + 1
45.       auxmsgat ← min{auxmsgat, msgatraso_i[novoℓ]}
46.       para cada vizinho j ∈ Viz_i faça
47.         se novoℓ − ultenv_i[j] > 1 e maxtrf_i[j] ≤ novoℓ então
48.           execlivre ← FALSO
49.       se não execlivre então
50.         novoℓ ← novoℓ − 1
51.     se novoℓ ≠ ℓ_i então
52.       se une_msgs então
53.         Φ' ← APLIC_i(novoℓ, ∪_{ℓ_i<ℓ'≤novoℓ} Φ_i[ℓ'])
54.       senão
55.         Φ' ← Φ' ∪ APLIC_i(ℓ', Φ_i[ℓ']) para cada ℓ' ∈ ]ℓ_i, novoℓ]
56.       ℓ_i ← novoℓ
57.     envie APLI_MSG(ℓ_i, S_i^+(ℓ_i, j), T_i^+(ℓ_i, j), Φ'[j]) para cada vizinho j ∈ Viz_i
58.   enquanto novoℓ + 1 > auxmsgat
```

No intuito de simplificar o algoritmo, omitimos os trechos responsáveis pela terminação. Para determinar o fim da execução de Γ, a ferramenta emprega o algoritmo definido em [10]. Isto obriga Γ a ter somente uma tarefa iniciando sua execução e, por enquanto, esta é uma limitação de Γ. Neste algoritmo de terminação, a terminação global é determinada pela tarefa que inicia a execução - no caso, a tarefa iniciadora - quando todas as tarefas se encotram no estado de terminação local. Um mecanismo de propagação de mensagens é utilizado para identificar tal situação. O estado de terminação local em uma tarefa é alcançado quando a mesma se encontra inativa e não possui mensagens a serem recebidas por eventos em relógios lógicos futuros. Quando estado de terminação global é

alcançado, uma mensagem com essa informação é propagada pela tarefa iniciadora.

A variável ℓ_i terá o papel de relógio lógico. Os eventos ocorrerão, portanto, de acordo com ela. As demais variáveis de SINC_i serão explicadas adiante.

3.3.1. Mensagens

O algoritmo da ferramenta só trata dois tipos de mensagens: INI_MSG e APLI_MSG. As mensagens do tipo INI_MSG são trocadas apenas na parte inicial do algoritmo por meio de uma propagação simples e têm o objetivo de tornar as tarefas prontas para o início da execução da aplicação Γ. Elas levam consigo o parâmetro *inst_ini* que é um conteúdo determinado pela tarefa iniciadora - possivelmente a instância da aplicação - a ser passado para APLIC_i na sua primeira chamada em todas as tarefas i.

As variáveis $iniciado_i$ e $contini_i$ estão diretamente ligadas a INI_MSG. Quando uma tarefa i recebe INI_MSG pela primeira vez, determina pai_i, altera $iniciado_i$ para $VERDADE$, encaminha a mesma mensagem para os demais vizinhos, com exceção de pai_i. Através de $contini_i$, i sabe se recebeu essa mensagem de todos os vizinhos. Quando isso acontece, a tarefa i envia INI_MSG para pai_i e já pode iniciar a execução de Γ, chamando pela primeira vez APLIC_i. Nesta primeira execução de APLIC_i, ℓ_i ainda é 0 e *inst_ini*, na condição de $\Phi(\ell_i)$, deverá ser sua entrada.

Mensagens do tipo APLI_MSG são enviadas para todos os vizinhos de uma tarefa quando ela realiza um evento de Γ, ou seja, ao chamar APLIC. Ao ser enviada pela tarefa i para a tarefa j, $\text{APLI\_MSG}(\ell_i, S_i^+(\ell_i, j), T_i^+(\ell_i, j), \Phi'[j])$ tem dois objetivos. O primeiro é avisar a j que i realizou o evento de relógio ℓ_i, indicando também os valores das funções passo e limite de saída para este relógio. O segundo é carregar em $\Phi'[j]$ o conteúdo das mensagens geradas por Γ resultantes deste evento e destinadas, obviamente, a j. Ainda que $\Phi'[j]$ seja vazio, a mensagem deverá ser enviada por conta do seu primeiro objetivo. As mensagens geradas por Γ na tarefa i são a saída de APLIC_i, como é possível perceber no algoritmo.

3.3.2. Funcionamento

Ao receber uma mensagem APLI_MSG($j\ell, js, jt, \Phi'[i]$), a tarefa i atualiza a variável $ultenv_i[j]$ com $j\ell$. Dessa forma, $ultenv_i[j]$ sempre armazena o valor de relógio correspondente ao evento mais recente de j cuja mensagem APLI_MSG já chegou em i. Como os canais de comunicação são FIFO, $ultenv_i[j]$ é sempre crescente no decorrer da execução da ferramenta. Em seguida, são calculados, através de js e jt, os valores das variáveis $apmin$ e $apmax$ as quais determinam, respectivamente, os relógios mínimo e máximo dos eventos que poderão receber a mensagem em $\Phi'[j]$. A variável $mintrf_i[j]$ armazena o valor do último $apmin$ calculado para uma mensagem recebida de j e é utilizada no cálculo do $apmin$ seguinte. Isto garante que a mensagem do $apmin$ a ser calculado não seja entregue a um evento de relógio menor ao de um evento que receberá uma mensagem anterior enviada por j. A variável $maxtrf_i[j]$ guarda o máximo $apmax$ das as mensagens vindas de j. A sua utilidade será explicada mais adiante. Após o cálculo de $apmin$ e $apmax$, se $apmax < apmin$, então o intervalo de relógios atribuído ao evento que deverá receber esta mensagem é vazio. Isto não é permitido, pois caracteriza uma execução de Γ em que esta mensagem não é recebida, como foi colocado na seção anterior quando enumeramos algumas propriedades de execução da ferramenta. A execução é então finalizada. Caso $apmax \geq apmin$, a mensagem é então armazenada em $\Phi_i[apmin]$, espaço de memória disponível para as mensagens a serem recebidas pelo evento realizado no relógio $apmin$. O evento de maior relógio que pode receber as mensagens em um determinado $\Phi_i[p]$ é o valor do menor $apmax$ das suas mensagens. Assim, é necessário uma variável que armazene este valor, a qual deverá ser atualizada sempre que uma nova mensagem for inserida em $\Phi_i[p]$. A variável para este propósito é $msgatraso_i[p]$. Logo, após o cálculo de $apmin$ e $apmax$, se $msgatraso_i[apmin] > apmax$, então $msgatraso_i[apmin]$ receberá o valor de $apmax$.

O próximo passo do algoritmo é verificar se os eventos relacionados aos próximos relógios daquela tarefa podem ser realizados. O laço da linha 28 será explicado mais adiante. Ao final do laço da linha 31, a variável $novo\ell$ terá o valor do relógio até quando

poderão ser realizados eventos, a partir de $\ell_i + 1$. Se $novo\ell = \ell_i$, então nenhum evento poderá ser realizado. As variáveis *auxmsgat* e *execlivre* auxiliam na determinação de $novo\ell$ e três condições são verificadas no laço para que $novo\ell$ seja incrementada. A primeira é *execlivre* ser $VERDADE$. O valor de *execlivre* se torna $FALSO$ se houver a possibilidade de algum vizinho j de i enviar uma mensagem que só poderá ser recebida até $novo\ell$. Isso é verificado na linha 34 e, se ocorrer, $novo\ell$ volta a ter seu valor antigo ($novo\ell - 1$) e o laço não é mais executado, por conta do valor de *execlivre*. A segunda condição é $novo\ell + 1 \leq auxmsgat$ e se deve ao seguinte fato. Para que o conjunto de eventos com relógios no intervalo de l_r até l_s, $l_s \geq l_r$, possa ser executado como um único evento, é necessário que $msgatraso_i[k]$, $\forall l_r \leq k \leq l_s$, seja menor ou igual a l_s. A variável *auxmsgat* guarda, portanto, o menor valor de $msgatraso_i[k]$, $\ell_i + 1 \leq k \leq novo\ell$. Se $novo\ell + 1 > auxmsgat$ para algum $novo\ell$, então o evento de relógio $novo\ell + 1$ não pode ser realizado em conjunto com os eventos do intervalo $[\ell_i + 1, novo\ell]$. No entanto, é possível que o evento de relógio $novo\ell + 1$ tenha condições de ser executado imediatamente depois dos eventos do intervalo citado serem realizados. Esta é a razão do laço da linha 28. A terceira condição é $novo\ell + 1 \leq \max_{mintrf_i[j], \forall j \in Viz_i}$. Isso garante que somente os eventos cujos relógios receberam uma mensagem APLI_MSG, e isso é determinado no cálculo de *apmin*, sejam executados. Após a execução do laço da linha 31, se $\ell_i \neq novo\ell$, então todos os eventos do intervalo $[\ell_i + 1, novo\ell]$ podem ser realizados.

4

Algoritmos Distribuídos para o Problema de Fluxo Máximo

Um dos nossos objetivos neste trabalho é mostrar que algoritmos parcialmente síncronos podem ser boas alternativas diante de algoritmos síncronos e assíncronos. A nosso conhecimento, não estão disponíveis na literatura bons algoritmos distribuídos para ambientes de memória distribuída para problema de fluxo máximo, o qual definiremos adiante. Existem algoritmos seqüenciais para esse problema, ao contrário, que são bastante eficientes. Por essa razão, encontrar algoritmos distribuídos que sejam expressivamente melhores que os algoritmos seqüencias talvez não seja viável e, de fato, esse não é o nosso objetivo. No entanto, o problema de fluxo máximo é um problema que ocorre com freqüência como subproblema de outros, de forma que uma boa implementação distribuída para ele pode ser útil.

O problema do fluxo máximo trata de situações em que um fluxo material deve atravessar uma rede de canais de forma a usufruir da melhor maneira possível dos recursos destes canais, isto é, de forma a permitir que o máximo deste fluxo possa atravessar a rede. Líquidos escoando por dutos, corrente elétrica passando por fios e informação transitando em redes de comunicação são exemplos dessas situações dentre as mais comuns.

No início do capítulo, definiremos através do modelo de grafos, a noção de redes de fluxo e, em seguida, o problema. Na Seção 4.1 mostraremos a noção intuitiva da abordagem do pré-fluxo, suas operações básicas e o seu algoritmo seqüencial. Na seção seguinte, descreveremos as versões distribuídas que implementamos.

4.1. Redes de fluxo

O modelo matemático usado para representar essas redes as quais chamaremos de *redes de fluxo*, é um grafo $G = (V, E)$ direcionado onde V é o conjunto de vértices e cada arco $(u, v) \in E$ possui uma *capacidade* $c(u, v) \geq 0$. Se $(u, v) \notin E$ então $c(u, v) = 0$. Existem dois vértices especiais chamados *fonte* (s) e *sumidouro* (t). Consideraremos que $n = |V|$ e $m = |E|$. Vamos presumir ainda que para todo vértice $v \in V$, existe um caminho que vai da fonte ao sumidouro que passa por v. Um fluxo numa rede de fluxo é uma função $f : V \times V \rightarrow \mathbb{R}$ com as seguintes propriedades:

1. *Restrição de capacidade*: $f(u, v) \leq c(u, v), \forall u, v \in V$.
2. *Simetria*: $f(u, v) = -f(v, u), \forall u, v \in V$.
3. *Conservação de fluxo*: $\forall u \in V - \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$.

Encontrar o máximo fluxo que possa atravessar um rede de fluxo é o objetivo deste problema que é conhecido como *o problema do fluxo máximo*. Matematicamente, de acordo com os conceitos de redes de fluxo: se $\mid F \mid = \sum_{v \in V} f(v, s)$ então o problema do fluxo máximo consiste em achar o maior $\mid F \mid$ que obedeça às restrições de f.

4.2. Algoritmo segundo a abordagem do pré-fluxo

Diversas algoritmos existem para a resolução do problema do fluxo máximo. Em [11] encontramos um estudo comparativo de dez desses algoritmos. Os algoritmos relacionados neste trabalho seguem a abordagem conhecida como *pré-fluxo* [6]. Um pré-fluxo é uma

função $f : V \times V \to \mathbb{R}$ que possui as propriedades de simetria e restrição de capacidade, mas é flexível quanto à conservação de fluxo da seguinte forma: para todo $u \in V - \{s\}$, $\sum_{v \in V} f(v, u) \geq 0$.

Para o entendimento desta abordagem, precisamos conhecer alguns conceitos. Considere uma rede de fluxo G e um pré-fluxo f sobre G.

Rede residual: Rede formada pelos arcos da rede de fluxo que admitem algum fluxo adicional. A quantidade de fluxo que ainda pode passar do vértice u ao vértice v de V será conhecida por *capacidade residual*, denotada por $c_f(u, v)$ e calculada da seguinte forma: $c_f(u, v) = c(u, v) - f(u, v)$. Assim, a rede residual de G relativa a f será $G_f = (V, E_f)$ tal que $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$.

Excesso de fluxo: Fluxo sobre um vértice u de V dado por $e(u) = \sum_{v \in V} f(v, u)$.

Vértice Ativo: Vértice que possui excesso de fluxo.

A seguir, de acordo com [9], mostraremos a intuição por trás da idéia do pré-fluxo e suas operações básicas.

4.2.1. A idéia do algoritmo e suas operações básicas

O funcionamento do algoritmo pode ser resumido da seguinte forma. No momento inicial, a fonte é escolhida para enviar o máximo de fluxo possível através de seus arcos, isto é, fluxo necessário para preencher a capacidade de todos os arcos que partem dela. A partir desse pré-fluxo inicial, a execução progride iterativamente através de uma seqüência de pré-fluxos até que um fluxo máximo seja encontrado.

No algoritmo de pré-fluxo, os vértices possuem duas propriedades especiais. A primeira diz que cada vértice possui um "reservatório", o qual retém o excesso de fluxo que chega neste vértice. A segunda atribui a cada vértice um *rótulo*. Este rótulo é uma medida de altura do vértice, determinando como o fluxo é encaminhado pela rede: só é possível enviar fluxo de vértices mais altos para vértices mais baixos.

Os rótulos dos vértices variam durante a execução do algoritmo. Inicialmente, todos os vértices possuem rótulo 0, exceção feita à fonte que recebe rótulo n. Em cada iteração, um vértice que possui fluxo em seu reservatório descarrega parte desse fluxo, o máximo possível, pelos seus arcos. Quando um vértice recebe um fluxo, ele o armazena no seu reservatório. Em algum momento, pode acontecer a seguinte situação. Para todo vértice u, se u possui excesso e os arcos que partem dele possuem ainda alguma capacidade de receber fluxo, então os vértices v da outra extremidade desses arcos possuem rótulo maior ou igual ao rótulo de u. Uma operação denominada *re-rotulamento* é realizada e consiste em re-rotular o vértice u que esteja na situação descrita, com um valor uma unidade maior do que o rótulo do vértice vizinho de menor rótulo cujo arco possua capacidade para receber fluxo. Dessa forma, estamos fazendo com que u "suba" em relação a pelo menos um de seus vizinhos que pode receber fluxo, possibilitando assim que este fluxo possa escoar. Assim, após essa operação, u já pode enviar fluxo por pelo menos um de seus arcos.

Pode ocorrer de um vértice u abaixo da fonte ter enviado todo o fluxo possível, mas ainda possuir excesso. Ou seja, os únicos vértices para onde ele pode enviar fluxo são vértices acima dele. Isso significa que todo fluxo partindo de u que pode chegar ao sumidouro foi enviado. Nesta situação, u deve, portanto, devolver à fonte o que ainda possui. As mesmas operações de envio de fluxo e re-rotulamento são realizadas e u ficará acima da fonte.

4.2.2. Operações básicas

Durante a execução do algoritmo de acordo com o pré-fluxo, são duas as operações básicas realizadas, conhecidas como *DESCARGA* e *RE-ROTULAMENTO*. Sejam uma rede de fluxo G, vértices u e v de G e um pré-fluxo f em um determinado momento da execução do algoritmo. Vamos tratar os valores capacidade residual de (u, v), fluxo em (u, v), excesso de u e altura de u respectivamente por $c[u, v]$, $f[u, v]$, $e[u]$ e $h[u]$. No início da execução do algoritmo, temos, para todo par de vértices u e v, que $c[u, v] = c(u, v)$ e $f[u, v] = 0$.

As operações básicas estão descritas a seguir.

$DESCARGA(u, v)$: É a operação de envio de fluxo de um vértice u a outro vértice v através do arco (u, v). É permitido que uma descarga seja realizada quando u possui excesso e, para algum v, $c[u, v] > 0$ e $h[u] = h[v]+1$, onde $h[u]$ é o valor do rótulo de u. Consiste em enviar $min\{e[u], c[u, v]\}$ unidades de fluxo de u para v. No decorrer do texto, também chamaremos a operação de descarga de *envio de fluxo*.

DESCARGA(u, v)
1. $aux \leftarrow min\{e[u], c[u, v]\}$
2. $f[u, v] \leftarrow f[u, v] + aux$
3. $f[v, u] \leftarrow -f[u, v]$
4. $e[u] \leftarrow e[u] - aux$
5. $e[v] \leftarrow e[v] + aux$
6. $c[u, v] \leftarrow c[u, v] - f[u, v]$
7. $c[v, u] \leftarrow c[v, u] - f[v, u]$

$RE - ROTULAMENTO(u)$: É a operação de re-rotulamento. É realizada quando u possui excesso e, para todo $v \in V$, se $c[u, v] > 0$ então $h[u] \leq h[v]$. Consiste em aumentar o valor do rótulo de u.

RE-ROTULAMENTO(u)
1. $h[u] \leftarrow 1 + min_{(u,v)\in E_f} h[v]$

As operações básicas nunca são realizadas pela fonte e pelo sumidouro. Assim, nos algoritmos descritos a seguir, considere que estes vértices nunca são escolhidos para a realizarem alguma dessas operações, ainda estando ativos.

4.2.3. Algoritmo seqüencial

A seguir, apresentaremos uma versão geral para o algoritmo seqüencial do pré-fluxo de acordo com [6]. Mais adiante, após a descrição de algumas heurísticas, descreveremos a versão implementada nesta dissertação.

O algoritmo seqüencial do pré-fluxo recebe como entrada o grafo $G = (V, E)$ e os vértices fonte s e sumidouro t. Após a inicialização das variáveis, são realizadas, sucessivamente e enquanto for possível, as operações de *DESCARGA* e *RE-ROTULAMENTO* através do método DESC_RE-ROT.

```
    DESC_RE-ROT(G, u)
1.  enquanto o vértice u ∈ V estiver ativo e houver v tal que c[u, v] > 0 e h[u] = h[v] + 1 faça
2.      Execute DESCARGA(u, v)
3.  se o vértice u ∈ V estiver ativo então
4.      Execute RE-ROTULAMENTO(u)
```

```
    Algoritmo PREFLUXO_GERAL(G, s, t)
1.  para cada vértice u ∈ V faça
2.      e[u] ← h[u] ← 0
3.  h[s] ← n
4.  para cada arco (u, v) ∈ E faça
5.      f[u, v] ← f[v, u] ← 0
6.  para cada vértice u tal que (s, u) ∈ E faça
7.      e[u] ← f[s, u] ← c(s, u)
8.      f[u, s] ← −c(s, u)
9.  enquanto algum vértice u ∈ V estiver ativo faça
10.     Execute DESC_RE-ROT(G, u)
```

A implementação feita nesta dissertação difere, como dito anteriormente, de PRE-FLUXO_GERAL pela inserção de diversas heurísticas. As informações a respeito dessas

heurísticas foram, basicamente, extraídas de [13].

Um fator essencial para o desempenho do algoritmo descrito acima é como escolher o vértice ativo u da linha 10. Em [8], Goldberg apresenta as seguintes abordagens de escolha.

FIFO algorithm: Nesta abordagem, os vértices, ao se tornarem ativos, são inseridos em uma fila e retirados à medida que se necessite escolher um. Sua complexidade é $O(n^3)$.

Highest-label algorithm: Nesta abordagem, o vértice ativo escolhido é sempre o de maior altura. Os vértices ativos são mantidos em uma *heap* para facilitar a escolha. Sua complexidade é $O(n^2 m^{1/2})$.

A abordagem implementada neste trabalho, tanto no algoritmo seqüencial quanto nos distribuídos, foi a segunda.

Um fator que pode ser nocivo ao desempenho do algoritmo é a forma como os vértices alteram suas alturas para poderem realizar a operação de envio de fluxo. O que pode ocorrer no decorrer do algoritmo é a formação de conjuntos de vértices os quais, em cada um destes conjuntos, os vértices aplicam as operações básicas sucessivamente somente enviando fluxo uns para os outros até que atinjam alturas superiores à da fonte, quando, finalmente, o excesso de fluxo que estava "circulando" por eles pode voltar para ela. Nos referiremos a esse efeito mais algumas vezes adiante e vamos chamá-lo de *efeito de operações em ciclo*. Quando um conjunto de vértices inicia este efeito é sinal de que o fluxo que circula por esses vértices não alcançará o sumidouro. Na verdade, não há mais caminhos na rede residual para que este fluxo alcance o sumidouro. O ideal seria, portanto, que estes vértices que não contribuirão mais para a acréscimo de fluxo no sumidouro fossem logo transportados para a altura da fonte.

Para amenizar o efeito de operações em ciclo, acrescentamos a heurística denominada *gap relabeling* apresentada por Ahuja e Orlin em [12] e descrita a seguir. Se há um valor h,

$1 \leq h \leq n$, tal que não existe nenhum vértice da rede com altura h, então os vértices com alturas maiores que h podem ter suas alturas aumentadas para o valor da altura da fonte. A questão é saber *quando* aplicar esta heurística no decorrer da execução do algoritmo. Se ela for aplicada poucas vezes, pode não proporcionar tanta economia de trabalho quanto poderia. Se for aplicada com muita freqüência, o trabalho extra desempenhado por ela pode não compensar os ganhos que ela traz. A maneira como escolhemos aplicar a *gap relabeling* em PreFluxo_Geral se mostrou satisfatória e foi da seguinte forma. Definimos, a princípio, a noção de *ciclo de trabalho* que é todo o trabalho (aplição das operações) realizado até que uma *gap relabeling* seja aplicada. No início do um ciclo de trabalho, a altura do vértice que está no topo da *heap* é armazenada. Este é o vértice ativo mais alto no momento. No decorrer do ciclo de trabalho, só podem entrar na *heap* vértices com altura menor ou igual a esta altura. Assim, quando a *heap* se tornar vazia um ciclo de trabalho é terminado e uma *gap relabeling* é aplicada. Os vértices ativos entram na *heap* e um novo ciclo de trabalho é realizado.

Empregamos ainda uma outra idéia proposta por Goldberg em [7], que é a execução do algoritmo do pré-fluxo em duas fases. Na primeira fase, as operações são realizadas somente nos vértices com alturas menores que a altura da fonte. A segunda fase é realizada quando já não existirem mais vértices ativos com alturas menores do que a da fonte. As operações são, então, realizadas sobre os demais vértices ativos.

Uma terceira modificação foi feita no PreFluxo_Geral e se baseia, em parte, na heurística conhecida como *global relabeling*, encontrada em [7] e [8]. Cada vértice, com exceção da fonte, recebe, o valor de altura que é a sua distância (o tamanho do menor caminho) ao sumidouro. Um método similar a uma busca em largura é empregado ($O(nm)$) para tal. Em nossa implementação, essa atribuição de alturas aos vértices é feita apenas no início do algoritmo, enquanto que a heurística do *global relabeling* realiza essa atribuição algumas vezes no decorrer da execução.

O algoritmo do pré-fluxo implementado está descrito a seguir.

Algoritmo PreFluxo(G, s, t)
1. **para** cada vértice $u \in V - s$ **faça**
2. $\quad h[u] \leftarrow$ distância para o sumidouro
3. $\quad e[u] \leftarrow 0$
4. $h[s] \leftarrow n$
5. $e[s] \leftarrow 0$
6. **para** cada arco $(u, v) \in E$ **faça**
7. $\quad f[u, v] \leftarrow f[v, u] \leftarrow 0$
8. **para** cada vértice u tal que $(s, u) \in E$ **faça**
9. $\quad e[u] \leftarrow f[s, u] \leftarrow c[s, u]$
10. $\quad f[u, s] \leftarrow -c[s, u]$
11. $fase \leftarrow 1$
12. **enquanto** algum vértice de V estiver ativo **e** $fase = 1$ **faça**
13. $\quad maxalt = \max_{w \in V, e[w] > 0, h[w] < n} h[w]$
14. $\quad$ **enquanto** algum vértice $v \in V$ estiver ativo **e** $d[v] \leq maxalt$ **faça**
15. $\quad\quad$ **selecione** v tal que $d[v] = \max_{w \in V, e[w] > 0, d[w] \leq maxalt} d[w]$
16. $\quad\quad$ **Execute** Desc_Re-Rot(G, v)
17. $\quad$ **se** existe $\min_{1 < a < n} a$ tal que $\{x : x \in V, h[x] = a\} = \emptyset$ **então**
18. $\quad\quad$ **para** cada $x \in V$ tal que $n > h[x] > a$ **faça**
19. $\quad\quad\quad h[x] \leftarrow n$
20. $\quad$ **se** não há nenhum vértice ativo $u \in V$ tal que $h[u] < n$ **então**
21. $\quad\quad fase \leftarrow 2$
22. **enquanto** algum $u \in V$ estiver ativo **faça**
23. $\quad$ **Execute** Desc_Re-Rot(G, u)

4.3. Algoritmos distribuídos

Os algoritmos distribuídos definidos nesta seção se baseiam no algoritmo seqüencial que descrevemos há pouco. As tarefas deverão ter seus canais de comunicação determinados de acordo com a forma da distribuição da rede de fluxo - instância do problema - por elas. As funções de evento das tarefas poderão receber como entrada informações a respeito de

operações básicas realizadas por outras tarefas para que possam atualizar suas variáveis, assim como suas saídas dizem respeito à operações básicas de pré-fluxo realizadas por elas. Outras informações podem compor a entrada e a saída, mas são específicas de cada algoritmo distribuído que descreveremos adiante.

Seja G uma rede de fluxo, a qual deverá ser entrada para os algoritmos. G deverá ser distribuída entre as tarefas que resolverão o problema de forma que cada tarefa i ficará responsável por um subconjunto de vértices de G, o qual será denotado por V_i. Duas tarefas i e j terão um canal de comunicação entre elas se existir um arco da rede de fluxo formado por vértices de V_i e V_j. Uma exceção é feita à tarefa iniciadora, a qual poderá se comunicar com todas as demais tarefas independentemente dos vértices da rede de fluxo que ela possua. Uma tarefa i só poderá realizar operações básicas do pré-fluxo sobre vértices de V_i e, para duas tarefas quaisquer i e j, temos que $1 \geq |V_i| - |V_j| \geq -1$. A tarefa iniciadora, que, como já havíamos dito, é reponsável pela leitura e distribuição da instância do problema, também se encarrega de receber dados de todas as demais tarefas no decorrer do algoritmo a fim de aplicar uma heurística. O seu papel será melhor entendido mais adiante. Os dados de capacidade, capacidade residual, fluxo nos arcos, rótulo e excesso que interessam a cada tarefa i não se limitam somente aos vértices de V_i. É necessário que i conheça esses dados de todos os vértices que se relacionam com V_i para que seja possível a realização das operações básicas em V_i. Assim, o conjunto de vértices dos quais i possui as informações relacionadas será conhecido como Vt_i e definido da seguinte forma. $Vt_i = \{v : (u, v)$ ou $(v, u) \in E$ e $u \in V_i\}$. O subgrafo de um tarefa i será, portanto, o subgrafo de G induzido por Vt_i, denotado por $Gt_i = (Vt_i, Et_i)$.

Os algoritmos distribuídos de pré-fluxo devem seguir o modelo de TAREFA$_t$. Para obtermos execuções síncronas ou parcialmente síncronas, é necessário, no entanto, empregar a ferramenta de atribuiçao de relógios lógicos. Assim, para esses algortimos, descreveremos a função de evento e as funções que determinam as restrições de sincronização. As funções de evento não enviam, mas constroem as mensagens que serão enviadas pela ferramenta. Já o algoritmo assíncrono, não necessita da ferramenta por conta das suas

restrições de sincronização. Assim, o próprio algoritmo também é responsável pelo envio das mensagens.

Cada tarefa dos algoritmos distribuídos receberá seu subgrafo correspondente e inicializará suas variáveis de acordo com a função de inicialização a seguir.

```
    INICIALIZA_PREFLUXO(Gt_i, s, t, h)
1.  para cada vértice u ∈ Vt_i faça
2.      e_i[u] ← 0
3.  para cada arco (u, v) ∈ Et_i faça
4.      f_i[u, v] ← f_i[v, u] ← 0
5.  para cada vértice u tal que (s, u) ∈ Et_i faça
6.      e_i[u] ← f_i[s, u] ← c_i[s, u]
7.      f_i[u, s] ← −c_i[s, u]
8.  atualize h_i a partir de h
```

Esta função de inicialização é comum a todas as versões distribuídas do algoritmo do pré-fluxo. Através dela, cada tarefa i recebe seu subgrafo Gt_i, além da identificação dos vértices fonte e sumidouro e das alturas dos vértices de Gt_i no vetor h.

A seguir, descreveremos as versões distribuídas implementadas. Como já havíamos comentado, estas serão a assíncrona, a síncrona e a parcialmente síncrona, as duas últimas executadas com a ferramenta.

4.3.1. Algoritmo assíncrono

No algoritmo assíncrono, cada tarefa, através da função de evento, realiza o máximo possível de operações DESCARGA e RE-ROTULAMENTO. Em seguida, caso ela tenha operado sobre algum ou alguns vértices que interessam a outra tarefa, ela deverá enviar-lhe uma mensagem cujo conteúdo é formado por essas operações. Uma tarefa, ao receber uma mensagem com operações realizadas por outra, terá que modificar dados de altura e/ou capacidade residual e, daí, a possibilidade de novos vértices ficarem ativos.

Neste algoritmo assíncrono, não é permitido, ao contrário das outras versões distribuídas, que uma tarefa deixe algum vértice ativo ao final de um evento. Isto é necessário, porque não há garantias de que ela venha a receber alguma mensagem futuramente, o que a impossibilitaria de operar com o vértice ativo.

Na execução deste algoritmo assíncrono é possível que ocorra um efeito indesejado. Considere as tarefas i e j que possuem, respectivemente, os vértices u e v e considere a existência da arco (u, v) na rede de fluxo. Se i decide que u deverá enviar fluxo para v, então $h_i[u] = h_i[v] + 1$ em i nesse momento. No entanto, quando j é informado desse envio de fluxo através de uma mensagem, é possível que $h_j[u] \neq h_j[v] + 1$, pois j poderá ter alterado a altura de v. Esse fluxo não é aceito e j envia essa informação para i [1].

Uma outra questão decorre da aplicação da *gap relabeling*. Como os vértices estão distribuídos entre as tarefas, em alguns momentos a tarefa centalizadora requisita suas alturas para aplicar a heurística. Quando uma tarefa envia as alturas dos seus vértices para a tarefa iniciadora, ela deveria manter essas alturas inalteradas até uma ordem da tarefa iniciadora. No entanto, para que as tarefas não fiquem ociosas, permitimos que elas trabalhem normalmente. Porém, as alturas enviadas para a tarefa iniciadora são guardadas e sobre elas é aplicada a heurística, caso a tarefa iniciadora encontre uma altura sem vértices.

O algoritmo PreFluxoAssinc realiza as operações de Pré-fluxo. Em seguida, o algoritmo assíncrono A_PreFluxo.

PreFluxoAssinc(ENTRADA)

1. **para** cada informação de ENTRADA **faça**
2. **se** *RE-ROTULAMENTO* **ou** *DESCARGA devolvida* **então**
3. **atualize** as variáveis correspondentes
4. **senão se** *DESCARGA normal* através de (u, v) **então**
5. **se** $h_i[u] = h_i[v] + 1$ **então**
6. **atualize** as variáveis correspondentes
7. **senão**
8. **atualize** SAIDA com fluxo *devolvido* através de (v, u)
9. **enquanto** existir $u \in V_i$ ativo **faça**
10. **Execute** Desc_Re-Rot(G, u)
11. **atualize** SAIDA com fluxo através de (u, v) ou com $h_i[u]$
12. **retorne** SAIDA

A_PreFluxo$_i$

1. **se** i é tarefa iniciadora **então**
2. **leia** o grafo $G = (V, E)$ do problema, os vértices fonte s e sumidouro t
3. **para** cada vértice $u \in V - s$ **faça**
4. $h[u] \leftarrow$ distância para o sumidouro
5. **determine** Gt_j para cada tarefa j com s e t pertencendo à tarefa iniciadora
6. **envie** mensagem INI para cada tarefa j com Gt_j, s, t e as alturas h de G
7. Inicializa_PreFluxo(Gt_i,s,t,h)
8. $saida \leftarrow$ PreFluxoAssinc()
9. **envie** mensagem OP para as tarefas vizinhas de acordo com *saida*
10. $contgap \leftarrow 0$
11. $auxh_i \leftarrow h_i$
12. **envie** mensagem GAPREQ para todos os vizinhos
13. **enquanto** não for determinada a terminação global **faça**
14. **receba** msg_i de $j \in Viz_i$
15. **se** msg_i = INI(Gt_i,s,t,h) **então**
16. Inicializa_PreFluxo(Gt_i,s,t,h)
17. $saida \leftarrow$ PreFluxoAssinc()
18. **envie** mensagem OP para as tarefas vizinhas de acordo com *saida*
19. **senão se** msg_i = OP(*oper*) **então**
20. $saida \leftarrow$ PreFluxoAssinc(*oper*)
21. **envie** mensagem OP para as tarefas vizinhas de acordo com *saida*
22. **senão se** msg_i = GAPREQ **então**
23. $auxh_i \leftarrow h_i$
24. **envie** mensagem GAPINFO para a tarefa iniciadora com as alturas h_i dos vértices de G_i

```
25.          se msg_i = GAPINFO(inteiro a ou vetor de alturas h_j) então
26.              se i é tarefa iniciadora então
27.                  se contgap = |Viz_i| então
28.                      se há min_{1<a<n} a tal que {x : h[x] = a} = ∅ e {y : n > h[y] > a} ≠ ∅ então
29.                          para cada x ∈ V(Gt_i) tal que n > auxh_i[x] > l faça
30.                              h_i[x] ← n
31.                          envie uma mensagem GAPINFO com a para todos os vizinhos
32.                      contgap ← 0
33.                      auxh_i ← h_i
34.                      envie uma mensagem GAPREQ para todos os vizinhos
35.                  senão
36.                      contgap ← contgap + 1
37.                      atualize h com as alturas h_j de j
38.              senão
39.                  para cada x ∈ V(Gt_i) tal que n > auxh_i[x] > a faça
40.                      h_i[x] ← n
```

A tarefa iniciadora inicia a execução fazendo a leitura e distribuição do grafo de entrada. Em seguida, ela executa PreFluxoAssinc pela primeira vez e requisita, através de uma mensagem GAPREQ as alturas dos vértices das demais tarefas para aplicar *gap relabeling*. Veja que, antes disso, ela copia suas alturas h_i no vetor $auxh_i$. Uma tarefa ao receber uma mensagem INI, inicializa suas variáveis e realiza PreFluxoAssinc. Uma mensagem OP faz com que uma tarefa execute PreFluxoAssinc com o conteúdo (*oper*) daquela mensagem.

Uma mensagem GAPREQ é sempre enviada pela tarefa iniciadora. A tarefa que a recebe faz uma cópia das alturas atuais dos seus vértices e as envia para a tarefa iniciadora em uma mensagem GAPINFO. Quando recebe uma mensagem desse tipo, a tarefa iniciadora atualiza o vetor h com as alturas enviadas nessa mensagem. Após receber de todas as tarefas, ela busca uma altura sem vértices, de acordo com a heurística *gap relabeling*. Se encontrar, envia esse valor para as demais tarefas em uma mensagem GAPINFO.

Em seguida, ela requisita mais uma vez as alturas dos vértices das demais tarefas para uma nova aplicação da heurística. Quando uma tarefa que não é a iniciadora recebe uma mensagem GAPINFO, o conteúdo desta é a altura encontrada na *gap relabeling*.

A terminação do algoritmo, cuja descrição foi omitida, está de acordo com [10].

4.3.2. Algoritmo síncrono

No Capítulo 2 definimos execuções síncronas e assíncronas de acordo com funções passo e prazo. Dessa forma, neste algortimo síncrono de pré-fluxo - assim como deve ser feito em qualquer outro - consideraremos que as funções de passo e prazo são todas constantes iguais a 1.

Os algoritmos síncronos têm, como uma de suas principais características, a possibilidade de realizarem execuções que tendem a serem parecidas com a de um algoritmo seqüencial, com a propriedade, naturalmente, de que muito trabalho é realizado simultaneamente nas tarefas. A natureza do modelo síncrono permite que heurísticas utilizadas em uma versão seqüencial de um algoritmo possam ser, com uma relativa facilidade, herdadas pela versão distribuída síncrona correspondente. Além disso, efeitos colaterais que venham a ocorrer nas versões distribuídas assíncronas têm a possibilidade de serem evitados, visto que tais efeitos não são inerentes a um algoritmo seqüencial.

Neste sentido, procuramos desenvolver um algoritmo síncrono para resolver o problema do fluxo máximo que fosse semelhante ao algoritmo seqüencial que já tínhamos. Procuramos evitar o efeito de fluxo negado que ocorre no algoritmo assíncrono através da restrição - por altura - dos vértices ativos a serem operados em cada pulso por cada tarefa. Procuramos também aplicar as heurísticas com os mesmos critérios utilizados em PreFluxo.

A função PreFluxoSinc realiza as operações de pré-fluxo. Em seguida, a função S_PreFluxo$_i$, que é a função evento para as execuções síncronas.

PreFluxoSinc(ENTRADA)

1. **para** cada informação de ENTRADA **faça**
2. **atualize** as variáveis correspondentes
3. **enquanto** existir vértices ativos $u \in V_i$ tais que $h_i[u] = maxalt$ **faça**
4. **enquanto** $h[u] < maxalt$ **faça**
5. **execute** *DESCARGA* ou *RE-ROTULAMENTO* sobre u
6. **atualize** SAÍDA com fluxo através de (u, v) ou com $h_i[u]$
7. **retorne** SAÍDA

S_PreFluxo$_i(\ell_i, MSG_i)$

1. **se** $\ell_i = 0$ **então**
2. $fase \leftarrow 1$
3. $estado \leftarrow TRABALHO$
4. **determine** Gt_i, s e t a partir de MSG_i
5. **se** i é tarefa iniciadora **então**
6. **para** cada vértice $u \in V - s$ **faça**
7. $h[u] \leftarrow$ distância para o sumidouro
8. $maxalt \leftarrow \max_{e[u]>0, h[u]<n, u \in V-\{s,t\}} h[u]$
9. **Incorpore** $maxalt$ e as alturas h de G para cada tarefa j à SAÍDA

10. **senão se** $\ell_i > 0$ **então**
11. **se** $\ell_i = 1$ **então**
12. **se** i não é tarefa iniciadora **então**
13. **atualize** h e $maxalt$ a partir de $MSG_i(\ell_i)$
14. INICIALIZA_PREFLUXO(Gt_i, s, t, h)
15. **se** $estado = GAP1$ **então**
16. $estado \leftarrow GAP2$
17. **se** i é tarefa iniciadora **então**
18. **atualize** h com as alturas das demais tarefas em $MSG_i(\ell_i)$
19. **se** existe a tal que $\{x : x \in V, h[x] = a\} = \emptyset$ **então**
20. $a \leftarrow \min_{1<a<n} a$ tal que $\{x : x \in V, h[x] = a\} = \emptyset$
21. **senão**
22. $a \leftarrow n$
23. **Incorpore** a para todas as tarefas à SAÍDA
24. **senão se** $estado = GAP2$ **então**
25. $estado \leftarrow GAP3$
26. **se** i não é tarefa iniciadora **então**
27. **atualize** a enviado pela tarefa iniciadora em $MSG_i(\ell_i)$
28. **para** cada $x \in Vt_i$ tal que $n > h[x] > a$ **faça**
29. $h[x] \leftarrow n$
30. $maxalt_i \leftarrow 0$
31. $maxalt_i \leftarrow \max_{e_i[u]>0, h_i[u]<n, u \in V_i - \{s,t\}} h_i[u]$
32. **Incorpore** $maxalt_i$ para a tarefa iniciadora à SAÍDA
33. **senão se** $estado = GAP3$ **então**
34. $estado \leftarrow GAP4$
35. **se** i é tarefa iniciadora **então**
36. $maxalt \leftarrow \max_{k \in \{i\} \cup Viz_i} maxalt_k$
37. **Incorpore** $maxalt$ para cada tarefa à SAÍDA

```
38.     senão
39.         se estado = GAP4 então
40.             estado ← TRABALHO
41.             atualize maxalt enviado pela tarefa iniciadora em MSG_i(ℓ_i)
42.             se maxalt = 0 então
43.                 fase ← 2
44.                 maxalt ← n
45.         saida ← PreFluxoSinc(MSG_i(ℓ_i))
46.         se i é tarefa iniciadora e fase = 2 e e_i[s] + e_i[t] = 0 então
47.             determine terminação global
48.         senão
49.             Incorpore saida para as tarefas vizinhas à SAÍDA
50.             maxalt ← maxalt − 1
51.             se fase = 1 e maxalt = 0 então
52.                 estado ← GAP1
53.                 se i é tarefa iniciadora então
54.                     atualize h com as alturas h_i
55.                 senão
56.                     Incorpore as alturas h_i para a tarefa iniciadora à SAÍDA
57.             senão se fase = 2 e maxalt = n − 1 então
58.                 estado ← GAP3
59.                 maxalt_i ← max_{e_i[u]>0, h_i[u]>=n, u∈V_i−s} h_i[u]
60.                 se i não é tarefa iniciadora então
61.                     Incorpore maxalt_i para a tarefa iniciadora à SAÍDA
62.     retorne SAÍDA
```

Durante a execução de S_PreFluxo, as tarefas podem estar nos estados $GAP1$, $GAP2$, $GAP3$, $GAP4$ ou $TRABALHO$. Nos estados $GAP1$ e $GAP2$ as tarefas estão envolvidas na aplicação de *gap relabeling*. No estado $TRABALHO$, as tarefas executam PreFluxoSinc. Em PreFluxoSinc, são realizadas as operações somente nos vértices que estiverem com altura $maxalt$. O valor de $maxalt$ é determinado pela tarefa iniciadora uma vez no relógio lógico inicial e, no decorrer da execução do algoritmo, no estado $GAP3$.

Uma vez no estado $TRABALHO$, as tarefas só saem dele quando *maxalt* se tornar 0, na fase 1, ou $n-1$, na fase 2. Do contrário, elas realizam um novo evento, no relógio lógico seguinte, no mesmo estado e decrementam *maxalt* de uma unidade.

No relógio lógico inicial, a tarefa iniciadora realiza a distribuição do grafo do problema. No relógio lógico seguinte, as tarefas inicializam suas variáveis e, já no estado $TRABALHO$ e com *maxalt* determinado, executam PreFluxoSinc. A terminação é, então, testada pela tarefa iniciadora. Ela ocorre quando as tarefas estiverem na fase 2 e quando o fluxo que parte da fonte for igual ao que chega no sumidouro. Caso não aconteça a terminação, *maxalt* é decrementado.

Quando *maxalt* chega a 0, na fase 1, a tarefas passam para o estado $GAP1$ e enviam a alturas dos seus vértices para a tarefa iniciadora para que ela possa iniciar a aplicação da *gap relabeling*. A partir de $GAP1$, as tarefas assumem os estados $GAP2$, $GAP3$ e $GAP4$ em relógios lógicos consecutivos. No estado $GAP1$, a tarefa iniciadora busca a altura sem vértices da heurística *gap relabeling*. No estado $GAP2$ as tarefas procedem a atualização das alturas dos seus vértices decorrente da aplicação da heurística, de acordo com o valor de altura enviado pela tarefa iniciadora ao final do evento do relógio lógico anterior, no estado $GAP1$. Ao final da execução em $GAP2$, as tarefas determinam em $maxalt_i$ a maior altura dentre as de seus vértices ativos e enviam este valor para a tarefa iniciadora. No relógio lógico seguinte, já no estado $GAP3$, a tarefa iniciadora calcula o *maxalt* global a partir dos $maxalt_i$ locais de todas as tarefas e envia este valor para elas. No evento do próximo relógio lógico, as tarefas estão no estado $GAP4$ e recebem o valor *maxalt* determinado pela tarefa iniciadora no evento do relógio lógico anterior. Se esse valor for 0, então as tarefas passam para o estado fase 2, pois já não possuem vértices ativos abaixo da fonte.

Quando *maxalt* é decrementado na fase 2 e se torna $n-1$, as tarefas vão para o estado $GAP3$, pois não há a aplicação da heurística *gap relabeling* nessa fase. Elas determinam seus $maxalt_i$ locais, tal como já foi explicado, e enviam para a tarefa iniciadora.

4.3.3. Algoritmo parcialmente síncrono

O algoritmo parcialmente síncrono reúne características dos algoritmos síncrono e assíncrono. A aplicação da heurística *gap relabeling* no algoritmo síncrono, por exemplo, é mais fácil e, como veremos pelos resultados, mais eficiente do que no algoritmo assíncrono. Assim, o algoritmo parcialmente síncrono herda do síncrono toda a idéia da aplicação da heurística. O algoritmo síncrono, no entanto, realiza uma grande quantidade de eventos, porque em cada evento no estado $TRABALHO$, somente os vértices com um determinado valor de altura podem ser operados. Apesar de essa abordagem fazer com que a execução síncrona realize uma quantidade bastante próxima de operações da execução seqüencial, o custo de comunicação para a realização de muitos eventos é alto.

O algoritmo parcialmente síncrono adota a seguinte abordagem. Durante a execução dos eventos em que ocorrem as operações básicas do pré-fluxo, são operados os vértices que possuem alturas em um determinado intervalo. Esse intervalo pode mudar de evento para evento. Além disso, os eventos nesses relógios são realizados com uma restrição de sincronização mais relaxada, de forma que a execução nesse período não é síncrona. Isto possibilitará que os fluxos negados inerentes ao algoritmo assíncrono aconteçam. No entanto, o benefício da independência entre os relógios e a maior quantidade de operações realizadas por relógio compensam este efeito negativo.

O algoritmo parcialmente síncrono é bastante parecido com o síncrono. Suas maiores diferenças ocorrem entre as funções PreFluxoSinc e PreFluxoPS. Em PreFluxoPS, os vértices a serem operados também dependem de *maxalt*. Serão operados todos aqueles que tiverem altura maior ou igual a este valor e menor ou igual a *maxaltini*, o qual é determinado no início de cada ciclo de execução de eventos no estado $TRABALHO$. A restrição imposta por *maxaltini* evita que vértices sofram re-rotulamentos excessivos quando eles podem ir diretamente para a altura da fonte na próxima aplicação de *gap relabeling*. Veja que essa restrição de alturas só é imposta na fase 1. Na segunda fase, todos os vértices ativos podem ser operados.

Em PS_PreFluxo, uma diferença com S_PreFluxo ocorre no decremento de *maxalt*. No algoritmo síncrono, a variável é decrementada de uma unidade, enquanto no parcialmente síncrono, ela é divida por 2. Essa diferença, como já comentamos, permite que mais vértices tenham a possibilidade de serem operados. Uma segunda e última diferença ocorre no fato de que o algoritmo parcialmente síncrono não impõe restrições às operações de vértices na segunda fase da execução. Como também não há aplicação de *gap relabeling* nesta fase, a partir do relógio em que ele tem início a execução ocorre de forma assíncrona.

PreFluxoPS(ENTRADA)

1. **para** cada informação de ENTRADA **faça**
2. **se** *RE-ROTULAMENTO* **ou** *DESCARGA devolvida* **então**
3. **Atualize** as variáveis correspondentes
4. **senão se** *PUSH normal* através de (u, v) **então**
5. **se** $h_i[u] = h_i[v] + 1$ **então**
6. **Atualize** as variáveis correspondentes
7. **senão**
8. **Incorpore** o fluxo *devolvido* através de (v, u) à SAÍDA
9. **se** $fase = 1$ **então**
10. **enquanto** existir vértices ativos $u \in V_i$ tais que $h_i[u] >= maxalt$ **e** $h_i[u] <= maxaltini$ **faça**
11. **enquanto** $h[u] < maxaltini$ **faça**
12. **Execute** *DESCARGA* ou *RE-ROTULAMENTO* sobre u
13. **Incorpore** o fluxo através de (u, v) ou $h_i[u]$ à SAÍDA
14. **senão**
15. **enquanto** existir vértices ativos $u \in V_i$ **faça**
16. **Execute** Desc_Re-Rot(G, u)
17. **Incorpore** o fluxo através de (u, v) ou $h_i[u]$ à SAÍDA
18. **retorne** SAÍDA

PS_PreFluxo(ℓ_i, MSG_i)

Entrada: $\ell_i = 0$, $MSG_i(0) = \emptyset$

1. $fase \leftarrow 1$
2. **determine** Gt_i, s e t a partir de MSG_i
3. **se** i é tarefa iniciadora **então**
4. **para** cada vértice $u \in V - s$ **faça**
5. $h[u] \leftarrow$ distância para o sumidouro
6. $maxalt \leftarrow \max_{e[u]>0, h[u]<n, u \in V-\{s,t\}} h[u]$
7. **Incorpore** $maxalt$ e as alturas h de G à SAÍDA

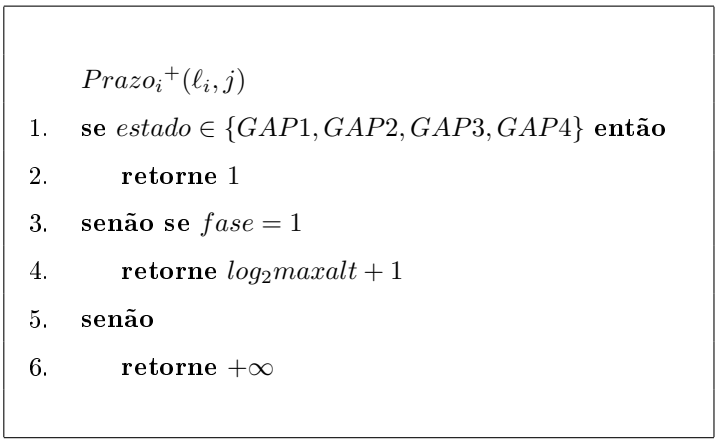
Entrada: $\ell_i > 0$, $MSG_i(\ell_i)$

9. **se** $\ell_i = 1$ **então**
10. **Atualize** h e $maxalt$ enviado pela tarefa iniciadora
11. INICIALIZA_PREFLUXO(Gt_i, s, t, h)
12. **se** $estado = GAP1$ **então**
13. $estado \leftarrow GAP2$
14. **se** i é tarefa iniciadora **então**
15. **Atualize** h com as alturas das demais tarefas em $MSG_i(\ell_i)$
16. **se** existe a tal que $\{x : x \in V, h[x] = l\} = \emptyset$ **então**
17. $a \leftarrow \min_{1<a<n} l$ tal que $\{x : x \in V, h[x] = a\} = \emptyset$
18. **senão**
19. $a \leftarrow n$
20. **Incorpore** a para todas as tarefas à SAÍDA
21. **senão se** $estado = GAP2$ **então**
22. $estado \leftarrow GAP3$
23. **se** i não é tarefa iniciadora **então**
24. **Atualize** a enviado pela tarefa iniciadora em $MSG_i(r)$
25. **para** cada $x \in Vt_i$ tal que $n > h[x] > l$ **faça**
26. $h[x] \leftarrow n$
27. $maxalt_i \leftarrow \max_{e_i[u]>0, h_i[u]<n, u \in V_i - \{s,t\}} h_i[u]$
28. **Incorpore** $maxalt_i$ para a tarefa iniciadora à SAÍDA
29. **senão se** $estado = GAP3$ **então**
30. $estado \leftarrow GAP4$
31. **se** i é tarefa iniciadora **então**
32. $maxalt \leftarrow \max_{k \in \{i\} \cup Viz_i} maxalt_k$
33. **Incorpore** $maxalt$ para cada tarefa à SAÍDA

```
34.     senão
35.         se estado = GAP4 então
36.             estado ← TRABALHO
37.             Atualize maxalt enviado pela tarefa iniciadora em MSG_i(ℓ_i)
38.             se maxalt = 0 então
39.                 fase ← 2
40.         saida ← PreFluxoSinc(MSG_i(ℓ_i))
41.         se i é tarefa iniciadora e fase = 2 e e_i[s] + e_i[t] = 0 então
42.             Determine terminação global
43.         senão
44.             Incorpore saida para as tarefas vizinhas à SAÍDA
45.             se fase = 1 então
46.                 maxalt ← maxalt/2
47.                 se maxalt = 0 então
48.                     estado ← GAP1
49.                     se i é tarefa iniciadora então
50.                         Atualize h com as alturas h_i
51.                     senão
52.                         Incorpore as alturas h_i para a tarefa iniciadora à SAÍDA
53.     Retorne SAÍDA
```

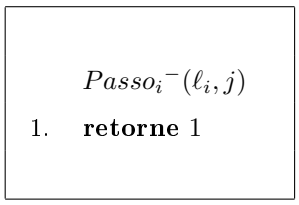
Seguem as funções passo e prazo para a execução parcialmente síncrona.

```
    Passo_i^+(ℓ_i, j)
1.  retorne 1
```



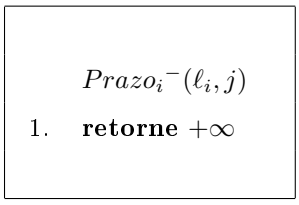
$Prazo_i{}^+(\ell_i, j)$

1. **se** $estado \in \{GAP1, GAP2, GAP3, GAP4\}$ **então**
2. **retorne** 1
3. **senão se** $fase = 1$
4. **retorne** $log_2 maxalt + 1$
5. **senão**
6. **retorne** $+\infty$



$Passo_i{}^-(\ell_i, j)$

1. **retorne** 1



$Prazo_i{}^-(\ell_i, j)$

1. **retorne** $+\infty$

As funções $Passo_i{}^-(\ell_i, j)$ e $Prazo_i{}^-(\ell_i, j)$ não estabelecem nenhuma restrição de relógios aos eventos do algoritmo parcialmente síncrono, ficando estas retrições sob responsabilidade das funções $Passo_i{}^+(\ell_i, j)$ e $Prazo_i{}^+(\ell_i, j)$. $Passo_i{}^+(\ell_i, j)$ sempre retorna 1, indicando que qualquer mensagem enviada pode por i ser recebida por j no relógio $\ell_i + 1$. A função $Prazo_i{}^+(\ell_i, j)$ é quem estabalece o sincronismo durante os eventos em que é aplicada a *gap relabeling* e o assincronimo nos eventos restantes. Lembre-se de que as funções passo são executadas ao final de um evento e seus valores são enviados em mensagens. Já as funções limite são executadas no início de um evento e seus resultados só iteressam à tarefa local. Assim, para entender $Prazo_i{}^+(\ell_i, j)$, é preciso notar que ao final de um evento em que uma tarefa, por exemplo, iniciou no estado $GAP1$, ela já estará no estado $GAP2$ e será neste estado que ela executará $Prazo_i{}^+(\ell_i, j)$. Assim, quando as tarefas estiverem nos estados $GAP1$, $GAP2$, $GAP3$ ou $GAP4$, a função retorna 1, o que ocasiona um sincronismo na execução. Sempre que as tarefas entram para estado trabalho,

todas elas conhecem o valor de $maxalt$. No primeiro evento, em cada vez que as tarefas entram neste estado, elas operam com as alturas no intervalo $[maxalt, maxalt/2]$ e atualizam $maxalt$ com o valor de $\lfloor maxalt/2 \rfloor$. Isso é repetido nos eventos seguintes até que $maxalt$ chegue em 0, quando este ciclo de eventos no estado $TRABALHO$ é encerrado. Durante este ciclo, portanto, ${Prazo_i}^+(\ell_i, j)$ retorna, em cada evento, $(log_2 maxalt) + 1$, que é a quantidade de relógios que faltam para o seu encerramento. Isso significa que qualquer mensagem enviada dentro do ciclo de estado $TRABALHO$ pode ser recebida por qualquer evento dentro desse ciclo, não depois.

4.3.4. Outros algoritmos distribuídos

Uma versão síncrona e uma assíncrona do pré-fluxo podem ser encontradas em [1]. A versão assíncrona é semelhante à nossa versão, com a diferença de não possuir heurísticas. A versão síncrona, no entanto, possui uma abordagem diferente, que é a seguinte. As tarefas realizam re-rotulamentos e envios de fluxo nos seus vértices em eventos alternados. Assim, nos relógios lógicos ímpares as tarefas realizam todos os re-rotulmentos necessários nos vértices ativos e nos pares, todos os envios de fluxo possíveis. Dessa forma, o efeito de fluxo negado do algoritmo assíncrono também é evitado. Para fins de comparação, realizamos os testes também com esta versão, acrescentando somente a heurística *gap relabeling* da forma como em S_PreFluxo.

5

Experimentos Computacionais

Neste capítulo, nos dedicaremos a mostrar uma comparação do desempenho das diversas versões do algoritmo de pré-fluxo mostradas no capítulo anterior. Certamente, nossa implementação do algoritmo do pré-fluxo poderia sofrer melhoras consideráveis através da adição ou troca de heurísticas, bem como através da mudança de estruturas de dados utilizadas. Isto é percebido quando comparamos os tempos obtidos pelo nosso programa seqüencial com os obtidos pela versão seqüencial desenvolvida por Andrew V. Goldberg [11] e disponibilizada pelo DIMACS - *Center for Discrete Mathematics and Theoretical Computer Science of Rutgers and Princeton University.* No entanto, não é um dos nossos objetivos implementar a melhor versão seqüencial para o problema do pré-fluxo, mas verificar o desempenho da versão parcialmente síncrona em comparação com as versões síncrona e assíncrona. Também não era nosso objetivo, a princípio, desenvolver uma versão parcialmente síncrona com melhor desempenho do que o da seqüencial, visto que as versões distribuídas implementadas nem sequer se aproximaram disso. Porém, o tempo de execução do algoritmo seqüencial sempre foi nossa meta no processo de desenvolvimento e aprimoramento da versão parcialmente síncrona, de maneira que, para muitas instâncias do problema, alcançamos esta meta.

5.1. Equipamento utilizado

Os testes iniciais foram realizados em um *cluster* Itautec com 16 nós de processamento dual-processados pertencente ao NACAD - Núcleo de Atendimento em Computação de Alto Desempenho - situado na COPPE, UFRJ.

No entanto, os tempos de execução mostrados neste capítulo mais adiante, foram obtidos em testes realizados em um *cluster* do Departamento de Computação da UFC. É um *cluster* Itautec que possui 28 nós de processamento dual-processados *Xeon* 1.4GHz, cada um com 1GB de memória RAM e um disco rígido de 18GB, interconectados por uma rede gigabit ethernet.

5.2. Instâncias

As instâncias utilizadas foram geradas a partir de um programa obtido no DIMACS, o gerador WLM (Washington-Line-Moderate), desenvolvido para o DIMACS *challenge workshop* acontecido em 1991 na *Rutgers University* (dimacs.rutgers.edu/Challenges/).

Utilizamos em nossos testes a classe *Random Level Graph*. Na representação gráfica de grafos dessa classe, os vértices estão dispostos em níveis a partir da fonte de forma que cada nível tem a mesma quantidade de vértices, com exceção do primeiro e do último nível, que possuem somente a fonte e o sumidouro, respectivamente. Os vértices possuem arestas direcionadas sempre para vértices de níveis seguintes e as arestas têm capacidade arbitrárias dentro de um limite. Dizemos que cada nível é uma coluna e os vértices que estão na mesma posição em cada nível formam uma linha. A razão entre as quantidades de linhas e colunas bem como o limite de capacidade das arestas podem variar. Um estudo comparativo entre algoritmos seqüenciais de pré-fluxo é apresentado em [12] e neste são utilizados grafos de uma classe similar à *Random Level Graph*. Os autores afirmam que grafos desta classe são representativos para testes das implementações de pré-fluxo e que grafos com mesmo tamanho e diferentes razões entre linhas e colunas apresentam

tempos de execução similares, a menos que a quantidade de linhas seja muito maior que a quantidade de colunas, ou o inverso. Baseado nisso, nossos testes foram realizados com grafos que possuiam a razão $1:1$ entre linhas e colunas, os quais classificamos como *Classe* $1:1$, e com grafos que tinham razão $2:1$, classificados como *Classe* $2:1$.

Para cada uma das classes, geramos grafos com os tamanhos de 20.000, 50.000 e 100.000 vértices.

5.3. Resultados

No decorrer dos testes com os nossos programas, tivemos a percepção, principalmente para o seqüencial, de que a quantidade de operações básicas de pré-fluxo realizadas para uma instância era mais determinante para o seu tempo de execução do que a quantidade de vértices desta instância. Além disso, a quantidade de operações básicas realizadas com instâncias de mesmo tamanho variava bastante quando se variava, por exemplo, a limite de capacidade das arestas nas instâncias. Com isso, procuramos realizar testes com instâncias cujas quantidades de operações realizadas fossem condizentes com seus tamanhos. O que queremos dizer, por exemplo, é que procuramos utilizar grafos de 100.000 vértices cuja quantidade de operações foi significantemente maior que a de instâncias de 20.000.

Os tempos apresentados nas tabelas estão em segundos. As primeiras tabelas mostram os tempos das execuções com todas as implementações. Os tempos mostradas são a média dos tempos obtidos com 4 instâncias para cada tipo de grafo. A primeira tabela mostra os resultados dos algoritmos seqüenciais por instância. Na segunda e quarta coluna, o tempo do seqüencial do Goldberg e na terceira e quinta, o tempo do nosso seqüencial. As tabelas seguintes mostram os tempos obtidos pelos programas distribuídos. Cada tabela se refere a uma instância cujo tamanho é mostrado no topo da tabela. As versões distribuídas são, na ordem: assíncrona, síncrona 1 (versão mostrada no Capítulo 4), síncrona 2 (versão de [1]) e parcialmente síncrona. As quatro linhas das tabelas mostram os resultados obtidos, respectivamente, com 2, 4, 8 e 16 tarefas. O símbolo ∞ indica um tempo de execução

muito acima dos tempos obtidos para aquela instância. Faremos um comentário a respeito disso adiante.

Tempos das execuções seqüenciais				
	Classe 2 : 1		*Classe* 1 : 1	
Número de vértices	Goldberg	Seqüencial	Goldberg	Seqüencial
20.000	2,3	3,1	2,1	3,0
50.000	7,7	12,9	4,5	14,8
100.000	21,0	33,1	15,6	50,2

Tabela 5.1. *Tempos de execução das versões seqüenciais*

Instâncias de 20.000 vértices								
	Classe 2 : 1				*Classe* 1 : 1			
Número de tarefas	Assinc	Sinc 1	Sinc 2	PS	Assinc	Sinc 1	Sinc 2	PS
2	12,3	32,6	16,3	4,3	11,4	60,1	22,5	5,2
4	$+\infty$	48,2	14,2	3,2	$+\infty$	96,6	19,8	4,0
8	$+\infty$	61,2	14,0	2,9	$+\infty$	127,5	20,6	3,9
16	$+\infty$	129,1	17,9	5,9	$+\infty$	283,9	33,0	9,8

Tabela 5.2. *Tempos de execução das versões distribuídas com instâncias de 20.000 vértices.*

As tabelas seguintes procuram mostrar resultados mais específicos para execuções com duas instâncias: uma de 20.000 vértices e outra de 50.000 vértices, ambas da classe 2:1. Os resultados mostrados são representativos para as classes de grafos utilizadas aqui, de forma que não acrescentamos resultados para mais instâncias. A Tabela 5.5 mostra a quantidade de operações básicas de pré-fluxo realizadas pelos programas seqüenciais. As Tabelas 5.6 e 5.7 mostram esses resultados para as execuções distribuídas - soma das operações realizadas por todas as tarefas durante a execução - em percentuais além da nossa versão seqüencial. Por exemplo, a Tabela 5.6 mostra que a execução parcialmente síncrona do grafo de 20.000 vértices realizou 9% a mais de operações que a nossa versão seqüencial com o mesmo grafo.

Instâncias de 50.000 vértices								
	Classe 2 : 1				*Classe* 1 : 1			
Número de tarefas	Assinc	Sinc 1	Sinc 2	PS	Assinc	Sinc 1	Sinc 2	PS
2	56,3	111,2	78,5	18,2	65,0	222,2	109,6	22,8
4	$+\infty$	149,7	60,4	11,9	$+\infty$	336,7	91,5	17,0
8	$+\infty$	198,2	56,7	9,1	$+\infty$	468,8	85,3	12,6
16	$+\infty$	389,7	63,4	11,8	$+\infty$	881,7	102,1	16,3

Tabela 5.3. *Tempos de execução das versões distribuídas com o instâncias de 50.000 vértices.*

Instâncias de 100.000 vértices								
	Classe 2 : 1				*Classe* 1 : 1			
Número de tarefas	Assinc	Sinc 1	Sinc 2	PS	Assinc	Sinc 1	Sinc 2	PS
2	137,2	285,3	225,8	45,5	312,0	921,6	431,1	74,8
4	$+\infty$	393,0	169,6	31,2	$+\infty$	1.398,6	370,0	51,0
8	$+\infty$	511,0	147,9	20,6	$+\infty$	1.978,6	385,1	43,1
16	$+\infty$	993,3	168,3	22,7	$+\infty$	$+\infty$	467,9	58,4

Tabela 5.4. *Tempos de execução das versões distribuídas com instâncias de 100.000 vértices.*

Operações de pré-fluxo		
	Goldberg	Seqüencial
20.000	4.460.039	1.490.382
50.000	15.023.986	5.993.395

Tabela 5.5. *Quantidade de operações realizadas pelas versões seqüencias com instâncias de 20.000 e 50.000 da classe* 2 : 1.

Operações de pré-fluxo - 20.000				
	Assinc	Sinc 1	Sinc 2	PS
2 tarefas	+10%	+2%	+72%	+9%
4 tarefas	$+\infty$	+3%	+74%	+11%
8 tarefas	$+\infty$	+4%	+76%	+17%
16 tarefas	$+\infty$	+4%	+79%	+19%

Tabela 5.6. *Quantidades de operações realizadas pelas versões distribuídas com uma instância de 20.000 vértices da classe* $2:1$*, em termos de percentual a mais do que a quantidade de operações da nossa versão seqüencial.*

Operações de pré-fluxo - 50.000				
	Assinc	Sinc 1	Sinc 2	PS
2 tarefas	+11%	+3%	+84%	+11%
4 tarefas	$+\infty$	+5%	+85%	+12%
8 tarefas	$+\infty$	+6%	+86%	+14%
16 tarefas	$+\infty$	+7%	+86%	+16%

Tabela 5.7. *Quantidades de operações realizadas pelas versões distribuídas com uma instância de 50.000 vértices da classe* $2:1$*, em termos de percentual a mais do que a quantidade de operações da nossa versão seqüencial.*

As tabelas seguintes detalham os tempos das execuções distribuídas, mostrando-os na forma T_p, T_s, T_m, onde:

- T_p é o tempo da realização da função de pré-fluxo, sem considerar a leitura e construção da saída. É, basicamente, o tempo da realização das operações básicas.
- T_s, o *tempo de sobrecarga*, é o tempo da construção e leitura da saída de dados dentro da função de pré-fluxo. É um trabalho que não é inerente à ferramenta, mas é inerente às versões distribuídas da aplicação de pré-fluxo, de forma que não ocorre nas versões seqüenciais.
- T_m, o *tempo de comunicação*, é o tempo de comunicação e computação realizada pela ferramenta.

Vamos, algumas vezes, nos referir a $T_p + T_s$ como *tempo de computação*. Assim, $T_p + T_s + T_m$ deverá ser o tempo total da execução.

Distribuição do tempo - 20.000												
Número de tarefas	Assinc			Sinc 1			Sinc 2			PS		
	T_p	T_s	T_m	T_p	T_s	T_m	T_p	T_s	T_m	T_p	T_s	T_m
2	6,6	4,1	0,4	3,2	2,1	26,3	6,8	4,0	6,5	1,8	1,3	1,5
4	$+\infty$			2,1	1,9	43,4	5,4	2,9	5,9	1,0	0,8	1,3
8	$+\infty$			1,6	1,3	53,9	3,0	1,5	9,5	0,6	0,5	1,8
16	$+\infty$			1,2	0,8	134,6	1,8	0,9	15,9	0,4	0,3	5,2

Tabela 5.8. *Tempos gastos com realização de operações de pré-fluxo, leitura e construção das mensagens da aplicação e comunicação, respectivamente, das versões distribuídas com uma instância de 20.000 vértices da classe* $2:1$.

As tabelas seguintes informam as quantidades de eventos, em média, realizadas por cada tarefa em cada uma das execuções.

Distribuição do tempo - 50.000												
Número de tarefas	Assinc			Sinc 1			Sinc 2			PS		
	T_p	T_s	T_m	T_p	T_s	T_m	T_p	T_s	T_m	T_p	T_s	T_m
2	32,1	20,6	1,7	16,1	9,3	84,7	33,1	22,4	23,0	8,1	5,4	5,3
4	$+\infty$			10,7	5,9	137,1	23,1	13,1	24,2	4,8	3,1	4,7
8	$+\infty$			6,3	3,8	190,0	15,9	10,8	30,0	2,6	2,0	4,4
16	$+\infty$			4,1	2,2	382,8	9,9	7,9	45,6	1,1	1,4	9,3

Tabela 5.9. *Tempos gastos com realização de operações de pré-fluxo, leitura e construção das mensagens da aplicação e comunicação, respectivamente, das versões distribuídas com uma instância de 50.000 vértices da classe* $2:1$.

Quantidade de eventos - 20.000				
	Assinc	Sinc 1	Sinc 2	PS
2 tarefas	2.720	161.286	14.286	2.434
4 tarefas	$+\infty$	159.157	16.334	4.105
8 tarefas	$+\infty$	161.090	17.422	4.910
16 tarefas	$+\infty$	163.226	19.650	6.738

Tabela 5.10. *Quantidade de eventos realizados pelas versões distribuídas com uma instância de 20.000 vértices da classe* $2:1$.

Quantidade de eventos - 50.000				
	Assinc	Sinc 1	Sinc 2	PS
2 tarefas	5.275	513.008	28.745	5.034
4 tarefas	$+\infty$	511.191	32.458	8.625
8 tarefas	$+\infty$	508.134	34.994	10.394
16 tarefas	$+\infty$	514.547	35.715	14.136

Tabela 5.11. *Quantidade de eventos realizados pelas versões distribuídas com uma instância de 50.000 vértices da classe* $2:1$.

5.4. Análise dos resultados

5.4.1. Versões seqüenciais

A versão seqüencial do Goldberg nos dá uma noção dos tempos de execuções seqüenciais que podem ser obtidos quando estruturas de dados e heurísticas eficientes são implementadas. Os tempos obtidos são sempre melhores do que os obtidos pela nossa versão, de acordo com a Tabela 5.1. Veja na Tabela 5.5 que a quantidade de operações realizadas pela versão do Goldberg é sempre maior do que a quantidade de operações realizadas pela nossa versão. Isso reforça a percepção de que podemos melhorar ainda mais as estruturas de dados e suas implementações utilizadas em nossa versão.

5.4.2. Versão assíncrona

A versão assíncrona em ambas as classes apresentou um desempenho melhor do que as síncronas com duas tarefas, mas bastante pior com mais tarefas. Com duas tarefas, esta versão conseguiu algum sucesso na aplicação da heurística *gap relabeling*. Com mais tarefas, não conseguimos determinar o tempo de execução para nenhuma instância, o qual se tornou maior que 3600s. O símbolo $+\infty$ indica isto.

O mau desempenho da versão assíncrona se dá por duas razões. A primeira, é a quantidade de fluxo negado. Este efeito colateral, sobre o qual falamos no capítulo anterior, induz muitas operações de envio de fluxo que posteriormente são desfeitas, de forma que muito trabalho inútil é realizado. A segunda razão decorre do efeito de operações em ciclo, sobre o qual também falamos no capítulo anterior. Mesmo nas versões seqüenciais, este efeito é capaz de tornar inviável a execução do programa. Sem a heurística *gap relabeling*, a qual, quando bem aplicada, ameniza sobremaneira esse efeito, seria inviável realizar testes com as instâncias que utilizamos e com qualquer versão. As execuções assíncronas com mais de duas tarefas não conseguiram aplicar com sucesso a heurística. Percebemos, localmente nas tarefas, a formação de diversos conjuntos de vértices que induzem o efeito de operações em ciclo no decorrer das execuções. Além disso, quando a tarefa iniciadora

dava início à aplicação da heurística procurando uma altura sem vértices, não obtinha sucesso em achar tal altura.

Perceba nas Tabelas 5.8 e 5.9 que nas versões assíncronas com duas tarefas, o tempo com comunicação é irrelevante em relação ao tempo de computação. Isso ocorre, porque não há nenhum custo de sincronização. As quantidades de eventos realizados (Tabelas 5.10 e 5.11) também foram baixas, se comparadas com as versões síncronas, pelo fato de que, em cada evento, o máximo possível de operações são realizadas. O tempo de sobrecarga, no entanto, foi relevante, assim como em com todas as versões distribuídas. Falaremos mais sobre isso adiante.

5.4.3. Versões síncronas

Na nossa versão síncrona (Sinc 1), em cada evento quando as tarefas aplicam as operações, todas elas operam sobre vértices de um mesma altura. Dessa forma, é evitado o efeito de fluxo negado. Em eventos sucessivos, as tarefas trabalham com alturas menores até alcançarem a altura do sumidouro, quando iniciam a aplicação da *gap relabeling*. Dessa maneira, a heurística também é aplicada com sucesso. No entanto, esta versão apresentou desempenho que piorava com o aumento da quantidade de tarefas. A razão para isso decorre do pouco trabalho que se realiza em cada evento, já que somente vértices de uma determinada altura são operados em cada evento que se trabalha com os vértices. Perceba que a quantidade de relógios realizados é bastante grande quando comparada com a versão parcialmente síncrona (Tabelas 5.10 e 5.11). Esta quantidade de relógios não aumenta significativamente com a quantidade de tarefas. O tempo aumenta, no entanto, pois, cada relógio de uma execução síncrona dura o tempo do evento de maior duração dentre os eventos realizados pelas tarefas naquele relógio. Isto é particularmente grave para uma execução quando se tem uma quantidade excessiva de relógios. Por diversas vezes, percebemos que poucas tarefas - às vezes somente uma - trabalhavam em um mesmo relógio lógico. O custo da sincronização se tornou, portanto, cada vez maior.

A versão síncrona de [1] (Sinc 2) também evita o efeito de fluxo negado, pois alterna

as operações de envio de fluxo e re-rotulamento no decorrer dos pulsos. Além disso, a heurística *gap relabeling* também conseguiu ser aplicada com sucesso. O número de operações básicas realizadas com os vértices, no entanto, (Tabelas 5.6 e 5.7) mostrou-se consideravelmente maior que nas demais versões. Assim, apesar de esta versão apresentar um desempenho que melhorava com o aumento do número de tarefas, até, em geral, o limite de 16, o número de eventos realizados ainda foi elevado (Tabelas 5.10 e 5.11). O número de relógios lógicos, como comentamos há pouco, é diretamente proporcional ao custo da sincronização.

O custo da sincronização em ambas as versões síncronas pode ser percebido nos tempos de comunicação mostrados nas Tabelas 5.8 e 5.9. Estes tornam-se bastante altos em relação aos tempos de computação com o aumento do número de tarefas. Constata-se que a execução síncrona é parecidade com a seqüencial, mas o custo de sincronização é alto.

5.4.4. Versão parcialmente síncrona

A versão parcialmente síncrona tem como objetivo conciliar os pontos positivos e negativos das versões síncrona (nossa versão) e assíncrona, tornando, obviamente, mais evidentes os pontos positivos. No algoritmo assíncrono, cada tarefa, nos eventos em que opera com os vértices, realiza muitas operações - ou melhor, o máximo de operações possíveis que ela pode realizar naquele evento - e isto é um fator positivo. No entanto, vimos que esta abordagem compromete a aplicação da *gap relabeling* e gera muito fluxo negado. Nas versões síncronas, os efeitos colaterais são evitados e isto é um ponto positivo. No entanto, o número de relógios lógicos, em especial na nossa versão, é elevado. Procuramos na versão parcialmente síncrona, realizar uma quantidade de trabalho considerável por evento. Em média, não tanto trabalho quanto se faz na versão assíncrona, pois isto poderia acarretar muito fluxo negado, mas, certamente, mais trabalho do que se realiza nas versões síncronas. Além disso, a versão parcialmete síncrona deve ser capaz de aplicar, com o sucesso das versões síncronas, a *gap relabeling*, pois isto é fundamental.

É para alcançar este objetivo que a execução da versão parcialmente síncrona possui trechos síncronos e não síncronos de forma alternada. Nos trechos síncronos, a heurística *gap relabeling* é aplicada da forma como nos algoritmos síncronos. Nos trechos não síncronos, adotamos uma idéia semelhante a da nossa versão síncrona, de forma que, em cada evento, cada tarefa trabalha com vértices dentro de uma faixa de alturas. Nesses trechos, quando os vértices alcançam uma altura limite eles são ignorados até o final do trecho e isso é fundamental para o sucesso da *gap relabeling*. Esses trechos têm também a característica de as tarefas serem independentes umas das outras. Dessa forma, é minimizado o efeito de tarefas ociosas esperarem por outras não ociosas como ocorre nos relógios lógicos das execuções síncronas. Algum fluxo negado ocorre, mas ele não compromete o desempenho do algoritmo. Nas Tabelas 5.6 e 5.7 vemos as quantidades de operações realizadas, as quais são sempre próximas das quantidades de operações da versão seqüencial.

As operações, além de terem quantidades próximas às da nossa versão seqüencial, conseguem ser bem divididas entre as tarefas. Os gráficos das Figuras 5.1, 5.2, 5.3 e 5.4 mostram essa distribuição para as execuções com 4 e 8 tarefas das instâncias de 20.000 e 50.000 vertices da classe $2:1$, da seguinte forma. Sejam, para uma determinada execução Ξ, $min_{\Xi}(\ell)$, a quantidade mínima de operações realizadas dentre todas as tarefas no relógio lógico ℓ, $max_{\Xi}(\ell)$, a quantidade máxima de operações realizadas dentre todas as tarefas no relógio lógico ℓ, e $tot_{\Xi}(\ell)$, a quantidade total de operações realizadas em ℓ. Assim, sendo f a função representada nos gráficos, temos que $f(\ell) = \frac{max_{\Xi}(\ell) - min_{\Xi}(\ell)}{tot_{\Xi}(\ell)}$. Os gráficos mostram apenas o comportamento das execuções nos primeiros 1.000 relógios lógicos, mas é um trecho representativo para o decorrer das execuções. Perceba que em todos os gráficos, a maior parte dos valores de $f(\ell)$ são próximos de 0, indicando uma boa distribuição de trabalho entre as tarefas.

Como a versão parcialmente sícrona realiza uma quantidade de operações próxima da nossa versão seqüencial e essas operações possuem uma boa distribuição entre as tarefas, era de ser esperar que o tempo de computação fosse, proporcionalmente à quantidade de tarefas, menor que o tempo da execução seqüencial. De fato, isto ocorre se considerarmos

somente o tempo da realização de operações de pré-fluxo mostrado nas Tabelas 5.8 e 5.9. No entanto, a complexidade da realização das operações de pré-fluxo é compatível com a complexidade de construir as mensagens com as operações realizadas. Perceba que, ao extrair um vértice do topo da *heap* para a realização de uma operação de re-rotulamento, por exemplo, o algoritmo de pré-fluxo percorre a lista com o conjunto de vizinhos do vértice no intuito de achar o mais adeqüado para a realização da operação. Para construir compor a mensagem de saída com esta operação, o algoritmo percorre uma lista com as tarefas que precisam conhecer as informações sobre esse vértice e preenche uma saída para cada tarefa com o novo rótulo do vértice. Essas listas - a lista de vizinhos do vértice e a lista de tarefas interessadas pelo vértice - podem se tornar próximas de acordo com o grau do vértice e a quantidade de tarefas. O tempo de sobrecarga, portanto, é bastante expressivo e isso torna o tempo de computação alto em relação ao tempo seqüencial.

A quantidade de relógios realizados nas execuções parcialmente síncronas (Tabelas 5.10 e 5.11) foi significativamente menor que nas execuções da nossa versão síncrona. Isto é resultado da maior carga de trabalho atribuído em cada evento a cada tarefa do que o que acontece nas versões síncronas. Os trechos de execução não síncrona induziram a um aumento do número de relógios realizados quando se aumentou o número de tarefas envolvidas na execução.

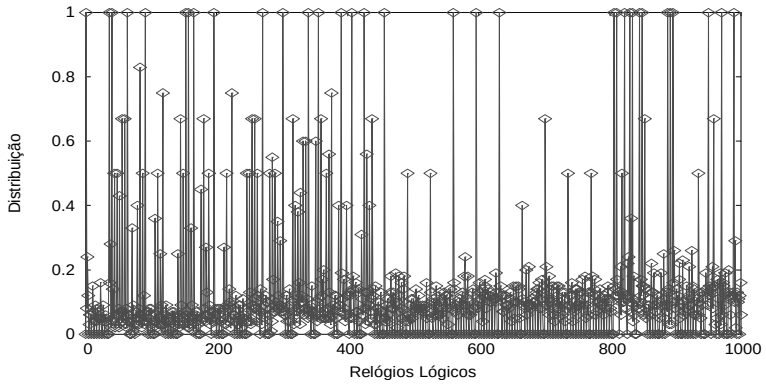


Figura 5.1. *Gráfico da distribuição de operações por tarefa por relógio lógico da execução parcialmente síncrona do grafo de 20.000 vértices com 4 tafefas.*

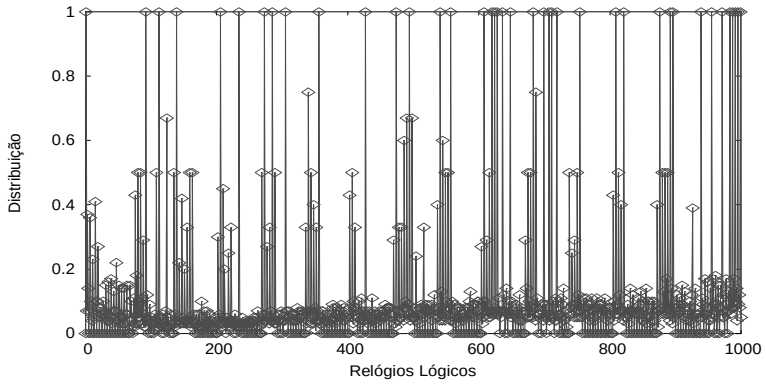


Figura 5.2. *Gráfico da distribuição de operações por tarefa por relógio lógico da execução parcialmente síncrona do grafo de 20.000 vértices com 8 tafefas.*

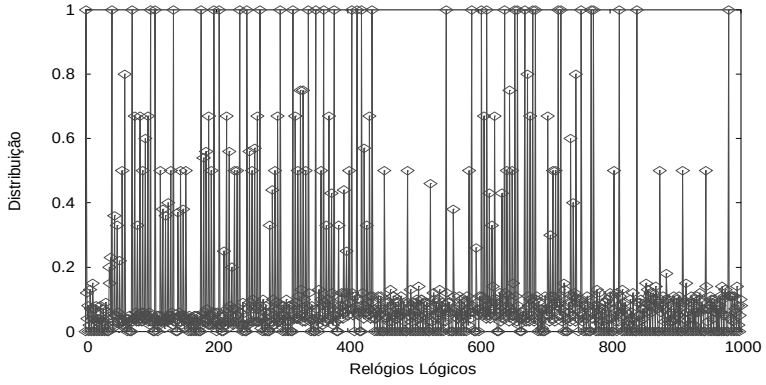


Figura 5.3. *Gráfico da distribuição de operações por tarefa por relógio lógico da execução parcialmente síncrona do grafo de 50.000 vértices com 4 tafefas.*

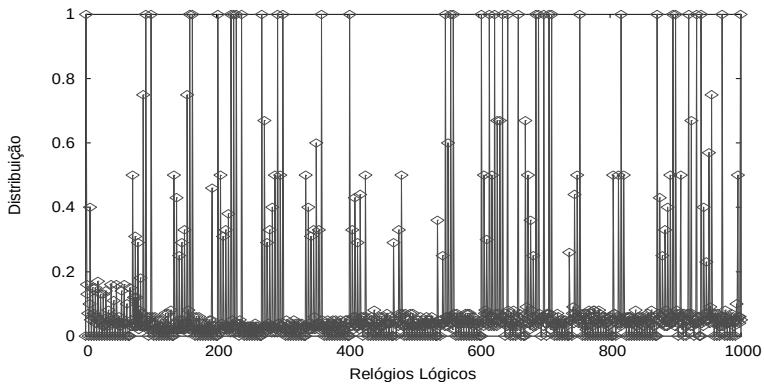


Figura 5.4. *Gráfico da distribuição de operações por tarefa por relógio lógico da execução parcialmente síncrona do grafo de 50.000 vértices com 8 tafefas.*

6

Conclusões

Execuções assíncronas possuem um grau de indeterminismo inerente às ordens em que as mensagens enviadas chegam aos seus destinos, já que sistemas de memória distribuída não podem dar garantias sobre tais ordens. Uma execução assíncrona pode convergir mais rapidamente ou não para o fim, de acordo com a ordem de chegada das suas mensagens. Um mecanismo de controle sobre essa ordem em tal execução pode controlar o seu indeterminismo e, principalmente, induzi-la a alcançar mais rapidamente a solução do seu problema. Execuções síncronas, por exemplo, precisam controlar a ordem das suas mensagens.

Nossos principais objetivos nesse trabalho foram construir um mecanismo dessa natureza, a ferramenta de atribuição de relógios lógicos, e avaliar seu funcionamento utilizando-o ao induzir a execução de uma aplicação distribuída. Controlar as mensagens de uma execução distribuída, no entanto, acarreta custos. Um desses é o custo do algoritmo da ferramenta quando ela manipula as mensagens geradas pela aplicação. Outro custo decorre da dependência que se cria entre as tarefas de uma aplicação distribuída, pois tarefas podem se tornar ociosas durante a execução ao esperarem por mensagens. Conciliar tais custos e a vantagem de se poder controlar uma execução distribuída é papel do usuário da ferramenta.

A aplicação escolhida para avaliar o desempenho da ferramenta foi o problema do fluxo máximo. Este problema possui soluções seqüenciais eficientes baseadas na abordagem de pré-fluxo. Ele é, no entanto, um problema de difícil paralelização, de forma que não são conhecidas soluções distribuídas eficientes. O custo de leitura e construção das mensagens referentes às operações básicas de pré-fluxo, por exemplo, é, em geral, quase o mesmo custo de se realizar as operações básicas. De maneira geral para todas as versões testadas e com a classe de grafos que utilizamos, percebemos que os tempos de execução estavam mais intimamente ligados à quantidade de operações de pré-fluxo realizadas do que propriamente com o tamanho da instância.

Apesar do fraco desempenho, os algoritmos síncrono e assíncrono que implementamos utilizando a abordagem de pré-fluxo têm as suas vantagens. O algoritmo assíncrono é capaz de realizar o máximo de operações de pré-fluxo possível antes de enviar suas mensagens e encerrar um evento. Além disso, suas tarefas possuem independência durante a execução, isto é, não precisam esperar por várias mensagens de diversas outras tarefas antes de iniciarem um evento. Esse comportamento acarreta efeitos colaterais, como a necessidade de se desfazer trabalho e a dificuldade na aplicação da heurística de *gap relabeling*, essencial para o bom desempenho do algoritmo. O algoritmo síncrono aplica de forma eficiente a heurística de *gap relabeling* e realiza uma quantidade próxima de operações da versão seqüencial. No entanto, realiza poucas operações por evento, o que causa uma excessiva quantidade de eventos na sua execução.

Dessa forma, desenvolvemos uma solução distribuída que conseguiu, razoavelmente, adaptar as vantagens dos algoritmos síncrono e assíncrono. Nossa solução manteve uma certa independência entre as tarefas nos trechos de execução em que se aplicam as operações de pré-fluxo dando uma carga maior de trabalho a esses eventos do que no algoritmo síncrono, mas manteve um sincronismo nos trechos da aplicação da heurística. A ferramenta apresentada aqui torna possível esse tipo de execução parcialmente síncrona.

De fato, a execução parcialmente síncrona obteve um desempenho melhor do que as outras soluções distribuídas e bastante próximo da solução seqüencial correspondente.

Conseguimos através dessa abordagem, diminuir consideravelmente a quantidade de eventos, aumentando a carga de trabalho realizado por evento em relação à versão síncrona. De forma geral, o melhor desempenho da versão parcialmente síncrona em relação às demais versões distribuídas se deveu à possibilidade de se intercalar trechos síncronos e não síncronos. Os trechos síncronos, os quais possuem um custo de comunicação considerável, foram aplicados somente quando necessários, isto é, durante a realização da heurística.

De maneira geral, a ferramenta de atribuição de relógios lógicos pode ser útil para aplicações distribuídas que encontrem benefícios em um controle mais rígido do assincronismo das suas execuções ou que possam se beneficiar da utilização de *pontos de sincronização*. Esses pontos são relógios lógicos da execução quando todas as tarefas devem realizar um evento antes de continuarem. Na nossa implementação parcialmente síncrona do pré-fluxo, por exemplo, utilizamos tais pontos para a aplicação da heurística de *gap relabeling*.

Um último fato relevante a ser comentado é a possibilidade de se melhorar os códigos desenvolvidos neste trabalho. É possível que estruturas de dados mais eficientes sejam utilizadas nos algoritmos de pré-fluxo. Além disso, estruturas de dados e métodos mais adequados podem ser utilizados no algoritmo da ferramenta no intuito de diminuir sua participação no tempo de uma execução distribuída.

A

Códigos Fonte da Ferramenta de Atribuição de Relógios Lógicos e das Versões de Pré-Fluxo Implementadas

---

A seguir, os códigos fonte dos programas desenvolvidos neste trabalho, com exceção da versão seqüencial do Goldberg e a versão síncrona de [1]. Os programas foram desenvolvidos com a linguagem C e a biblioteca MPI - *Message Passing Insterface* - para a comunição entre as tarefas.

O primeiro código mostrado é o da ferramenta. São apresentadas a função principal, chamada LOTA, funções e variáveis relacionadas. A utilização da ferramenta se dá pela chamada da função LOTA. Os parâmetros de LOTA são, pela ordem:

1. Um *buffer* com o grafo de tarefas da seguinte forma.

   $buffer = QMem + id_1 + nviz_1 + viz_{11} + viz_{12} + \ldots + viz_{1nviz_1} +$
   $id_2 + nviz_2 + viz_{21} + viz_{22} + \ldots + viz_{2nviz_1} + \ldots$, onde :

   - $QMem$ é o espaço em bytes ocupado por *buffer*.

     Para cada $i \geq 1$,

   - id_i é a identificação da i-ésima tarefa listada.

- $nviz_1$ é a quantidade de vizinhos da tarefa id_i.
- viz_{ij} é a j-ésima tarefa vizinha da tarefa id_1, $1 \leq j \leq nviz_i$.

2. Um *buffer* correspondendo à instância da aplicação.
3. A função de saída da aplicação. Essa função será chamada pela ferramenta ao fim da execução.
4. As restrições de sincronização correspondentes, respectivamente, a T^+, T^-, S^- e S^+, explicadas no Capítulo 3.
5. A variável correspondente a une_msgs, explicada no Capítulo 3.

Os códigos mostrados em seguida são, respectivamente, o da versão parcialmente síncrona, síncrona e assíncrona. Para cada um deles, mostramos apenas a função de evento - a qual realiza as operações de fluxo e procede a leitura e construção das mensagens - com algumas funções relaicionadas. As funções principais, as quais apenas lêem a instância do problema, inicialiam a rede de comunicação e chamam a função da ferramenta - nos casos síncrono e parcialmente síncrono - foram omitidas.

Por fim, apresentamos o código fonte da nossa versão seqüencial.

Referências Bibliográficas

[1] Barbosa, Valmir C., "An Introduction to Distributed Algorithms", The MIT Press, 1996.

[2] Ruggiero, Márcia A. Gomes e Lopes, Vera Lúcia da R., "Cálculo Numérico. Aspectos Teóricos e Computacionais." 2a edição, Makron, 1996.

[3] Lamport, Leslie, "Time, Clocks and the Ordering of Events in a Distributed System", Communications of the ACM, Vol 21, Number 7:558-565, 1978.

[4] C. Fidge, "Logical Time in Distributed Systems", Computer 24, 28-33, 1991.

[5] B. Charron-Bost, "Concerning the Size of Logical Clocks in Distributed Systems", Information Processing Letters 39, 11-16, 1991.

[6] Goldberg, Andrew V., "A New Max-Flow Algorithm", PhD thesis, Massachusetts Institute of Technology, 1985.

[7] Goldberg, A. V. and Tarjan, R. E. "A new approach to the maximum flow problem", Proceedings of the 18th Annual ACM symposium on Theory of Comuting, pages 136-146, 1986.

[8] Goldberg, A. V. and Tarjan, R. E., "A new approach to the maximum flow problem", Journal of the ACM, 35:921-940, 1988.

[9] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. and Stein, Clifford, "Introduction to Algorithms", Columbia University McGraw-Hill, 2001.

[10] Dijkstra, E. W. and Scholten, C. S., "Termination detection for diffusing computations", Inf. Process. Lett. 11, 1-4, 1980.

[11] Ahuja, Ravindra K., Kodialam, Murali, Mishra Ajay K., and Orlin, James B., "Computational Investigations of Maximum Flow Algorithms", European Journal on Operational Research 97, 509-542, 1997.

[12] Ahuja, Ravindra K. and Orlin, James B., "Distance-directed augmenting path algorithms for maximum and parametric maximum problems", Naval Research Logistics, 38:413-430, 1991.

[13] Anderson, Charles L., "Implementations of the Pseudoflow Algorithm for the Maximum Flow Problem", PhD thesis, University of California, 2001.