Universidade Federal do Ceará
Centro de Ciências
Mestrado e Doutorado em Ciência da Computação

Uma *Interface* de Programação Distribuída para Aplicações em Otimização Combinatória

Allberson Bruno de Oliveira Dantas

Fortaleza – Ceará
Setembro 2011

Universidade Federal do Ceará
Centro de Ciências
Mestrado e Doutorado em Ciência da Computação

Uma *Interface* de Programação Distribuída para Aplicações em Otimização Combinatória

Autor
Allberson Bruno de Oliveira Dantas

Orientador
Ricardo Cordeiro Corrêa

Dissertação apresentada à Coordenação do Programa de Pós-graduação em Ciência da Computação da Universidade Federal do Ceará como parte dos requisitos para obtenção do grau de ***Mestre em Ciência da Computação****.*

Fortaleza – Ceará
Setembro 2011

Resumo

Este trabalho foi motivado pela necessidade da exploração do potencial do paralelismo distribuído em aplicações em Otimização Combinatória. Para tanto, propomos uma *interface* de programação distribuída, na qual prezamos dois requisitos principais: eficiência e reuso.

O primeiro advém da necessidade de aplicações de CAD (Computação de Alto Desempenho) exigirem máximo desempenho possível. Assim sendo, especificamos esta *interface* como uma extensão da biblioteca MPI, a qual é assumida como eficiente para aplicações distribuídas. O requisito reuso deve tornar compatíveis duas características importantes: assincronismo e operações coletivas. O assincronismo deve estar presente na *interface*, uma vez que as aplicações em Otimização Combinatória, em sua maioria, possuem uma natureza assíncrona. Operações coletivas são funcionalidades que devem estar disponíveis na *interface*, de modo que possam ser utilizadas por aplicações em suas execuções.

Tendo em vista atender o requisito reuso, baseamos esta *interface* nos Modelos de Computação Distribuída Dirigidos por Eventos e por Pulsos, pois os mesmos são assíncronos e permitem a incorporação de operações coletivas.

Implementamos parcialmente a *inteface* definida neste trabalho. Tendo em vista validar uso dessa *inteface* por aplicações em Otimização Combinatória, selecionamos duas aplicações e as implementamos utilizando nossa *interface*. São elas a técnica *Branch-and-Bound* e o Problema do Conjunto Independente Máximo (CIM). Fornecemos também alguns resultados experimentais.

Abstract

This work was motivated by the need of exploiting the potential of distributed paralelism in combinatorial optimization applications. To achieve this goal, we propose a distributed programming interface, in which we cherish two main requirements: efficiency and reuse.

The first stems from the need of HPC (High Performance Computing) applications require maximum possible performance. Therefore, we specify our interface as an extension of the MPI library, which is assumed to be efficient for distributed applications. The "reuse" requirement must make compatible two important features: asynchronism and collective operations. Asynchronism must be present at our interface, once most of combinatorial optimization applications have an asynchronous nature. Collective operations are features that should be available in the interface, so that they can be used by applications in their execution.

In order reach the reuse requirement, we based this interface on the Event- and Pulse-driven Models of Distributed Computing, once they are asynchronous and allow the incorporation of collective operations.

We implemented partially the interface defined in this work. In order to validate the use of the inteface by combinatorial optimization applications, we selected two applications and implemented them using our interface. They are the Branch-and-Bound technique and the Maximum Stable Set Problem (MSSP). We also provide some experimental results.

Dedicatória

Dedico este trabalho a minha avó e mãe Maria Liduina. Obrigado pelo amor, apoio constante e confiança irrestrita em todos os momentos.

Agradecimentos

Gostaria de agradecer primeiramente a Deus, pelo dom da vida e por me proporcionar saúde e perseverança para alcançar todos os objetivos da minha vida.

Ao professor Ricardo Corrêa pela confiança em mim depositada e pela competência na orientação deste trabalho.

Ao professor Heron Carvalho pelos conselhos, incentivos e disposição para ajudar nas dificuldades deste trabalho.

À UFC, mais especificamente o Departamento de Computação, pela oportunidade da realização deste trabalho.

À minha avó Liduina que me criou como um filho e me ensinou tudo na vida e me fez ser tudo o que sou hoje.

À minha mãe Alba que me deu o dom da vida e é para mim exemplo de perseverança.

Ao meu pai Willdson (in memoriam) que foi em vida exemplo de simplicidade e dedicação.

Às minhas tias Fabiane, Kátia e Altamira que muito ajudaram na minha criação.

Ao Serpro, empresa que me acolheu tão bem e permitiu liberação de horas de trabalho para a realização deste mestrado.

Aos meus chefes Carlos Augusto, Elan Lima e Leonardo Fonseca, que tanto me apoiaram e incentivaram para a concretização deste sonho.

Sumário

Capítulo 1

Introdução

O termo Computação de Alto Desempenho, ou CAD, está associado ao nicho de aplicações que requerem um alto poder computacional, frequentemente alcançado através do uso intensivo de paralelismo. A CAD é usada largamente, por exemplo, em modelos físicos para previsão climática e de catástrofes naturais, na descoberta de fármacos através da modelagem molecular, na genômica e bioinformática e no setor de finanças através da previsão de movimentações de mercado [14]. Tais aplicações são oriundas, sobretudo, das Ciências Computacionais e Engenharias [20]. Nesse contexto, usa-se o termo CAD para designar a área que estuda modelos, técnicas e algoritmos que permitem extrair o máximo de desempenho computacional em cada aplicação. Além das aplicações citadas anteriormente, ressaltamos a Otimização Combinatória, a qual, através de problemas sobre grafos, é uma das motivações deste trabalho.

No vasto domínio de CAD, dedicamos nossa atenção ao *paralelismo distribuído*, no qual o poder computacional é obtido utilizando-se um conjunto de nós interconectados em uma rede por canais de comunicações bidirecionais. Nessa rede, cada nó possui sua própria memória, à qual ele tem acesso exclusivo em suas computações locais. As interações entre os nós ocorre através de trocas de mensagens. Todos os canais de comunicação são confiáveis, ou seja, se uma mensagem é enviada através do referido canal, sua chegada ao nó de destino é garantida, mesmo que com algum atraso finito. Os canais também não podem garantir ordem nas mensagens que trafegam neles. Cada nó é capaz de realizar computações sequenciais, as quais são guiadas por um relógio local. Entretanto, não existe um relógio global que dita o passo em que a computação distribuída se desenrola.

1.1 Motivações e Objetivo

Neste trabalho, buscamos explorar o potencial do paralelismo distribuído em aplicações assíncronas, particularmente em Otimização Combinatória. Para tanto, propomos uma *interface* de programação distribuída que preza dois requisitos principais, ambos em um contexto de computações assíncronas: eficiência e reuso. Ambos advêm da natureza de aplicações de CAD. Tendo em vista o princípio "eficiência" já estar presente no projeto da *Message Passing Interface (MPI)* [18], a *interface* aqui proposta é uma extensão da MPI.

O requisito de reuso merece uma discussão um pouco mais detalhada. Naturalmente, a ideia básica, comum às situações em que esse termo "reuso" é evocado, é a possibilidade do

uso da *interface* por um número grande de aplicações. Porém, essa ideia traz consigo algumas considerações específicas às características das aplicações em Otimização Combinatória. A primeira delas é o assincronismo, característica presente na maioria das estratégias de paralelização desse tipo de aplicações e que deve estar presente na *interface* [6, 7, 10]. A segunda diz respeito ao estabelecimento de operações coletivas assíncronas, funcionalidades previamente incorporadas à *interface*, que podem ser requeridas por aplicações em Otimização Combinatória no momento de suas execuções. A biblioteca de programação por troca de mensagens MPI já inclui operações coletivas. Entretanto, as mesmas possuem alguma forma de sincronismo, em razão da forte motivação de aplicações numéricas. Entre outros fatos, é comum em MPI que uma função associada a uma operação coletiva exija, para o seu perfeito funcionamento, a invocação espontânea e em uma ordem pré-estabelecida da mesma por diversos nós da rede. Esse fato pressupõe alguma forma de sincronismo entre os nós. A forma que encontramos para compatibilizar o assincronismo e as funções de operações coletivas da *interface* foi através da utilização de modelos distribuídos assíncronos em que as computações locais são guiadas pelas mensagens recebidas. Assim, é possível expressar formalmente as operações coletivas de forma totalmente distribuída e assíncrona apenas pela troca adequada de mensagens. As únicas funções que precisam ser executadas coletivamente são aquelas que configuram e disparam os modelos. Dessa forma, as operações coletivas ficam disponíveis para aplicações que sejam executadas dentro de algum desses modelos.

1.2 Os Modelos de Computação Distribuída

Os modelos de computação distribuída consistem em um formalismo para a descrição precisa de computações distribuídas [1]. Com a ajuda de um tal modelo, identificamos padrões no comportamento coletivo existentes nas aplicações assíncronas. Nos Capítulos 2 e 3, fazemos uma descrição completa desses modelos. Nos atemos aqui somente a uma visão geral das computações que podem ser expressas a partir destes dois modelos.

Para os modelos, fornecemos como entrada um grafo $\Gamma = (N, L)$, no qual $N = \{1, 2, \ldots, n\}$ e L correspondem ao conjunto dos nós da rede computacional e o conjunto de canais bidirecionais entre esses nós, respectivamente. Refere-se a um canal através da notação ij, em que $i, j \in N$ são os nós referentes às duas extremidades do canal. Definimos também a vizinhança de um nó $i \in N$ como o conjunto $N(i)$ de todos os nós tais que existe um canal entre cada um deles e o nó i. Mais precisamente, $N(i) = \{j \in N \mid ij \in L\}$. Um nó somente pode enviar mensagens para um nó que pertence à sua vizinhança.

1.2.1 Computações Distribuídas Dirigidas por Eventos

A noção primordial desse tipo de computações é a de *evento*, denominação dada ao ato de receber uma mensagem e, em resposta a ela, realizar uma computação de maneira atômica, possivelmente alterando o estado local do nó.

Um estado inicial de uma computação distribuída, chamado de *estado inicial global*, é composto pelos estados iniciais dos nós e nenhuma mensagem em trânsito na rede. Uma computação distribuída é definida como as computações feitas em cada nó (e as correspondentes interações entre os nós) a partir de estado inicial global até um estado de terminação global. Em uma computação dirigida por eventos, a transição local de cada nó é guiada pelo recebimento de uma

mensagem enviada por um de seus vizinhos e, em seguida, por uma computação local associada à mensagem recebida, a qual caracteriza o processamento de um evento.

O **Algoritmo 1.1** é uma descrição genérica de uma computação distribuída dirigida por eventos em um nó identificado por i da rede Γ. No algoritmo abaixo, o recebimento de uma mensagem msg_i de um vizinho de i altera o estado local do nó i através da execução da função $EVENT_i$ que corresponde a uma computação local particular associada ao nó i. Além de alterar o estado local do nó i, $EVENT_i$ gera um conjunto de mensagens MSG_i, possivelmente vazio, em que cada mensagem é destinada a um vizinho de i. A primeira chamada da função $EVENT_i$, consiste do evento espontâneo, que é gerado sem a recepção de mensagem. O algoritmo termina quando o nó tem o conhecimento do final da computação distribuída (linha 4).

Algoritmo 1.1: Computação feita no nó i no modelo de computações dirigidas por eventos

Entrada: Conjunto $N(i)$ de vizinhos de i em Γ, estado local inicial para i

1. Registre o estado local inicial do nó i
2. $EVENT_i(-, MSG_i)$
3. Envie cada mensagem em MSG_i para seu respectivo vizinho
4. **enquanto** a terminação global não for conhecida por i **faça**
5. **se** uma mensagem msg_i de um vizinho de i for recebida **então**
6. $EVENT_i(msg_i, MSG_i)$
7. Envie cada mensagem em MSG_i para seu respectivo vizinho

1.2.2 Computações Distribuídas Dirigidas por Pulsos

Enquanto o modelo de computações dirigidas por eventos é adequado para descrever situações em que as mudanças nos estados locais acontecem exclusivamente por reação a uma recepção de mensagem, há computações distribuídas em que essas mudanças de estado podem ocorrer mesmo quando mensagens não são recebidas.

Nessa classe de computações, a evolução da computação distribuída, do estado inicial até o estado final, é guiada por um mecanismo que gera uma sequência de pulsos (mecanismo de geração de pulsos), o qual governa a evolução do estado de cada nó i. Tal mecanismo é implementado através das funções abstratas *getCurrent* e *hasAdvanced* do **Algoritmo 1.2** a seguir. A primeira função é usada para notificar o mecanismo de geração de pulsos que o nó i começou sua computação local associada ao pulso gerado mais recente. A segunda é uma função booleana utilizada pelo mecanismo de geração de pulsos para a sinalização da geração de um novo pulso. Se *hasAdvanced* retorna **true** então ela não retornará **true** novamente antes de *getCurrent* ser chamada. $EVENT_i$ tem a função somente de obter informações relevantes da mensagem recebida, não gerando mensagens. A transição do estado do nó é executada pela função $PULSE_i$, a qual recebe como parte de sua entrada o número do pulso corrente.

Algoritmo 1.2: Computação feita no nó i no modelo de computações dirigidas por pulsos
Entrada: Conjunto $N(i)$ de vizinhos de i em Γ, estado local inicial para i
1. $\ell_i \leftarrow getCurrent()$
2. $PULSE_i(\ell_i, MSG_i)$
3. Envie cada mensagem em MSG_i para seu respectivo vizinho
4. **enquanto** a terminação global não for conhecida por i **faça**
5. **se** $hasAdvanced()$ **então**
6. $\ell_i \leftarrow getCurrent()$
7. $PULSE_i(\ell_i, MSG_i)$
8. Envie cada mensagem em MSG_i para seu respectivo vizinho
9. **senão se** uma mensagem msg_i de um vizinho de i for recebida **então**
10. $EVENT_i(msg_i)$

1.3 Operações Coletivas

Operações são chamadas de coletivas por envolverem diversas computações locais em mais de um nó de processamento em sua realização. O possível assincronismo presente nessas operações está relacionado ao fato de as computações locais envolvidas não estarem restritas a seguir uma única ordem de execução. Em nosso estudo, enumeramos diversas operações coletivas assíncronas e, dentre elas, escolhemos algumas, as quais consideramos abranger uma gama maior de aplicações. As mesmas são definidas a seguir.

1.3.1 Registro dos Estados Locais Iniciais

Nos referidos modelos, supomos que o grafo Γ que descreve a rede e os estados iniciais dos nós são conhecidos inicialmente pelo nó 0. Assim, devemos difundir essas informações para os demais nós da rede. A difusão dessas informações para todos os outros nós da rede constitui o problema do registro dos estados locais iniciais.

Uma propagação de uma mensagem consiste em um nó i inicialmente enviá-la para todos os seus vizinhos e cada vizinho j, recursivamente, enviá-la a seus vizinhos, com exceção de i. Observe que uma propagação de uma mensagem gera uma árvore em que cada nó é filho do nó que primeiro lhe enviou a mensagem.

Uma mensagem de realimentação de j para i é uma mensagem enviada como resposta a uma mensagem anterior que foi enviada de i para j.

Neste trabalho, desenvolvemos, nos Capítulos 2 e 3, algoritmos que utilizam um mecanismo de propagação com realimentação nos dois modelos para a resolução do problema do registro dos estados locais iniciais.

1.3.2 Ordenação de Mensagens

As computações distribuídas dirigidas por eventos e por pulsos descritas anteriormente são não-determinísticas. Tal não-determinismo deve-se pelo fato de as computações locais nos nós da rede serem assíncronas e pelo fato de os atrasos nas transferências das mensagens no modelo serem não-determinísticos [4]. O não-determinismo em computações distribuídas torna difícil projetar e testar programas distribuídos, uma vez que para uma mesma entrada externa, o programa pode apresentar múltiplos comportamentos. Uma solução bastante utilizada para a redução do não-determinismo em programas distribuídos é a imposição mecanismos de ordenação às mensagens trocadas na rede [9]. Em algumas aplicações, como o problema dos leitores/escritores, o uso

de protocolos de ordenação de mensagens para o controle do não-determinismo revela ganho significativo de desempenho para a aplicação [16].

Dentre os mecanismos de ordenação de mensagens mais utilizados, escolhemos incorporamos à nossa *interface* as ordens FIFO (do inglês *First In, First Out*) e Causal. Buscamos na literatura algoritmos para a essas ordens [4, 9]. Na *interface*, o usuário, no momento da execução de uma aplicação que utiliza um dos modelos, pode escolher, através da atribuição de um parâmetro, se utilizará algum mecanismo de ordenação e, caso sim, qual ordem utilizará. A descrição dessas ordens, bem como os algoritmos adotados serão detalhados posteriormente.

1.3.3 Detecção da Terminação Global

O estado global em uma computação distribuída é definido pela união dos estados locais dos nós mais os estados dos canais de comunicação. Nos modelos supomos que nenhum nó possui o conhecimento completo do estado global e não supomos a existência de um relógio global. Logo, determinar se a computação distribuída terminou se torna um problema que requer um tratamento cuidadoso [13]. Dessa forma, propomos também neste trabalho algoritmos baseados no algoritmo *Dijkstra-Scholten* [17] nos dois modelos para o problema da detecção da terminação global, detalhados nos Capítulos 2 e 3. Na *interface*, permitimos que o usuário escolha, através de um parâmetro passado ao modelo, se o modelo deverá detectar a terminação global ou se a própria aplicação indicará ao modelo que a terminação foi detectada.

1.4 Aplicações

Um problema de otimização combinatória, de uma forma genérica, tem como entrada uma descrição compacta de um conjunto enumerável $\mathcal{S}$ de possíveis soluções e uma função f que atribui um valor $f(S)$ a cada elemento S de $\mathcal{S}$. O problema consiste em determinar $S \in \mathcal{S}$ que seja "melhor" que todos os demais. Essa noção de "melhor" pode ser de dois tipos. Em um problema de *minimização*, deseja-se encontrar $S \in \mathcal{S}$ tal que $\forall S' \in \mathcal{S} \Rightarrow f(S) \leq f(S')$. Por outro lado, um problema de *maximização* é definido como a procura de $S \in \mathcal{S}$ tal que $\forall S' \in \mathcal{S} \Rightarrow f(S) \geq f(S')$. Em ambos os casos, a solução S é uma *solução ótima*.

A título de ilustração, considere $G = (V, E)$ um grafo. Um *conjunto independente* S é um subconjunto de V cujos elementos são mutuamente não adjacentes em G. Um problema de maximização associado a conjuntos independentes, denominado problema do *Conjunto Independente Máximo (CIM)*, é definido tomando $\mathcal{S}$ como o conjunto de todos os conjuntos independentes de G e a definindo a função $f(S) = |S|$, para todo $S \in \mathcal{S}$. A forma compacta usualmente adotada para descrever $\mathcal{S}$ neste caso é o próprio grafo G. A dificuldade de resolução desse problema para grafos arbitrários reside no fato de que $|\mathcal{S}|$ é, para muitos grafos, exponencial em $|V|$, inviabilizando a enumeração exaustiva de $\mathcal{S}$ [21, 22]. Utilizamos a notação $\alpha(G)$ para representar o conjunto independente máximo para G.

O Problema CIM é considerado computacionalmente difícil, pois pertence à classe NP-Difícil [8], fato esse que está ligado à possibilidade de existência de um número exponencial de conjuntos independentes em um grafo (esse número pode chegar a $3^{\frac{n}{3}}$) [8].

A dificuldade de resolução exibida pelo CIM, ligada ao grande número de elementos em $\mathcal{S}$, é reproduzida em muitos problemas de otimização. Existe uma fonte de paralelismo intrínseca aos

métodos para resolução de problema de otimização combinatória difíceis, a qual está ligada ao fato de diversas soluções serem enumeradas (mesmo que em número reduzido com relação a $|\mathcal{S}|$, a quantidade de soluções enumeradas tende a ser grande). Uma vez que vários nós de processamento estão disponíveis, pode-se acelerar a enumeração visitanto diversas soluções em paralelo. Seguindo esse princípio, a estratégia natural de paralelização consiste em repartir as soluções a enumerar entre os nós de processamento. Caso essa repartição pudesse ser feita de forma equitável *a priori*, os nós poderiam trabalhar independentemente (e de forma totalmente assíncrona) em suas respectivas enumerações. A dificuldade reside no fato de não se conhecer, de antemão, as soluções a serem enumeradas, visto que elas são escolhidas durante a própria enumeração, segundo a heurística utilizada para guiá-la [3, 5–7, 10].

Apesar de o assincronismo ser um elemento essencial para a obtenção de algoritmos paralelos eficientes baseados em métodos enumerativos para problemas de otimização combinatória, interações entre diferentes nós devem ocorrer durante uma computação distribuída. O objetivo dessas interações é manter um bom balanceamento das enumerações realizadas por diferentes nós a fim de minimizar o número de soluções enumeradas na versão paralela que não o são na versão sequencial. Os modelos baseados em eventos ou pulsos são adequados a esse cenário.

O *branch-and-bound* é um método genérico para encontrar soluções ótimas em problemas de Otimização Combinatória. Consiste de uma enumeração sistemática de todas as soluções candidatas, onde grandes subconjuntos de soluções candidatas infrutíferas são descartados em massa, utilizando limites superiores e inferiores estimados para a quantidade a ser otimizada [15]. O *branch-and-bound* para o CIM é descrito a seguir. Inicialmente, determina-se uma estimativa, na forma de um limite superior, para $\alpha(G)$. Em seguida, escolhe-se um vértice $v \in V$ e divide-se a enumeração em duas partes: em uma, são considerados apenas os conjuntos independentes que contêm v; na outra, somente aqueles que *não* contêm v. Cada uma dessas partes recebe o nome de *subproblema*. A enumeração prossegue para o primeiro dos subproblemas (aquele dos conjuntos independentes contendo v), que pode ser vista como a enumeração dos conjuntos independentes do subgrafo de G definido por todos os vértices que não são adjacentes a v. Dessa forma, aplica-se recursividade (calculando uma estimativa para o subproblema e dividindo a sua enumeração) até que se determine o maior dos conjuntos independentes contendo v. Para tratar dos conjuntos independentes que não contêm v, compara-se o melhor conjunto independente já obtido com uma estimativa do maior que se pode obter sem incluir v. Caso a estimativa indique a possibilidade de existência de uma solução melhor, procede-se à enumeração da segunda parte, também recursivamente. Os detalhes desse algoritmo (incluindo o cálculo da estimativa de cada subproblema e o critério utilizado para escolher o vértice usado para gerar subproblemas) estão em [23], e alguns deles são apresentados no Capítulo 5.

1.5 Organização do Documento

Este documento está organizado da seguinte forma: no Capítulo 2 será feita uma descrição do Modelo de Computação Distribuída Dirigida por Eventos e a definição das operações coletivas através deste modelo. Em seguida, no Capítulo 3, descrevemos o Modelo de Computação Distribuída Dirigida por Pulsos e as operações coletivas neste modelo. O Capítulo 4 mostra a *interface* de programação distribuída fruto deste trabalho.

O Capítulo 5 é destinado às aplicações mencionadas. Nele, fazemos a descrição da técnica *branch-and-bound*, do problema CIM e de suas versões nos modelos de computações distribuídas. No Capítulo 6, descrevemos os experimentos efetuados e avaliamos nossa implementação mediante os experimentos feitos. Por fim, no Capítulo 7, delineamos as principais dificuldades encontradas neste trabalho, nossas contribuições, os produtos gerados e propostas para trabalhos futuros. Logo após, estão as referências bibliográficas.

Capítulo 2

O Modelo de Computação Distribuída Dirigida por Eventos

Neste capítulo, descrevemos o Modelo de Computação Distribuída Dirigida por Eventos. Este modelo apresenta um formalismo para a perfeita caracterização de computações descritas pelo **Algoritmo 1.1** do Capítulo 1. Através deste formalismo, podemos definir precisamente o significado, dentro deste modelo, das operações coletivas mencionadas na Seção 1.3. Apresentamos também os algoritmos que realizam tais operações coletivas.

2.1 Computações Distribuídas como Conjunto de Eventos

A partir do **Algoritmo 1.1** do Capítulo 1, podemos caracterizar um evento ξ por uma 6-tupla definida como:

$$\xi = \langle i, r, msg, \sigma, \sigma', MSG \rangle$$

em que:

- i é o nó onde o evento ocorre;
- r é o tempo no relógio local de i em que o evento ocorreu;
- msg é a mensagem que dispara o evento, se existe alguma;
- σ é o estado do nó i antes da ocorrência do evento;
- σ' é o estado do nó i após da ocorrência do evento;
- MSG é o conjunto de mensagens geradas como resultado da ocorrência do evento. Este conjunto pode ser vazio.

Essa definição de evento é baseada na premissa de que cada nó $i \in N$ opera como uma máquina de estados cujas transições são os eventos ocorrendo em i durante a computação distribuída. Uma computação distribuída pode ser vista simplesmente como um conjunto coleção de eventos $\Xi = \Xi_1 \cup \Xi_2 \cup ... \cup \Xi_n$ e cada Ξ_i, $i \in N$, é o conjunto de eventos que ocorrem em i.

As dependências causais dos eventos em uma computação distribuída são descritas formalmente através das ordens parciais $\rightarrow$ e $\rightsquigarrow$ sobre Ξ que passamos a definir. Seja ξ um evento. As funções $node(\xi)$ e $time_i(\xi)$, $i \in N$, retornam, respectivamente, o nó em que ξ ocorre e um valor r_i se ξ é o r_i-ésimo evento que ocorre no nó i. Mais ainda, $msg_i(\xi)$ representa a mensagem recebida no nó i e que dispara o evento ξ'. De maneira análoga, $MSG_i(\xi)$ representa o conjunto de mensagens resultantes da ocorrência do evento ξ.

A relação $\rightarrow$ é tal que $\xi \rightarrow \xi'$, para $\xi, \xi' \in \Xi$, se, e somente se, vale uma das duas condições a seguir:

- $i = node(\xi) = node(\xi') = j$ e $time_i(\xi) = time_i(\xi') - 1$;
- $i = node(\xi) \neq node(\xi') = j$ e $msg_j(\xi') \in MSG_i(\xi)$.

Essas condições incluem na relação $\rightarrow$ pares de eventos que são consecutivos em um mesmo nó ou são a origem e o destino de uma mesma mensagem. Observe que essas são as situações que refletem a intuição natural sobre a dependência causal direta entre eventos.

A relação $\rightsquigarrow$ reflete a situação em que a ocorrência de um evento é desencadeada pela ocorrência de uma sequência de eventos relacionados por $\rightarrow$. Posto de outra forma, $\rightsquigarrow$ é o fecho transitivo de $\rightarrow$. Portanto, se $\xi_1, \ldots, \xi_t \in \Xi$, para algum $t \geq 1$, se $\xi_1 \rightsquigarrow \xi_t$, então $\xi_s \rightarrow \xi_{s+1}$, para cada $s \in 1, ..., t-1$.

Na figura adiante, ilustramos uma computação distribuída dirigida por eventos. As linhas horizontais representam as linhas da evolução do tempo de três nós $i, j, k \in N$. Um círculo preto indica a ocorrência de um evento no respectivo nó. Uma seta entre dois eventos indica que eles se relacionam através de $\rightarrow$. Um exemplo de par da relação $\rightsquigarrow$ é representada través de uma seta ondulada.

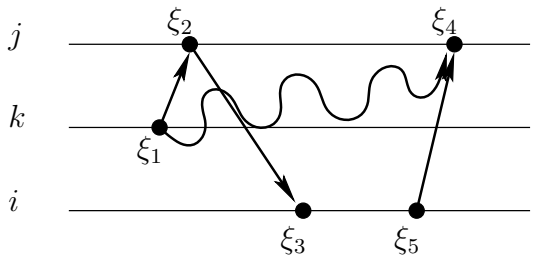

Figura 2.1: Exemplo de eventos no modelo de computação distribuída dirigida por eventos

2.2 Um Algoritmo Genérico

Nesta seção, descrevemos um algoritmo genérico, dotado das operações coletivas descritas no Capítulo 1, para o modelo de computações dirigidas por eventos. Por simplicidade, os controles relativos às ordenações de mensagens não estão presentes nesse algoritmo.

As operações coletivas estão sempre associadas a uma computação específica, denominada de substrato. Assim sendo, dois tipos de mensagens circulam durante uma execução de um tal algoritmo: as mensagens do substrato e as mensagens destinadas ao controle das operações

coletivas. A apresentação feita neste capítulo mostra em detalhes apenas mensagens das operações coletivas e os respectivos eventos.

Na linha 1 é feita inicialização das variáveis de controle locais do algoritmo. Somente o nó 0 recebe como entrada o grafo Γ que descreve a rede e os estados locais iniciais dos nós. Na linhas 2 e 3, o nó 0 inicia a propagação destas informações segundo a operação coletiva de registro dos estados locais iniciais. Enquanto o nó não tiver recebido essas informações, as mensagens do substrato são acumuladas em Φ_i e as mensagens de controle do registro dos locais iniciais são processadas (linhas 5-10).

Por conveniência, criamos a função $SPONTANEOUS\_EVENT_i$ para tratar os eventos espontâneos, que são eventos gerados sem a recepção de mensagens. O algoritmo para a detecção da terminação global exige que somente o nó 0 efetue evento espontâneo. Nas linhas 11 a 13, o nó 0 processa o evento espontâneo. Se o retorno das funções $SPONTANEOUS\_EVENT_i$ e $EVENT_i$ é 0, isso significa que o nó se manteve ativo após o último evento gerado. Quando isso ocorre, computações referentes à detecção da terminação global são efetuadas. Uma observação importante é que solicitações aos modelos feitas internamente às funções $SPONTANEOUS\_EVENT_i$ e $EVENT_i$ podem desencadear computações locais do modelo e envio de mensagens de controle (linhas 11-13, 19-21 e 27-29). Trataremos este fato com mais detalhes no Capítulo 4.

Para os demais nós, as mensagens em Φ_i são processadas (linhas 14-21). Se a mensagem for do substrado, um evento é processado. Se a mensagem for de controle, a computação associada é realizada. A computação então prossegue até que a terminação global seja detectada (linhas 22-29). De forma semelhante às mensagens contidas em Φ_i, uma computação é feita para cada mensagem recebida.

Algoritmo 2.1: Algoritmo genérico utilizando o modelo de computações dirigidas por eventos no nó i

Entrada: identificação $i \in N$, grafo $\Gamma = (N, L)$, conjunto $\Sigma = \{\sigma_j \mid j \in N\}$ de estados locais iniciais

1. Inicialização das variáveis de controle locais
2. **se** $i = 0$ **então**
3. Inicia registro de estados locais iniciais
4. $\Phi_i \leftarrow \emptyset$
5. **enquanto** não estiver pronto para computação do substrato **faça**
6. aguarde chegada de mensagem msg_i
7. $\triangleright$ type(msg_i) = mensagem de controle de registro de estados locais iniciais:
8. realize computação correspondente
9. $\triangleright$ type(msg_i) $\neq$ mensagem de controle de registro de estados locais iniciais:
10. $\Phi_i \leftarrow \Phi_i \cup \{msg_i\}$
11. **se** $SPONTANEOUS\_EVENT_i(MSG_i) = 0$ **então**
12. computações referentes à terminação global
13. envie cada mensagem em MSG_i para seu respectivo vizinho
14. **enquanto** $\Phi_i \neq \emptyset$ **faça**
15. remova msg_i de Φ_i
16. $\triangleright$ type(msg_i) = mensagem de controle:
17. realize computação correspondente
18. $\triangleright$ type(msg_i) = mensagem do substrato:
19. **se** $EVENT_i(msg_i) = 0$ **então**
20. computações referentes à terminação global
21. envie cada mensagem em MSG_i para seu respectivo vizinho
22. **enquanto** terminação global não for conhecida por i **faça**
23. aguarde chegada de mensagem msg_i
24. $\triangleright$ type(msg_i) = mensagem de controle:
25. realize computação correspondente
26. $\triangleright$ type(msg_i) = mensagem do substrato:
27. **se** $EVENT_i(msg_i) = 0$ **então**
28. computações referentes à terminação global
29. envie cada mensagem em MSG_i para seu respectivo vizinho

2.3 Operações Coletivas

2.3.1 Registro dos Estados Locais Iniciais

A estratégia para difundir o grafo Γ e os estados iniciais dos nós utiliza o seguinte mecanismo de propagação com realimentação: o nó 0 inicia a propagação, enviando as informações para seus vizinhos [1]. Cada nó $i \in N$, ao receber essas informações vindas de um nó $j \in N(i)$, marca j como seu pai na árvore da propagação. Em seguida, i dá prosseguimento à propagação, enviando mensagens contendo as informações para seus vizinhos, com exceção do seu pai. Em seguida, i aguarda o recebimento de todas as mensagens de realimentação referentes às mensagens enviadas. Ao receber todas elas, i envia uma mensagem de realimentação para seu pai. Observe que este processo segue o mesmo principio do modelo de computações dirigidas por eventos, em que cada nó reage a mensagens recebidas.

No algoritmo proposto neste trabalho, primeiramente, o nó 0 faz uma propagação das informações com realimentação. Ao receber a realimentação de todas as mensagens da primeira propagação, o nó 0 inicia uma segunda propagação, destinada a avisar a cada nó que ele pode iniciar sua execução, ou seja, pode executar seu evento espontâneo. Um canal FIFO é tal que se duas mensagens forem enviadas pelo canal, elas são entregues na mesma ordem em que foram enviadas. No entanto, como no modelo não há suposição de que os canais sejam FIFO, as mensagens do substrato que alcançarem um nó que ainda não tenha sido alcançado pela segunda propagação têm

o recebimento retardado para após o evento espontâneo.

Propomos o **Algoritmo 2.2**, que é uma extensão do **Algoritmo 1.1** do Capítulo 1, para o registro dos estados locais iniciais no modelo de computações dirigidas por eventos. Nessa extensão, o nó iniciador inclui uma propagação de informação com realimentação no início da computação [1] com o objetivo de informar aos demais nós o grafo Γ sobre o qual o modelo é definido e os estados locais iniciais. Os tipos de mensagens de modelo utilizadas são:

- $config(\Gamma, \Sigma)$: indica a primeira propagação, contendo o grafo e o conjunto dos estados locais iniciais dos nós;
- $init\_msg$: indica segunda propagação, destinada a avisar a cada nó que ele pode realizar evento espontâneo.

Segue a descrição das variáveis adicionais utilizadas:

- $parent_i$: é utilizada para marcar, no nó i, seu pai na árvore da primeira propagação;
- $deficit_i$: conta o número de mensagens a ser recebidas pelo nó i na primeira propagação;
- $init_i$: indica se o nó i já foi atingido pela segunda propagação;
- Φ_i: acumula mensagens do substrato recebidas antes da execução do evento espontâneo no nó i. Tais mensagens são entregues à aplicação após a execução do evento espontâneo.

Algoritmo 2.2: Computação feita no nó i no modelo de computações dirigidas por eventos com inicialização de estados locais iniciais

Entrada (quando $i = 0$): grafo $\Gamma = (N, L)$, estado local inicial σ_j, para todo $j \in N$

1. $deficit_i \leftarrow 0$
2. $parent_i \leftarrow NULL$
3. $init_i \leftarrow$ **false**
4. $\Phi_i \leftarrow \emptyset$
5. **se** $i = 0$ **então**
6. Registre σ_i e $N(i)$
7. Envie $config(\Gamma, \Sigma)$ para todo vizinho em $N(i)$
8. $deficit_i \leftarrow deficit_i + |N(i)|$
9. **senão**
10. **quando** uma $msg_i = config(\Gamma, \Sigma)$ de um vizinho j de i for recebida **faça**
11. Registre σ_i e $N(i)$
12. $parent_i \leftarrow j$
13. Envie $config(\Gamma, \Sigma)$ para todo vizinho em $N(i)$, exceto para $parent_i$
14. $deficit_i \leftarrow deficit_i + |N(i)|$
15. **enquanto** $deficit_i > 0$ **faça**
16. **se** uma mensagem msg_i de um vizinho de i for recebida **então**
17. $deficit_i \leftarrow deficit_i - 1$
18. **se** $parent_i \neq NULL$ **então**
19. Envie $config(-, -)$ para $parent_i$
20. $parent_i \leftarrow NULL$
21. **senão**
22. Envie $init\_msg$ para todo vizinho em $N(i)$
23. $init_i \leftarrow$ **true**
24. $SPONTANEOUS\_EVENT_i(MSG_i)$
25. Envie cada mensagem em MSG_i para seu respectivo vizinho
26. **enquanto** a terminação global não for conhecida por i **faça**
27. $\triangleright$ **se** uma mensagem msg_i de um vizinho j de i for recebida **então**
28. **se** $i \neq 0$ e $parent_i = NULL$ **então**
29. $parent_i \leftarrow j$
30. **se** $msg_i = init\_msg$ **então**
31. **se** $\neg init_i$ **então**
32. $init_i \leftarrow$ **true**
33. $SPONTANEOUS\_EVENT_i(MSG_i)$
34. Envie cada mensagem em MSG_i para seu respectivo vizinho
35. **enquanto** $\Phi_i \neq \emptyset$ **faça**
36. $\Phi_i \leftarrow \Phi_i \backslash \{msg_i\}$
37. $EVENT_i(msg_i, MSG_i)$
38. Envie cada mensagem em MSG_i para seu respectivo vizinho
39. **senão**
40. **se** $init_i$ **então**
41. $EVENT_i(msg_i, MSG_i)$
42. Envie cada mensagem em MSG_i para seu respectivo vizinho
43. **senão**
44. $\Phi_i \leftarrow \Phi_i \cup \{msg_i\}$

Primeiramente, é feita a propagação dos estados iniciais (linhas 1-20). A propagação inicial é incluída de uma forma que pode acarretar alteração de ordem de entrega de mensagens, pois somente as mensagens do tipo *config* são tratadas antes da linha 26. A justificativa é apenas a conveniência de estruturação do código. Uma possível consequência é o acúmulo de mensagens de computação (ou qualquer outra de tipo diferente de *config*) nos *buffers* dos canais até que a realimentação da propagação inicial tenha fim. Naturalmente, supõe-se que esses *buffers* tenham tamanho suficiente para esse acúmulo de mensagens, o que é verdadeiro para os casos práticos.

Em seguida, o nó 0 inicia a segunda propagação para ativar o evento espontâneo de cada nó (linhas 21-25). Se o nó recebe uma mensagem de inicialização ($init\_msg$), ele realiza o evento espontâneo e em seguida realiza os eventos relacionados às mensagens atrasadas (linhas 26-38). Do

ponto de vista do substrato ocorrendo no nó i, este evento pode ser classificado como um evento espontâneo, visto que ele não ocorre em reação a uma mensagem de configuração do modelo.

Se o nó recebe uma mensagem do substrato e já executou o evento espontâneo ele processa o evento relativo a essa mensagem (linhas 40-42). Caso não, o nó acumula a mensagem em Φ_i (linhas 43-44).

2.3.2 Detecção da Terminação Global

Nos referidos modelos, supomos que nenhum nó possui o conhecimento completo do estado global de uma computação distribuída, o qual é definido pela união entre os estados locais dos nós e os estados dos canais de comunicação. Esse fato, aliado à suposição da inexistência de um relógio global, faz com que detectar se a computação distribuída terminou se torne um problema que merece um tratamento cuidadoso [13].

Uma computação distribuída é considerada globalmente terminada quando cada nó se tornou localmente terminado e todas as mensagens enviadas pelos canais da rede foram entregues. Dizemos que um nó se tornou localmente terminado quando ele terminou sua computação local e não reiniciará a menos que receba uma mensagem. A linha 4 do **Algoritmo 1.1**, visto no Capítulo 1, supõe que no nó corrente possamos inferir se a computação distribuída se encontra globalmente terminada. Muitas vezes, a própria aplicação que utiliza um dos modelos se encarrega de informar ao modelo que a computação se tornou globalmente terminada. Para situações em que isso não ocorre, adicionamos aos modelos um mecanismo, o qual é descrito adiante, que se encarrega de detectar se a computação distribuída se tornou globalmente terminada. Em aplicações que utilizem o modelo de computações dirigidas por eventos, as funções $EVENT_i$ e $SPONTANEOUS\_EVENT_i(MSG_i)$ informam ao modelo a terminação a terminação local do nó i.

Utilizamos como base para nossos algoritmos para detecção de terminação o algoritmo desenvolvido por Dijkstra e Scholten [17]. Uma condição para a utilização desse algoritmo é que a computação tenha apenas um evento espontâneo. Dada a forma de registro dos estados locais iniciais, essa condição é satisfeita por nossa implementação do modelo. No algoritmo genérico adiante, considera-se uma estrutura em árvore tal que um nó se junta à árvore para realizar computação. Supomos o nó 0 como a raiz da árvore. Quando um nó pertencente à árvore termina sua computação local e não tem mais filhos na árvore, ele comunica a seu nó pai na árvore o encerramento de sua computação e se desliga da árvore. No caso em que esse nó seja o nó 0, a terminação global foi detectada. Uma mensagem de reconhecimento, $ack(x)$, é enviada por um nó i para um nó j como confirmação do recebimento de x mensagens enviadas por j para i.

Algoritmo 2.3: *Dijkstra-Scholten* para detecção de terminação global no nó i

```
1.  ▷ No recebimento de uma mensagem do substrato de um vizinho j
2.      se i ainda não está na computação então
3.          j se torna pai de i na árvore
4.      senão
5.          Envie mensagem ack(1) para j
6.  ▷ se i se tornou ocioso e não tem mais filhos então
7.      se i ≠ 0 então
8.          i se desliga da árvore enviando ack(1) para seu pai na árvore
9.      senão
10.         A terminação global foi detectada
```

Desenvolvemos, a partir do **Algoritmo 2.3**, o **Algoritmo 2.4**, que detecta a terminação global no Modelo de Computação Distribuída Dirigida por Eventos. No algoritmo que segue, supõe-se que o registro dos estados iniciais é feito conforme o **Algoritmo 2.2** (a parte anterior ao laço delimitado pela condição de terminação, que corresponde às linhas 1 a 25, é omitida no algoritmo a seguir).

Após a detecção da terminação global pelo nó 0, este inicia uma propagação, sem realimentação, que efetivamente encerra as computações dos nós. Dizemos que um nó está ativo quando sua computação local ainda não foi terminada. Segue a descrição das variáveis adicionais utilizadas:

- $acks_i[j]$: conta o número de confirmações devidas pelo nó i a $j \in N(i)$;
- $active_i$: indica se o nó i permaneceu ativo após o último evento;
- $term_i$: conta o número de mensagens da propagação final recebidas pelo nó i. Quando esse contador atinge o valor correspondente ao número de vizinhos em $N(i)$, o nó i encerra a sua computação local.

Os tipos de mensagens de modelo utilizadas são:

- $ack(x)$: representa a confirmação de x mensagens;
- $term\_msg$: representa a mensagem de propagação final.

Algoritmo 2.4: Computação feita no nó i no modelo de computações dirigidas por eventos com terminação

Configuração inicial das variáveis:
1. $term_i \leftarrow 0$
2. $parent_i \leftarrow NULL$
3. **para** $j \in N(i)$ **faça**
4. $\quad acks_i[j] \leftarrow 0$
5. **se** $i = 0$ **então**
6. $\quad active_i \leftarrow SPONTANEOUS\_EVENT_i(MSG_i)$
7. $\quad deficit_i \leftarrow deficit_i + |MSG_i|$
8. $\quad$ Envie cada mensagem em MSG_i para seu respectivo vizinho
9. **senão**
10. $\quad active_i \leftarrow$ **false**

Tratamento de mensagens de terminação:
11. **se** $msg_i = ack(x)$ **então**
12. $\quad deficit_i \leftarrow deficit_i - x$
13. **senão**
14. $\quad$ **se** $msg_i = term\_msg$ **então**
15. $\quad\quad$ **se** $i \neq 0$ **e** $term_i = 0$ **então**
16. $\quad\quad\quad$ Envie $term\_msg$ para todo vizinho em $N(i)$
17. $\quad\quad term_i \leftarrow term_i + 1$

Tratamento de mensagens do substrato ou inicialização:
18. $acks_i[j] \leftarrow acks_i[j] + 1$
19. **se** $parent_i = NULL$ **e** $i \neq 0$ **e** $\neg active_i$ **então**
20. $\quad parent_i \leftarrow j$
21. $\quad active_i \leftarrow$ **true**
22. **senão**
23. $\quad$ **se** $init_i$ **então**
24. $\quad\quad active_i \leftarrow EVENT_i(msg_i, MSG_i)$
25. $\quad\quad deficit_i \leftarrow deficit_i + |MSG_i|$
26. $\quad\quad$ Envie cada mensagem em MSG_i para seu respectivo vizinho
27. $\quad$ **senão**
28. $\quad\quad \Phi_i \leftarrow \Phi_i \cup \{msg_i\}$
29. **se** $init_i$ **e** $j \neq parent_i$ **e** $acks_i[j] \geq 16$ **então**
30. $\quad$ Envie $ack(acks_i[j])$ para j
31. $\quad acks_i[j] \leftarrow 0$
32. **se** $init_i$ **e** $deficit_i = 0$ **e** $\neg active_i$ **então**
33. $\quad$ **para todo** $j \in N(i)$ tal que $acks_i[j] > 0$ **faça**
34. $\quad\quad$ Envie $ack(acks_i[j])$ para j
35. $\quad\quad acks_i[j] \leftarrow 0$
36. $\quad$ **se** $parent_i \neq NULL$ **então**
37. $\quad\quad$ Envie $ack(1)$ para $parent_i$
38. $\quad\quad parent_i \leftarrow NULL$
39. $\quad$ **senão**
40. $\quad\quad$ Envie $term\_msg$ para todo vizinho em $N(i)$

No recebimento de uma mensagem de confirmação ($ack(x)$), x mensagens são confirmadas (linhas 11 e 12). No recebimento de uma mensagem de propagação final ($term\_msg$), a mesma é propagada para os vizinhos do nó (linhas 14-17). No recebimento de uma mensagem do substrato ou de inicialização, se o nó não está na árvore e está ativo ele se junta á árvore (linhas 19-21). O tratamento para mensagens de inicialização foi omitido pois já está descrito no **Algoritmo 2.2** (linhas 30-37).

Se o número de *acks* pendentes para um nó j que não é pai de i é maior ou igual a 16, então essa quantidades de *acks* é enviada (linhas 29-31). O valor 16 é apenas uma convenção e pode ser

alterado de acordo com cada aplicação. Se o nó recebeu todas as suas confirmações e está inativo, ele se desconecta da árvore enviando os *acks* pendentes para seus vizinhos e um *ack* para o seu pai (36-38). No caso em que o nó é o nó 0, a terminação foi detectada e ele inicia a propagação da terminação (linhas 39 e 40).

Observe que, no modelo de computações dirigidas por eventos, um nó, após se desconectar da árvore da computação, pode voltar à computação (se juntar novamente à árvore). Para tanto, basta receber uma mensagem do substrato vinda de um de seus vizinhos.

2.3.3 Ordenação de Mensagens

Como descrito no Capítulo 1, os modelos em estudo são não-determinísticos, permitindo distintas computações para um mesmo algoritmo e uma mesma entrada (a distinção entre esssas computações está na ordem em que as mensagens são entregues aos seus nós de destino). Devido a esse não-determinismo, em algumas aplicações, certas computações distribuídas apresentam propriedades relativas à eficiência ou a outros aspectos que são indesejáveis. Uma forma de prevenir a ocorrência de uma tal computação é a restrição do não-determinismo do modelo. Uma forma de se restringir o não-determinismo dos modelos é através de mecanismos de ordenação de mensagens. Apresentamos adiante os tipos de ordenação de mensagens mais comuns na literatura [9] e seus respectivos algoritmos, os quais foram adaptados para o Modelo de Computação Distribuída Dirigida por Eventos.

O estabelecimento de ordem entre mensagens envolve o estabelecimento de uma ordem de recebimento baseada na ordem em que as mesmas são enviadas. Os envios de mensagens são ordenados segundo os eventos em que ocorrem. A título de ilustração, tomemos duas mensagens msg_1 e msg_2. A comparação entre os eventos ξ_1 e ξ_2 tais que $msg_1 \in MSG(\xi_1)$ e $msg_2 \in MSG(\xi_2)$ define a comparação entre msg_1 e msg_2. Nesse sentido, $msg_1 \rightarrow msg_2$ se e somente se $\xi_1 \rightarrow \xi_2$, assim como $msg_1 \rightsquigarrow msg_2$ se e somente se $\xi_1 \rightsquigarrow \xi_2$.

Observamos que a relação $\rightarrow$ é irreflexiva, ou seja para um evento ξ, $\xi \not\rightarrow \xi$. Visto que $\rightarrow$ é irreflexiva, a relação $\rightarrow$ não inclui pares de mensagens originadas simultanamente. Assim sendo, se duas mensagens msg_1 e msg_2 são enviadas por um mesmo evento, então elas são incomparáveis na relação $\rightarrow$, ou seja, $msg_1 \not\rightarrow msg_2$ e $msg_2 \not\rightarrow msg_1$. Esse fato tem implicações nos algoritmos descritos posteriormente.

Ordem FIFO

A ordem FIFO constitui o mecanismo de ordenação de mensagens menos exigente. Determina que dadas duas mensagens enviadas de um nó $i \in N$ para um nó $j \in N(i)$, então elas devem ser recebidas na mesma ordem em que foram enviadas. Considere dois eventos ξ_1 e ξ_2 disparados por msg_1 e msg_2, respectivamente. A ordem FIFO é expressa formalmente através da seguinte regra:

- se msg_1 e msg_2 são originadas no mesmo nó, $msg_1 \rightsquigarrow msg_2$ e $node(\xi_1) = node(\xi_2)$, então $\xi_1 \rightsquigarrow \xi_2$

A Figura 2.2 a seguir ilustra um cenário que infringe a ordem FIFO no modelo de computações

dirigidas por eventos. As linhas horizontais representam as linhas do tempo de cada nó. Uma seta entre dois nós representa uma mensagem trocada entre os dois nós.

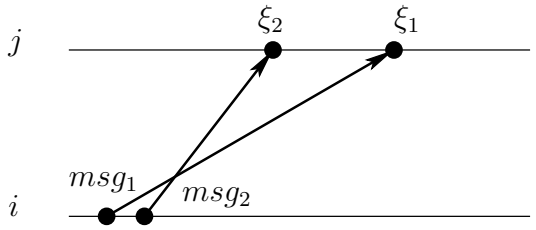

Figura 2.2: Troca de mensagens que não respeita a ordem FIFO no modelo de computações dirigidas por eventos

O **Algoritmo 2.5**, que é uma extensão do algoritmo **Algoritmo 1.1** do Capítulo 1, implementa a ordem FIFO no modelo de computações dirigidas por eventos. Sua ideia básica consiste no uso de uma sequência de número para as mensagens. Quando uma mensagem é recebida fora de ordem ela deve ser atrasada. Dizemos que uma mensagem está habilitada quando sua recepção não infringe a ordem de mensagens imposta. Segue a descrição das variáveis utilizadas:

- N_i: é um vetor contendo $|N(i)|$ entradas. Dado um vizinho j de i, se temos $N_i[j] = k$, então j é o k-ésimo elemento de uma sequência crescente dos nós de $N(i)$ com base nos índices dos nós (j é o k-ésimo vizinho de i). Todas as entradas começam com valor 0;
- S_i: é um vetor com $|N(i)|$ entradas. $S_i[j]$ indica, a cada evento ξ que ocorre em i durante a computação distribuída, o número de mensagens enviadas de i para $N_i[j]$ em eventos que precedem ξ na relação $\rightsquigarrow$. Todas as entradas começam com valor 0;
- s_i: é um vetor com $|N(i)|$ entradas. $s_i[j]$ indica se $S_i[j]$ já foi incremetado no evento corrente. Todas as entradas começam com valor 0;
- R_i: é um vetor com $|N(i)|$ entradas. $R_i[j]$ indica, a cada evento ξ que ocorre em i durante a computação distribuída, o número de mensagens recebidas em ordem no nó i, as quais foram enviadas pelo nó $N_i[j]$, em eventos que precedem ξ na relação $\rightsquigarrow$. Todas as entradas começam com valor 0;
- $S(msg_i)$: é chamado de sequencial da mensagem, é um valor k obtido de uma mensagem msg_i enviada por j. Indica que msg_i foi a k-ésima mensagem enviada de j para i;
- Φ_i: acumula mensagens recebidas fora de ordem.

Algoritmo 2.5: Computação feita no nó i no modelo de computações dirigidas por eventos com ordem FIFO

Envio de mensagens:
1. **para toda** mensagem $msg_i \in MSG_i$, destinada a $N_i[j]$, **faça**
2. **se** $s_i[j] = 0$ **então**
3. $S_i[j] \leftarrow S_i[j] + 1$
4. $s_i[j] \leftarrow 1$
5. Anexe $S_i[j]$ a msg_i
6. Envie msg_i para $N_i[j]$

Recepção de mensagem:
7. $\Phi_i \leftarrow \Phi_i \cup \{msg_i\}$
8. **se** existe mensagem $msg_i \in \Phi_i$, destinada a $N_i[j]$, tal que $S(msg_i) = R_i[j] + 1$ **então**
9. $\Phi_i \leftarrow \Phi_i \backslash \{msg_i\}$
10. $R_i[j] \leftarrow R_i[j] + 1$
11. Receba mensagem msg e realize evento correspondente

Nas linhas 2 a 4, verificamos se $S_i[j]$ já foi incrementado para o evento que acabou de ocorrer. Isto visa garantir a incomparabilidade entre mensagens geradas no mesmo evento e destinadas a um mesmo nó. Se existe alguma mensagem habilitada em Φ_i, ou seja, se o sequencial da mensagem é igual ao número de mensagens já recebidas em i mais um, então processamos o evento relativo a essa mensagem (linhas 8-11).

Ordem Causal

Uma ordenação de mensagens mais restrita que a ordem FIFO é a ordem causal. De forma geral, impõe a restrição que uma mensagem não possa ser ultrapassada por uma sequência de mensagens. Suponha que ξ_1 e ξ_2 sejam os eventos disparados por msg_1 e msg_2, respectivamente. A ordem causal é formalmente descrita pela seguinte regra:

- se $msg_1 \rightsquigarrow msg_2$ e $node(\xi_1) = node(\xi_2)$, então $\xi_1 \rightsquigarrow \xi_2$

Nas Figura 2.3, ilustramos um cenário que infringe a ordem causal no modelo de computações dirigidas por eventos. No exemplo, o nó i envia uma mensagem para o nó j que é ultrapassada por uma sequência de mensagens iniciada no nó i e finalizada no nó j.

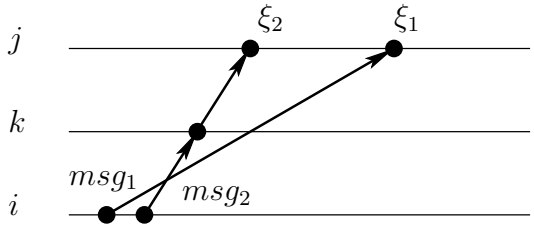

Figura 2.3: Troca de mensagens que não respeita a ordem causal no modelo de computações dirigidas por eventos

O **Algoritmo 2.6** implementa a ordem causal no modelo de computações dirigidas por eventos. As variáveis auxiliares utilizadas são:

- M_i: é uma matriz com $|N|*|N|$ entradas. $M_i[j][k]$ indica, ao conhecimento de i, a cada evento ξ que ocorre em j durante a computação distribuída, o número de mensagens enviadas pelo nó j para o nó k, em eventos que precedem ξ na relação $\rightsquigarrow$. Todas as entradas começam com valor 0;
- m_i: é um vetor com $|N|$ entradas. $m_i[j]$ indica se $M_i[i][j]$ já foi incremetado no evento corrente. Todas as entradas começam com valor 0;
- Φ_i: acumula mensagens recebidas fora de ordem;
- W_i é uma matriz com $|N|*|N|$ entradas. É extraída de uma mensagem e representa seu histórico causal.

Algoritmo 2.6: Computação feita no nó i no modelo de computações dirigidas por eventos com ordem causal

Envio de mensagens:

1. **para toda** mensagem $msg_i \in MSG_i$, destinada a j, **faça**
2. **se** $m_i[j] = 0$ **então**
3. $M_i[i][j] \leftarrow M_i[i][j] + 1$
4. $m_i[j] \leftarrow 1$
5. Anexe M_i a msg_i
6. Envie msg_i para j

Recepção de mensagem:

7. $\Phi_i \leftarrow \Phi_i \cup \{msg_i\}$
8. **se** existe mensagem $msg_i \in \Phi_i$, destinada a j, com matriz W_j, tal que $W_j[j][i] = M_i[j][i] + 1$ **então**
9. **se** $\forall k \neq j : M_i[k][i] \geq W_j[k][i]$ **então**
10. $\Phi_i \leftarrow \Phi_i \backslash \{msg_i\}$
11. $M_i \leftarrow max(M_i, W_j)$
12. Receba mensagem msg e realize evento correspondente

No envio de uma mensagem msg_i para $j \in N(i)$, verificamos se $M_i[i][j]$ já foi incrementado no evento corrente e anexamos a matriz M_i a msg_i (linhas 1-6). Nas linhas 8 e 9, verificamos se alguma mensagem $msg_i \in \Phi_i$ está habilitada. Para tanto, primeiramente, verificamos se a mensagem msg_i foi ultrapassada por outra mensagem também enviada por j para i (linha 8). Observe que se para algum k, $W_j[k][i] > M_i[k][i]$, então existe uma mensagem que foi enviada no histórico causal de msg_i que ainda não foi recebida, ou seja, foi ultrapassada por uma sequência de mensagens cuja mensagem final é msg_i. Caso isso não se verifique, msg_i está habilitada (linha 9). Se msg_i está habilitada, a informação na matriz M_i é atualizada com a matriz W_j extraída de msg_i (linha 11) [4, 9].

Capítulo 3

O Modelo de Computação Distribuída Dirigida por Pulsos

O Modelo de Computação Distribuída Dirigida por Pulsos caracteriza-se pela existência de um mecanismo de geração de pulsos que governa a evolução das computações locais em cada nó. Esse fato demanda um formalismo no qual as recepções de mensagens, atreladas a eventos, são separadas das computações, efetuadas nos pulsos. Neste capítulo, mostramos os detalhes desse formalismo, inicialmente considerando que o mecanismo de geração de pulsos é dado pelas funções *getCurrent* e *hasAdvanced* conforme mostrado no Algoritmo 1.2. Em seguida, traçamos considerações sobre a implementação do citado mecanismo utilizando troca de mensagens. A apresentação neste capítulo é fortemente baseada em [4].

3.1 Computações Distribuídas como Sequências de Pulsos

Conforme estabelecido no Algoritmo 1.2, a evolução dos estados locais dos nós é determinada por uma sequência de pulsos ocorrendo nesse nó. Podemos caracterizar um pulso λ por uma 6-tupla definida como:

$$\lambda = \langle i, r, \sigma, \sigma', \Xi, MSG \rangle$$

em que:

- i é o nó onde o pulso ocorre;
- r é o tempo no relógio local de i em que o pulso ocorre;
- σ é o estado do nó i antes da ocorrência do pulso;
- σ' é o estado do nó i após da ocorrência do pulso;
- Ξ é o conjunto de eventos associados ao pulso;
- MSG é o conjunto de mensagens geradas como resultado da ocorrência do pulso. Este conjunto pode ser vazio.

Um evento ξ é, por sua vez, caracterizado por apenas 3 dos parâmetros, pois não altera o estado do nó:

$$\xi = \langle i, msg, \lambda \rangle$$

em que:

- i é o nó onde o evento ocorre;
- msg é a mensagem que dispara o evento;
- λ é o pulso em que o evento ocorre, que deve ser tal que r, o tempo em que λ ocorre, deve ser maior que o tempo de ocorrência do pulso que gera msg;

Uma computação distribuída dirigida por pulsos é caracterizada por um conjunto $\Lambda = \Lambda_1 \cup \Lambda_2 \cup ... \cup \Lambda_n$, onde Λ_i, $i \in N$, é a sequência de pulsos que ocorrem no nó i. Para $\lambda_i \in \Lambda_i$, $MSG_i(\lambda_i)$ é o conjunto de mensagens geradas pela ocorrência do pulso λ_i, pulso esse associado a uma execução da função $PULSE_i$ no **Algoritmo 1.2** do Capítulo 1. Utilizamos a função $rank_i(\lambda_i)$ para se obter uma ordem nos pulsos em i baseada na cronologia em que os pulsos são executados. Utilizamos também a função $pulse_i$ que retorna, para um evento ξ_i, o pulso λ_i que antecedeu o evento ξ_i. Formalmente, $\lambda_i = pulse_i(\xi_i)$.

Podemos definir relações de causalidade entre eventos e entre pulsos para o modelo de computações dirigidas por pulsos, como segue.

A relação $\rightarrow$ definida para eventos é tal que para dois eventos ξ e ξ', $\xi \rightarrow \xi'$, se, e somente se, vale uma das duas condições a seguir:

- $i = node(\xi) = node(\xi') = j$ e $rank_i(pulse_i(\xi)) < rank_i(pulse_i(\xi'))$;
- $i = node(\xi) \neq node(\xi') = j$ e λ e λ' são pulsos tais que $\lambda = pulse_i(\xi)$, $\lambda' = pulse_j(\xi')$ e $msg_j(\xi') \in MSG_i(\lambda)$.

A notar que a causalidade entre eventos em um mesmo nó é determinada pela sequência de pulsos desse nó. Em particular, não há ordem causal entre eventos de um mesmo pulso. Quando comparamos eventos ocorrendo em nós distintos, são as trocas de mensagens que estabelecem as ordens causais. A ordem $\rightarrow$ para pulsos é definida de forma que para dois pulsos λ e λ', $\lambda \rightarrow \lambda'$ implica uma das duas condições a seguir:

- $node(\lambda) = i$, $node(\lambda') = j$, $i = j$ e $rank_i(\lambda) = rank_i(\lambda') - 1$;
- $node(\lambda) = i$, $node(\lambda') = j$, $i \neq j$ e $rank_i(\lambda) = rank_j(\lambda') - 1$.

A ordem parcial $\rightsquigarrow$ entre eventos ou entre pulsos baseada na relação $\rightarrow$ acima é definida de forma semelhante ao modelo de computações dirigidas por eventos.

Na Figura 3.1, ilustramos uma computação no modelo de computações dirigidas por pulsos. Semelhantemente ao modelo de computações dirigidas por eventos, um círculo preto indica a ocorrência de um evento no nó. Um traço vertical numerado indica a ocorrência de um pulso. Diferentemente do modelo de computações dirigidas por eventos, uma seta entre um pulso λ e um evento ξ indica que o $msg_j(\xi) \in MSG_i(\lambda)$.

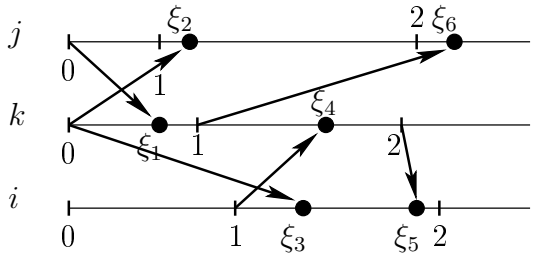

Figura 3.1: Exemplo de computação no modelo de computação distribuída dirigida por pulsos

3.2 Um Algoritmo Genérico

Da mesma forma que no Capítulo 2, fazemos adiante a definição de um algoritmo genérico, dotado das operações coletivas descritas no Capítulo 1, para o modelo de computações dirigidas por pulsos. Novamente, os controles das ordenações de mensagens são omitidas.

As linhas 2 a 10 dizem respeito ao registro dos estados locais iniciais. Caso uma mensagem do substrato seja recebida antes do registro do estado inicial do nó, essa mensagem é armazenada em Φ_i. Após o registro das informações, as mensagens em Φ_i são tratadas (linhas 11-17). Caso msg_i seja uma mensagem do substrato, um evento é gerado e são feitas computações com respeito à detecção da terminação global. A função $PULSE_i$ deve retornar 0 se o nó permanecer ativo após a execução do pulso. Solicitações aos modelos feitas internamente às funções $PULSE_i$ e $EVENT_i$ podem desencadear computações locais do modelo e envio de mensagens de controle (linhas 16 e 17, 27 e 28 e 31-33). Trataremos este fato com mais detalhes no Capítulo 4.

Após a variável Φ_i se tornar vazia, o pulso inicial é executado (linhas 18-21). Para cada mensagem recebida na rede, o tratamento deve ser semelhante às mensagens contidas em Φ_i (linhas 23-28). Caso não exista mensagem a ser recebida e o mecanismo de pulsos tenha avançado (um novo pulso pode ser gerado), um pulso é executado (linhas 29-33).

Algoritmo 3.1: Algoritmo genérico utilizando o modelo de computações dirigidas por pulsos no nó i

Entrada: identificação $i \in N$, grafo $\Gamma = (N, L)$, conjunto $\Sigma = \{\sigma_j \mid j \in N\}$ de estados locais iniciais

```
1. Inicialização das variáveis de controle locais
2. se i = 0 então
3.       Inicia registro de estados locais iniciais
4. Φ_i ← ∅
5. enquanto não estiver pronto para computação do substrato faça
6.       aguarde chegada de mensagem msg_i
7.       ▷ type(msg_i) = mensagem de controle de registro de estados locais iniciais:
8.            realize computação correspondente
9.       ▷ type(msg_i) ≠ mensagem de controle de registro de estados locais iniciais:
10.           Φ_i ← Φ_i ∪ {msg_i}
11. enquanto Φ_i ≠ ∅ faça
12.      remova msg_i de Φ_i
13.      ▷ type(msg_i) = mensagem de controle:
14.           realize computação correspondente
15.      ▷ type(msg_i) = mensagem do substrato:
16.                EVENT_i(msg_i)
17.                computações referentes à terminação global
18. ℓ_i ← getCurrent()
19. se PULSE_i(ℓ_i, MSG_i) = 0 então
20.      computações referentes à terminação global
21. Envie cada mensagem em MSG_i para seu respectivo vizinho
22. enquanto terminação global não for conhecida por i faça
23.      se uma mensagem msg_i de um vizinho de i for recebida
24.           ▷ type(msg_i) = mensagem de controle:
25.                realize computação correspondente
26.           ▷ type(msg_i) = mensagem do substrato:
27.                EVENT_i(msg_i)
28.                computações referentes à terminação global
29.      se hasAdvanced() então
30.           ℓ_i ← getCurrent()
31.           se PULSE_i(ℓ_i, MSG_i) = 0 então
32.                computações referentes à terminação global
33.           Envie cada mensagem em MSG_i para seu respectivo vizinho
```

3.3 Mecanismo de Geração de Pulsos

O **Algoritmo 1.2** do Capítulo 1 descreve genericamente o mecanismo de geração de pulsos através das funções *getCurrent* e *hasAdvanced* [4]. Propomos, portanto, o **Algoritmo 3.2**, o qual apresenta uma forma de implementar o mecanismo de geração de pulsos (que substitui as funções *getCurrent* e *hasAdvanced*) usando unicamente trocas de mensagens. Sua ideia consiste em avançar a computação local de cada nó $i \in N$ (gerando uma sequência de pulsos em i) de forma que, a cada mensagem recebida, i tenha executado pelo menos tantos pulsos quanto o nó que enviou a mensagem anteriormente a esse envio.

Utilizamos a variável auxiliar ℓ_i para contar o número de pulsos executados no nó i. A cada mensagem a ser enviada, ℓ_i é anexado. Para uma mensagem msg recebida, $\ell(msg_i)$ representa o valor do pulso que a gerou. A função $PULSE_i$ informa a terminação local de i ao modelo. Se o nó torna-se inativo, a geração de pulsos é interrompida até terminação global ser detectada ou alguma mensagem do substrato chegar.

Algoritmo 3.2: Computação feita no nó i no modelo de computações dirigidas por pulsos

Entrada: Conjunto $N(i)$ de vizinhos de i em Γ, estado local inicial para i

```
1.  ℓ_i ← 0
2.  PULSE_i(ℓ_i, MSG_i)
3.  Envie cada mensagem em MSG_i para seu respectivo vizinho, anexando ℓ_i
4.  enquanto a terminação global não for conhecida por i faça
5.      enquanto houver uma mensagem msg_i de um vizinho de i disponível faça
6.          Receba msg_i
7.          enquanto ℓ(msg_i) > ℓ_i faça
8.              ℓ_i ← ℓ_i + 1
9.              PULSE_i(ℓ_i, MSG_i)
10.             Envie cada mensagem em MSG_i para seu respectivo vizinho,
                anexando ℓ_i
11.         EVENT_i(msg_i)
12.     ℓ_i ← ℓ_i + 1
13.     PULSE_i(ℓ_i, MSG_i)
14.     Envie cada mensagem em MSG_i para seu respectivo vizinho,
        anexando ℓ_i
```

3.4 Operações Coletivas

3.4.1 Registro dos Estados Locais Iniciais

No Capítulo 2, definimos o problema do registro dos estados locais iniciais dos nós. No **Algoritmo 3.3**, desenvolvemos uma abordagem para o registro dos estados locais iniciais no modelo de computações dirigidas por pulsos. Da mesma forma que no modelo de computações dirigidas por eventos, o nó 0 recebe como entrada o grafo Γ que descreve a rede e a rede e os estados locais iniciais dos nós. O nó 0 inicia uma propagação com realimentação. Ao receber a realimentação da primeira propagação, inicia uma segunda propagação, destinada a ativar cada nó para a execução do primeiro pulso. As mensagens do substrato que alcançarem um nó que ainda não tenha sido alcançado pela segunda propagação têm o recebimento retardado para após o primeiro pulso. As variáveis utilizadas são:

- $parent_i$: é utilizada para marcar, no nó i, a árvore na realimentação da primeira propagação;
- $deficit_i$: conta número de mensagens a serem recebidas pelo nó i na propagação com realimentação;
- $init_i$: indica se o nó i já foi atingido pela segunda propagação;
- Φ_i: acumula as mensagens retardadas para após o primeiro pulso no nó i.

Os tipos de mensagens utilizados são:

- $config(\Gamma, \Sigma)$: indica a primeira propagação, contendo o grafo e o conjunto dos estados locais iniciais;
- $init\_msg$: indica a segunda propagação, disparando a realização do primeiro pulso.

Algoritmo 3.3: Computação feita no nó i no modelo de computações dirigidas por pulsos com inicialização de estados locais iniciais

Entrada (quando $i = 0$): grafo $\Gamma = (N, L)$, estado local inicial σ_j, para todo $j \in N$

1. $\ell_i \leftarrow 0$
2. $deficit_i \leftarrow 0$
3. $parent_i \leftarrow NULL$
4. $init_i \leftarrow$ **false**
5. **se** $i = 0$ **então**
6. Registre σ_i e $N(i)$
7. Envie $config(\Gamma, \Sigma)$ para todo vizinho em $N(i)$
8. $deficit_i \leftarrow deficit_i + |N(i)|$
9. **senão**
10. **quando** uma $msg_i = config(\Gamma, \Sigma)$ de um vizinho j de i for recebida **faça**
11. Registre σ_i e $N(i)$
12. $parent_i \leftarrow j$
13. Envie $config(\Gamma, \Sigma)$ para todo vizinho em $N(i)$, exceto $parent_i$
14. $deficit_i \leftarrow deficit_i + |N(i)|$
15. **enquanto** $deficit_i > 0$ **faça**
16. **se** uma msg_i de um vizinho de i for recebida **então**
17. $deficit_i \leftarrow deficit_i - 1$
18. **se** $parent_i \neq NULL$ **então**
19. Envie $config(-, -)$ para $parent_i$
20. $\Phi_i \leftarrow \emptyset$
21. $parent_i \leftarrow NULL$
22. **senão**
23. Envie $init\_msg$ para todo vizinho em $N(i)$
24. $init_i \leftarrow$ **true**
25. **enquanto** a terminação global não for conhecida por i **faça**
26. **enquanto** houver uma mensagem msg_i de um vizinho j de i disponível **faça**
27. Receba msg_i
28. **se** $i \neq 0$ e $parent_i = NULL$ **então**
29. $parent_i \leftarrow j$
30. **se** $msg_i = init\_msg$ **então**
31. $init_i \leftarrow$ **true**
32. Torne todas as mensagens em Φ_i disponíveis
33. **senão**
34. **se** $init_i$ **então**
35. **enquanto** $\ell(msg_i) > \ell_i$ **faça**
36. $\ell_i \leftarrow \ell_i + 1$
37. $PULSE_i(\ell_i, MSG_i)$
38. Envie cada mensagem em MSG_i para seu respectivo vizinho, anexando ℓ_i
39. $EVENT_i(msg_i)$
40. **senão**
41. $\Phi_i \leftarrow \Phi_i \cup \{msg_i\}$
42. **se** $init_i$ **então**
43. $\ell_i \leftarrow \ell_i + 1$
44. $PULSE_i(\ell_i, MSG_i)$
45. Envie cada mensagem em MSG_i para seu respectivo vizinho, anexando ℓ_i

Primeiramente, é feita a propagação de Γ e os estados locais iniciais dos nós (linhas 1-21). Em seguida, o nó 0 inicia a segunda propagação para habilitar o nó para a execução do primeiro pulso (linhas 22-24). Se o nó recebe uma mensagem de inicialização ($init\_msg$), ele torna todas as mensagens em Φ_i disponíveis (linhas 30-32). Se o nó recebe uma mensagem do substrato e já recebeu a mensagem de inicialização, ele prossegue sua execução normal de acordo com o mecanismo de geração de pulsos (linhas 34-39 e 42-45). Caso contrário, o nó acumula a mensagem em Φ_i (linhas 40 e 41).

3.4.2 Detecção da Terminação Global

De forma semelhante ao Capítulo 2, apresentamos uma abordagem para a detecção da terminação global no Modelo de Computação Distribuída Dirigida por Pulsos. O procedimento apresentado faz parte do mecanismo de geração de pulsos e é uma adaptação do algoritmo de Dikjstra e Scholten.

Da mesma forma que no modelo de computações dirigidas por eventos, supõe-se que o registro dos estados iniciais é feito conforme o **Algoritmo 3.3** (a parte anterior ao laço delimitado pela condição de terminação é omitida no algoritmo adiante). Para reduzir o número de mensagens de *ack*, cada nó acumula um certo número de mensagens de cada vizinho antes de enviar todas as confirmações devidas em uma só mensagem de *ack*.

Diferentemente do modelo de computações dirigidas por eventos, todos os nós começam ativos e uma vez alcançando o estado inativo, o nó não mais poderá se tornar ativo (não gerando pulsos, portanto) até que a detecção da terminação global ou a chegada de uma mensagem do substrato. Assim, quando o nó estiver inativo e receber uma mensagem do substrato ele acumulará *ack* para o nó que enviou a mensagem e retorna ao estado ativo. Após a detecção da terminação global pelo nó 0, esse inicia uma propagação, sem realimentação, que efetivamente encerra as computações dos nós.

Seguem as variáveis adicionais utilizadas:

- $acks_i[j]$: representa o número de confirmações devidas pelo nó i a $j \in N(i)$;
- $deficit_i$: conta número de mensagens enviadas que não foram confirmadas por *ack*. Quando o nó se torna inativo e $deficit_i = 0$, isso significa que o nó terminou sua computação e iniciará a propagação da terminação. Entretanto, no modelo diridido por pulsos, como ressaltado anteriormente, um nó após se desconectar da árvore da computação o mesmo não poderá voltar a fazer computação. Assim, fazemos $deficit_i \leftarrow -1$ para que o nó propague a terminação somente uma vez;
- $active_i$: indica se o nó i permaneceu ativo após o último pulso. Todos os nós começam ativos;
- $term_i$: conta o número de mensagens da propagação final recebidas pelo nó i. Quando esse contador atinge o valor correspondente ao número de vizinhos em $N(i)$, o nó i encerra a sua computação local.

As mensagens adicionais utilizadas são:

- $ack(x)$: confirmação de x mensagens;
- $term\_msg$: representa a mensagem de propagação final.

Algoritmo 3.4: Computação feita no nó i no modelo de computações dirigidas por pulsos com inicialização e terminação

Entrada: conjunto $N(i)$ de vizinhos i em Γ, estado local inicial para i

1. $\ell_i \leftarrow term_i \leftarrow deficit_i \leftarrow 0$
2. $init_i \leftarrow$ **false**
3. $parent_i \leftarrow NULL$
4. **para** $j \in N(i)$ **faça**
5. $acks_i[j] \leftarrow 0$
6. $active_i \leftarrow PULSE_i(\ell_i, MSG_i)$
7. $deficit_i \leftarrow deficit_i + |MSG_i|$
8. Envie cada mensagem em MSG_i para seu respectivo vizinho, anexando ℓ_i
9. **enquanto** $term_i < |N(i)|$ **faça**
10. **se** $\neg active_i$ **então**
11. Aguarde chegada de mensagem
12. **enquanto** houver uma mensagem msg_i de um vizinho j de i disponível **faça**
13. Receba msg_i
14. **se** $msg_i = ack(x)$ **então**
15. $deficit_i \leftarrow deficit_i - x$
16. **senão se** $msg_i = term\_msg$ **então**
17. **se** $i \neq 0$ **e** $term_i = 0$ **então**
18. Envie $term\_msg$ para todo vizinho em $N(i)$
19. $term_i \leftarrow term_i + 1$
20. **senão** * mensagem do substrato ou inicialização *\
21. $acks_i[j] \leftarrow acks_i[j] + 1$
22. **se** $parent_i = NULL$ **e** $i \neq 0$ **então**
23. $parent_i \leftarrow j$
24. $active_i \leftarrow true$
25. **senão**
26. Envie $ack(1)$ para j
27. **se** $msg_i = init\_msg$ **então**
28. **se** $\neg init_i$ **então**
29. $init_i \leftarrow$ **true**
30. $active_i \leftarrow PULSE_i(\ell_i, MSG_i)$
31. $deficit_i \leftarrow deficit_i + |MSG_i|$
32. Envie cada mensagem em MSG_i para seu respectivo vizinho, anexando ℓ_i
33. **enquanto** $\Phi_i \neq \emptyset$ **faça**
34. $\Phi_i \leftarrow \Phi_i \backslash \{msg_i\}$
35. $j \leftarrow orig(msg_i)$
36. **enquanto** $\ell(msg_i) > \ell_i$ **faça**
37. $\ell_i \leftarrow \ell_i + 1$
38. $active_i \leftarrow PULSE_i(\ell_i, MSG_i)$
39. $deficit_i \leftarrow deficit_i + |MSG_i|$
40. Envie cada mensagem em MSG_i para seu respectivo vizinho, anexando ℓ_i
41. $EVENT_i(msg_i)$
42. **senão**
43. **se** $init_i$ **então**
44. **enquanto** $\ell(msg_i) > \ell_i$ **faça**
45. $\ell_i \leftarrow \ell_i + 1$
46. $active_i \leftarrow PULSE_i(\ell_i, MSG_i)$
47. $deficit_i \leftarrow deficit_i + |MSG_i|$
48. Envie cada mensagem em MSG_i para seu respectivo vizinho, anexando ℓ_i
49. $EVENT_i(msg_i)$
50. **senão**
51. $\Phi_i \leftarrow \Phi_i \cup \{msg_i\}$
52. **se** $init_i$ **e** $j \neq parent_i$ **e** $acks_i[j] \geq 16$ **então**
53. Envie $ack(acks_i[j])$ para j
54. $acks_i[j] \leftarrow 0$
55. **se** $init_i$ **então**
56. $\ell_i \leftarrow \ell_i + 1$
57. $active_i \leftarrow PULSE_i(\ell_i, MSG_i)$
58. $deficit_i \leftarrow deficit_i + |MSG_i|$
59. Envie cada mensagem em MSG_i para seu respectivo vizinho, anexando ℓ_i

```
60.          se ¬active_i então
61.               para todo j ∈ N(i) tal que j ≠ parent_i e acks_i[j] > 0 faça
62.               Envie ack(acks_i[j]) para j
63.               acks_i[j] ← 0
64.     se init_i e deficit_i = 0 e ¬active_i então
65.          deficit_i ← −1
66.          se parent_i ≠ NULL então
67.                    Envie ack(acks_i[parent_i]) para parent_i
68.                    acks_i[parent_i] ← 0
69.                    parent_i ← NULL
70.          senão
71.                    Envie term_msg para todo vizinho em N(i)
```

A computação no nó termina quando o nó recebe a propagação final de todos os seus vizinhos (laço da linha 9). Ao receber uma mensagem ($ack(x)$), x mensagens são confirmadas (linhas 14 e 15). No recebimento de uma mensagem de propagação final ($term\_msg$) por um nó $i \neq 0$, se o nó ainda não tiver propagado a terminação, a mesma é propagada para seus vizinhos (linhas 16-19). Relembre que o algoritmo *Dijkstra-Scholten*, que serve como base para este algoritmo, mantém uma estrutura em árvore para a detecção da terminação global (ver Capítulo 2).

No recebimento de uma mensagem do substrato ou de inicialização, se o nó não está na árvore e está ativo ele se junta à árvore (linhas 22-24). Se o nó está inativo, uma mensagem *ack* é devolvida imediatamente (linhas 25 e 26). Se o nó se tornou inativo e ainda possui *acks* pendentes para um nó que não é seu nó pai, ele envia as confirmações para esses nós (linhas 60-63). Se o nó recebeu todas as suas confirmações e está inativo, ele se desconecta da árvore enviando um *ack* para seu pai na árvore (linhas 66-69). Nesse caso se o nó é o nó 0, a terminação foi detectada e ele inicia a propagação da terminação (linhas 70 e 71).

3.4.3 Ordenação de Mensagens

Estendemos, nessa seção, os conceitos relativos aos mecanismos de ordenação de mensagens descritos no Capítulo 2 para o modelo de computações dirigidas por pulsos.

Dadas duas mensagens msg_1 e msg_2 de uma computação dirigida por pulsos. A comparação entre essas mensagens se dá através dos pulsos λ_1 e λ_2 tais que $msg_1 \in MSG(\lambda_1)$ e $msg_2 \in MSG(\lambda_2)$. Dessa forma, $msg_1 \rightarrow msg_2$ se e somente se $\lambda_1 \rightarrow \lambda_2$, assim como $msg_1 \rightsquigarrow msg_2$ se e somente se $\lambda_1 \rightsquigarrow \lambda_2$.

De forma análoga ao modelo de computações dirigidas por eventos, a relação $\rightarrow$ entre pulsos é irreflexiva. Assim, se duas mensagens msg_1 e msg_2 são enviadas por um mesmo pulso, então elas são incomparáveis na relação $\rightarrow$.

Ordem FIFO

A definição formal da ordem FIFO no modelo de computações dirigidas por pulsos é semelhante à definição do capítulo anterior. Sejam dois eventos ξ_1 e ξ_2 disparados por msg_1 e msg_2, respectivamente. Podemos expressar a ordem FIFO no modelo de computações dirigidas por pulsos formalmente através da seguinte regra:

- se msg_1 e msg_2 são originadas no mesmo nó, $msg_1 \rightsquigarrow msg_2$ e $node(\xi_1) = node(\xi_2)$, então $\xi_1 \rightsquigarrow \xi_2$

Na Figura 3.2, ilustramos um cenário em que a ordem FIFO é violada no modelo de computações dirigidas por pulsos.

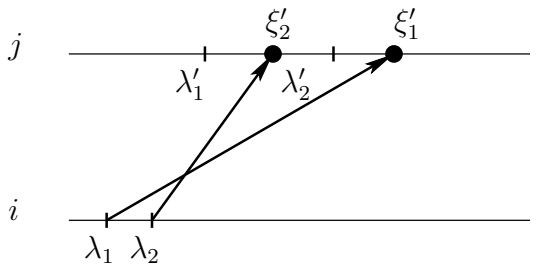

Figura 3.2: Troca de mensagens que não respeita a ordem FIFO no modelo de computações dirigidas por pulsos

No **Algoritmo 3.5**, adaptamos a ordem FIFO para o modelo de computações dirigidas por pulsos. Sua ideia é semelhante à ideia do **Algoritmo 2.5** do Capítulo 2.

Seguem as variáveis auxiliares utilizadas:

- N_i: é um vetor contendo $|N(i)|$ entradas. Dado um vizinho j de i, se temos $N_i[j] = k$, então j é o k-ésimo elemento de uma sequência crescente dos nós de $N(i)$ com base nos índices dos nós (j é o k-ésimo vizinho de i). Todas as entradas começam com valor 0;
- S_i: é um vetor com $|N(i)|$ entradas. $S_i[j]$ indica, a cada pulso λ que ocorre em i durante a computação distribuída, o número de mensagens enviadas de i para $N_i[j]$ em eventos que precedem λ na relação $\rightsquigarrow$. Todas as entradas começam com valor 0;
- s_i: é um vetor com $|N(i)|$ entradas. $s_i[j]$ indica se $S_i[j]$ já foi incremetado no evento corrente. Todas as entradas começam com valor 0;
- R_i: é um vetor com $|N(i)|$ entradas. $R_i[j]$ indica, a cada pulso λ que ocorre em i durante a computação distribuída, o número de mensagens recebidas em ordem no nó i, as quais foram enviadas pelo nó $N_i[j]$, em pulsos que precedem λ na relação $\rightsquigarrow$. Todas as entradas começam com valor 0;
- $S(msg_i)$: é chamado de sequencial da mensagem, é um valor k obtido de uma mensagem msg_i enviada por j. Indica que msg_i foi a k-ésima mensagem enviada de j para i;
- Φ_i: acumula mensagens recebidas fora de ordem;

Algoritmo 3.5: Computação feita no nó i no modelo de computações dirigidas por pulsos com ordem FIFO

Entrada: Conjunto $N(i)$ de vizinhos de i em Γ, estado local inicial para i

```
1.  ℓ_i ← 0
2.  PULSE_i(ℓ_i, MSG_i)
3.  para toda mensagem msg_i ∈ MSG_i, destinada a N_i[j], faça
4.      Anexe ℓ_i a msg_i
5.      se s_i[j] = 0 então
6.          S_i[j] ← S_i[j] + 1
7.          s_i[j] ← 1
8.      Anexe S_i[j] a msg_i
9.      Envie msg_i para N_i[j]
10. enquanto a terminação global não for conhecida por i faça
11.     se existe mensagem msg_i ∈ Φ_i, destinada a N_i[j], tal que S(msg_i) = R_i[j] + 1 então
12.         Φ_i ← Φ_i \ {msg_i}
13.         R_i[j] ← R_i[j] + 1
14.         enquanto ℓ(msg_i) > ℓ_i faça
15.             ℓ_i ← ℓ_i + 1
16.             PULSE_i(ℓ_i, MSG_i)
17.             para l de 0 até |N(i)| − 1 faça
18.                 s_i[l] ← 0
19.             para toda mensagem msg_i ∈ MSG_i, destinada a N_i[k], faça
20.                 Anexe ℓ_i a msg_i
21.                 se s_i[k] = 0 então
22.                     S_i[k] ← S_i[k] + 1
23.                     s_i[k] ← 1
24.                 Anexe S_i[k] a msg_i
25.                 Envie msg_i para N_i[k]
26.         EVENT_i(msg_i)
27.     senão se uma mensagem msg_i de um vizinho de i for recebida então
28.         Φ_i ← Φ_i ∪ {msg_i}
29.     senão
30.         ℓ_i ← ℓ_i + 1
31.         PULSE_i(ℓ_i, MSG_i)
32.         para l de 0 até |N(i)| − 1 faça
33.             s_i[l] ← 0
34.         para toda mensagem msg_i ∈ MSG_i, destinada a N_i[j], faça
35.             Anexe ℓ_i a msg_i
36.             se s_i[j] = 0 então
37.                 S_i[j] ← S_i[j] + 1
38.                 s_i[j] ← 1
39.             Anexe S_i[j] a msg_i
40.             Envie msg_i para N_i[j]
```

A idéia deste algoritmo é semelhante à ordem FIFO no modelo de computações dirigidas por eventos. Para cada mensagem enviada, verificamos se $S_i[j]$ já foi incrementado para o pulso que acabou de ocorrer (linhas 5-7, 20-23 e 36-38). Com isto, garantimos a incomparabilidade entre mensagens geradas no mesmo pulso e destinadas a um mesmo nó. Se existe alguma mensagem habilitada em Φ_i, ou seja, se o sequencial da mensagem é igual ao número de mensagens já recebidas em i mais um, então geramos pulsos até que o número de pulsos de i seja igual ao número de pulsos do nó que enviou a mensagem. Em seguida, processamos um evento com relação a essa mensagem (linhas 11-26).

Ordem Causal

Suponha que ξ_1 e ξ_2 sejam os eventos disparados por msg_1 e msg_2, respectivamente. A ordem causal no modelo de computações dirigidas por pulsos é formalmente descrita pela seguinte regra:

- se $msg_1 \rightsquigarrow msg_2$ e $node(\xi_1) = node(\xi_2)$, então $\xi_1 \rightsquigarrow \xi_2$

A figura 3.3 ilustra um cenário em que a ordem causal é violada no modelo de computações dirigidas por pulsos.

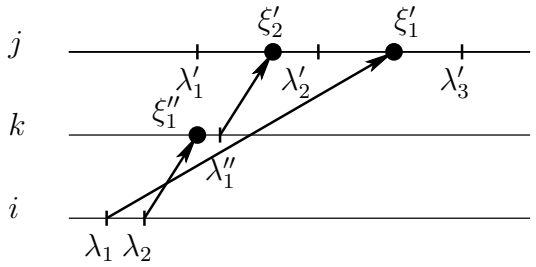

Figura 3.3: Ordem causal violada no modelo de computações dirigidas por pulsos

O **Algoritmo 3.6** descreve a ordem causal no modelo de computações dirigidas por pulsos. O mesmo incorpora ideias semelhantes ao **Algoritmo 2.6** do Capítulo 2.

As variáveis auxiliares utilizadas são:

- M_i: é uma matriz com $|N|*|N|$ entradas. $M_i[j][k]$ indica, ao conhecimento de i, a cada pulso λ que ocorre em j durante a computação distribuída, o número de mensagens enviadas pelo nó j para o nó k, em eventos que precedem λ na relação $\rightsquigarrow$. Todas as entradas começam com valor 0;
- m_i: é um vetor com $|N|$ entradas. $m_i[j]$ indica se $M_i[i][j]$ já foi incremetado no pulso corrente. Todas as entradas começam com valor 0;
- Φ_i: acumula mensagens recebidas fora de ordem.
- W_i é uma matriz com $|N|*|N|$ entradas. É extraída de uma mensagem e representa seu histórico causal.

Algoritmo 3.6: Computação feita no nó i no modelo de computações dirigidas por pulsos com ordem causal

Entrada: Conjunto $N(i)$ de vizinhos de i em Γ, estado local inicial para i

1. $\ell_i \leftarrow 0$
2. $PULSE_i(\ell_i, MSG_i)$
3. **para toda** mensagem $msg_i \in MSG_i$, destinada a j, **faça**
4. Anexe ℓ_i a msg_i
5. **se** $m_i[j] = 0$ **então**
6. $M_i[i][j] \leftarrow M_i[i][j] + 1$
7. $m_i[j] \leftarrow 1$
8. Anexe M_i a msg_i
9. Envie msg_i para j
10. **enquanto** a terminação global não for conhecida por i **faça**
11. **se** existe mensagem $msg_i \in \Phi_i$, destinada a j, com matriz W_j, tal que $W_j[j][i] = M_i[j][i] + 1$ **então**
12. **se** $\forall k \neq j : M_i[k][i] \geq W_j[k][i]$ **então**
13. $\Phi_i \leftarrow \Phi_i \backslash \{msg_i\}$
14. $M_i \leftarrow max(M_i, W_j)$
15. **enquanto** $\ell(msg_i) > \ell_i$ **faça**
16. $\ell_i \leftarrow \ell_i + 1$
17. $PULSE_i(\ell_i, MSG_i)$
18. **para** l de 0 até $|N| - 1$ **faça**
19. $m_i[l] \leftarrow 0$
20. **para toda** mensagem $msg_i \in MSG_i$, destinada a k, **faça**
21. Anexe ℓ_i a msg_i
22. **se** $m_i[k] = 0$ **então**
23. $M_i[i][k] \leftarrow M_i[i][k] + 1$
24. $m_i[k] \leftarrow 1$
25. Anexe M_i a msg_i
26. Envie msg_i para k
27. $EVENT_i(msg_i)$
28. **senão se** uma mensagem msg_i de um vizinho de i for recebida **então**
29. $\Phi_i \leftarrow \Phi_i \cup \{msg_i\}$
30. **senão**
31. $\ell_i \leftarrow \ell_i + 1$
32. $PULSE_i(\ell_i, MSG_i)$
33. **para** l de 0 até $|N| - 1$ **faça**
34. $m_i[l] \leftarrow 0$
35. **para toda** mensagem $msg_i \in MSG_i$, destinada a j, **faça**
36. Anexe ℓ_i a msg_i
37. **se** $m_i[j] = 0$ **então**
38. $M_i[i][j] \leftarrow M_i[i][j] + 1$
39. $m_i[j] \leftarrow 1$
40. Anexe M_i a msg_i
41. Envie msg_i para j

Primeiramente, enviamos as mensagens do primeiro pulso, atentando para o fato de não incrementarmos $M_i[i][j]$ duas vezes no mesmo pulso (linhas 6-8). Em seguida, verificamos se alguma mensagem $msg_i \in \Phi_i$ está habilitada (linhas 11 e 12). Se msg_i está habilitada, a informação na matriz M_i é atualizada com a matriz W_j extraída de msg_i (linha 14). Enquanto o número de pulsos executados em i for menor que o número de pulsos do nó que enviou msg_i, executamos uma sequência de pulsos, atentando para o fato de não incrementarmos $M_i[i][k]$ duas vezes no mesmo pulso (linhas 15-26). Por fim, caso não exista mensagem a ser recebida, executamos um pulso, também verificando se já incrementamos $M_i[i][j]$ no pulso corrente (linhas 36-40).

Capítulo 4

Interface de Programação

Neste capítulo, descrevemos uma *interface* de programação voltada para o uso em aplicações em Otimização Combinatória. Esta *interface* deve ser eficiente, fornecer assincronismo e operações coletivas. Para atender a esses requisitos, propomos uma *interface* que é extensão da biblioteca MPI [18] e é baseada nos dois modelos de computação distribuída descritos anteriormente.

4.1 Preliminares

Esta *interface* é voltada essencialmente para programas que tenham suas execuções governadas por um dos modelos assíncronos: modelo de computações dirigidas por eventos ou modelo de computações dirigidas por pulsos. A *interface* é dividida em duas *subinterfaces*, uma referente ao modelo de computações dirigidas por eventos e uma referente ao modelo de computações dirigidas por pulsos. Cada uma é composta por funções que estendem a MPI, incluindo as funcionalidades relacionadas ao referido modelo. Dessa forma, o primeiro pressuposto é que uma implementação da interface deve ser usada com o MPI. Uma observação importante é que cada *subinterface* sendo uma extensão da MPI, as funções originais da MPI podem ser usadas diretamente. Todavia, caso isso aconteça, as mensagens e computações geradas não são capturadas pelo modelo adotado.

A base para as funções de operações coletivas definidas neste capítulo são funções coletivas já existentes na biblioteca MPI. Algumas adaptações para os modelos assíncronos são necessárias quando as funções coletivas da biblioteca MPI envolvem alguma forma de sincronismo. Isto ocorre, por exemplo, na função `MPI_Bcast`. Ela utiliza um mecanismo bloqueante (síncrono) para a difusão de informações [19]. Buscando fornecer operações de difusão implementadas de maneira totalmente assíncrona, criamos as funções `MPI_Event_Bcast_feedback`, `MPI_Event_Bcast`, `MPI_Pulse_Bcast_feedback` e `MPI_Pulse_Bcast` para a difusão de informações utilizando o mecanismo de propagação, com ou sem realimentação, descrito nos Capítulos 2 e 3.

As funções de operações coletivas, além de tomar os modelos de eventos ou pulsos como referência (o que inclui, entre outras características, o fato de somente utilizar os canais de comunicação representados no grafo que descreve o sistema distribuído), são assíncronas. Esse assincronismo consiste em não haver bloqueio nos nós durante suas execuções nem imposição de ordem de recepção de mensagens. Para atingir esse objetivo, a realização de uma operação coletiva não exige que todos os nós envolvidos façam chamadas a uma função específica. Ao contrário,

basta que um dos nós faça uma chamada à função localmente, e os demais nós são integrados à execução da operação através de trocas de mensagens deflagradas pela chamada de função original.

Tipos de dados e topologias são definidos usando as funções padrão do MPI. Os parâmetros dos modelos são configurados com funções específicas de cada modelo e estabelecem propriedades específicas. Quando a implementação de propriedades específicas de um modelo requer adicionar algum propósito especial a uma aplicação no modelo, isso é feito através do uso de funções de empacotamento do MPI.

Quando a implementação de certas propriedades de um modelo requer mensagens específicas, então um comunicador específico é criado e usado somente para mensagens exclusivas do modelo. A aplicação deve definir um comunicador referente às mensagens do substrato e informá-lo ao modelo. Chamadas diretas de funções de comunicação do MPI (em vez de funções de comunicação definidas nesta *interface*) utilizando o comunicador do substrato podem interferir no comportamento das operações coletivas de uma computação assíncrona, não garantindo o perfeito funcionamento daquelas. Uma observação importante é que as funções de comunicação do modelo de computações dirigidas por eventos são ativadas somente dentro de execuções de funções de evento e evento espontâneo. Da mesma maneira, as funções de comunicação do modelo de computações dirigidas por pulsos são ativadas somente dentro de execuções de funções pulso e evento. Uma chamada a uma função de comunicação de um modelo fora dessas funções gera um erro.

Cada *subinterface* está divida em grupos de funções. Por exemplo, no modelo de computações dirigidas por eventos, temos rotinas de configuração de parâmetros, rotinas que podem ser chamadas dentro de funções de evento e evento espontâneo e rotinas de execução e difusão. Na definição de cada subseção correspondente, é feita uma descrição mais detalhada de cada grupo.

No final da definição de cada *subinterface*, é mostrado um exemplo de aplicação genérica utilizando algumas das funções da *subinterface* correspondente. Estes exemplos aparecem em detalhes no Capítulo 5.

4.2 A Biblioteca MPI

Os nomes de todas as entidades MPI (rotinas, constantes, tipos etc.) para a linguagem de programação C iniciam-se com `MPI_`. Os nomes das funções seguem o padrão `MPI_Xxxxx` e os nomes das constantes são escritos em letra maiúscula. Buscando seguir os mesmos princípios do MPI, nossa *interface* também segue esse padrão de nomenclatura.

Algumas das funções que propomos na *interface* correspondem a operações locais a um nó, como envio ou recepção de mensagem. Nesses casos, mesmo havendo funções similares na biblioteca MPI, a especificação da referida operação sofre alterações, pois chamadas sucessivas dessas funções podem estar associadas de alguma forma. Essa possível associação não pode ser capturada entre chamadas sucessivas das funções originais do MPI, pois as mesmas atuam isoladamente. Essa associação é presente, por exemplo, no envio de diferentes mensagens pertencentes a um mesmo evento ou a um mesmo pulso. Nesses casos, novas funções são definidas.

Grupos e comunicadores são dois dos principais conceitos da biblioteca MPI. Através de um conjunto de rotinas, podemos definir novos grupos de processos como subconjuntos de grupos existentes. Podemos também criar um comunicador e associá-lo a um grupo. Um comunicador pode ser usado para permitir a comunicação entre os processos de um grupo, a qual pode ser feita

através de rotinas de comunicação ponto-a-ponto ou rotinas de comunicação coletiva.

Quando criamos um comunicador para um grupo de processos, é criada uma estrutura de comunicação entre esses processos em forma de um grafo completo (quaisquer dois processos podem trocar mensagens). Entretanto, podemos definir uma topologia virtual para o caso em que não se deseja que todos os processos possam se comunicar diretamente. Dentre as diversas topologias virtuais existentes no MPI, ressaltamos a topologia em grafo. Nela, os processos MPI e os canais de comunicação representam, respectivamente, os vértices e as arestas de um grafo. Na implementação da *interface*, uilizamos uma topologia virtual em grafo para definir virtualmente a rede representada pelo grafo Γ passado como entrada para os modelos.

A biblioteca MPI fornece funções de comunicação ponto-a-ponto síncronas e assíncronas. Nas versões síncronas, há um protocolo de tempo que garante que uma operação específica inicie após o recebimento de uma indicação que outra operação anterior foi completada. Existem também as funções de envio e recepção assíncronas, as quais não fazem uso de chamadas não-bloqueantes. Observe que essa noção de assincronismo difere do conceito de assincronismo dos modelos, uma vez que nos modelos essa propriedade está ligada à possibilidade das computações não estarem restritas a seguir uma única ordem de execução. Observe também que uma computação pode ser assíncrona, mesmo usando funções síncronas. Por enquanto, em nossa *interface*, somente comunicações síncronas foram implementadas.

4.3 Modelo de Computação Distribuída Dirigida por Eventos

4.3.1 Rotinas de Configuração de Parâmetros

Correspondem a rotinas que permitem a configuração dos parâmetros do modelo de computações dirigidas por eventos. De acordo com o **Algoritmo 2.1** do Capítulo 2, o qual descreve um algoritmo genérico para um programa que utiliza o modelo de computações dirigidas por eventos, a aplicação deve fornecer ao modelo informações de configuração necessárias às computações que seguem esse modelo. Dentre elas, estão as configurações sobre operações coletivas, funções de evento e funções de evento espontâneo.

`MPI_Event_set_model`

Informa ao modelo as funções de evento e evento espontâneo para a computação que segue. Esta rotina associará as funções de evento e evento espontâneo à computação executada após sua chamada. Ao longo da computação distribuída, estas funções podem ser alteradas.

SINTAXE

```
#include <ed_mpi.h>
int MPI_Event_set_model(int(*event)(MPI_Status *),
  int(*spontaneous_event)(MPI_Status *))
```

PARÂMETROS DE ENTRADA

event

Função de evento responsável pela execução de uma computação local associada a um nó no modelo de computações dirigidas por eventos (função).

spontaneous_event

Função de evento responsável por iniciar uma computação no modelo de computações dirigidas por eventos. Ocorre somente no nó 0 e não necessita a recepção de mensagem (função).

PARÂMETROS DE SAÍDA

IERROR

Status do erro (inteiro).

`MPI_Event_set_config_param`

Configura parâmetros do modelo de computações dirigidas por eventos. Os possíveis parâmetros a ser configurados são:

ED_TERMINATION

Define se o modelo de computações dirigidas por eventos detectará a terminação global. Os valores possíveis são: ED_TERMINATION_OFF (a aplicação notificará o modelo de computações dirigidas por eventos quando a computação distribuída terminar) e ED_TERMINATION_MODEL (o modelo detectará a terminação global).

ED_MSG_ORDER

Define o mecanismo de ordenação de mensagens imposto às mensagens trocadas na rede. Os possíveis valores são: ED_MSG_ORDER_OFF (sem ordenação de mensagens), ED_MSG_ORDER_FIFO e ED_MSG_ORDER_CAUSAL.

SINTAXE

```
#include <ed_mpi.h>
int MPI_Event_set_config_param(int param, int value)
```

PARÂMETROS DE ENTRADA

param

Parâmetro a ser configurado (inteiro não-negativo).

value

Valor a ser atribuído ao parâmetro (inteiro não-negativo).

PARÂMETROS DE SAÍDA

IERROR

Status do erro (inteiro).

4.3.2 Rotinas que podem ser chamadas dentro de funções de Evento e Evento Espontâneo

Considerando que as funções de evento e evento espontâneo são responsáveis pelas computações locais nos nós neste modelo, as rotinas que podem ser chamadas dentro destas funções são as rotinas da *interface* que atuam diretamente no substrato. Dessa forma, caso essas rotinas sejam chamadas fora dessas funções, um erro é gerado.

Uma observação importante pode ser feita com relação às rotinas `MPI_Event_Send` e `MPI_Event_Isend`. Elas têm função semelhante às suas versões originais do MPI. Todavia, caso uma mensagem seja enviada utilizando as versões originais do MPI, esta mensagem não será levada em consideração pelo modelo e nem pelas operações coletivas assíncronas.

`MPI_Event_neighbors_test`
Testa se um nó destino para uma mensagem é vizinho do nó corrente na topologia virtual. As operações coletivas são garantidas somente para os processos da topologia virtual. Portanto, um nó pode enviar ou receber mensagens de nós vizinhos.
As operações coletivas são garantidas somente para os processos da topologia virtual. Portanto, um nó pode enviar ou receber mensagens de nós vizinhos.
SINTAXE
`#include <ed_mpi.h>` `int MPI_Event_neighbors_test(int dest)`
PARÂMETROS DE ENTRADA
dest *Rank* do nó de destino (inteiro não-negativo).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

`MPI_Event_Send`
Realiza um envio modo padrão bloqueante durante a execução de um evento. Para uma explicação mais detalhada da semântica do *Send* modo padrão, consulte o Padrão MPI-1.
SINTAXE
`#include <ed_mpi.h>` `int MPI_Event_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag)`
PARÂMETROS DE ENTRADA
buf Endereço inicial do *buffer* de envio (escolha).
count Número de elementos a enviar (inteiro não-negativo).
datatype Tipo de dado de cada elemento do *buffer* de envio (manipulador).
dest *Rank* do nó de destino (inteiro).
tag *Tag* da mensagem (inteiro).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

`MPI_Event_Isend`
Realiza um envio modo padrão não-bloqueante durante a execução de um evento. Esta rotina não bloqueará a execução do programa. Tendo em vista se verificar se a mensagem já foi recebida pelo nó de destino, as propriedades do pedido de comunicação devem ser utilizadas. Para uma explicação mais detalhada da semântica do *Isend* modo padrão, consulte o Padrão MPI-1.
SINTAXE
`#include <ed_mpi.h>` `int MPI_Event_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Request *request)`
PARÂMETROS DE ENTRADA
buf Endereço inicial do *buffer* de envio (escolha).
count Número de elementos a enviar (inteiro não-negativo).
datatype Tipo de dado de cada elemento do *buffer* de envio (manipulador).
dest *Rank* do nó de destino (inteiro).
tag *Tag* da mensagem (inteiro).
request Pedido de comunicação (manipulador).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

4.3.3 Rotinas que podem ser chamadas dentro de funções de Evento

Representam rotinas quem podem ser chamadas exclusivamente dentro de funções de evento. De forma análoga à seção anterior, se `MPI_Event_Recv` for chamada fora de uma função de evento, será gerado um erro. Mais ainda, se `MPI_Recv` for chamada em vez de `MPI_Event_Recv`, a mensagem recebida não será considerada pelo modelo e nem pelas operações coletivas.

`MPI_Event_Recv`
Executa uma recepção modo padrão bloqueante durante a execução de um evento. Esta rotina bloqueará a execução até a mensagem ser recebiba pelo nó de destino. Para uma explicação mais detalhada da semântica do *Recv* modo padrão, consulte o Padrão MPI-1.
SINTAXE
`#include <ed_mpi.h>` `int MPI_Event_Recv(void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
PARÂMETROS DE ENTRADA
buf Endereço inicial do *buffer* de recepção (escolha).
count Número de elementos a receber (inteiro não-negativo).
datatype Tipo de dado de cada elemento do *buffer* de recepção (manipulador).
status Objeto status (Status).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

4.3.4 Rotinas de Difusão e Execução

Permitem difundir informações e iniciar uma execução de uma aplicação utilizando o modelo de computações dirigidas por eventos. `MPI_Event_Bcast_feedback` e `MPI_Event_Bcast` utilizam o mecanismo de propagação descrito no Capítulo 2. A diferença entre as duas reside no fato de a primeira utilizar também a realimentação. Os parâmetros de entrada para `MPI_Event_run` são os mesmos de `MPI_Event_Bcast_feedback` e `MPI_Event_Bcast`. O primeiro parâmetro de `MPI_Event_run`, `buffer`, deve ser informado somente pelo nó 0. `buffer` deve conter o grafo Γ que descreve a rede e os estados iniciais de todos os nós. Estas informações são, então, difundidas utilizando o algoritmo para o registro dos locais iniciais do Capítulo 2.

`MPI_Event_Bcast_feedback`

Executa a difusão de dados de acordo o mecanismo de propagação com realimentação definido para o modelo de computações dirigidas por eventos. A função MPI_Bcast utiliza um mecanismo bloqueante (síncrono) para difundir a mensagem. Diferentemente, MPI_Event_Bcast_feedback utiliza o mecanismo assíncrono de propagação com realimentação descrito nas operações coletivas do modelo de computações dirigidas por eventos.

SINTAXE

```
#include <ed_mpi.h>
int MPI_Event_Bcast_feedback(void *buffer, int count, MPI_Datatype datatype,
  int root, MPI_Comm comm)
```

PARÂMETROS DE ENTRADA

buffer

Endereço inicial do *buffer* (escolha).

count

Número de entradas do *buffer* (inteiro).

datatype

Tipo de dado do *buffer* (manipulador).

root

Rank do nó raiz da difusão (inteiro).

comm

Comunicador (manipulador).

PARÂMETROS DE SAÍDA

IERROR

Status do erro (inteiro).

`MPI_Event_Bcast`
Executa a difusão de dados de acordo o mecanismo de propagação (sem realimentação) definido para o modelo de computações dirigidas por eventos. A função MPI_Bcast utiliza um mecanismo bloqueante (síncrono) para difundir a mensagem. Diferentemente, MPI_Event_Bcast utiliza o mecanismo assíncrono de propagação descrito nas operações coletivas do modelo de computações dirigidas por eventos.
SINTAXE
`#include <ed_mpi.h>` `int MPI_Event_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
PARÂMETROS DE ENTRADA
buffer Endereço inicial do *buffer* (escolha).
count Número de entradas do *buffer* (inteiro).
datatype Tipo de dado do *buffer* (manipulador).
root *Rank* do nó raiz da difusão (inteiro).
comm Comunicador (manipulador).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

MPI_Event_run
Executa uma aplicação baseada no modelo de computações dirigidas por eventos. Os parâmetros de entrada devem ser fornecidos pelo nó 0 e são difundidos para os outros nós de acordo com o algoritmo para o registro dos estados locais iniciais. O parâmetro *buffer* deve conter o grafo Γ e os estados iniciais de todos os nós. Esta rotina gera uma sequência de eventos até que a aplicação se torne globalmente terminada.
SINTAXE
`#include <ed_mpi.h>` `int MPI_Event_run(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Status * comp_status)`
PARÂMETROS DE ENTRADA
buffer Endereço inicial do *buffer* (escolha).
count Número de entradas do *buffer* (inteiro).
datatype Tipo de dado do *buffer* (manipulador).
root *Rank* do nó raiz da difusão (inteiro).
comm Comunicador (manipulador).
comp_status Status da computação corrente (Status).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

4.3.5 Uma Aplicação Genérica

Para a criação de qualquer aplicação MPI, algumas rotinas básicas, como `MPI_Init`, `MPI_Comm_rank`, `MPI_Comm_size` e `MPI_Finalize` [18] são necessárias. Todavia, por simplicidade, as mesmas são omitidas da aplicação que segue.

Primeiramente, é necessária a configuração dos parâmetros relativos ao modelo de computações dirigidas por eventos. São eles a detecção da terminação global, o mecanismo de ordenação de mensagens utilizado e as funções de evento e evento espontâneo (linhas 1-5). Nessa aplicação, a detecção da terminação global ficará a cargo do modelo e a ordem de mensagens escolhida foi a causal. Devido ao mecanismo da terminação descrito no Capítulo 2, as funções de evento e evento espontâneo devem retornar 0 se o nó está ativo após o evento anterior. Dentro dessas funções devem ser inseridas as rotinas da *subinterface* que atua diretamente no substrato, como rotinas de envio e recepção (linhas 9-19).

Para o registro dos estados iniciais, cada nó deve criar um *buffer* capaz de armazenar Γ e os estados iniciais de todos os nós (linha 6). Esse *buffer* é preenchido inicialmente pelo nó 0. Ao ser feita a chamada à rotina de execução, os Γ e os estados iniciais são difundidos (linha 8).

Algoritmo 4.1: Aplicação genérica utilizando a *subinterface* referente ao modelo de computações dirigidas por eventos no nó i

Entrada (quando $i = 0$): grafo $\Gamma = (N, L)$ e os estados iniciais dos nós
Configuração:
1. **inclua** <ed_mpi.h>
2. **se** $i = 0$ **então**
3. MPI_Event_set_config_param (ED_TERMINATION, ED_TERMINATION_MODEL)
4. MPI_Event_set_config_param (ED_MSG_ORDER, ED_MSG_ORDER_FIFO)
5. MPI_Event_set_model (event$_i$, spontaneous_event$_i$)

Difusão e Execução:
6. Construa um *buffer* graph_buf capaz de armazenar Γ e os estados iniciais dos nós
7. **se** $i = 0$ **então**
8. MPI_Event_run(graph_buf, size_of_graph_buf, MPI_CHAR, ...)

Definição das funções de evento e evento espontâneo:
Função 4..1: event$_i$(MPI_Status *Status)
...
9. MPI_Event_Recv(...)
...
10. MPI_Event_ISend(...)
...
11. **se** o nó está ocioso **então**
12. **retorne 1**
13. **senão**
14. **retorne 0**

Função 4.2: spontaneous_event$_i$(MPI_Status *Status)
...
15. MPI_Event_Isend(...)
...
16. **se** o nó está ocioso **então**
17. **retorne 1**
18. **senão**
19. **retorne 0**

4.4 Modelo de Computação Distribuída Dirigida por Pulsos

Descrevemos adiante a *subinterface* com relação ao modelo de computações dirigidas por pulsos. Muitas das considerações são semelhantes às feitas com respeito ao modelo de computações dirigidas por eventos e por esse motivo serão omitidas.

4.4.1 Rotinas de Configuração de Parâmetros

Correspondem a rotinas que permitem a configuração dos parâmetros do modelo de computações dirigidas por pulsos. O parâmetro `PD_PULSE_GENERATION` define se a aplicação seguirá o mecanismo de geração pulsos contido no **Algoritmo 3.2** do Capítulo 3. Deve ser atribuida a constante `PD_PULSE_GENERATION_AUTOMATIC` caso o mecanismo de pulsos seja o do algoritmo citado. Deve ser atribuída a constante `PD_PULSE_GENERATION_OFF` se a aplicação guiará a geração de pulsos. Para tanto, a aplicação pode fazer uso da função `MPI_Pulse_advance_pulse_counter` para gerar uma sequência de pulsos até um pulso alvo. `MPI_Pulse_postpone_msg` pode ser utilizada para atrasar a recepção de uma mensagem, a qual será entregue após um certo pulso acontecer. Estas duas funções serão detalhadas mais adiante.

`MPI_Pulse_set_model`

Informa ao modelo as funções de pulso e evento para a computação que segue. Essa rotina associará as funções de pulso e evento à computação executada após sua chamada. Ao longo da computação distribuída, estas funções podem ser alteradas.

SINTAXE

```
#include <pd_mpi.h>
int MPI_Pulse_set_model(int(*pulse)(long long, MPI_Status *),
  int(*event)(MPI_Status *))
```

PARÂMETROS DE ENTRADA

pulse

Função de pulso responsável pela execução de uma computação local associada a um nó no modelo de computações dirigidas por pulsos (função).

event

Função de evento responsável pelo tratamento de mensagens (função).

PARÂMETROS DE SAÍDA

IERROR

Status do erro (inteiro).

MPI_Pulse_set_config_param
Configura parâmetros do modelo de computações dirigidas por pulsos.Os possíveis parâmetros a ser configurados são:
PD_TERMINATION Define se o modelo de computações dirigidas por pulsos detectará a terminação global. Os valores possíveis são: PD_TERMINATION_OFF (a aplicação notificará o modelo de computações dirigidas por pulsos quando a computação distribuída terminar) e PD_TERMINATION_MODEL (o modelo detectará a terminação global).
PD_MSG_ORDER Define o mecanismo de ordenação de mensagens imposto às mensagens trocadas na rede. Os possíveis valores são: PD_MSG_ORDER_OFF (sem ordenação de mensagens), PD_MSG_ORDER_FIFO e PD_MSG_ORDER_CAUSAL.
PD_PULSE_GENERATION Define se o mecanismo de geração de pulsos será controlado pela aplicação ou pelo modelo. Os possíveis valores são: PD_PULSE_GENERATION_OFF (a aplicação controlará o mecanismo de geração de pulsos utilizando as funções MPI_Pulse_advance_pulse_counter e MPI_Pulse_postpone_msg) e PD_PULSE_GENERATION_AUTOMATIC (o modelo de computações dirigidas por pulsos controlará a geração de pulsos).
SINTAXE
`#include <pd_mpi.h>` `int MPI_Pulse_set_config_param(int param, int value)`
PARÂMETROS DE ENTRADA
param Parâmetro a ser configurado (inteiro não-negativo).
value Valor a ser atribuído ao parâmetro (inteiro não-negativo).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

4.4.2 Rotinas que podem ser chamadas dentro de funções de Evento

São rotinas quem podem ser chamadas exclusivamente dentro de funções de evento do modelo de computações dirigidas por pulsos. Destinam-se ao tratamento de mensagens.

MPI_Pulse_Recv
Executa uma recepção modo padrão bloqueante durante a execução de um evento. Para uma explicação mais detalhada da semântica do *Recv* modo padrão, consulte o Padrão MPI-1.
SINTAXE
`#include <pd_mpi.h>` `int MPI_Pulse_Recv(void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
PARÂMETROS DE ENTRADA
buf Endereço inicial do *buffer* de recepção (escolha).
count Número de elementos a receber (inteiro não-negativo).
datatype Tipo de dado de cada elemento do *buffer* de recepção (manipulador).
status Objeto status (Status).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

MPI_Pulse_postpone_msg
Atrasa uma mensagem até um certo pulso ser executado. Esta rotina atrasará uma mensagem recebida em um evento. A mensagem será recebida em um evento que ocorra imediatamente após o pulso representado por *targetpulse*.
SINTAXE
`#include <pd_mpi.h>` `int MPI_Pulse_postpone_msg(long long targetpulse)`
PARÂMETROS DE ENTRADA
targetpulse Número do pulso para depois do qual a mensagem deve ser atrasada (escolha).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

4.4.3 Rotinas que podem ser chamadas dentro de funções de Pulso

Correspondem a rotinas quem podem ser chamadas dentro de funções de pulso.

`MPI_Pulse_neighbors_test`

Testa se um nó destino para uma mensagem é vizinho do nó corrente na topologia virtual. As operações coletivas são garantidas somente para os processos da topologia virtual. Portanto, um nó pode enviar ou receber mensagens de nós vizinhos.

SINTAXE

```
#include <pd_mpi.h>
int MPI_Pulse_neighbors_test(int dest)
```

PARÂMETROS DE ENTRADA

dest

Rank do nó de destino (inteiro não-negativo).

PARÂMETROS DE SAÍDA

IERROR

Status do erro (inteiro).

`MPI_Pulse_Send`

Realiza um envio modo padrão bloqueante durante a execução de um pulso. Esta rotina bloqueará a execução até a mensagem ser enviada ao nó de destino. Para uma explicação mais detalhada da semântica de um envio modo padrão, consulte o Padrão MPI-1.

SINTAXE

```
#include <pd_mpi.h>
int MPI_Pulse_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag)
```

PARÂMETROS DE ENTRADA

buf

Endereço inicial do *buffer* de envio (escolha).

count

Número de elementos a enviar (inteiro não-negativo).

datatype

Tipo de dado de cada elemento do *buffer* de envio (manipulador).

dest

Rank do nó de destino (inteiro).

tag

Tag da mensagem (inteiro).

PARÂMETROS DE SAÍDA

IERROR

Status do erro (inteiro).

MPI_Pulse_Isend
Realiza um envio modo padrão não-bloqueante durante a execução de um pulso. Esta rotina não bloqueará a execução do programa. Tendo em vista se verificar se a mensagem já foi recebida pelo nó de destino, as propriedades do pedido de comunicação devem ser utilizadas. Para uma explicação mais detalhada da semântica do *Isend* modo padrão, consulte o Padrão MPI-1.
SINTAXE
`#include <pd_mpi.h>` `int MPI_Pulse_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Request *request)`
PARÂMETROS DE ENTRADA
buf Endereço inicial do *buffer* de envio (escolha).
count Número de elementos a enviar (inteiro não-negativo).
datatype Tipo de dado de cada elemento do *buffer* de envio (manipulador).
dest *Rank* do nó de destino (inteiro).
tag *Tag* da mensagem (inteiro).
request Pedido de comunicação (manipulador).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

MPI_Pulse_advance_pulse_counter
Produz um avanço no mecanismo de geração de pulsos. Esta rotinha gerará uma sequência de pulsos que terminará quando se alcançar o pulso representado por *newpulse*.
SINTAXE
`#include <pd_mpi.h>` `int MPI_Pulse_advance_pulse_counter (long long newpulse)`
PARÂMETROS DE ENTRADA
newpulse Número do pulso para o qual o mecanismo de geração de pulsos deve avançar (escolha).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

4.4.4 Rotinas de Difusão e Execução

Permitem difundir informações e iniciar uma execução de uma aplicação utilizando o modelo de computações dirigidas por pulsos. Novamente, os parâmetros de entrada para `MPI_Pulse_run` são os mesmos de `MPI_Pulse_Bcast_feedback` e `MPI_Pulse_Bcast`. O primeiro parâmetro de `MPI_Pulse_run`, `buffer`, deve ser informado somente pelo nó 0. `buffer` deve conter o grafo Γ que descreve a rede e os estados iniciais de todos os nós. A difusão destas informações é feita conforme o algoritmo **Algoritmo 3.3** do Capítulo 3.

`MPI_Pulse_Bcast_feedback`
Executa a difusão de dados de acordo o mecanismo de propagação com realimentação definido para o modelo de computações dirigidas por pulsos. A função MPI_Bcast utiliza um mecanismo bloqueante (síncrono) para difundir a mensagem. Diferentemente, MPI_Pulse_Bcast_feedback utiliza o mecanismo assíncrono de propagação com realimentação descrito nas operações coletivas do modelo de computações dirigidas por pulsos.
SINTAXE
`#include <pd_mpi.h>` `int MPI_Pulse_Bcast_feedback(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
PARÂMETROS DE ENTRADA
buffer Endereço inicial do *buffer* (escolha).
count Número de entradas do *buffer* (inteiro).
datatype Tipo de dado do *buffer* (manipulador).
root *Rank* do nó raiz da difusão (inteiro).
comm Comunicador (manipulador).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

`MPI_Pulse_Bcast`
Executa a difusão de dados de acordo o mecanismo de propagação (sem realimentação) definido para o modelo de computações dirigidas por pulsos. A função MPI_Bcast utiliza um mecanismo bloqueante (síncrono) para difundir a mensagem. Diferentemente, MPI_Pulse_Bcast utiliza o mecanismo assíncrono de propagação descrito nas operações coletivas do modelo de computações dirigidas por pulsos.
SINTAXE
`#include <pd_mpi.h>` `int MPI_Pulse_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
PARÂMETROS DE ENTRADA
buffer Endereço inicial do *buffer* (escolha).
count Número de entradas do *buffer* (inteiro).
datatype Tipo de dado do *buffer* (manipulador).
root *Rank* do nó raiz da difusão (inteiro).
comm Comunicador (manipulador).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

MPI_Pulse_run
Executa uma aplicação baseada no modelo de computações dirigidas por pulsos. Os parâmetros de entrada devem ser fornecidos pelo nó 0 e são difundidos para os outros nós de acordo com o algoritmo para o registro dos estados locais iniciais. O parâmetro *buffer* deve conter o grafo Γ e os estados iniciais de todos os nós. Esta rotina gera uma sequência de pulsos até que a aplicação se torne globalmente terminada.
SINTAXE
`#include <pd_mpi.h>` `int MPI_Pulse_run(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Status * comp_status)`
PARÂMETROS DE ENTRADA
buffer Endereço inicial do *buffer* (escolha).
count Número de entradas do *buffer* (inteiro).
datatype Tipo de dado do *buffer* (manipulador).
root *Rank* do nó raiz da difusão (inteiro).
comm Comunicador (manipulador).
comp_status Status da computação corrente (Status).
PARÂMETROS DE SAÍDA
IERROR Status do erro (inteiro).

4.4.5 Uma Aplicação Genérica

O algoritmo que segue representa uma aplicação genérica que utiliza a *subinterface* relativa ao modelo de computações dirigidas por pulsos. As idéias são semelhantes à sua versão no modelo de computações dirigidas por eventos. A principal diferença reside no parâmetro `PD_PULSE_GENERATION`, o qual determina se o modelo ou a aplicação guiará a geração de pulsos (linha 5). Nessa aplicação, o mecanismo de geração de pulsos fica a cargo do modelo.

Algoritmo 4.2: Aplicação genérica utilizando a *subinterface* referente ao modelo de computações dirigidas por pulsos no nó i

Entrada (quando $i = 0$): grafo $\Gamma = (N, L)$ e os estados iniciais dos nós
Configuração:

```
1. inclua <pd_mpi.h>
2. se i = 0 então
3.       MPI_Pulse_set_config_param (PD_TERMINATION,
         PD_TERMINATION_MODEL)
4.       MPI_Pulse_set_config_param (PD_MSG_ORDER, PD_MSG_ORDER_CAUSAL)
5.       MPI_Pulse_set_config_param (PD_PULSE_GENERATION,
         PD_PULSE_GENERATION_AUTOMATIC)
6.       MPI_Pulse_set_model (pulse_i, event_i)
```

Difusão e Execução:

```
7. Construa um buffer graph_buf capaz de armazenar Γ e os estados iniciais dos nós
8. se i = 0 então
9.       MPI_Pulse_run(graph_buf, size_of_graph_buf, MPI_CHAR, ...)
```

Definição das funções de pulso e evento:

Função 4.3: pulse$_i$(long long pulsecount, MPI_Status *Status)

```
...
10. MPI_Pulse_ISend(...)
...
11. se o nó está ocioso então
12.       retorne 1
13. senão
14.       retorne 0
```

Função 4.4: event$_i$(MPI_Status *Status)

```
...
15. MPI_Pulse_Recv(...)
...
```

Capítulo 5

Aplicações

Tendo em vista validar o uso dos modelos em questão por aplicações em Otimização Combinatória, implementamos a aplicação descrita neste capítulo utilizando uma implementação da *interface* do Capítulo 4. Nesta aplicação, propomos uma abordagem distribuída para o Problema do Conjunto Independente Máximo.

5.1 Algoritmo *Branch-and-Bound* Sequencial para o Problema do CIM

Retomemos $G = (V, E)$ como um grafo arbitrário no qual $n = |V|$, $m = |E|$ e que G é não-direcionado, simples, não-vazio e conexo.

Tomamos por base o algoritmo proposto por *Tomita* e *Kameda* em [23], que possui complexidade $O(3^{\frac{n}{3}})$, para encontrar o conjunto independente máximo em um grafo. Passamos a uma descrição desse algoritmo como preâmbulo para as versões distribuídas.

Nesse algoritmo, cada subproblema é representado por uma lista de *vértices candidatos*, assim chamados por definirem o subgrafo de G no qual deve-se encontrar um conjunto independente máximo associado ao referido subproblema. Naturalmente, o problema inicial é representado pela lista contendo todos os vértices do grafo. Além disso, é estabelecida uma ordem geral para os vértices que depende do grau de cada vértice. Essa ordem geral e outros parâmetros locais a cada subproblema são usados de forma a que os vértices sejam dispostos na lista em uma ordem que permita a determinação da estimativa do subproblema, conforme detalhado adiante.

Tomemos um subproblema definido por uma lista R, de tamanho r. Os índices de 0 a $r-1$ indicam as posições na lista, ou seja, $R[i]$, com $i \in \{0, 1, \ldots, r-1\}$, denota o i-ésimo vértice na ordem dos vértices estabelecida pela lista R. A geração do i-ésimo subproblema referente a R se faz pela seleção do vértice $R[i]$ (denominado *pivô*) e pela criação de uma nova lista (representando o novo subproblema), formada por todos os vértices $R[j]$ tais que $0 \leq j < i$ e $R[i]R[j] \notin E$ (não existe aresta entre $R[i]$ e $R[j]$).

O processo de enumeração decorrente da operação acima gera, eventualmente, subproblemas "vazios", seja porque $i = 0$, seja porque não há vértices candidatos em R não adjacentes a $R[i]$. Conforme indicado na Figura 5.1, a ocorrência desse fato identifica um conjunto independente de G

formado pelos pivôs da sequência de gerações recursivas de subproblemas desde o problema original. Esses pivôs são guardados em uma pilha. Por conveniência, a partir deste ponto, utilizamos, em vez de i, a notação piv para nos referirmos ao índice do pivô corrente.

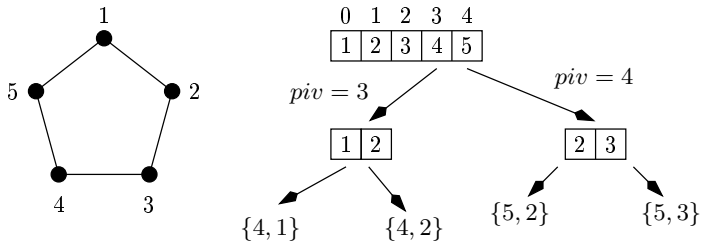

Figura 5.1: Sequências recursivas de geração de subproblemas para um grafo. Nessas sequências, três conjuntos independentes são enumerados

Conforme mencionado na Introdução, a cada subproblema R gerado durante a enumeração é associada uma estimativa, na forma de limite superior para o tamanho dos conjuntos independentes que podem ser enumerados a partir de R. A estimativa utilizada é derivada de uma cobertura por cliques do subgrafo correspondente a R. Uma *clique* é um subconjunto de vértices mutuamente adjacentes, enquanto uma *cobertura por cliques* $\mathcal{C}$ é um conjunto (ou família) de cliques que cobrem todos os vértices do subgrafo. Uma observação direta é que todo conjunto independente possui, no máximo, um vértice de cada clique de uma cobertura por cliques, pois se existisse dois vértices de um conjunto independente pertencentes a uma mesma clique, existiria uma aresta entre eles, o que violaria a condição do conjunto independente. Dessa forma, o subgrafo correspondente a R não possui conjuntos independentes maiores que $|\mathcal{C}|$. Uma observação imediata é que, caso um conjunto independente de tamanho pelo menos $|\mathcal{C}|$ seja enumerado antes de R, esse subproblema pode ser descartado da enumeração.

A geração de um subproblema R é seguida do cálculo de uma cobertura por cliques $\mathcal{C}$ do subgrafo correspondente. Nessa cobertura, as cliques são numeradas de 1 a $|\mathcal{C}|$. Os vértices são então ordenados na lista em ordem crescente do número da clique a que pertencem. Voltando à Figura 5.1, suponhamos que o grafo mostrado seja, na realidade, subgrafo de um grafo maior, subgrafo esse associado ao subproblema representado pela lista ocupando a raiz da árvore de subproblemas mostrada na figura. Um exemplo de cobertura por cliques desse subgrafo é composto pelas cliques $\{1, 2\}$, $\{3, 4\}$ e $\{5\}$, indicando não ser possível a existência de um conjunto independente no subgrafo de tamanho maior que 3. Esse é a conclusão usada quando $piv = 4$. Já para o caso $piv = 3$, verifica-se não ser possível haver um conjunto independente no subgrafo definido pelos vértices 1, 2, 3 e 4 com tamanho maior que 2 (note que a cobertura a ser considerada é formada pelas cliques $\{1, 2\}$ e $\{3, 4\}$.

O algoritmo *branch-and-bound* original utiliza as variáveis R, a qual corresponde à lista corrente, Q, que corresponde a um conjunto independente encontrado antes R, e $Qmax$, que consiste no tamanho do maior conjunto independente encontrado até o momento. O algoritmo executa o procedimento recursivo a seguir recebendo como entrada uma lista de vértices candidatos R e um pivô piv. Na primeira chamada do procedimento recursivo, R é uma lista formada

pelos vértices de G, ou seja, $R = V$. Inicialmente, $piv = |R| - 1$. Em [23], é mostrada uma forma de calcular a cobertura por cliques associada a uma lista de vértices. Seja $\mathcal{C}$ a cobertura por cliques calculada para $R[0, \ldots, piv]$, que corresponde à sublista de R que começa em $R[0]$ e termina em $R[piv]$. Primeiramente, observe que o conjunto formado pelos pivôs de todas as listas de um caminho entre a raiz (a lista que representa V) e uma lista folha na árvore de ramificações corresponde a um conjunto independente. Por exemplo, na figura, $\{5, 3\}$ é um conjunto independente. Mantemos em Q o conjunto independente formado pelos pivôs das listas do caminho entre a raiz e $R[0, \ldots, piv]$.

Veja que $|\mathcal{C}| + |Q|$ corresponde ao tamanho do maior conjunto independente que pode ser encontrado entre a raiz e uma folha, passando por $R[0, \ldots, piv]$. Desta maneira, se $|\mathcal{C}| + |Q| \leq Qmax$, então o conjunto independente que vai da raiz até uma folha, passando por $R[0, \ldots, piv]$, não pode melhorar a estimativa $Qmax$. Quando isso ocorre, descartamos $R[0, \ldots, piv]$ e atualizamos $Qmax$ para $Q + 1$, se $Q + 1 > Qmax$. Quando $|\mathcal{C}| + |Q| > Qmax$, criamos uma nova lista Rp a partir de $R[piv]$. Se $Rp \neq \emptyset$, passamos a analizar a lista Rp e guardamos $R[0, \ldots, piv - 1]$ (chamamos o procedimento recursivo para $R[0, \ldots, piv - 1]$). O algoritmo termina quando $R = \emptyset$. No final da execução, o tamanho do conjunto independente máximo de G é $Qmax$.

5.2 Algoritmos *Branch-and-Bound* Distribuídos Aleatórios

5.2.1 Aspectos Gerais

Os algoritmos distribuídos que passamos a descrever são adaptações da versão sequencial resumida acima tomando por base uma combinação de ideias apresentadas e analisadas em [12]. Há alguns fatores que exigem essas adaptações. O primeiro deles é o fato de o algoritmo sequencial adotado não se enquadrar diretamente em nenhum dos dois casos considerados em [12], detalhados adiante. De um lado, o algoritmo sequencial, ao realizar uma busca em profundidade na árvore de geração de subproblemas, se assemelha ao caso denominado *backtracking*. Por outro lado, o uso de podas a partir das estimativas dos subproblemas (no algoritmo serial, as coberturas por cliques representam essas estimativas) somente está presente no caso denominado *branch-and-bound*. O segundo fator que nos obriga a fazer adaptações vem das diferenças entre os modelos de computação distribuída dos capítulos anteriores e aquele adotado em [12]. Dois aspectos deste último modelo não são encontrados nos modelos computações distribuídas presentes neste trabalho: o sincronismo na execução dos diferentes nós e a conexão dos mesmos em um grafo completo.

Observando o algoritmo sequencial, verificamos que o mesmo atua aplicando sucessivas operações de *ramificação*. O processo do ramificação consiste em resolver um subproblema R (designamos um subproblema pela lista de vértices que o define) diretamente, ou dividí-lo em subproblemas $R_{r-1}, R_{r-2}, \ldots, R_0$ (r denota o tamanho de R, ver Figura 5.1) de tal forma que a solução para R está em um dos subproblemas gerados. Para cada subproblema analisado, o mesmo também pode ser ramificado recursivamente se $r > 0$. Observe que o processo gera uma árvore de ramificação com raiz no problema original e folhas constituídas pelos subproblemas para os quais $r = 0$.

Uma observação importante é que existem dois grafos em questão: o do problema CIM, $G = (V, E)$, e o que define os canais dos modelos de computação, $\Gamma = (N, L)$.

Nos algoritmos *branch-and-bound* distribuídos aleatórios descritos a seguir, o estado inicial da computação consiste do grafo G difundido entre todos os nós da rede. Nesses algoritmos, existem três tipos de passos: *ramificação*, *emparelhamento* e *doação*. Esses passos, inspirados em [12], se transformam em eventos ou pulsos nos algoritmos para os respectivos modelos. A seguir, descrevemos a função de cada um:

- Passo de ramificação: cada nó $i \in N$ mantém uma pilha Q_i de subproblemas que fazem parte da enumeração. Em um passo desse tipo, efetua-se a geração de subproblema na sequência natural da enumeração. Uma observação importante é que pelo fato de este algoritmo utilizar um mecanismo de busca em profundidade (inerente ao *backtracking*), um conjunto formado pelos pivôs de um caminho entre a lista que está na base e a lista que está no topo de Q_i é um conjunto independente.
- Passo de emparelhamento: quando a pilha de subproblemas de $i \in N$ está vazia (i está ocioso), ele envia uma mensagem de emparelhamento para pedir subproblemas a um possível nó doador. Tal nó doador é um vizinho de i escolhido ao acaso. A aleatoriedade do algoritmo deve-se a este fato.
- Passo de doação: consiste em um nó $i \in N$ doar trabalho a um nó solicitante. Através dela, i envia a metade direita da lista que está na base da pilha Q_i para o nó solicitante. Por este fato, um subproblema agora deve ser representado por $R[beg, \ldots, piv]$, em que beg e piv correspondem aos índices inicial e final da lista analizada.

Desenvolvemos adiante versões para o CIM utilizando o *branch-and-bound* distribuído aleatório nos Modelos de Computações Distribuídas Dirigidas por Eventos e Pulsos. Esses algoritmos se baseiam nos **Algoritmos 4.1** e **4.2** do capítulo anterior.

5.2.2 Modelo de Computação Distribuída Dirigida por Eventos

No algoritmo adiante, o nó 0 (da rede) é responsável por calcular a ordem geral dos vértices, a qual é propagada inicialmente aos demais nós. Além disso, o nó 0 calcula uma cobertura por cliques para a lista inicial (que representa V) e inicializa a pilha com essa lista antes de entrar nos modelo.

Todas as pilhas dos nós são vazias inicialmente, com exceção do nó 0, cuja pilha possui inicialmente o problema original V. As funções `event`$_i$ e `spontaneous_event`$_i$ devem ser informadas ao modelo. O algoritmo termina quando cada nó i se torna ocioso (quando a pilha de i, Q_i, se torna vazia) e todas as mensagens enviadas na rede foram recebidas por seus destinatários. A terminação global é detectada através do algoritmo **Algoritmo 2.3** visto no Capítulo 2. De acordo com o algoritmo da terminação global (visto no Capítulo 1), o retorno das funções `event`$_i$ e `spontaneous_event`$_i$ deve ser 1 caso o nó esteja ocioso, e 0, caso contrário.

As variáveis utilizadas adicionais utilizadas são:

- *tem_doacao_pendente*$_i$: indica se o nó i solicitou doação de subproblemas e ainda não os recebeu. Possui inicialmente o valor `false`;
- *pode_doar*$_i$: indica que o nó realizou uma ramificação e pode doar subproblemas. Possui inicialmente o valor `false`;

Para a execução do algoritmo, devemos primeiramente difundir o grafo G e a ordem inicial dos vértices. O grafo G é representado por uma matriz de adjacências que usa um *bit* por aresta. Dessa forma, devemos construir um *buffer* `graph_buf` com tamanho capaz de conter três informações: um inteiro com o número de vértices de G, a ordem geral dos vértices e a matriz de adjacências de G. A difusão de `graph_buf` e a execução da aplicação é feita através da chamada de `MPI_Event_run(graph_buf, (`$n+1$`)*sizeof(int)+size_graph, MPI_CHAR, ...)`. A variável `size_graph` denota o tamanho da matriz de adjacências, em *bytes*. Em cada nó $i \in N$, há uma variável $\texttt{has\_graph}_i$ que indica se i já possui o grafo ou não. Se $i = 0$, essa variável é inicializada com **true**. Caso contrário, o valor inicial é **false**.

Considere $i \in N$ e $j \in N(i)$ dois nós vizinhos de Γ. Os tipos de mensagens utilizadas são:

- "i deseja emparelhar-se a j": representa uma mensagem de emparelhamento, na qual i solicita a doação de subproblemas a um nó j;
- "i doa D_i para j": representa uma mensagem de doação. Através dela, i doa o conjunto de subproblemas a j. D_i pode não pode ser vazio;
- i não doa subproblemas para j": representa uma negação de doação. Caso $i \in N$ não possa doar subproblemas a j, uma mensagem desse tipo é enviada;
- "i gera evento em j": é denominada de mensagem artificial. Tem por finalidade gerar uma cadeia de eventos que garante que a computação distribuída termine. Será melhor detalhada mais adiante.

Como dito anteriormente, os passos do *branch-and-bound* se tranformam em eventos, como é descrito a seguir. Quando o nó i está ocioso, ele executa um evento de emparelhamento, gerando uma mensagem de emparelhamento, na qual solicita a doação de subproblemas a um nó j, escolhido ao acaso. Quando um nó i recebe um pedido de emparelhamento de um nó j, realiza um evento de doação, devolvendo uma mensagem de doação de subproblemas a j caso possua subproblemas para doar ou uma mensagem de negação de doação, caso contrário. Em um evento de ramificação, um nó i somente expande um subproblema de sua pilha local (não gerando mensagens). Observe que, com exceção dos eventos de ramificação, cada evento gera uma mensagem correspondente. O fato de termos um tipo de eventos que não gere mensagem pode fazer que a computação não termine, uma vez que as computações locais nesse modelo dependem da recepção de mensagens. Dessa forma, para esse tipo de evento (ramificação), devemos utilizar uma mensagem artificial que tem por finalidade somente gerar um evento em um outro nó j, que permanece fixo durante toda a computação distribuída. Através desse artifício, geramos uma cadeia causal de eventos que garante que a computação distribuída termine. Nesse caso, dizemos que i é o nó de geração de j. Tendo em vista diminuir o possível efeito da sobrecarga gerada pela utilização dessas mensagens artificiais, podemos aplicar algumas otimizações, como, por exemplo, alocar os nós i e j em um mesmo processador, caso esse processador possua dois núcleos de processamento. Esse fato será tratado em detalhes no Capítulo 6.

Algoritmo 5.1: *Branch-and-Bound* Distribuído Aleatório no nó i utilizando o Modelo de Computação Distribuída Dirigida por Eventos

Entrada (quando $i = 0$): grafo $G = (V, E)$
Inicialização:
1. **inclua** <ed_mpi.h>
2. **se** $i = 0$ **então**
3. $R_i \leftarrow V$
4. $beg_i \leftarrow 0$
5. $piv_i \leftarrow |R_i| - 1$
6. Calcule cobertura por cliques $\mathcal{C}_i$ para $R_i[beg_i, \ldots, piv_i]$
7. Ordene $R_i[beg_i, \ldots, piv_i]$
8. $Q_i \leftarrow \{\langle R_i, beg_i, piv_i, \mathcal{C}_i \rangle\}$
9. Crie *buffer* $graph\_buf_i$ como indicado
10. Armazene em $size\_graph_i$ o tamanho da matriz de adjacências de G
11. $has\_graph_i \leftarrow$ **true**
12. **senão**
13. $Q_i \leftarrow \emptyset$
14. $R_i \leftarrow \emptyset$
15. $beg_i \leftarrow 0$
16. $piv_i \leftarrow 0$
17. Crie *buffer* $graph\_buf_i$ vazio
18. $size\_graph_i \leftarrow 0$
19. $has\_graph_i \leftarrow$ **false**
20. $tem\_doacao\_pendente_i \leftarrow$ **false**
21. $pode\_doar_i \leftarrow$ **false**
22. $Qmax_i \leftarrow 0$

Configuração:
23. MPI_Event_set_config_param (ED_TERMINATION, ED_TERMINATION_MODEL)
24. MPI_Event_set_model (event$_i$, spontaneous_event$_i$)

Difusão e Execução:
25. MPI_Event_run(graph_buf, $(n+1)$*sizeof(int)+size_graph, MPI_CHAR, ...)

Definição das funções de evento e evento espontâneo:

Função 5.1: event$_i$(MPI_Status *Status)

26. **se** $has_graph = $ **false então**
27. Leia $graph\_buf$
28. $has\_graph \leftarrow$ **true**
29. MPI_Event_Recv(msg_i, ...)
30. **caso** msg_i **seja**: "j deseja emparelhar-se a i" **então**
31. **se** $Q_i \neq \emptyset$ **e** $pode\_doar_i$ **então**
32. Seja $\langle R'_i, beg'_i, piv'_i, \mathcal{C}'_i \rangle$ a base da pilha Q_i
33. $beg''_i \leftarrow \left\lfloor \frac{piv'_i}{2} \right\rfloor$
34. $piv''_i \leftarrow piv'_i$
35. $R''_i \leftarrow R'_i[beg''_i, \ldots, piv''_i]$
36. Calcule cobertura por cliques $\mathcal{C}''_i$ para $R''_i[beg''_i, \ldots, piv''_i]$
37. Ordene $R''_i[beg''_i, \ldots, piv''_i]$
38. Seja D_i o conjunto $\{\langle R''_i, beg''_i, piv''_i, \mathcal{C}''_i \rangle\}$
39. $piv'_i \leftarrow \left\lfloor \frac{piv'_i}{2} \right\rfloor - 1$
40. Recalcule cobertura por cliques $\mathcal{C}'_i$ para $R'_i[beg'_i, \ldots, piv'_i]$
41. Ordene $R'_i[beg'_i, \ldots, piv'_i]$
42. $pode\_doar_i \leftarrow$ **false**
43. MPI_Event_ISend("i doa D_i para j", j, ...)
44. **senão**
45. MPI_Event_ISend("i não doa subproblemas para j", j, ...)
46. **retorne** 0
47. **caso** msg_i **seja**: "j doa D_j para i" **então**
48. $Q_i \leftarrow Q_i \cup D_j$
49. $tem\_doacao\_pendente_i \leftarrow$ **false**
50. **caso** msg_i **seja**: "j não doa subproblemas para i" **então**
51. $tem\_doacao\_pendente_i \leftarrow$ **false**
52. **caso** msg_i **seja**: "j gera evento em i" **então** * mens. artificial *\
53. Não faça nada
54. **inclua** o trecho de código ramificar$_i$
55. **se** $\neg tem\_doacao\_pendente_i$ **então**
56. $tem\_doacao\_pendente_i \leftarrow$ **true**
57. Seja $j \in N(i)$ um nó escolhido ao acaso
58. MPI_Event_ISend("i deseja emparelhar-se a j", j, ...)
59. **retorne** 1
60. **retorne** 0

Função 5.2: spontaneous_event$_i$(MPI_Status *Status)

61. **inclua** o trecho de código ramificar$_i$
62. **retorne** 1

Trecho 5.1: ramificar$_i$

63. **se** $Q_i \neq \emptyset$ **então**
64. Desempilhe $\langle R_i, beg_i, piv_i, \mathcal{C}_i \rangle$ de Q_i
65. **se** $|\mathcal{C}_i| + |Q_i| > Qmax_i$ **então**
66. Calcule uma nova lista Rp_i a partir do pivô $R_i[piv_i]$
67. **se** $Rp_i \neq \emptyset$ **então**
68. $Q_i \leftarrow Q_i \cup \{\langle R_i, beg_i, piv_i - 1, \mathcal{C}_i \rangle\}$
69. $beg_i \leftarrow 0$
70. $piv_i \leftarrow |Rp_i| - 1$
71. Calcule cobertura por cliques $\mathcal{C}p_i$ para $Rp_i[beg_i, \ldots, piv_i]$
72. Ordene $Rp_i[beg_i, \ldots, piv_i]$
73. $R_i \leftarrow Rp_i$
74. $\mathcal{C}_i \leftarrow \mathcal{C}p_i$
75. $pode\_doar_i \leftarrow$ **true**
76. **senão**
77. **se** $|Q_i| + 1 > Qmax_i$ **então**
78. $Qmax_i \leftarrow |Q_i| + 1$
79. Seja $j \in N(i)$ o nó de geração de i
80. MPI_Event_ISend("i gera evento em j", j, ...)
81. **retorne** 0

Se o nó recebe um pedido de emparelhamento e pode doar, ele faz a doação da metade direita da lista base de sua pilha (linhas 30-38). Em seguida, as informações referentes à nova lista base da pilha de i (formada pela metade esquerda da lista base antiga) são calculadas (linhas 39-41). Caso não possua problemas para doar, i envia uma mensagem de negação de doação (linhas 44 e 45). No recebimento de uma mensagem de doação, o subproblema doado é incorporado (linhas 47-49). Se o nó possui listas em sua pilha, desempilhamos uma lista de Q_i e a expandimos (linhas 63 e 64). Se $|\mathcal{C}_i| + |Q_i| > Qmax_i$, então calculamos uma nova lista Rp (que será a nova lista corrente) e guardamos $R_i[beg_i, \ldots, piv_i - 1]$ (linhas 63-75). Se $|\mathcal{C}_i| + |Q_i| \leq Qmax_i$, verificamos se o tamanho do conjunto independente corrente (que corresponde a um conjunto formado pelos pivôs do caminho entre a base e o topo de Q_i) mais um é maior que o tamanho do maior conjunto independente encontrado até o momento, $Qmax_i$ (76-78). Em seguida, uma mensagem de geração é enviada para o nó de geração de i (linhas 79 e 80). Quando a pilha de i está vazia e não existe doação pendente, i envia um pedido de emparelhamento a um nó escolhido ao acaso (linhas 55-58). O algoritmo termina quando as pilhas de todos os nós se tornam vazias. Quando isso ocorre, o tamanho do conjunto independente máximo é o maior $Qmax_i$ entre todos os nós.

5.2.3 Modelo de Computação Distribuída Dirigida por Pulsos

O algoritmo **Algoritmo 5.2** a seguir utiliza ideias semelhantes às empregadas na versão no modelo de computação distribuída dirigida por eventos, associando um passo do *branch-and-bound* a um tipo de pulso. Pelo fato da existência de um mecanismo de geração de pulsos que garante a evolução da computação distribuída, não há a necessidade da utilização de mensagens artificiais nos pulsos de ramificação. Podemos elencar duas possíveis análises acerca das mensagens artificiais: uma diz respeito à mensuração de quanto a sobrecarga gerada pelas mensagens artificiais influi no desempenho da versão no modelo de computação distribuída dirigida por eventos e a outra recai sobre uma possível vantagem da implementação no modelo de computação distribuída dirigida por pulsos ser devida pelo não uso de mensagens artificiais. Como é mostrado nos experimentos (Capítulo 6), não fazemos essas análises.

Utilizamos as mesmas variáveis empregadas no **Algoritmo 5.1**. Além destas, utilizamos as seguintes variáveis auxiliares:

- $pedidos\_emparelhamento_i$: é utilizada para armazenar os nós que solicitaram doação de subproblemas ao nó i. É inicializada como um conjunto vazio;
- $subproblemas\_doados_i$: armazena os subproblemas doados aos nós que solicitaram doação. É inicializada como um conjunto vazio.

Algoritmo 5.2: *Branch-and-Bound* Distribuído Aleatório no nó i utilizando o Modelo de Computação Distribuída Dirigida por Pulsos

Entrada (quando $i = 0$): grafo $G = (V, E)$

Inicialização:

1. **inclua** <pd_mpi.h>
2. **se** $i = 0$ **então**
3. $R_i \leftarrow V$
4. $beg_i \leftarrow 0$
5. $piv_i \leftarrow |R_i| - 1$
6. Calcule cobertura por cliques $\mathcal{C}_i$ para $R_i[beg_i, \ldots, piv_i]$
7. Ordene $R_i[beg_i, \ldots, piv_i]$
8. $Q_i \leftarrow \{\langle R_i, beg_i, piv_i, \mathcal{C}_i\rangle\}$
9. Crie *buffer* $graph\_buf_i$ como indicado
10. Armazene em $size\_graph_i$ o tamanho da matriz de adjacências de G
11. $has\_graph_i \leftarrow$ **true**
12. **senão**
13. $Q_i \leftarrow \emptyset$
14. $R_i \leftarrow \emptyset$
15. $beg_i \leftarrow 0$
16. $piv_i \leftarrow 0$
17. Crie *buffer* $graph\_buf_i$ vazio
18. $size\_graph_i \leftarrow 0$
19. $has\_graph_i \leftarrow$ **false**
20. $tem\_doacao\_pendente_i \leftarrow$ **false**
21. $pode\_doar_i \leftarrow$ **false**
22. $Qmax_i \leftarrow 0$
23. $subproblemas\_doados_i \leftarrow \emptyset$
24. $pedidos\_emparelhamento_i \leftarrow \emptyset$

Configuração:

25. MPI_Pulse_set_config_param (PD_TERMINATION, PD_TERMINATION_MODEL)
26. MPI_Pulse_set_model (pulse$_i$, event$_i$)
27. MPI_Pulse_set_config_param (PD_PULSE_GENERATION, PD_PULSE_GENERATION_AUTOMATIC)

Difusão e Execução:

28. MPI_Pulse_run(graph_buf, $(n+1)$*sizeof(int)+size_graph, MPI_CHAR, ...)

Definição das funções de pulso e evento:

Função 5.3: pulse$_i$(long long pulsecount, MPI_Status *Status)

29. **se** has_graph = **false então**
30. Leia $graph\_buf$
31. $has\_graph \leftarrow$ **true**
32. **se** $pedidos\_emparelhamento_i \neq \emptyset$ **então**
33. Seja j o nó há mais tempo em $pedidos\_emparelhamento_i$
34. $pedidos\_emparelhamento_i \leftarrow pedidos\_emparelhamento_i \backslash \{j\}$
35. Seja D_i o conjunto de listas há mais tempo em $subproblemas\_doados_i$
36. $subproblemas\_doados_i \leftarrow subproblemas\_doados_i \backslash \{D_i\}$
37. **se** $D_i \neq \emptyset$ **então**
38. MPI_Pulse_ISend("i doa D_i para j", j, ...)
39. **senão**
40. MPI_Pulse_ISend("i não doa subproblemas para j", j, ...)
41. **retorne** 0
42. **se** $Q_i \neq \emptyset$ **então**
43. Desempilhe $\langle R_i, beg_i, piv_i, \mathcal{C}_i \rangle$ de Q_i
44. **se** $|\mathcal{C}_i| + |Q_i| > Qmax_i$ **então**
45. Calcule uma nova lista Rp_i a partir do pivô $R_i[piv_i]$
46. **se** $Rp_i \neq \emptyset$ **então**
47. $Q_i \leftarrow Q_i \cup \{\langle R_i, beg_i, piv_i - 1, \mathcal{C}_i \rangle\}$
48. $beg_i \leftarrow 0$
49. $piv_i \leftarrow |Rp_i| - 1$
50. Calcule cobertura por cliques $\mathcal{C}p_i$ para $Rp_i[beg_i, \ldots, piv_i]$
51. Ordene $Rp_i[beg_i, \ldots, piv_i]$
52. $R_i \leftarrow Rp_i$
53. $\mathcal{C}_i \leftarrow \mathcal{C}p_i$
54. $pode\_doar_i \leftarrow$ **true**
55. **senão**
56. **se** $|Q_i| + 1 > Qmax_i$ **então**
57. $Qmax_i \leftarrow |Q_i| + 1$
58. **retorne** 0
59. **se** $\neg tem\_doacao\_pendente_i$ **então**
60. $tem\_doacao\_pendente_i \leftarrow$ **true**
61. Seja $j \in N(i)$ um nó escolhido ao acaso
62. MPI_Pulse_ISend("i deseja emparelhar-se a j", j, ...)
63. **retorne** 1
64. **retorne** 0

Função 5.4: event$_i$(MPI_Status *Status)

65. MPI_Pulse_Recv(msg_i, ...)
66. **caso** msg_i **seja**: "j deseja emparelhar-se a i" **então**
67. **se** $Q_i \neq \emptyset$ **e** $pode\_doar_i$ **então**
68. Seja $\langle R'_i, beg'_i, piv'_i, \mathcal{C}'_i \rangle$ a base da pilha Q_i
69. $beg''_i \leftarrow \left\lfloor \frac{piv'_i}{2} \right\rfloor$
70. $piv''_i \leftarrow piv'_i$
71. $R''_i \leftarrow R'_i[beg''_i, \ldots, piv''_i]$
72. Calcule cobertura por cliques $\mathcal{C}''_i$ para $R''_i[beg''_i, \ldots, piv''_i]$
73. Ordene $R''_i[beg''_i, \ldots, piv''_i]$
74. Seja D_i o conjunto $\{\langle R''_i, beg''_i, piv''_i, \mathcal{C}''_i \rangle\}$
75. $piv'_i \leftarrow \left\lfloor \frac{piv'_i}{2} \right\rfloor - 1$
76. Recalcule cobertura por cliques $\mathcal{C}'_i$ para $R'_i[beg'_i, \ldots, piv'_i]$
77. Ordene $R'_i[beg'_i, \ldots, piv'_i]$
78. $pode\_doar_i \leftarrow$ **false**
79. **senão**
80. $D_i \leftarrow \emptyset$
81. $pedidos\_emparelhamento_i \leftarrow pedidos\_emparelhamento_i \cup \{j\}$
82. $subproblemas\_doados_i \leftarrow subproblemas\_doados_i \cup \{D_i\}$
83. **retorne**
84. **caso** msg_i **seja**: "j doa D_j para i" **então**
85. $Q_i \leftarrow Q_i \cup D_j$
86. $tem\_doacao\_pendente_i \leftarrow$ **false**
87. **retorne**
88. **caso** msg_i **seja**: "j não doa subproblemas para i" **então**
89. $tem\_doacao\_pendente_i \leftarrow$ **false**
90. **retorne**

Ao receber um pedido de doação, se o nó pode doar, ele armazena essas doações para enviá-las em pulsos posteriores (linhas 66-78). Caso não possa doar, ele armazena um conjunto vazio para indicar que deve enviar uma negação de doação ao nó solicitante (linhas 79 e 80). Ao executar um pulso, se o nó possui pedidos de emparelhamento, ele executa as doações ou negações, enviando primeiramente os pedidos mais antigos (linhas 32-41).

Capítulo 6

Experimentos

Este capítulo destina-se à apresentação de experimentos feitos com a aplicação do Capítulo 5. A análise destes experimentos é baseada em uma comparação entre uma implementação serial do algoritmo de *Tomita* e *Kameda* [23] e uma implementação das versões distribuídas definidas no Capítulo 5. Através desses experimentos, avaliamos o desempenho da implementação da *interface* proposta no Capítulo 4.

6.1 Parâmetros de Avaliação

O tempo sequencial de um programa corresponde ao tempo entre o começo da execução do programa até o seu fim em uma máquina sequencial. O tempo paralelo corresponde ao tempo entre o começo da execução da computação paralela até o final da execução do último elemento de processamento [14]. Representamos o tempo sequencial por T_S e o tempo paralelo por T_P.

Quando avaliamos o desempenho de um programa paralelo, estamos geralmente interessados em calcular o benefício relativo entre a versão paralelizada e a versão sequencial do problema. O ganho de performance obtido através da paralelização pode ser obtido pela grandeza aceleração, a qual é definida através da razão entre o tempo levado para resolver o problema em um único elemento de processamento utilizando o melhor algoritmo sequencial conhecido e o tempo necessário para resolver o mesmo problema paralelamente em p elementos de processamento. Denotamos a aceleração pela notação S. Formalmente, temos

$$S(p) = \frac{T_S}{T_P(p)}.$$

Outra grandeza, a eficiência, aqui denotada por E, revela a fração de tempo a qual cada elemento de processamento é utilizado efetivamente. É dada pela razão entre a aceleração e o número de nós da rede, $|N|$. Em um sistema paralelo ideal, temos aceleração igual a $|N|$ e eficiência igual a 1. Todavia, na prática, a eficiência é um valor entre 0 e 1, e, além disso, quanto mais próximo de 1, melhor cada nó será utilizado. Matematicamente,

$$E(|N|) = \frac{S(|N|)}{|N|}.$$

6.2 Descrição dos Experimentos

Em toda a nossa implementação, utilizamos a linguagem de programação C. Na implementação da *interface* de programação distribuída, utilizamos a biblioteca MPI e a distribuição escolhida foi a MPICH2 [19]. A escolha dessa distribuição deve-se simplesmente pelo fato de a mesma já ser adotada em projetos do nosso grupo de pesquisa. A partir dessa implementação, foram realizados os experimentos descritos adiante.

Os experimentos foram realizados em *cluster* formado por 16 nós, cada um possuindo 2 núcleos de processamento e conectados por uma rede *fast-ethernet*, do *cluster* do Laboratório de Pesquisa em Computação (LIA), localizado no Departamento de Computação da UFC. Utilizando este *cluster*, calculamos os tempos seriais e os tempos das versões distribuídas do problema CIM.

Antes de fazermos a descrição completa das execuções, fazemos adiante algumas considerações importantes:

Tempo sequencial Como base para o tempo sequencial para o problema CIM, implementamos a versão sequencial do algoritmo original proposto por Tomita e Kameda [23].

Instâncias Para a realização dos testes, usamos como instância os complementos de 10 grafos contidos no jogo de testes para o problema de clique máxima em grafos (de onde decorre o fato de tomarmos os complementos desses grafos) utilizados no *Second DIMACS Implementation Challenge for Cliques, Coloring and Satisfiability* [11]. Concentramos nossa análise em instâncias escolhidas dentre as que refletem os casos mais difíceis para o algoritmo de Tomita e Kameda [23]. Escolhemos também 8 instâncias de grafos aleatórios densos (seus complementos são, naturalmente, esparsos). A geração desses grafos aleatórios se dá pela escolha de um conjunto de vértices e, em seguida, pelo lançamento de uma moeda para cada dois vértices. Caso o lançamento dê "cara", é criada uma aresta entre esses dois vértices. Caso contrário, não é criada uma aresta entre eles. A densidade de um grafo $G = (V, E)$, $D(G)$, é dada por $|E|/\{n \times (n-1)/2\}$, em que $n = |V|$. Grafos densos permitem uma melhor análise do comportamento das implementações distribuídas uma vez que geram, em geral, um número grande de ramificações.

Mensagens Artificiais no *Branch-and-bound com Eventos* Como visto no **Algoritmo 5.1** do Capítulo 5, para cada evento de expansão em um nó $i \in N$, é gerada uma mensagem, destinada a um nó $j \in N(i)$ (i é o nó de geração de j), que tem por objetivo gerar uma cadeia causal de eventos. Para reduzir a sobrecarga gerada pela criação dessa mensagem adicional, fazemos também com que o nó de geração de j seja i. Mais ainda, fazemos que i e j estejam no mesmo nó de processamento durante as execuções no *cluster*.

Execuções Nas versões distribuídas, rodamos execuções com 4, 8 e 16 processos MPI. No modelo de computações dirigidas por pulsos, utilizamos um mapeamento um-para-um entre os nós do *cluster* e os processos MPI. Dessa forma, por exemplo, em execuções com 16 processos MPI, utilizamos 16 nós do *cluster*. No modelo de computações dirigidas por eventos, cada nó roda dois processos, tendo em vista reduzir a sobrecarga gerada pelas mensagens artificiais. Nesse caso, por exemplo, em execuções com 8 processos MPI, utilizamos 4 nós do *cluster*, em que cada nó abriga dois processos. Neste trabalho, não fizemos a análise de quanto

esta estratégia seria melhor do que uma distribuição um-para-um entre os processos e nós no modelo de computações dirigidas por eventos. O nó a qual cada processo pertence é fixo durante toda a execução. Considerando o não-determinismo dos modelos, para cada instância relacionada e para cada número de processos (4, 8 e 16), rodamos as versões distribuídas 5 vezes. O valor 5 foi deduzido a partir de alguns experimentos prévios, que tiveram por finalidade estabelecer um tempo razoável para a execução de todos os experimentos. Os valores aferidos finais são obtidos através de uma média dos valores aferidos nas 5 execuções.

Valores Aferidos Na versão sequencial, medimos o tempo de execução e o número de ramificações realizadas pelo algoritmo. Nas versões distribuídas, além dos valores calculados anteriormente, calculamos também o número de pedidos de doação feitos pelos nós (que corresponde ao número de passos de emparelhamento), o número de negações de doações feitas pelos nós, a aceleração (S) e a eficiência (E). De forma análoga à aceleração e a eficiência, defimos também as grandezas *aceleração potencial* e *eficiência potencial*, porém, nestas, comparamos as quantidades de ramificações (e não o tempo de execução). Essas medidas especiais se aplicam nos casos em que o algoritmo sequencial não é capaz de resolver o problema dentro de limite de 1800 segundos (por exemplo, `p_hat500-3`, `brock400_1` e `MANN_a45` dentre os grafos DIMACS). O limite de 1800 segundos foi obtido através de experimentos prévios que tiveram por finalidade estabelecer um tempo razoável para a execução dos experimentos finais. Em tais casos, naturalmente, não é possível fazer as medidas de aceleração e eficiência, mas apenas uma estimativa desses valores. A aceleração potencial, estimativa da aceleração, é calculada em duas etapas. Na primeira etapa, ajustamos o número de ramificações que a execução sequencial realiza, da seguinte forma. Sejam RAM_s, T_s e T_p, respectivamente, o número de ramificações e o tempo da execução sequencial, e o tempo de uma execução distribuída, todos medidos para uma mesma instância. Tomamos, então,

$$RAM_s^{ajust}(|N|) = \frac{T_p(|N|)RAM_s}{T_s}.$$

Observe que em algumas instâncias (Tabela 6.3), as execuções distribuídas também não foram concluídas dentro do tempo limite de 1800 segundos. Nesses casos, $RAM_s^{ajust} = RAM_s$. A segunda etapa é a comparação dos números de ramificação nas execuções sequencial e distribuída, obtendo assim a aceleração potencial na forma

$$SP(|N|) = \frac{RAM_p(|N|)}{RAM_s^{ajust}(|N|)}.$$

A eficiência potencial, naturalmente, é dada por

$$EP(|N|) = \frac{SP(|N|)}{|N|}.$$

6.3 Experimentos

As tabelas adiante descrevem os valores das execuções para o problema CIM. Para cada instância G, n, $\alpha(G)$ e $D(G)$ representam, respectivamente, o número de vértices, o tamanho do conjunto independente máximo calculado para a instância e a densidade. O valor da densidade apresentado nas tabelas é o do grafo original no jogo de testes. Uma vez que os algoritmos considerados utilizam os complementos desses grafos, a densidade real das execuções é calculada por $1 - D(G)$. Atribuimos um tempo máximo de 1800 segundos para cada experimento. Assim sendo, quando tivermos em um experimento um tempo próximo de 1800 segundos, isto significa que o experimento foi interrompido e os valores foram aferidos até este momento. Para esses experimentos, os valores aceleração e eficiência não são calculados. Na coluna "Ramificações", os valores apresentados foram obtidos dividindo o número de ramificações do experimento por 10^5.

Tabela 6.1 Tempos seriais para o Problema CIM

Instância	**Tempo (s)**	**Ramificações ($\div 10^5$)**
(n, $\alpha(G)$, $D(G)$)		
Grafos DIMACS		
p_hat300-3	30,65	20,69
(300, 36, 0.744)		
p_hat500-3	1800,07	667,59
(500, 50, 0.75)		
brock400_1	1800,02	1876,52
(400, 27, 0.75)		
brock400_2	1540,94	1198,89
(400, 29, 0.75)		
brock400_4	1298,28	1159,99
(400, 33, 0.75)		
DSJC500.5	7,84	13,12
(500, 13, 0.502)		
DSJC1000.5	706,39	911,61
(1000, 15, 0.50)		
MANN_a27	8,17	0,38
(378, 126, 0.9901)		
MANN_a45	1802,77	10,57
(1035, 345, 0.996)		
san400_0.9_1	20,22	2,62
(400, 100, 0.9)		
Grafos Aleatórios		
g200_90	1504,28	984,92
(200, 40, 0.90)		
g300_70	45,67	58,85
(300, 20, 0.70)		
g300_90	1800,02	1147,72
(300, 44, 0.9)		
g300_95	1800,01	739,76
(300, 66, 0.95)		
g400_70	649,94	754,65
(400, 21, 0.70)		
g400_90	1800	1149,71
(400, 47, 0.9)		
g400_95	1800,16	833,64
(400, 71, 0.95)		
g1000_50	662,96	852,2
(1000, 15, 0.5)		

Tabela 6.2 Tempos para o Problema CIM com *Branch-and-bound* Distribuído Aleatório utilizando o Modelo de Computação Distribuída Dirigido por Eventos (Grafos Dimacs)

Instância	Tempo (s)	Ram.	*SP*	*EP*	*S*	*E*	Neg. Doa.	Neg. Doa. (%)
p_hat300-3								
4	9,03	21,1	3,46	0,87	3,39	0,85	189,5	10 (5,28)
8	5,84	20,41	5,18	0,65	5,25	0,66	517,6	36,6 (7,07)
16	4,58	20,64	6,67	0,42	6,69	0,42	478	59,2 (12,38)
p_hat500-3								
4	1472,18	1902	3,48	0,87	1,22	0,31	425	11 (2,59)
8	755,73	1956,41	6,98	0,87	2,38	0,3	147	13 (8,84)
16	382,05	1910,22	13,48	0,84	4,71	0,29	2821,5	61 (2,16)
brock400_1								
4	957,96	3275,69	3,28	0,82	1,88	0,47	346,67	11 (3,17)
8	461,03	3040,49	6,33	0,79	3,9	0,49	859	24,67 (2,87)
16	263,77	3425,13	12,46	0,78	6,82	0,43	1630	67,67 (4,15)
brock400_2								
4	375,16	969,36	3,32	0,83	4,11	1,03	301,1	11,2 (3,72)
8	206,91	1110,67	6,9	0,86	7,45	0,93	731	28 (3,83)
16	140,75	1625,32	14,84	0,93	10,95	0,68	1398,8	66,8 (4,78)
brock400_4								
4	335,45	978,22	3,26	0,82	3,87	0,97	271,9	11,5 (4,23)
8	151,82	1055,2	7,78	0,97	8,55	1,07	603	26,2 (4,34)
16	24,47	189,87	8,69	0,54	53,06	3,32	1082,8	59 (5,45)
DSJC500.5								
4	6,16	12,82	1,24	0,31	1,27	0,32	176,2	10,3 (5,85)
8	2,73	12,91	2,82	0,35	2,87	0,36	213	28,4 (13,33)
16	2,29	12,39	3,24	0,2	3,43	0,21	127,6	48,4 (37,93)
DSJC1000.5								
4	230,26	907,12	3,05	0,76	3,07	0,77	166,33	11,33 (6,81)
8	102,98	895,65	6,74	0,84	6,86	0,86	1903,67	30 (1,58)
16	59,49	873,94	11,38	0,71	11,87	0,74	2075	101,67 (4,9)
MANN_a27								
4	2,13	0,38	3,87	0,97	3,84	0,96	22	9 (40,91)
8	1,12	0,39	7,42	0,93	7,28	0,91	49,8	19,8 (39,76)
16	0,71	0,39	11,84	0,74	11,58	0,72	102,6	40,2 (39,18)
MANN_a45								
4	1328,85	29,63	3,8	0,95	1,36	0,34	49,8	11,1 (22,29)
8	660,56	29,52	7,62	0,95	2,73	0,34	138,6	24,4 (17,6)
16	341,56	29,56	14,76	0,92	5,28	0,33	367	45,5 (12,4)
san400_0.9_1								
4	0,21	0,06	2,39	0,6	96,24	24,06	10,9	8,6 (78,9)
8	0,18	0,12	5,42	0,68	114,09	14,26	19,4	16,6 (85,57)
16	0,16	0,36	17,32	1,08	127,28	7,96	42	34,6 (82,38)

Tabela 6.3 Tempos para o Problema CIM com *Branch-and-bound* Distribuído Aleatório utilizando o Modelo de Computação Distribuída Dirigido por Eventos (Grafos Aleatórios)

Instância	**Tempo (s)**	**Ram.**	*SP*	*EP*	*S*	*E*	**Neg. Doa.**	**Neg. Doa. (%)**
g200_90								
4	413,99	952,42	3,51	0,88	3,63	0,91	104,33	10,67 (10,22)
8	169,01	731,53	6,61	0,83	8,9	1,11	249	23 (9,24)
16	88,54	769,28	13,27	0,83	16,99	1,06	632	52,33 (8,28)
g300_70								
4	14,43	59,46	3,2	0,8	3,17	0,79	195,67	9 (4,6)
8	7,6	59,95	6,12	0,77	6,01	0,75	370,5	34 (9,18)
16	4,12	57,42	10,82	0,68	11,09	0,69	863,33	64,33 (7,45)
g300_90								
4	1800,05	3920,47	3,42	0,85			27,4	11,3 (41,24)
8	1800,03	7288,65	6,35	0,79			62	23,8 (38,39)
16	1800,08	10974,86	9,56	0,6			113,2	47 (41,52)
g300_95								
4	1800,06	2350,18	3,18	0,79			54,7	14,5 (26,51)
8	1800,09	4705,24	6,36	0,8			78,8	26 (32,99)
16	1800,11	6989	9,45	0,59			137,4	48,8 (35,52)
g400_70								
4	200,99	749,95	3,21	0,8	3,23	0,81	44,5	11,2 (25,17)
8	100,43	761,67	6,53	0,82	6,47	0,81	79	25 (31,65)
16	53,33	784,34	12,67	0,79	12,19	0,76	96	40,5 (42,19)
g400_90								
4	1800,02	3862,14	3,36	0,84			42,1	11,4 (27,08)
8	1800,04	7172,51	6,24	0,78			52,2	20,2 (38,7)
16	1800,07	13637,75	11,86	0,74			117	48 (41,03)
g400_95								
4	1800,02	2296,16	2,75	0,69			24,9	10,7 (42,97)
8	1800,08	4793,47	5,75	0,72			50	21 (42)
16	1800,06	9156,32	10,98	0,69			108,5	47 (43,32)
g1000_50								
4	197,51	847,85	3,34	0,83	3,36	0,84	1501,4	24,2 (1,61)
8	106,07	857,36	6,29	0,79	6,25	0,78	2053,8	33,8 (1,65)
16	73,51	864,97	9,15	0,57	9,02	0,56	3106	59 (1,9)

Tabela 6.4 Tempos para o Problema CIM com *Branch-and-bound* Distribuído Aleatório utilizando o Modelo de Computação Distribuída Dirigido por Pulsos (Grafos DIMACS)

Instância	Tempo (s)	Ram.	*SP*	*EP*	*S*	*E*	Neg. Doa.	Neg. Doa. (%)
p_hat300-3								
4	10,95	21,01	2,84	0,71	2,8	0,7	193,3	11,8 (6,1)
8	8,33	20,91	3,72	0,46	3,68	0,46	280	27,4 (9,79)
16	6,96	21,55	4,59	0,29	4,41	0,28	249,8	53,2 (21,3)
p_hat500-3								
4	1357,16	1908,94	3,79	0,95	1,33	0,33	453	11,33 (2,5)
8	691,91	1992,47	7,76	0,97	2,6	0,33	1249,67	29 (2,32)
16	361,98	1908,15	14,21	0,89	4,97	0,31	2558,33	54,67 (2,14)
brock400_1								
4	957,96	3378,16	3,38	0,85	1,88	0,47	346,67	11 (3,17)
8	461,03	3039,51	6,32	0,79	3,9	0,49	859	24,67 (2,87)
16	263,77	3630,9	13,2	0,83	6,82	0,43	1630	67,67 (4,15)
brock400_2								
4	387,1	967,98	3,21	0,8	3,98	1	344,3	14,1 (4,1)
8	193,78	1117,44	7,41	0,93	7,95	0,99	730,2	32,8 (4,49)
16	132,03	1665,37	16,21	1,01	11,67	0,73	1425,8	63,6 (4,46)
brock400_4								
4	266,34	1014,26	4,26	1,07	4,87	1,22	301	13,7 (4,55)
8	129,16	1202,62	10,42	1,3	10,05	1,26	536,8	31,4 (5,85)
16	24,86	229,94	10,35	0,65	52,23	3,26	1025	63,7 (6,21)
DSJC500.5								
4	6,58	13,13	1,19	0,3	1,19	0,3	221,1	17,6 (7,96)
8	3,67	12,92	2,1	0,26	2,14	0,27	142,2	28,2 (19,83)
16	2,4	12,5	3,11	0,19	3,27	0,2	107,6	44,4 (41,26)
DSJC1000.5								
4	204,84	907,14	3,43	0,86	3,45	0,86	166,33	6 (3,61)
8	102,98	896,66	6,75	0,84	6,86	0,86	1903,67	41,67 (2,19)
16	64,78	884,87	10,59	0,66	10,91	0,68	2075	69 (3,33)
MANN_a27								
4	2,07	0,38	3,97	0,99	3,96	0,99	26,8	9,5 (35,45)
8	1,05	0,39	8,06	1,01	7,76	0,97	67,8	22,8 (33,63)
16	0,56	0,4	15,2	0,95	14,54	0,91	130,6	44,4 (34)
MANN_a45								
4	1800,24	35,62	3,38	0,84			51,5	10,4 (20,19)
8	1277,86	29,52	3,94	0,49	1,41	0,18	127,2	25,2 (19,81)
16	633,92	29,56	7,95	0,5	2,84	0,18	186,8	52,8 (28,27)
san400_0.9_1								
4	1,46	0,05	0,26	0,07	13,89	3,47	19,9	11,6 (58,29)
8	0,1	0,18	14,33	1,79	207,68	25,96	29,2	18,6 (63,7)
16	0,35	1,03	22,52	1,41	57,41	3,59	54,8	37,2 (67,88)

Tabela 6.5 Tempos para o Problema CIM com *Branch-and-bound* Distribuído Aleatório utilizando o Modelo de Computação Distribuída Dirigido por Pulsos (Grafos Aleatórios)

Instância	Tempo (s)	Ram.	*SP*	*EP*	*S*	*E*	Neg. Doa.	Neg. Doa. (%)
g200_90								
4	393,16	954,38	3,71	0,93	3,83	0,96	111	11,33 (10,21)
8	156,45	738,44	7,21	0,9	9,62	1,2	326,33	24 (7,35)
16	553,42	784,38	2,16	0,14	2,72	0,17	421,67	56,33 (13,36)
g300_70								
4	13,05	60,6	3,6	0,9	3,5	0,87	247,33	12 (4,85)
8	7,31	59,83	6,35	0,79	6,25	0,78	327	25,33 (7,75)
16	4,12	58,48	11,02	0,69	11,09	0,69	863,33	64,33 (7,45)
g300_90								
4	1800,76	4161,37	3,62	0,91			37,8	12,6 (33,33)
8	1800,68	7902,27	6,88	0,86			69,6	24,4 (35,06)
16	1800,72	15520,03	13,52	0,84			88,4	43,6 (49,32)
g300_95								
4	1800,47	2538,23	3,43	0,86			80,7	15,5 (19,21)
8	1800,47	4958,74	6,7	0,84			82,4	27,2 (33,01)
16	1800,52	9903,29	13,38	0,84			131,4	44,2 (33,64)
g400_70								
4	180,74	760,27	3,62	0,91	3,6	0,9	290,33	10,33 (3,56)
8	91,82	763,2	7,16	0,89	7,08	0,88	733	23,33 (3,18)
16	48,98	796,14	14	0,87	13,27	0,83	1533,67	66 (4,3)
g400_90								
4	1800,8	4225,17	3,67	0,92			35,6	11,3 (31,74)
8	1800,69	7755,02	6,74	0,84			72	22,4 (31,11)
16	1800,68	14977,9	13,02	0,81			137	49,4 (36,06)
g400_95								
4	1800,51	2581,24	3,1	0,77			39	10,9 (27,95)
8	1800,51	5274,85	6,33	0,79			74	25,6 (34,59)
16	1800,3	1802,55	2,16	0,14			53,2	36,4 (68,42)
g1000_50								
4	194,51	768,95	3,08	0,77	3,41	0,85	733,8	16,2 (2,21)
8	105,2	859,56	6,36	0,79	6,3	0,79	1348,2	33,2 (2,46)
16	82,41	871,33	8,23	0,51	8,05	0,5	2322	62,4 (2,69)

6.4 Avaliação dos Experimentos

Segundo experimentos relatados na literatura, são os seguintes fatores que determinam a eficiência de uma execução distribuída de algoritmos do tipo *branch-and-bound* [3, 5, 12, 14]:

Número de ramificações: em razão de o percurso na árvore de ramificações diferir entre a execução sequencial e as execuções distribuídas, o número total de ramificações nos diversos casos também podem diferir. Em geral, uma boa paralelização não causa um aumento significativo nesses números, embora eles possam até diminuir. Para analisar esse aspecto, os totais de ramificações são medidos nos diversos experimentos.

Balanceamento: a distribuição igualitária das ramificações pelos nós da rede é um fato essencial para o efetivo aproveitamento do paralelismo potencial inerente ao uso de diversos nós. Observe que quanto melhor for a distribuição de trabalho entre os nós, mais nós possuirão subproblemas para doar e teremos um menor número de negações de doações. Dessa forma, as quantidades de pedidos de doação e o percentual dos não atendidos servem como critério para analisar esse fator.

menor será o número de negações de doações.

Comunicações e sincronização: interrupções no ritmo de computações dos nós podem afetar o tempo total das execuções distribuídas. Dessa forma, processamento de comunicações de mensagens de controle ou pontos de sincronização devem ser limitados. A acelerações e eficiências potenciais nos grafos que não são resolvidos pela implementação sequencial servem como uma estimativa para esse fator, na medida em que as execuções distribuídas sofrem menor impacto da fase de terminação.

Uma observação inicial é que em grafos difíceis cujas execuções sequenciais foram concluídas dentro do tempo limite de 1800 segundos (`brock400_2`, `brock400_4` e `g200_90`), temos um percentual pequeno de negações de doações (Tabelas 6.2 e 6.3). Isto revela que um grande número de ramificações tende a reduzir os efeitos de desbalanceamento. Vale ressaltar que o desbalanceamento dessa implementação é proveniente exclusivamente do algoritmo *branch-and-bound* adotado e não da *interface* de programação que definimos neste trabalho. Nestes casos, ao se observar a eficiência, percebe-se que os nós trabalharam efetivamente em média 75% do tempo.

Para grafos fáceis (`p_hat300-3`, `DSJC500.5`, `g300_70` etc.), ao se observar o número de negações de doações, temos em geral um desbalanceamento maior (Tabelas 6.2, 6.3, 6.4 e 6.5). Entretanto, para essas execuções, esse desbalanceamento não interfere de forma significante na eficiência, pois o ganho com a paralelização da enumeração é maior, o que resulta, em algumas ocasiões, em uma *aceleração superlinear* [14]. Segundo [14], a aceleração superlinear é uma anomalia comum a paralelizações de algoritmos de busca como o *branch-and-bound*.

Outra análise possível diz respeito aos grafos muito difíceis, os quais não foram resolvidos pela implementação sequencial em até 1800 segundos (`g300_90`, `g300_90`, `g300_70` `g400_95`). Para esses casos, observando a coluna EP (Eficiência Potencial), concluimos que, até o momento da interrupção do experimento, os nós trabalharam em média 80% do tempo (Tabelas 6.2 e 6.4). Observe também que, para esses experimentos, temos um número grande de negações de doações

(em torno de 40%). Por esses dois fatos, concluímos que o processo de enumeração do algoritmo começa lentamente e acelera gradualmente, acompanhando o crescimento da árvore de enumeração. Dessa forma, conluimos que para essas execuções é mantido um bom balanceamento.

Podemos fazer uma análise da *interface* sob dois aspectos: sua conveniência para a elaboração e implementação de algoritmos distribuídos e a eficiência dos resultados obtidos. Com relação à elaboração de algoritmos, essa tarefa pode ser facilitada através do uso dos modelos, uma vez que os mesmos fornecem um formalismo para a definição de aplicações. No tocante à implementação, o requisito reuso surge como elemento principal. Além do reuso herdado da biblioteca MPI, a *interface* fornece tanto o assincronismo necessário às aplicações em Otimização Combinatória, quanto um conjunto de operações coletivas úteis a essas aplicações.

Para que possamos avaliar a *interface* em termos de eficiência, devemos observar o comportamento da implementação quando há pouco desbalanceamento (percentual de negação de doações baixo), pois nesse caso há uma carga menor de sincronização. Para instâncias em que isso ocorre (`p_hat500-3`, `brock400_1`, `g1000_50` etc.), a eficiência se aproxima de 55% (Tabelas 6.2, 6.3, 6.4 e 6.5). Em comparação com a eficiência para instâncias difíceis, temos uma diferença de 20%, o que que revela que o uso da *inteface* implica em uma perda de 20% em eficiência.

Outra análise possível pode ser feita para os casos em que os grafos possuem tanto S e E quanto SP e EP. Nesses casos, para a maioria dos grafos, S-SP e E-EP são valores próximos de zero (Tabelas 6.2, 6.3, 6.4 e 6.5). Dessa forma, para grafos que só possuem SP e EP, essas grandezas são suficientes para uma análise aproximada de aceleração e eficiência.

Por fim, podemos estabelecer uma comparação entre os dois modelos com relação à aplicação em questão. Para grafos muito densos (densidade entre 0.9000 e e 0.9999), a versão no modelo de computação distribuída dirigida por pulsos revelou melhor desempenho que a versão no modelo de computação distribuída dirigida por eventos. Devido a essa alta densidade, existe um número grande de listas de vértices candidatas. Observe que no modelo de computações dirigidas por pulsos, através do mecanismo de geração de pulsos, pulsos são gerados enquanto existirem subproblemas a serem expandidos, impedindo que os nós fiquem ociosos por muito tempo e esvaziem suas pilhas mais rapidamente.

Para os outros grafos com menor densidade, a versão no modelo de computação distribuída dirigida por eventos foi mais eficiente. Pela existência de menos listas de vértices candidatas, os nós ficam ociosos mais rapidamente. Essa ociosidade faz com que os nós executem mais pedidos de doações. Os pedidos de doações e suas respostas (doação ou negação de doação) geram eventos nos nós, fazendo com que os nós trabalhem (quando recebem doações) ou continuem pedindo (quando recebem negações de doações). Dessa forma, quando existem menos subproblemas a serem examinados, as computações distribuídas dirigidas por eventos são mais adequadas a essa aplicação.

Capítulo 7

Conclusões

Neste capítulo, fazemos as considerações finais com relação a este trabalho.

7.1 Principais Dificuldades Encontradas

O objetivo deste trabalho consiste na definição de uma *interface* de programação distribuída que valorize os requisitos eficiência e reuso em aplicações em Otimização Combinatória. Ao escolhermos a biblioteca MPI como ponto de partida para a *interface*, tivemos que introduzir uma visão assíncrona na MPI, uma vez que a mesma tem sua inspiração principal em aplicações que envolvem um alto grau de sincronismo. A via empregada para para alcançar esse objetivo foi a utilização dos modelos teóricos e a proposta de expansão da MPI.

A implementação de um algoritmo *branch-and-bound* distribuído eficiente foi outro ponto que nos gerou bastante dificuldade. Primeiramente, encontramos dificuldades relativas à manipulação das estruturas de dados contidas nos passos do *branch-and-bound* do Capítulo 5. Em seguida, foi necessária a criação de mensagens artificiais nos passos de ramificação do *branch-and-bound* no Modelo de Computação Distribuída Dirigida por Eventos para se garantir que a computação terminasse. Outro ponto que mereceu uma análise mais profunda foi o ajuste de alguns parâmetros da aplicação de forma a melhorar o balanceamento de trabalho entre os nós da rede.

7.2 Contribuições

Nossas contribuições recaem principalmente sobre as versões de algoritmos criadas neste trabalho. São elas o mecanismo de geração de pulsos, os algoritmos para o Registro dos Estados Locais Iniciais nos dois modelos, os algoritmos para Detecção da Terminação Global nos dois modelos, os algoritmos para ordem FIFO nos dois modelos, os algoritmos para ordem Causal nos dois modelos e os algoritmos para o Problema CIM que utilizam o *Branch-and-Bound* Distribuído Aleatório nos dois modelos.

7.3 Produtos Gerados

O principal produto gerado foi uma *interface* de programação distribuída que se adequa a aplicações em Otimização Combinatória, tanto por sua conveniência na elaboração de algoritmos

distribuídos quanto por sua eficiência (ver seção 6.4). Foi gerada também uma implementação parcial dessa *interface*, a qual contém, com exceção das operações coletivas Registro dos Estados Locais Iniciais e Ordenação de Mensagens, todas as funcionalidades previamente definidas para ela.

Com relação a aplicações, geramos um *Branch-and-bound* Distribuído Aleatório eficiente para aplicações em Otimização Combinatória. Geramos também uma implementação para o Problema CIM, que é um problema fundamental na área de pesquisa em questão.

7.4 Trabalhos futuros

Algumas questões que podem ser abordadas futuramente, são:

- Implementar os mecanismos de Ordenação de Mensagens nos dois modelos;
- Incluir novas operações coletivas, como, por exemplo, o compartilhamento de recursos [2]. Através do gerenciamento do compartilhamento de recursos, os nós podem ter acesso a informações compartilhadas entre eles de forma justa;
- Implementar as funções de comunicação assíncronas que foram definidas na *interface*.

Referências Bibliográficas

[1] Barbosa, V. C. *Introduction to Distributed Algorithms*. A The MIT Press, Cambridge, MA, 1996.

[2] Chandy, K. M., and Misra, J. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems 6* (1984), 632–646.

[3] Corrêa, R. *Models for Parallel and Distributed Computation: Theory, Algorithmic Techniques, and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[4] Corrêa, R. C., and Barbosa, V. C. Partially ordered distributed computations on asynchronous point-to-point networks. *Parallel Computing 35* (2009), 12–28.

[5] Corrêa, R. C., Ferreira, A., and Porto, S. C. S. *Handbook of Combinatorial Optimization*, vol. 3. Kluwer, 1998, ch. Solving hard problems using parallel computers, pp. 407–456.

[6] Crainic, T. G., Cun, B. L., and Roucairol, C. Chapter 1 parallel branch-and-bound algorithms. *Tutorials on Emerging Methodologies and Applications in Operations Research* (2004), 44.

[7] David A. Bader, W. E. H., and Phillips, C. A. Chapter 5 parallel algorithm design for branch and bound. *Tutorials on Emerging Methodologies and Applications in Operations Research* (2004), 28.

[8] de Araújo, G. A. *Uma Nova Plataforma CCA para Aplicações de Alto Desempenho usando Conectores Exógenos*. Doutorado em Ciência da Computação, Departamento de Computação – Universidade Federal do Ceará(DC/UFC), 2010.

[9] Gark, V. K. *Concurrent and Distributed Computing in Java*. IEEE Press, Austin, TX, 2004.

[10] Gendron, B., and Crainic, T. G. Parallel branch-and-branch algorithms: Survey and synthesis. *Operations Research 42*, 6 (November/December 1994), 1042–1066.

[11] Johnson, D., and Trick, M., Eds. *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, vol. 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[12] Karp, R. M., and Zhang, Y. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM 40*, 3 (1993), 765–798.

[13] Kshemkalyani, A. D., and Singhal, M. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, New York, NY, USA, 2008.

[14] Kumar, V. *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[15] Land, A. H., and Doig, A. G. An automatic method of solving discrete programming problems. *Econometrica 28*, 3 (1960), 497–520.

[16] Lumezanu, C., Spring, N., and Bhattacharjee, B. Decentralized message ordering for publish/subscribe systems. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware* (New York, NY, USA, 2006), Middleware '06, Springer-Verlag New York, Inc., pp. 162–179.

[17] Misra, J., and Chandy, K. M. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst. 4*, 1 (1982), 37–43.

[18] Mpi. http://www-unix.mcs.anl.gov/mpi/. acesso em agosto de 2010.

[19] Mpich2. http://www.mcs.anl.gov/research/projects/mpich2/. acesso em julho de 2011.

[20] Post, D. E., and Votta, L. G. Computational Science Demands a New Paradigm. *Physics Today 58*, 1 (2005), 502–527.

[21] Robson, J. M. Algorithms for maximum independent sets. *Journal of Algorithms 7*, 3 (1986), 425 – 440.

[22] Tarjan, R. E., and Trojanowski, A. E. Finding a maximum independent set. *SIAM J. Comput.*.

[23] Tomita, E., and Kameda, T. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization 37*, 1 (2007), 95–111.