Universidade Federal Fluminense

ARIEL ALVES DA FONSECA

Análise Experimental do Controle de Fluxo de Mensagens na Execução Paralela de Aplicações em Grades Computacionais

NITERÓI

2010

ARIEL ALVES DA FONSECA

Análise Experimental do Controle de Fluxo de Mensagens na Execução Paralela de Aplicações em Grades Computacionais

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de ou Mestre. Área de concentração: Redes de Computadores e Sistemas Distribuídos e Paralelos.

Orientador:

Vinod Rebello, PhD

Universidade Federal Fluminense

NITERÓI

2010

Análise Experimental do Controle de Fluxo de Mensagens na Execução Paralela de Aplicações em Grades Computacionais

Ariel Alves da Fonseca

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre.

Aprovada por:

Prof. Vinod Rebello / IC-UFF (Presidente)

Profª. Maria Cristina Silva Boeres / IC-UFF

Prof. Felipe Maia Galvão França / COPPE-UFRJ

Niterói, abril de 2010.

Para meu pais, meus irmãos e orientador pela apoio nos momentos mais difíceis.

Agradecimentos

Foram várias pessoas que de forma direta e indireta contribuíram para a realização do meu sonho de fazer um mestrado. Por isso, não poderia esquecê-las nesse momento, inicialmente gostaria de agradecer a Deus por me permitir chegar até aqui. Na sequência aos meus pais e irmãos por sempre me apoiar. Tenho um agradecimento especial ao meu orientador pelas contribuições e dicas que foram importantes para a realização desta dissertação. Gostaria também agradecer ao Alexandre, Aline e Jacques que foram fundamentais para a conclusão deste trabalho.

Resumo

A utilização de grades computacionais para a execução de aplicações paralela é cada vez mais empregada para oferecer um alto poder computacional a um baixo custo. Porém, como os ambientes grades possuem natureza heterogênea, dinâmica, compartilhada e distribuída, adaptar as aplicações para a execução nesse tipo de ambiente de forma a obter um bom desempenho não é uma tarefa simples. Administrar a utilização da rede comunicação evitando a ocorrência de congestionamento de mensagens e permitindo uma eficiente utilização da capacidade de processamento e armazenamento oferecida por uma grade, é um desafio ainda maior quando esses recursos cujas características variam são utilizados por pesquisadores que não possuem conhecimento, nem habilidade para gerenciar ambientes computacionais complexos.

Uma abordagem já utilizada é adotar um sistema de gerencia de aplicações, como por exemplo, o SGA EasyGrid, que esconda do usuário as dificuldades existentes na utilização de uma grade computacional, permitindo que aplicações paralelas possam utilizar as grades computacionais sem grandes esforços. Esse nível de gerenciamento normalmente é capaz de criar aplicações que sejam auto-gerenciáveis se ajustando às mudanças que ocorram no ambiente fornecendo funcionalidades como auto-escalonamento e auto-recuperação de falhas. Entretanto, uma importante atribuição para a gerência da execução de uma aplicação paralela é administrar não só os recursos computacionais, mas também os recursos de comunicação e de armazenamento eficientemente com um mínimo de impacto adverso na execução da aplicação.

Nessa dissertação será estudado o problema de congestionamento da rede de comunicação ocasionado pela troca excessiva de mensagens durante a execução e gerência da aplicação. Será proposta uma nova camada dentro o gerenciador de aplicação *SGA EasyGrid*, permitindo a execução das aplicações autônomas MPI de grande escala sem o risco de congestionamento de mensagens alem de não deixar de executar devido a falta de memória para guardar os dados da aplicação.

Abstract

Grid Computing is now extensively being adopted to provide high computational power at low cost for parallel applications. As computational grids are in essence heterogeneous, dynamic, shared and distributed environments, developing grid enabled applications is extremely complex. The difficulty of managing network communication to avoid message congestion and allow the efficient use of the processing power and storage being offered, is even more acute for non-expert users given their lack of familiarity with the intricacies and complexities of computational grids.

A promising approach already being used to address these issues is the creation of autonomic applications through the adoption of Application Management Systems (AMS) that hide from the programmer and user the difficulties related to the grid computing, thus allowing parallel applications to execute efficiently on the grid without much effort. These grid systems generally offer functionalities such as self scheduling and tolerance to resource failures. However, an important aspect of controlling the execution of an application is managing not only processing resources but also communication and storage efficiently with minimal impact on performance. This dissertation focuses on the problem of managing network congestion caused by excessive communication message exchange during the execution of autonomic applications. This work proposes a new layer within the EasyGrid AMS to allow the execution of large scale MPI autonomic applications without the risk of message congestion or even execution failure due to a lack of memory space for the application.

Palavras-chave

1. Grades Computacionais
2. Controle do Fluxo de Mensagem
3. Gerenciamento de Memória da Aplicação
4. Aplicação MPI

Glossário

AMS	:	*Application Management System*
BoT	:	*Bag of Tasks*
GAD	:	Grafo Acíclico Direcionado
GM	:	*Global Manager*
HM	:	*Host Manager*
SGA	:	Sistema Gerenciador de Aplicações
SM	:	*Site Manager*

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A crescente necessidade da sociedade em utilizar computadores [25], coloca a computação em posição de destaque no cenário mundial. Os avanços tecnológicos têm permitido o aumento da capacidade de processamento e armazenamento, no entanto, a quantidade de problemas complexos que são tratados pela comunidade científica também cresce, exigindo um demanda computacional que pode não ser disponibilizada por uma única máquina de uma maneira fácil ou trivial.

Nos últimos anos, graça aos avanços tecnológicos do poder computacional dos computadores pessoais, a redução dos gastos de produção dos microprocessadores e a redução da latência em redes, surgiu-se como um novo desafio o estudo de formas de utilização de recursos de processamento de menor capacidade que estejam conectados através de uma rede de computadores.

A utilização de supercomputadores é a alternativa mais simples para a obtenção de um alto desempenho, mas o investimento necessário para a compra desse tipo de hardware pode inviabilizar essa possibilidade. Além disso, alguns supercomputadores requerem do desenvolvedor um conhecimento específico da arquitetura do sistema paralelo onde será executada a aplicação.

A utilização de sistemas distribuídos e computação paralela surgiu como uma solução de menor custo para a aquisição e manutenção de máquinas paralelas [17]. Com a computação em *cluster* [7] os pesquisadores tiveram acesso a recursos computacionais de pequeno e médio porte a um custo acessível. Os *clusters* costumam ser construídos através da interligação de computadores homogêneos e dedicados. Por isso, aumentar a capacidade computacional de um *cluster* pode não ser uma tarefa simples, pois esses sistemas são compostos de recursos homogêneos e como a evolução dos computadores é

constante, dificilmente alguns meses após a montagem de um *cluster* será possível obter o mesmo hardware adquirido no momento de sua criação. A compra de um novo *cluster* é a estratégia mais utilizada para a evolução desse tipo de sistema, mas tem como grande inconveniente a perda do investimento feito no *cluster* que será substituído.

Na tentativa de retirar proveito de múltiplos *clusters* juntos nasceu as grades computacionais [22] que diferentemente dos *clusters*, os recursos são heterogêneos e não são necessariamente dedicados. Devido a natureza dinâmica de uma grade, o gerenciamento de uma aplicação nesse ambiente costuma ser mais difícil do que em um *cluster*. Vários fatores precisam ser avaliados durante a execução da aplicação para uma utilização eficiente dos recursos de uma grade, como por exemplo:

- Controlar os recursos disponíveis - Não existe qualquer restrição que obrigue os recursos integrantes de uma grade a estarem sempre disponíveis para a execução. Uma falha no momento da execução paralela, por exemplo, poderia forçar a aplicação a rever todo a escalonamento das tarefas e re-executar toda a aplicação [13].

- O escalonamento das tarefas - Como as máquinas não são dedicadas um escalonamento obtido estaticamente provavelmente não seria uma boa opção para a execução da aplicação caso os recursos da grade começassem a ser compartilhados por outras aplicações.

- O tempo e a banda de comunicação pertencentes a uma mesma grade podem variar durante a execução, aumentando o tempo de envio das tarefas além do previsto estaticamente.

Para simplificar o desenvolvimento de aplicações para grades os pesquisadores têm gasto esforços na criação de uma camada de *software*, que tenta esconder do desenvolvedor da aplicação as particularidades da grade computacional. Esse *middleware*, também chamado de gerenciador de aplicação, deverá ser capaz de oferecer mecanismos de tolerância a falha, escalonamento dinâmico de tarefas e balanceamento dinâmico de carga sem trazer impactos adversos no tempo de execução da aplicação do usuário. Além dos mecanismos citados nesse parágrafo, algumas aplicações devido as suas características podem requerer que o gerenciador da aplicação também realize um controle dos dados enviados na rede comunicação, de forma a evitar que aplicações com grande demanda de comunicação possam congestionar a rede de computadores.

Para o devido gerenciamento desse tipo de aplicação é necessário coordenar a troca de

mensagens dinamicamente evitando a ocorrência de congestionamento entre as máquinas da rede utilizada.

Nesta dissertação será proposta uma nova camada de gerenciamento da rede de comunicação que conhecerá a arquitetura da grade possibilitando ao gerenciador aceitar apenas a quantidade de trabalho que seja capaz de controlar e executar. Essa camada será fundamental para a execução eficiente de aplicações com grande número de mensagens ou quando o tamanho das mensagens enviadas são grandes. Além do controle da rede de comunicação, a utilização da memória principal foi também abordado nesse trabalho, e assim outras implementações foram realizadas com o objetivo de permitir a execução de aplicações grande, com mais de 10 mil tarefas, e ainda obter um bom tempo de execução evitando falha na execução devido a falta de recurso de armazenamento.

1.1 Motivação

O esforço requerido para reescrever o código MPI de uma aplicação e incluir ferramentas de gerência para uma execução eficiente e segura em um ambiente grades computacionais pode, algumas vezes, desencorajar a utilização desse ambiente para a execução paralela. Nesse contexto, torna-se necessária a criação de uma ferramenta que possibilite administrar a demanda da aplicação de forma eficiente e robusta sem que seja necessária qualquer alteração no código fonte da aplicação, um sistema *system aware*. Assim, através da incorporação de um *middleware* ao código fonte da aplicação, essa poderá usufruir de serviços como balanceamento dinâmico de carga, identificação de falhas e monitoramento de forma automática.

Essa camada de gerenciadora de aplicação precisará também ocultar do desenvolvedor da aplicação detalhes da arquitetura e da rede de comunicação utilizada. Para isso, o gerenciador precisar ser capaz de identificar o comportamento da aplicação para evitar que a aplicação inunde a rede de comunicação com um grande número de mensagens ou um tráfego de dados pesado, caso isso não seja necessário. Garantindo que a comunicação entre as máquinas pertencentes ao ambiente paralelo funcione conforme previsto durante a execução da aplicação do usuário.

O *middleware* proposto em [12] [16] [39] foi utilizado como base para a incorporação de um esquema de controle do fluxo de mensagem. Esse *middleware* é composto por uma estrutura hierárquica, onde os gerenciadores se comunicam entre si, para controlar as tarefas da aplicação. A utilização desse *middleware*, garante a aplicação uma execução

eficiente e confiável para aplicações pequenas, por isso, a inclusão da nova camada de controle de fluxo de mensagens permite estender a eficiência já observada e comprovada também para as aplicações com grande número de tarefas.

1.2 Contribuicao

A principal contribuição deste trabalho é propor e validar um modelo de controle de fluxo de mensagens para a execução de aplicações paralelas com grande número de tarefas em *cluster* ou grades computacionais. A política de controle de fluxo proposta nessa dissertação, inclui no sistema gerenciador da aplicação uma nova camada de gerência que garantirá as aplicações BoTs, a execução de forma uniforme, reduzindo o risco de erros, atrasos relacionado ao congestionamentos das mensagens ou devido a falta de memória principal.

Para estudar a criação de uma política de controle de fluxo eficiente, foi incorporada uma nova camada no SGA *EasyGrid*, customizando a biblioteca responsável pela comunicação entre a aplicação do usuário e o gerenciador do nível mais alto da hierarquia de gerenciadores.

Com a utilização da nova camada que controla a comunicação, o SGA *EasyGrid* poderá oferecer ao seus usuários uma maior eficácia no tempo de execução de um número maior de aplicações paralelas. A utilização do controle de fluxo, fez com que a troca de mensagens deixasse de ser o principal limitante para a execução de aplicações de grande escala, direcionando as atenções para o uso da memória principal. Por isso, uma segunda contribuição deste trabalho é o estudo e a implementação de estratégia para melhorar a utilização dos recursos de armazenamento, assim como, apresentar a utilização do controle de fluxo para administrar parte da memória principal utilizada, como será descrito nos próximos capítulos.

1.3 Organizacao

Esta dissertação está dividida da seguinte forma, no Capitulo 1 foi apresentada as principais razões para a realização deste trabalho. No Capitulo 2 será apresentado uma visão geral do escalonamento de tarefas e conceitos básicos da execução de aplicações em grades computacionais e em uma rede de comunicação. No Capítulo 3 será feita uma breve descrição do Sistema Gerenciador de Aplicação (SGA). No Capítulo 4 será discutido o modelo

de controle de fluxo de mensagem que constitui a principal contribuição deste trabalho e o Capitulo 5 destina-se a apresentar uma análise dos resultados obtidos. A conclusão desta dissertação e trabalhos futuros são apresentadas no Capítulo 6.

Capítulo 2

Computação em Grades

Este capítulo abordará os conceitos básicos sobre a computação em grades computacionais e uma visão geral sobre o problema de escalonamento de tarefas, em seguida serão apresentados o protocolo de comunicação utilizado, a biblioteca de comunicação MPI e as características das aplicações BoTs (Bag Of Task).

2.1 Escalonamento de Tarefas

O problema de escalonamento de tarefas consiste na distribuição dos processos de uma aplicação paralela entre os múltiplos processadores de forma a atingir um desempenho desejado. Geralmente os principais objetivos são minimizar o tempo de execução (*makespan*), minimizar os atrasos de comunicação e melhorar a utilização dos recursos. O problema de escalonamento de tarefas de forma geral é dito NP-Completo [23, 44, 48] e se torna ainda mais difícil quando a execução ocorre em um ambiente de grade computacionais, onde os recursos são heterogêneos, de natureza dinâmica e compartilhada.

O escalonamento de tarefas pode acontecer em tempo de compilação, chamado de escalonamento estático, ou em tempo de execução chamado de escalonamento dinâmico. A maior vantagem do escalonamento estático é não concorrer por recursos de processamento durante a execução da aplicação, permitindo que a aplicação possa utilizar o processador por mais tempo. Entretanto, a dificuldade de estimar a quantidade de tempo que cada tarefa gastará e atrasos que poderão ocorrer na comunicação fazem com que o escalonamento estático dificilmente consiga prever com fidelidade a execução real da aplicação. Uma outra razão que pode dificultar a previsão do escalonamento estático está relacionada a característica dinâmica inerente a da execução de aplicações em ambientes reais, que se torne extremamente difícil desenvolver um algoritmo que identifique estaticamente

as possíveis variações no custo computacional e custos de comunicação que ocorrerão no momento da execução.

O escalonamento dinâmico por ocorre junto com a execução da aplicação do usuário, e por isso compete por recursos de processamento para coletar informações de controle sobre o progresso da execução e viabilizar a tomada de decisão de onde alocar ou realocar uma tarefa em tempo de execução. Por essa razão, é importante que o escalonador dinâmico tenha um algoritmo simples, fornecendo uma rápida resposta quando for necessária a revisão do escalonamento estático para não competir excessivamente por recursos de processamento com a aplicação. Uma grande vantagem do escalonamento dinâmico em relação ao estático é não obrigar o gerenciador a conhecer todo o comportamento da execução da aplicação [12] para realizar o escalonamento. Para as aplicações que criam novas tarefas dinamicamente, dependendo dos resultados obtidos na execução, apenas o escalonamento dinâmico poderá ser utilizado. Em ambiente grade computacional onde a quantidade de recursos de processamento e comunicação muda de forma inesperada o escalonamento dinâmico é fundamental para a obtenção de um bom escalonamento.

2.2 A camada de Rede - O Protocolo da Internet(IP)

Existem duas versões do protocolo IP em uso hoje, a versão 4 que é a mais utilizadas, denominada por IPv4 [26, 46] e possui endereços de 32 bits, e a versão 6 do protocolo IP, denomina por IPv6 [1, 27] que possui endereço de 128 bits.

A comunicação em uma rede IP é feita através do envio de pacotes da camada de rede, denominados de datagramas. O protocolo IP oferece um serviço não confiável de transmissão, chamado de melhor esforçom, por isso, os pacotes podem: chegar desordenados, serem duplicados ou mesmo serem perdidos. O cabeçalho de uma datagrama IP de 32 bits possui geralmente o tamanho de 20 bytes, quando os campos opcionais não são utilizados, e é formado pelos campos: versão, comprimento do cabeçalho, tipo de serviço, comprimento do datagrama, identificador de 16 bits, flags, deslocamento de fragmentação(13 bits), tempo de vida, protocolo da camada superior, soma de verificação do cabeçalho, endereço IP de 32 bits da fonte, endereço IP de 32 bits do destino, opções e dados. Maiores detalhes sobre cada campo pode ser obtido em [46].

O protocolo IP é responsável por transportar dados na rede e a quantidade máxima de dados que um quadro da camada de enlace pode carregar é denominada de unidade máxima de transmissão (Maximum Transmission Unit - MTU). Para a transmissão de uma

informação é comum que vários roteadores sejam utilizados. Em uma rede de Internet cada roteador pode possui as suas configurações próprias, como o seu tamanho do MTU [35], assim pode ocorre a fragmentação de um datagrama quando um roteador adotar um MTU menor do que o utilizado pelo datagrama recebido. A aplicação de destino utilizará os dados do cabeçalho dos datagramas para montar o datagrama original.

Um roteador pode eventualmente descartar pacotes quando o seu espaço de *buffer* estiver exaurido. A perda de um pacote pode acontecer na fila da porta de entrada ou na fila de saída do roteador e dependerá da carga de tráfego, da velocidade relativa do elemento de comutação e da taxa do elemento de comutação. Essa última taxa citada, representa a velocidade na que o elemento de comutação pode movimentar pacotes de suas portas de entrada até as portas de saída. As filas dentro do roteador acontecem quando a vazão com que os dados são recebidos é maior do que os dados são enviados.

Uma consequência da fila na porta de saída é que um escalonador de pacotes precisa ser utilizado para escolher o melhor pacote dentre os que estão na fila para a transmissão. Algumas estratégias de escalonamento podem ser utilizadas como:

- FCFS (First Come First Server) - A ordem com que os pacotes chegam é a mesma que os pacotes são enviados.
- WFQ (Weighted Fair Queuing) - Compartilha o enlace de saída com justiça entre as diferentes conexões fim-a-fim que têm pacotes na fila para transmissão. Nessa estratégia cada pacote de mesma classe são tratados através da estratégia de FCFS. Para cada envio é feita uma varredura cíclicas das filas que possuem pacotes de mesma classe onde é transmitido o primeiro pacote da fila que está a mais tempo sem ser atendido.

A possibilidade de perda de pacotes na camada de rede obriga a camada de transporte a controlar e se necessário reenviar os pacotes descartados, desta forma é possível garantir a confiabilidade da entrega dos dados tramitidos a aplicação, conforme veremos na próxima seção.

2.3 A Camada de Transporte - o TCP

O protocolo TCP (Transmission Control Protocol) foi criado com o objetivo de realizar a intercomunicação de computadores. O TCP [2, 6, 32, 36, 38] é o protocolo da camada de

transporte orientado à conexão, que oferece um serviço confiável. O TCP é um protocolo de propósito geral que pode ser alterado para ser usado com uma variedade de sistemas.

As principais características do TCP são:

- Orientado à conexão - A aplicação envia um pedido de conexão para o destino e usa a conexão para transferir dados.
- Ponto a ponto - uma conexão TCP é estabelecida entre dois processos. Cada conexão TCP possuem *buffers* próprios para o envio e recebimento de mensagens.
- Confiabilidade - O TCP utiliza várias técnicas para proporcionar uma entrega confiável dos pacotes de dados, que é a sua grande vantagem em relação ao UDP. O TCP permite a recuperação de pacotes perdidos pela camada inferior, a eliminação de pacotes duplicados, a recuperação de dados corrompidos, e pode recuperar a conexão em caso de problemas comunicação com a rede.
- Full duplex - É possível a transferência simultânea em ambas direções (cliente-servidor) durante toda a sessão.
- Apresentação de 3 vias - Mecanismo de estabelecimento e finalização de conexão onde são trocadas 3 mensagens que sincronizam a comunicação, definindo o número de sequência que será utilizado por cada aplicação para a comunicação e a reserva dos *buffers* e variáveis da conexão.
- Entrega ordenada - A aplicação faz a entrega ao TCP de blocos de dados com um tamanho arbitrário, tipicamente em octetos. O TCP divide esses dados em segmentos de tamanho especificado pelo valor MTU (Maximum Transmission Unit) [37]. A circulação dos pacotes ao longo da rede pode fazer com que os pacotes não cheguem ordenados. O TCP garante a reconstrução dos dados no destinatário utilizando os números de sequência.
- Controle de fluxo - O TCP usa o campo janela para controlar o fluxo. O receptor, à medida que recebe os dados, envia mensagens ACK (*Acknowledgement*), confirmando a recepção de um segmento. O receptor pode especificar junto a mensagens de ACK o tamanho máximo do *buffer* no campo (janela) do segmento TCP, determinando a quantidade máxima de *bytes* disponível em seu *buffer* de recebimento. O transmissor somente poderá enviar segmentos com tamanho menor ou igual que a sua janela e a do receptor. Comumente o tamanho da janela dos *buffer* de envio e recebimento no ambiente Linux é $128 Kbytes$ [38].

- Controle de congestionamento - O TCP utiliza a janela de controle de congestionamento para evitar a sobrecarga da rede. A quantidade de dados não reconhecidos não pode exceder ao mínimo entre o tamanho da janela de congestionamento e o tamanho da janela utilizado pelo controle de fluxo. O tamanho da janela de congestionamento inicialmente possui tamanho de 1 MSS (Maximum Segment Size) [45] e esse valor é aumentado exponencialmente, duplicando o valor da janela de congestionamento a cada RTT (Round Trip Time), até quando ocorrer um evento de perda de mensagem. A partir desse momento a janela de congestionamento é reduzida pela metade e começa a aumentar linearmente a janela de congestionamento em 1 MSS a cada RTT [2].

2.4 MPI

O MPI (Message Passing Interface) é uma biblioteca de troca de mensagem que utiliza o protocolo TCP na camada inferior. As principais funções, sub-rotinas e métodos do MPI são desenvolvidas utilizando, C, C++, Fortan-77 e Fortran-95 [18]. Os principais objetivos do MPI são:

- Permitir a execução de programas paralelos mesmo em sistemas heterogêneos.
- Permitir a comunicação eficiente entre processadores paralelos.
- Construir uma interface para programação paralela sem a necessidade de utilização de compiladores ou bibliotecas de sistema específicas.

A primeira versão do MPI, o MPI1 [20], é formada por 128 funções para a comunicação entre processos, integração e gerenciamento com o ambiente e comunicação de grupo. Nessa versão do MPI, os grupos de processos são estático, isso significa que nenhum novo processo pode ser acrescentado a um grupo após a a sua criação. Devido a essa característica, o MPI1, se torna incompatível com as necessidades para o gerenciamento dinâmico existentes em ambientes heterogêneos como uma grade computacional.

A segunda versão do MPI, o MPI2 [19], possui 200 novas funções e foi projetado para permitir a criação e a cooperação entre processos mesmo durante a execução da aplicação já ter iniciada. Através da função *MPI_Comm_spawn* é possível a criação de processos dinamicamente e estabelecimento de conexão com eles, retornando um intercomunicador contendo o processo pai no grupo local e o processo filho remoto. O *MPI_Comm_spawn*

ainda permite o balanceamento dinâmico da carga, pois é permitida a escolha da maquina onde a tarefa será disparada.

O MPI utiliza 3 protocolos para o envio de mensagens na rede:

- Short - Este protocolo transfere o cabeçalho MPI e o corpo de mensagem dentro de um simples pacote. O tamanho total da mensagem MPI é limitada pelo MTU, nesse caso o remetente não espera a mensagem ser retirada do *buffer* de recebimento para continuar a sua execução, e por isso esse protocolo é chamado de não bloqueante. O padrão é que mensagens com tamanho até 52104 bytes utilizam esse protocolo.

- Eager - Este protocolo transfere o corpo e o cabeçalho da mensagem em pacotes distintos. Como no protocolo *short*, essa mensagem não é bloqueante, pois o remetente não espera o destino confirmar o recebimento de qualquer uma das mensagens para continuar a sua computação. Para a mensagem utilizar esse protocolo é necessário que a mensagem esteja entre 52105 bytes e 65536 bytes.

- Rendezvous - Utilizam este protocolo o envio de mensagem com mais de 65536 bytes. A principal diferença deste protocolo em relação aos outros é que as mensagens são sincronizadas. Primeiro é enviado o cabeçalho da mensagem para o destino, apenas apos o destino responder que está pronto para receber a mensagem o remetente envia um novo pacote contendo os dados da mensagem.

2.5 LAM/MPI uma Versão do Padrão MPI

O LAM/MPI não é somente uma biblioteca que implementa o MPI padrão, mas também estabelece o ambiente de execução LAM/MPI. Esse ambiente LAM é baseado em *daemons* (processos que executam em *background* no sistema), que fornecem muitos dos serviços requeridos por programas MPI. LAM/MPI utiliza um *daemons* no nível do usuário (lamd) sob cada máquina, para criar um ambiente de execução persistente. Uma comunicação persistente tem o objetivo de reduzir o tempo de comunicação, pois permite que os argumento utilizados na comunicação sejam criados uma única vez e seja reutilizado a cada novo envio ou recebimento [49].

Um conjunto de daemons *lamd* é iniciado pelo comando *lamboot*, esse comando envia uma solicitação para cada máquina, tipicamente via *rsh* ou *ssh*, para que cada uma delas inicie um *lamd* local. Quando um *lamd* é criado, ele é preenchido com uma tabela de rotas contendo os endereços IPs dos outros processos *lamds*. Os processos *daemons* são sempre

usados na criação de processos MPI, mas podem ou não atuar como intermediários na comunicação.

Após a chamada à função MPI Init(), todos os processos MPI ficam cientes da existência dos outros processos MPI, formado por um comunicador consistente (MPI COMM WORLD). Todos os processos nesse comunicador precisam executar o MPI_Finalize() antes de terminar a execução. As chamadas às funções MPI Init() são consideradas ponto de sincronização entre os processos MPI.

A distribuição LAM permite a utilização de arquivos de esquemas, chamado de *appschemas*, nas funções *MPI_COMM_SPAWN()*. O *appschema* é um arquivo ASCII que permite ao programador especificar um conjunto arbitrário de CPUs ou nós onde serão disparados processos MPI. Caso o programador da aplicação não especifique nenhum arquivo de esquema para a criação dinâmica de processos, o LAM escalona os processos segundo o algoritmo *round robin*, ou seja, atribui um processo para cada nó LAM, a partir da CPU 0 (ou com menor identificador) e continua até a CPU com maior identificador. Quando o número de processos da aplicação é maior que o número de CPUs, o procedimento de criação de processos se repete a partir da CPU 0.

Atualmente, com o surgimento dos ambientes grades, novas implementações do padrão MPI vêm sendo propostas para atender aos requisitos específicos desse novo ambiente. A implementação LAM/MPI a partir da versão 7.0 passou a fornecer suporte para o Globus Toolkit [21] que permite a execução de aplicações paralelas em grades computacionais. Outra versão também habilitada para grade é o MPICH-G2, que é uma extensão da implementação MPICH. A versão LAM/MPI apresenta várias vantagens em relação ao MPICH-G2, em particular, o suporte às funções de gerenciamento dinâmico de processos através do comando *MPI_COMM_SPAWN()*, essencial para ambientes dinâmicos como os grades.

2.5.1 Modelo de Comunicação

O MPI oferece vários modelos de comunicação que permitem controlar o envio e o recebimento de mensagens. Dependendo das necessidades e características da aplicação podem ser utilizados modelos de comunicação onde o MPI guarda ou não a mensagem antes de enviar. Segue abaixo um detalhamento sobre as possíveis formas de envio de mensagem que o MPI disponibiliza:

- *Send* Pronto: A mensagem é enviada tão logo possível. Esse tipo de *Send* é chamado

de não bloqueante.

- *Send* Síncrono: Antes do envio de uma mensagem é enviada uma solicitação para o envio. Somente quando a aplicação destino estiver pronta para receber a mensagem o destinatário enviará a permissão para o envio da mensagem acontecer. Esse tipo de *Send* é chamado de bloqueante.

- *Send* Padrão: Pode ser utilizado o protocolo de comunicação do *Send* Pronto para mensagem curta e o protocolo do *Send* Síncrono para as mensagens maiores.

- *Send Buffered*: A remetente copia a mensagem dentro do *buffer* de envio e depois envia a mensagem como um *Send* não bloqueante.

Algumas operações de comunicação quando ficam pendentes podem consumir os limitados recursos do sistema em particular a memória do sistema, chegando a causar erro na execução da aplicação. O MPI deixa como responsabilidade da aplicação a gerência da quantidade de recursos pendentes para o envio e recebimento de mensagens. O desenvolvedor da aplicação deverá reservar recursos suficientes para suportadas o número máximo de envio que podem ficar pendentes aguardando a confirmação de recebimento pelo destino.

O MPI utilizando o modelo *Send Buffered* para permitir ao usuário definir um espaço em memória para as mensagens enviadas mas sem a confirmação de recebimento. Caso o número de mensagens pendentes ultrapasse o tamanho do *buffer* definido pelo usuário ocorrerá um *overflow* do *buffer* e um erro será sinalizado. Isso pode acontecer, por exemplo, no cenário onde uma aplicação repetidamente envia novos dados e a aplicação que recebe os dados não as utiliza na mesma velocidade.

Já um comando *MPI_Send()* padrão não completará caso o somatório dos tamanhos das mensagens pendentes no sistema ultrapasse o tamanho dos *buffers* do TCP de envio e recebimento. Nesse caso o comando *MPI_Send()* aguardará a liberação de quantidade suficiente de espaço no seu *buffer* do TCP para concluir. Assim, mesmo a utilização do comando *MPI_Send()* pode gerar falha na execução, caso a tarefa mestre e a tarefa escrava estejam com os *buffer* do TCP cheios e tentaram enviar uma mensagem um para o outro. O comando MPI_Send nunca concluirá, pois ambas as aplicações precisariam receber mensagens para liberar espaço suficiente para a conclusão do comando *MPI_Send()*.

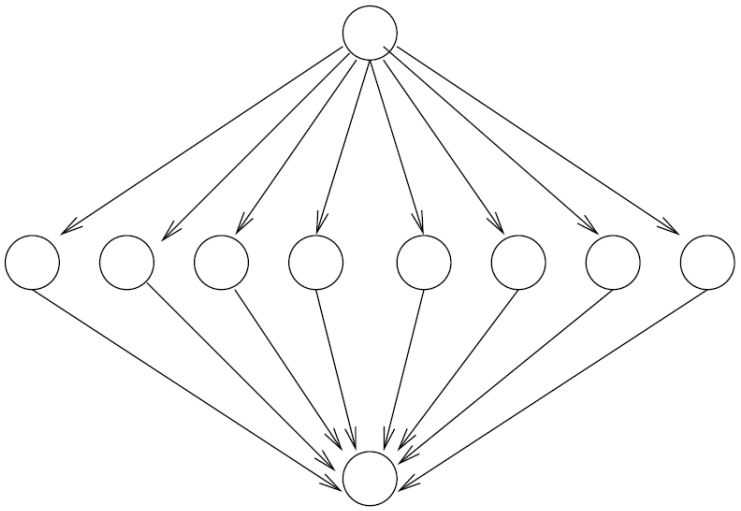

Figura 2.1: Representação de uma aplicação BOT em forma de GAD

2.6 Aplicações Bag of Tasks (BoTs)

As aplicações *Bag-of-Tasks (BoTs)* [8, 10, 30] se caracterizam por serem compostas por tarefas independentes [28, 50] que podem ser executadas sem uma ordem pré-definida. Vários tipos de aplicações paralelas são categorizadas como BoT, por exemplo, as aplicações mestre/escravo [11] que são constituídas de um mestre que distribui trabalho para os escravos processarem, como por exemplo, as aplicações SETI@home[4] e *parameter sweep* [9] . Com uma topologia simples e ausência de comunicação entre os escravos, as aplicações BoTs são amplamente executadas em ambientes heterogêneos, inclusive em grades computacionais. As aplicações BoTs podem ser facilmente representadas por uma GAD do tipo *fork/join*, conforme representado pela Figura 2.1, onde o nó fonte/destino é usado apenas para enviar/recolher dados das tarefas mestre/escravo.

A falta de dependência entre as tarefas permite ao gerenciador algumas facilidades como, reexecução das tarefas em qualquer ordem e para qualquer máquina e caso ocorra um atraso na execução de uma tarefa isso não implicar em impacto na execução de outras tarefas da aplicação. Entretanto, mesmo sem considerar as restrições de precedência, o problema de encontrar o escalonamento ótimo ou um bom escalonamento ainda é difícil [29] [31] [39].

2.7 Resumo

Esse capítulo apresentou de forma sucinta o conceito de escalonamento de tarefa, as ferramenta de comunicação utilizadas na rede, a biblioteca de comunicação MPI, os modos de comunicação disponibilizados por essa biblioteca e a aplicação utilizada nesse trabalho. Os conceitos apresentados aqui serão importante para o entendimento de como acontece o congestionamento de mensagens em aplicação que utilizam o MPI. No próximo capítulo serão introduzidas as características principais do SGA, o sistema gerenciador que foi utilizado como base nesse trabalho para a inclusão do controle de fluxo de mensagens.

Capítulo 3

Sistema de Gerenciamento de Aplicação

Neste capítulo será apresentado o sistema gerenciador de aplicação e os seus serviços para a execução de aplicações em Grades Computacionais, como monitoramento, escalonamento de tarefas e tolerância a falhas. Serão apresentadas também as características do SGA *EasyGrid* que fazem com que o controle de fluxo de mensagens seja um ferramenta obrigatória para a garantia de sucesso da execução da aplicação.

3.1 SGA EasyGrid

O SGA *EasyGrid* [12, 16, 39, 40], é um Sistema Gerenciador de Aplicação (SGA) que possui uma camada de serviço que gerência a aplicação para a execução de forma automática. O SGA é embutido automaticamente na aplicação MPI em tempo de compilação, fornecendo serviços de criação dinâmica, escalonamento dinâmico e tolerância a falhas. Isso torna as aplicações capazes de se adaptarem às características mais comuns das grades computacionais. O projeto de construção do SGA iniciou em 2004 com o objetivo de criar um *framework* para permitir que aplicações MPI criadas para *cluster* possam ser executadas em ambientes grades. O SGA tem como atribuição fornecer os serviços de gerenciamento básico da aplicação realizando a auto-adaptação às mudanças do sistema, frequentes em ambientes onde os recursos não são dedicados como as grades computacionais.

O SGA *EasyGrid* utiliza uma arquitetura hierárquica de processos gerenciadores dividida em três níveis gerenciais, conforme é mostrado na figura 3.1. Todos os níveis possuem serviços de monitoramento, gerenciamento e comunicação entre processos, escalonamento dinâmico e tolerância a falhas [12, 39] e são embutidos automaticamente na aplicação do usuário, permitindo que essas aplicações sejam capazes de se auto gerenciar (*autonomic*

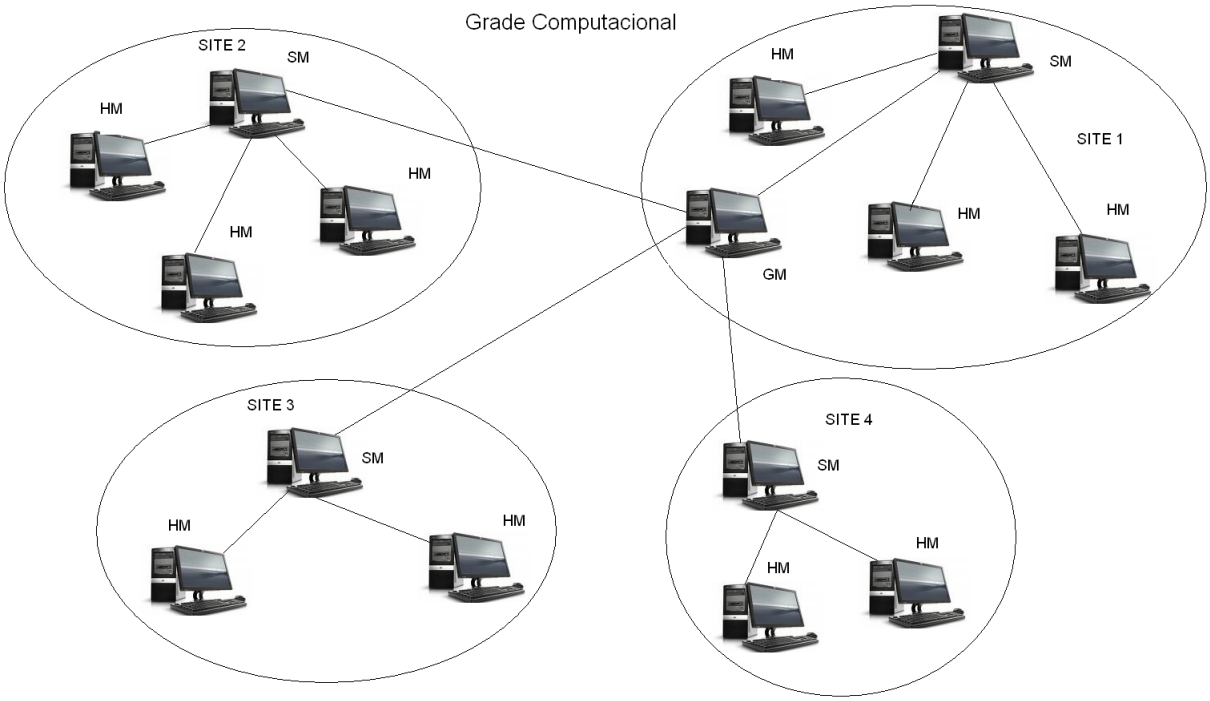

Figura 3.1: Hierarquia dos gerenciadores do SGA *EasyGrid* [16]

application). No nível zero está o *Global Manager*(GM) responsável por gerenciar todos os *sites* envolvidos na execução da aplicação. No nível 1 estão os *Sites Manage*(SM) que respondem pela alocação dos processos da aplicação nos seus recursos. No último nível, o nível 2, estão os *Host Manager*(HM) que são responsáveis pelo escalonamento, criação e execução dos processos atribuídos a cada máquina [16, 43].

Cada gerenciador do SGA é estruturado em camadas, conforme apresentado na figura 4.3 e a cada camada é atribuído um conjunto de funcionalidades que dependendo do nível do processo gerenciador na estrutura hierárquica. A camada mais baixa do SGA, a **Gerenciamento de Processos**, implementa as funcionalidades de gerência dos processos, criação dinâmica dos processos MPI da aplicação do usuário e o redirecionamento de mensagem . A camada de **Monitoramento da Aplicação** é responsável por fornecer informações necessárias para o reescalonamento e para a detecção de falhas de processos da aplicação essa camada é utilizada pelas camadas de escalonamento dinâmico e de tolerância a falhas [39]. A camada de **Escalonamento Dinâmico** realiza o balanceamento de carga no ambiente e a camada de **Tolerância a Falhas** possui a função de identificar e tratar as falhas [13].

3.1.1 Gerenciador Global

O gerenciador global, também chamado de *Global Manager* (GM), tem função fundamental para a execução da aplicação. A execução do SGA começa com um único processo, o gerenciador global da grade que tem o conhecimento de todas as máquinas pertencentes a arquitetura da grade que foi selecionada pelo usuário para a execução, e tem como principais atribuições criar os gerenciadores dos *sites*, identificar falhas nos SMs e controlar o processo de escalonamento e re-escalonamento entre os *sites*. O processo de criação dos gerenciadores, para a criação da arquitetura de gerência de processo, inicia-se disparando dinamicamente um processo gerenciador de *site* em cada *site*.

3.1.2 Gerenciador do Site

Os gerenciadores dos sites, também chamados de *Site Manager(SM)*, são criados dinamicamente pelo GM utilizando arquivos configurados pelo usuário ou pelo EasyGrid Portal [5], chamados de arquivos de esquema (*appschema*). A versão do MPI utilizada, LAM/MPI, permite a utilização de arquivos de esquema na chamada à função MPI_Comm_spawn() e desta forma é possível definir as máquinas que participarão de cada *site*. O SM tem uma função bastante similar ao GM, pois ele cria dinamicamente os gerenciadores das máquinas e administram as falhas e a carga de todas as máquinas existentes em seu *site*, além de realizar o escalonamento das tarefas entre os HM pertencentes aos *sites*.

3.1.3 Gerenciador da Máquina

Os gerenciadores das máquinas, também chamado de *Host Manager* (HM), são responsáveis apenas pela gerência das aplicações atribuídas à máquinas local, sem ter qualquer função direta sobre as demais máquinas envolvidas na execução. No momento da criação dos HM's, é feita a leitura de arquivos que especificam os parâmetros da execução da aplicação é o número máximo de processos da aplicação que podem ser executados simultâneamente na máquina onde foi disparado o HM. A capacidade de executar mais de um processo da aplicação concorrentemente na mesma máquina é uma importante característica do SGA, pois em processadores multi núcleos isso permite a aplicação do usuário utilizar todos os núcleos disponíveis na máquina, caso o usuário decida por isso.

3.1.4 O Portal do SGA

O *framework EasyGrid* possui um Portal de escalonamento que permite ao usuário escolher a politica de escalonamento e de tolerância a falhas apropriada para a aplicação, determinar uma alocação de processos inicial, compilar a aplicação, gerenciar credenciais de acesso a grade do usuário, criar o ambiente MPI, incluindo a transferência de arquivos necessários para garantir uma execução segura do gerenciador global, provendo um mecanismo de tolerância a falhas [5, 47].

3.1.5 Processos da Aplicação

Uma das primeiras tarefas dos gerenciadores GM, SM e HM é realizar a leitura dos arquivos com o escalonamento estático, isso permite identificar a ordem em que as tarefas devem ser executadas, as máquinas onde elas serão alocadas e as informações de precedência. Na versão do SGA anterior a esse trabalho, apenas o GM realizava a leitura desse arquivo e no momento da criação enviava todas as informações relacionadas as tarefas da aplicação e os parâmetros sobre o escalonamento definido pelo usuário utilizando o MPI.

Inicialmente é realizada uma triagem entre as tarefas para dividí-las entre prontas e pendentes, prontas serão todas as tarefas onde todos os dados necessários à sua execução estão disponíveis, essas tarefas serão inseridas na lista de prontos. As tarefas que precisam aguardar a execução de tarefas predecessoras ou aguardam uma mensagem do GM são inseridas na lista de pendentes. Para que uma tarefa possa realizar o seu processamento, é necessário que todos os dados de entrada estejam disponíveis, isso permite ao HM apenas criar os processos para as tarefas que já estão prontas para a execução. O número de processos prontos que são criados concorrentemente sob a gerência de uma mesmo HM é definido pelo usuário nos arquivos de esquema. Desta forma, o SGA não limita a aplicação do usuário a ter um número máximo de processos, e garante que a troca de contexto relacionada a concorrência de processos executando na mesma máquina possa ser administrada pelo usuário.

A criação dinâmica dos processos da aplicação apresenta várias vantagens em relação à tradicional criação estática do MPI, entre elas o controle do número de processos concorrentes em execução por *host*, a facilidade de re-escalonamento das tarefas ainda não criadas, a possibilidade de utilização de rotinas de tolerância a falhas para os processos que não executaram devidamente, e escalabilidade e eficiência na execução de aplicações de grande escala. [39, 41]

3.1.6 Comunicação entre os Processos da Aplicação

Com a utilização do SGA para a gerencia da aplicação não é permitida a comunicação diretamente entre processos, pois toda a troca de mensagens entre as tarefas da aplicação são realizadas através da hierarquia de gerenciadores, (GM, SM e HM). Intermediar a comunicação entre as tarefas é necessário para garantir tolerância a falhas e evitar que todos os processos da aplicação sejam criados no inicio da execução.

No SGA *EasyGrid* existem três diferentes tipos de comunicação entre as tarefas da aplicação do usuário:

- Entre tarefas que compartilham o mesmo recurso - uma tarefa t1 tenta enviar uma mensagem para a tarefa t2 e ambas são controladas pelo mesmo HM, apenas o HM gerencia a comunicação entre as duas tarefas. O tempo para a comunicação é dado pelo tempo para gerar, enviar e receber a mensagem.

- Entre tarefas que estão em recursos distintos do mesmo *site* - para o envio de uma mensagem entre uma tarefa t1 para uma tarefa t2 que estão em HM distinto é necessário o envolvimento do SM1 além dos HMs das duas tarefas. O HM1 recebe a mensagem da tarefa t1 e encaminha para o SM1, que encaminha para o HM2 de t2 para o posterior envio para t2. Nesse caso além do tempo gasto para cada processo gerar ou receber a mensagem existe a latência de comunicação entre os diferentes recursos.

- Entre tarefas que se encontram em recursos de sites diferentes - para a comunicação entre tarefas que estão em SM diferente todos os níveis da hierarquia de comunicação são envolvidos, assim, para o envio de uma mensagem da tarefa t1 para uma tarefa t2 que está em *sites* diferente de t1 o HM, SM e o GM são envolvidos. O HM recebe a mensagem da tarefa t1 e encaminha para o SM1, que encaminha para o GM e posteriormente para SM2, até ser encaminhado para HM2 e finalmente ser enviado para t2. Nesse caso além do tempo gasto para cada processo gerar ou receber a mensagem existe a latência de comunicação entre os diferentes recursos.

Quando uma tarefa envia uma mensagem e a tarefa destino ainda não foi criada o SGA guarda a mensagem até que a tarefa de destino seja criada. Esse controle é feito com a utilização de uma biblioteca, denominada por MEGmpi.h [16], que intercepta as chamadas de envio e recebimento de mensagens da aplicação substituindo-as por envio e recebimento redefinidas pela biblioteca. A biblioteca funciona como uma camada de

abstração (*wrapper*) que inclui funcionalidades adicionais as funções existentes na biblioteca do MPI. Essa camada de abstração é incluída na aplicação do usuário em tempo de compilação, necessitando apenas que seja declarada a biblioteca MEGmpi.h no fonte da aplicação. Maiores detalhes sobre a biblioteca citada podem ser encontrado em [16].

A inclusão da biblioteca MEGmpi.h garante o funcionamento do SGA, deixando a cargo dos processos gerenciadores a entrega das mensagens aos seus destinos, independente da máquina em que elas estejam executando. Desta forma, toda a comunicação entre as tarefas da aplicação e o GM e entre os gerenciadores é interceptada pela camada de abstração.

3.1.7 Mensagens de Monitoramento

As mensagens de monitoramento utilizadas pelo SGA *EasyGrid* ajudam a manter informações atualizadas tanto sobre a aplicação em execução como sobre o desempenho dos recursos da grade. Essas mensagens são disparadas sempre que uma tarefa da aplicação do usuário termina a sua execução em um determinado recurso e seu término deve ser avisado aos gerenciadores do site e global. Juntamente com esta mensagem indicativa de fim de execução seguem informações como porcentagem de utilização da CPU, tempo estimado para finalizar a execução da aplicação no recurso, quantas tarefas estão executando no momento, etc.

É possível que seja ativado um monitoramento ainda mais detalhado do estado da aplicação do usuário. Desta forma, é informado o momento de criação de cada processo (*init*), quando são realizadas as trocas de mensagens (*send* e *receive*) e quando cada processo termina sua execução (*finalize*). Entretanto, tal abordagem é mais intrusiva, pois um número maior de mensagens circulará no sistema devido a cada mensagem extra de monitoramento. Logo, apenas a mensagem que indica o fim da execução de um processo é obrigatoriamente monitorada, dando suporte por exemplo, à camada de escalonamento dinâmico.

Pode ocorrer que as tarefas da aplicação demorem muito tempo para terminar sua execução e mensagens de *heartbeat* são usadas para manter informações precisas sobre o estado do sistema e da aplicação. As informações empacotadas em uma mensagem de monitoramento são usadas pelas camadas de tolerância a falhas, cujas políticas estão em desenvolvimento e será fruto de uma tese [13, 42].

3.2 Resumo

Esse capítulo abordou as principais características do escalonador utilizado como base para o desenvolvimento do controle de fluxo. Algumas particularidades do SGA *EasyGrid* também foram apresentadas, como o funcionamento da camada de abstração e o modelo hierárquico dos processo gerenciadores. Foi apresentado também nesse capítulo o modelo de comunicação do *SGA EasyGrid*, onde toda a comunicação entre as tarefas é intermediada através dos gerenciadores. O próximo capítulo apresentará a camada de controle de fluxo proposta neste trabalho.

Capítulo 4

A Necessidade de Gerenciar a Comunicação e os Recursos de Memória

Para a execução eficiente de algumas aplicações em um ambiente paralelo é importante que o sistema gerenciador administre além da aplicação do usuário, mas também os recursos disponíveis. Administrar os recursos de armazenando e comunicação é fundamental para a execução de algumas aplicações. Nesse trabalho, foram consideradas aplicações grandes ou de larga escala as que possuíam mais de 10 mil tarefas ou as aplicações onde cada mensagem de comunicação é maior do que 1Kbyte. Uma aplicação grande necessita de um controle especial dos recursos de armazenamento e da rede de comunicação para evitar falhas ou o comprometimento do tempo esperado para sua execução.

A falta de sincronismo na comunicação entre dois processos é a principal razão para o congestionamento de mensagens na comunicação entre máquinas. Como será apresentado neste capítulo, o sistema gerenciador da aplicação precisa prever e agir antes que o congestionamento de mensagem aconteça.

O controle do uso da memória principal é outra necessidade para garantir a execução paralela de aplicações grandes. Gerenciar a quantidade de memória utilizada por cada gerenciador permite criar estratégias que reduzem os riscos de falha na execução devido a falta de recursos de armazenamento. O controle de fluxo de mensagens além de controlar a comunicação na rede também é uma importante ferramenta para o controle da memória utilizada no *SGA EasyGrid*. Isso pode ser feito administrando a quantidade de processos prontos e a quantidade de mensagens pendentes em cada gerenciador.

Este capítulo abordará uma estratégia para controlar o fluxo de mensagem e memória principal utilizada para aplicações BoTs de larga escala e apresentar os principais benefícios desses controles. Serão apresentadas também as otimizações no código e nas

estruturas de dados do SGA que foram implementadas para garantir uma eficiente execução do SGA para uma gama extensiva de instâncias.

4.1 Otimizações Realizadas no SGA

Antes de iniciar o desenvolvimento da nova camada de controle de fluxo, foi realizada uma análise criteriosa para identificar os gargalos e as principais otimizações que deveriam ser feitas no SGA a fim de permitir a investigação do real impacto da comunicação na execução de aplicações gerenciadas pelo SGA.

A versão inicial do SGA possuía uma simples estrutura de dados, um vetor, que armazenava e controlava as tarefas da aplicação. Essa estrutura permitia uma facilidade na implementação pois o MPI requer que os dados sejam transmitidos em vetores, e um rápido tempo de resposta para consultar e armazenar novas tarefas. O grande inconveniente da estrutura vetorial é a necessidade da memória contígua para o seu armazenamento. Erros por falta de memória pode acontecer, principalmente quando são executadas aplicação com grande número de tarefas.

Basicamente as otimizações propostas no SGA foram as seguintes: alteração na estrutura de armazenamento das tarefas da aplicação; a forma como os gerenciadores obtêm as informações das tarefas que serão escalonadas; a criação de estruturas de índices nas lista de tarefas de forma otimizar a inserção e procura de tarefas. Os tópicos que segue detalhará cada uma das otimizações realizadas.

4.1.1 Leitura Local dos Dados da Aplicação

Na primeira versão do SGA, a leitura dos dados da aplicação apenas acontecia no gerenciador global que carregava os arquivos com os detalhes da aplicação, e enviava esses dados de forma sequencial para cada gerenciador do *site*. Os gerenciadores do *site* quando recebiam as informações das tarefas as enviavam para os gerenciadores da máquina, também de forma sequencial.

O atraso na inicialização ocasionava em impacto direto no *makespan* da aplicação paralela, fazendo com que as máquinas utilizadas ficassem ociosas por vários segundos ou minutos dependendo do tamanho da aplicação. Cada segundo de atraso na inicialização representa na perda da capacidade de trabalho proporcional em segundos ao número de processadores utilizados.

A otimização realizada foi alterar a inicialização de cada gerenciador fazendo com que os dados da aplicação fossem obtidos localmente, através da leitura dos arquivos que contêm os dados da aplicação. Outra otimização realizada foi a criação de apontadores para o último elemento acessado e para o fim da lista de tarefas de cada gerenciador. Cada gerenciador possui uma lista de tarefa ordenada conforme o escalonamento estático ou dinâmico.

Para que o gerenciador obtenha ganhos com o apontador para último elemento acessado, é necessário que o escalador estático forneça um bom escalonamento, pois os dados da aplicação são inserido na lista de tarefas dos gerenciadores na ordem prevista pelo escalonamento estático. Assim, caso o escalonamento estático, ou parte dele, seja seguido no momento da execução o próximo elemento a ser executado está logo após ao elemento anteriormente utilizado. A inserção na lista de tarefas, também poderá ser feita mais rapidamente utilizando o apontador para o fim da lista, caso as tarefas do arquivo de entrada já estejam na ordem do escalonamento estático, cada nova tarefa será inserida logo após a anterior.

No próximo capítulo serão apresentados os resultados dos experimentos, que mostram uma melhora substancial no tempo de inicialização devido as otimizações citadas.

4.1.2 Tabela *Hash*

O controle das tarefas no SGA utilizava uma estrutura vetorial, onde cada elemento representava uma tarefa da aplicação com todos os seus atributos. O baixo custo para acessar um elemento garante aos vetores uma grande vantagens em relação as outras estruturas de dados. Um vetor, conhecido também como *array*, é formado por um conjunto de itens de dados, todos do mesmo tipo, em que a posição de cada item é referenciada de forma única, por um inteiro e o acesso é feito de forma direta.

O maior inconveniente do vetor é a necessidade de que todo o espaço ocupado por ele esteja disponível de forma contígua na memória. Por isso, um dos principais problemas observados no SGA para a execução de aplicações com grande número de tarefas era a necessidade de grande quantidade de memória contigua nos gerenciadores.

Uma *hash table* ou *hash map* é uma estrutura de dados que usa uma função para transformar chaves em um índice do*array* de elementos, onde os dados estão guardados [33, 34]. A idéia da tabela *Hash* é permitir que várias possíveis chaves possam ser mapeadas em um mesmo local do *array* através da utilização de uma função *hash*. Essa função pode

mapear mais de um elemento para um mesmo índice, as chamadas colisões entre os dados, sendo necessária a utilização de uma estratégia para permitir que mais de um elemento seja referenciado na mesma posição do vetor.

Algumas técnicas são conhecidas na literatura [33, 34] para a resolução do problema de colisão, como:

- *Linear Probing* - Este é o método mais simples de resolver o problema de colisão. Quando é identificada uma colisão é feita um busca sequencial na tabela iniciando pelo endereço *hash* da chave desejada até encontrar uma campo vazio. A tabela de armazenamento precisa ser circula, para que quando a última posição da tabela for atingida a busca retorne para a primeira posição da tabela. O principal problema dessa técnica é de aglomerar os dados e aumentar o tempo de armazenamento, pois quando ocorre um colisão os dados tendem a ser armazenados de forma sequencial na tabela, onde a inserção é mais lenta quando comparada a utilização da função *hash*.

- *Rehashing* - Esse método busca reduzir o problema de aglomeração dos dados no cenários de colisão. Quando ocorre uma colisão uma segunda função *hash* predefinida é aplicada a chave para obter o endereço da tabela desejado. Caso ocorra uma nova colisão, podem ser aplicadas novas funções *hash* predefinidas até que seja encontrada uma localização livre para guardar o dado a ser inserido.

- *Quadratic Probing* - Nesta estratégia os casos de colisão são resolvidos utilizando um função quadrática. Caso ocorra uma colisão na localização h da tabela é utilizada uma função h + i^2 para definir o endereço onde o dado será armazenado, onde i é um numero inteiro obtido de forma sequencial começando em 1.

- *Chaining* - Neste caso, cada elemento do tabela *hash* aponta para o primeiro elemento de uma lista encadeada. Nessa estratégia a lista apontada pela função *hash* receberá o novo elemento a ser inserido.

A estratégia utilizada para a resolução de colisão no SGA foi a *Chaining*, onde cada índice da tabela *hash* possui apenas ponteiros para o primeiro elemento de sua lista encadeada. A principal vantagem dessa estratégia é a economia de espaço, pois a tabela *hash* é um *array*, nesse caso como essa tabela possui apenas endereços para o primeiro elemento da lista, relativamente pouco espaço contíguo é necessário quando comparamos a quantidade de elementos e atributos que serão guardados na tabela *hash*.

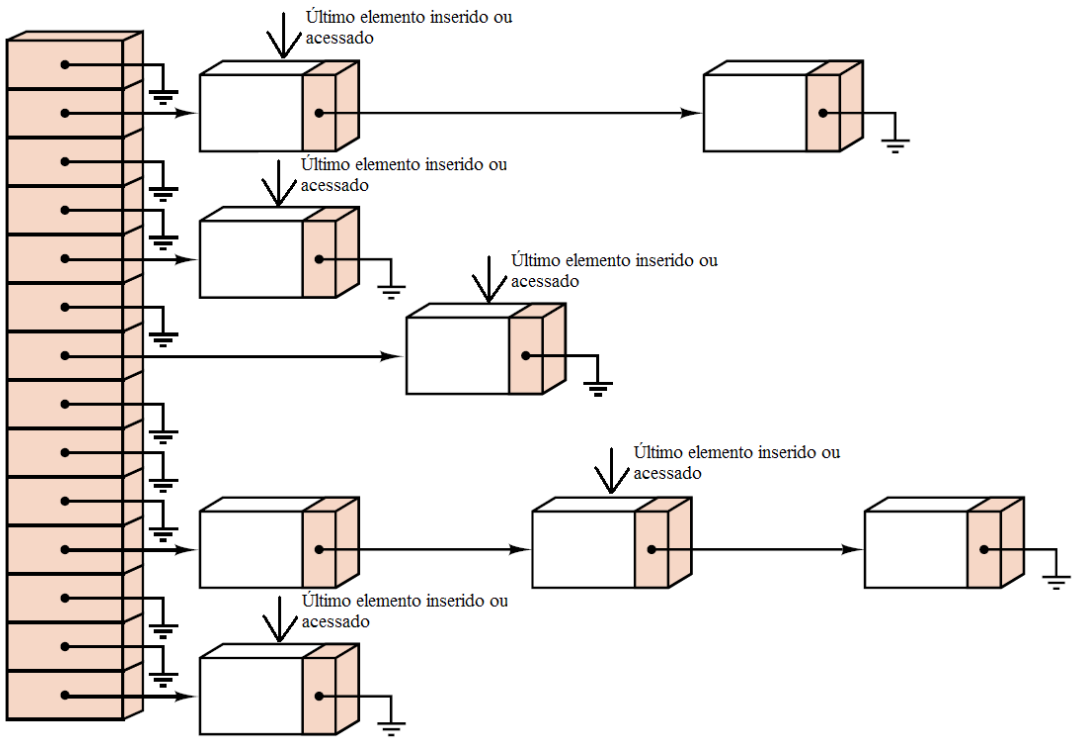

Figura 4.1: A Estratégia *Chaining* com o ponteiro utilizado para otimizar o acesso

O principal inconveniente dessa estratégia em relação as outras citadas, é o tempo gasto para acessar um elemento. Além de acessar o apontador da lista existente na tabela *hash*, pode ser necessário percorrer toda a lista para obter um elemento. Considerando que λ represente a divisão do número de elementos existentes pelo número de entradas na tabelas *hash*, supondo que os dados estão divididos uniformente pelas listas, uma procura na lista sem sucesso será realizada em no máximo λ comparações, pois é necessário percorrer todos os λ elementos da lista para concluir que a procura não terá sucesso. No caso de sucesso, em média, serão requeridos aproximadamente $1+\lambda/2$ comparações. Maiores detalhes sobre cada uma dessas estratégias podem ser encontrados em [33, 34].

As informações de uma tarefa do SGA é constituída por 11 inteiros e 3 *arrays* de caracteres que totalizam o tamanho de 664 *bytes*, para a execução de uma aplicação com 50 mil tarefas é necessária a existência de cerca de 32 *Mbytes* de memória contigua em cada um dos gerenciadores. A necessidade de espaço em memória principal pode ser agravada quando mais de um gerenciador compartilha a mesma máquina.

A nova estrutura de dados implementada no SGA consiste na utilização de um *hash* com estratégia de *Chaining* para a alocação. O vetor da tabela *hash* foi configurado com tamanho fixo de 1024 apontadores para uma lista encadeada. Os elementos da lista representam tarefas da aplicação e cada lista pode ser obtida através do resultado da

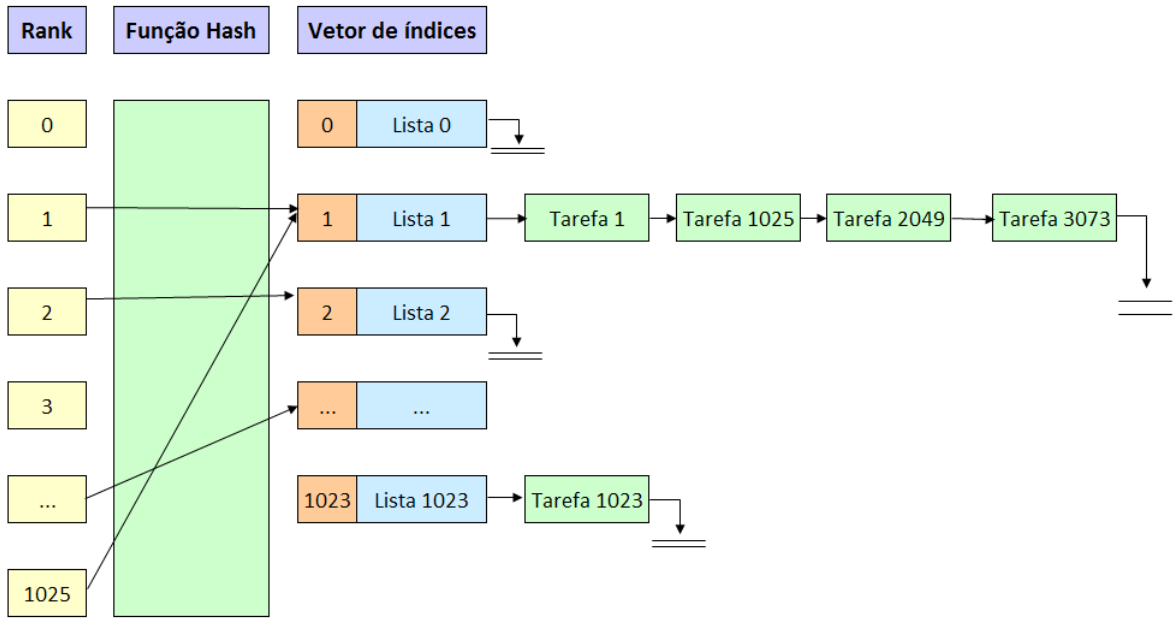

Figura 4.2: Tabela Hash

função *Hash* resultado do resto da divisão do identificador da tarefa (*rank* do processo) pelo tamanho do vetor. Na figura 4.2 é apresentado um exemplo da tabela *Hash* utilizada.

A lista de tarefas é ordenada pelo identificador da tarefa (*rank* do processo) e possui um apontador que guarda o último elemento inserido ou acessado em cada uma das listas da tabela *hash*. Geralmente as tarefas da aplicação aparecem nos arquivos de entrada de forma ordenada pelo identificador, com isso, existe uma grande probabilidade de que uma nova tarefa seja inserida imediatamente após a última tarefa inserida. A busca na lista de tarefas também possui um apontador para o último elemento acessando, conforme foi feito nas outras estruturas de dados do SGA. A utilização das listas permitiu que as tarefas que já foram executadas fossem removidas da tabela*hash*, fazendo com que o tempo médio de acesso a cada tarefa seja reduzido ao longo da execução.

Na versão do SGA utilizada nesse trabalho, os gerenciadores que participavam da execução paralela têm conhecimento apenas das tarefas da aplicações sob sua gerencia. Assim, apenas o escalonador global conhece todas as tarefas aplicação, e por isso, requer uma quantidade maior de memória para a sua execução.

A versão base que serviu como critério de comparação para a nova versão do SGA, ainda utiliza uma versão onde todos os gerenciadores conhecem todas as tarefas da aplicação. Isso agravava o problema de falta de memória para a execução dos gerenciadores

do SGA em máquinas com pouca memória principal.

O próximo capítulo mostrará que a extinção do vetor de tarefas deixou o SGA mais robusto permitindo a execução de um maior número de tarefas sem impactar de forma considerável no tempo de execução da aplicação.

4.2 O Controle do Fluxo de Mensagem

A execução de aplicações BoTs com grande número de tarefas, como os problemas relacionados a *data mining* [14], processamento genômico, renderização de imagens [24] e determinação de clima [15], inclui uma dificuldade maior aos gerenciadores de aplicação do que as aplicação menores, pois requer do sistema gerenciador um controle do fluxo de mensagens da aplicação para a obtenção de bons resultados. O sistema gerenciador da aplicação precisa ter estratégias para controlar o envio das mensagens evitando um congestionamento de mensagens na rede, a sobrecarga dos *buffers* de comunicação e o uso excessivo dos recursos de armazenamento.

Mesmo que o número de tarefas da aplicação seja muito grande, mas o SGA permite executar simultâneamente apenas o número de tarefas definidas pelo usuário para cada máquina. O controle do número de processos que são executados concorrentemente é importante para evitar que as trocas excessivas de contexto e viabilizar a execução de aplicações grandes. O controle do fluxo de mensagem tem o mesmo conceito, permitir que apenas um conjunto de tarefas mais prioritárias sejam consideradas e tratadas pelos gerenciadores, garantindo a redução do uso de memória e evitando que um congestionamento de mensagens impacte no resultado do escalonamento.

Nessa seção serão mostrados os impactos do congestionamento de mensagem para a aplicação e o funcionamento da rotina de controle de fluxo.

4.2.1 Como Funciona a Comunicação

O primeiro passo para a comunicação é guardar o pacote de dados, composto pela mensagem e o seu envelope no *buffer* de envio do TCP para a transmissão. O envelope é composto pelo identificadores da fonte, do destino, do comunicador e do tag. Essas serão as informações utilizadas para selecionar a mensagem pela operação de recebimento no destinatário. Após a transmissão da mensagem através da rede de comunicação, o pacote recém chegado ficará guardado no *buffer* do TCP até que ocorra um casamento dos

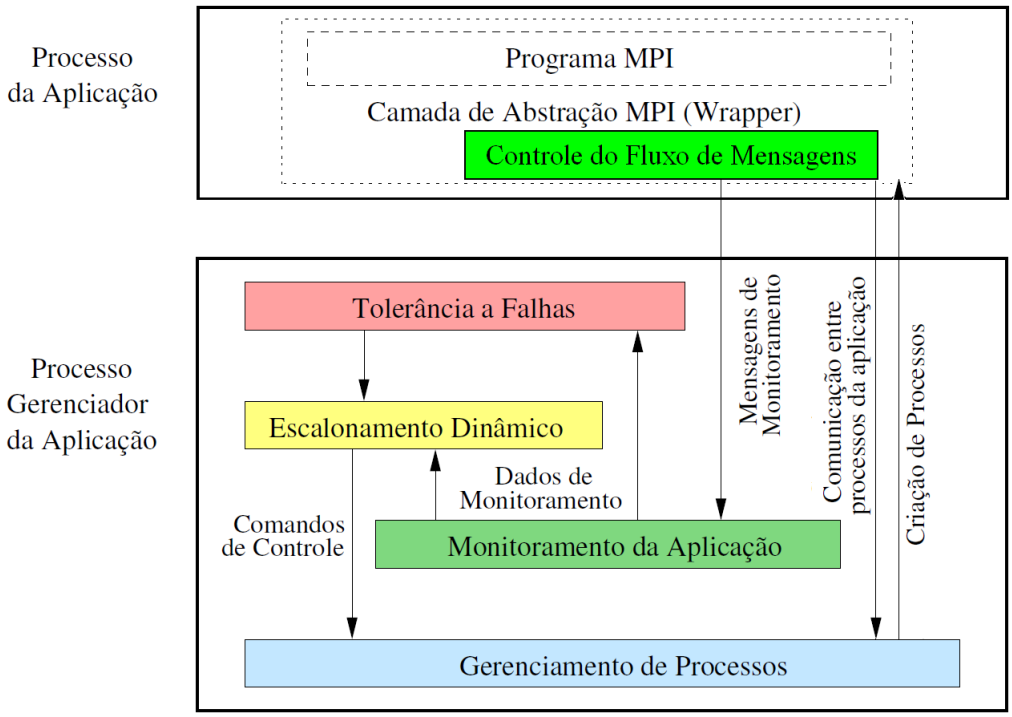

Figura 4.3: Arquitetura em camadas do SGA

parâmetros do envelope da mensagem com as informações esperadas pela aplicação, só nesse momento a mensagem é retirada do *buffer* do TCP e entregue para a aplicação. Para que ocorra o envio de uma mensagem entre dois processos é necessário que exista espaço disponível nos *buffers* de envio ou recebimento dos processos origem e destino da mensagem, caso contrário, o processo que enviou ficará bloqueado na operação de envio ou precisará manter em memória os dados a serem transmitidos.

4.2.2 A Troca de Mensagens no SGA

Para entender como acontece o congestionamento de mensagem no SGA é importante que conheça o funcionamento da troca de mensagens desse sistema gerenciador. Ao explorar os detalhes da implementação desse gerenciador aplicação foi possível identificar os impactos do congestionamento para a aplicação e onde a estratégia de controle de fluxo de mensagem deveria agir. A troca de mensagem no SGA é feita utilizando uma biblioteca desenvolvido no SGA a MEGmpi.h [16], que intercepta as chamadas de envio e recebimento de mensagens da aplicação substituindo-as por envio e recebimento redefinidas pela MEGmpi.h.

A única comunicação permitida do processo mestre da aplicação é com um processo

gerenciador global, quando o mestre da aplicação precisa enviar uma mensagem para as tarefas escravas a biblioteca MEGmpi.h interceptará a mensagem para forçar que essa mensagem seja transmitida pela hierarquia de gerenciadores até chegar ao destino. A comunicação das tarefas escravas com o SGA acontece apenas via o gerenciador da máquina, esse é responsável por receber e encaminhar a mensagem pela estrutura de gerenciamento até o destino desejado, conforme explicado no seção 3.1.6 do capitulo 3 dessa dissertação.

Utilizar a hierarquia de gerenciadores garante a criação dinâmica de processos, a tolerância a falhas e o escalonamento dinâmico das tarefas, entretanto faz com que uma simples mensagem do mestre da aplicação percorra pelo menos 3 gerenciadores, o GM, o SM e da HM, até chegar a tarefa escrava. A figura 4.4 ilustra as mensagens trocadas por cada gerenciador para uma aplicação BoT de 100 tarefas, onde são enviadas pelo menos 600 mensagens no total.

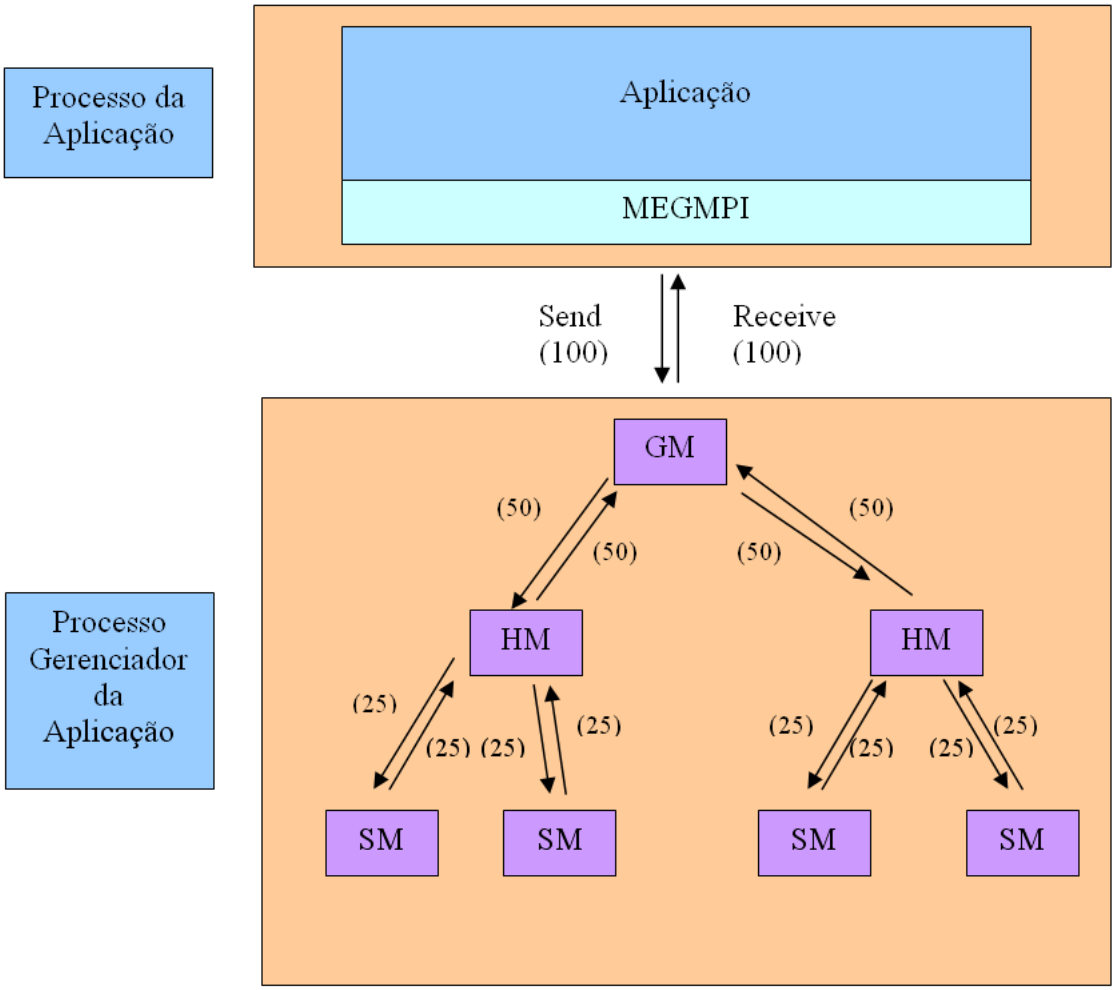

Figura 4.4: Troca de mensagens no SGA

O SGA utiliza em sua comunicação mensagens MPI do tipo *Send* e *Isend*. Para

evitar que os gerenciadores do nível superior aguardassem o envio de uma mensagem para continuar a sua execução, o gerenciador global e local utilizam sempre o envio não bloqueante (*Isend*) na comunicação. Isso evitar que um congestionamento apenas entre os gerenciadores GM e SM ou entre o HM e o SM impacte a comunicação entre os outros processos da aplicação. A troca de mensagens entre os gerenciadores da máquina (HM) e do site (SM) são feitas via *Send* porque a grande maioria das mensagens trocadas nesse sentido é para informar a conclusão da execução de um processo da aplicação, sendo por isso pequenas. As mensagens de tamanho menores que 52105 bytes são não bloqueantes por utilizar o protocolo de comunicação *short*.

No SGA o recebimento de mensagens da camada de nível superior é prioritário sobre o recebimento de mensagens do nível inferior. Por exemplo, para o GM uma mensagem da aplicação será tratada antes do que uma mensagens do SM. De forma similar, as mensagens do GM sobre as mensagens do SM e entre o SM e o HM. Essa estratégia é importante para garantir que as mensagens da aplicação cheguem mais rapidamente aos gerenciadores SM e da HM, viabilizando uma melhor utilização dos recursos de processamento.

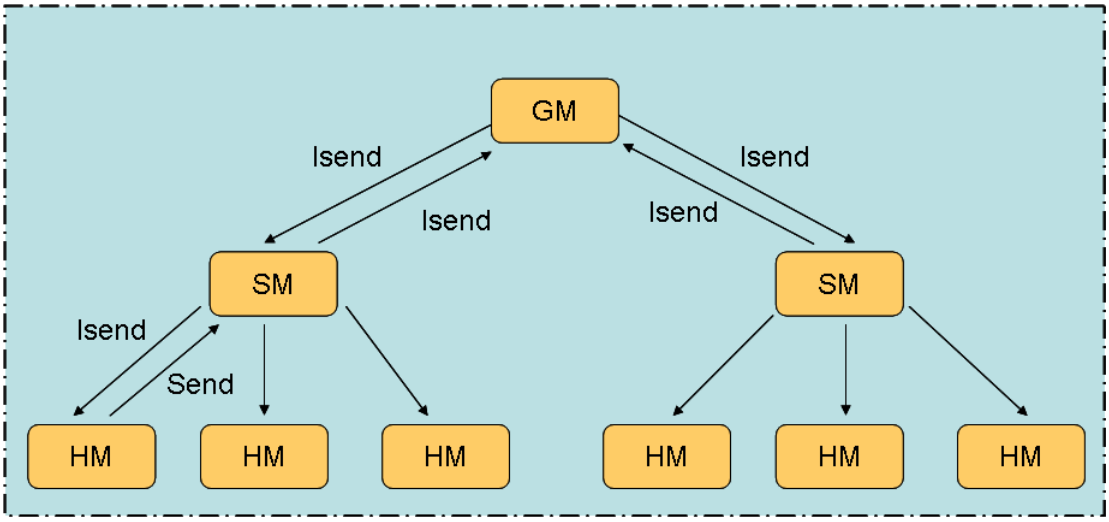

Figura 4.5: Tipos de mensagens MPI utilizados pelo SGA

As trocas de mensagens no SGA são controladas por 19 *tags* que possibilitam oferecer escalonamento dinâmicos, monitoramento da aplicação e tolerância a falhas. As principais mensagens trocadas no SGA utilizam os *tags* de envio de mensagens para a aplicação é para informar que uma tarefa foi concluída. A grande maiorias das outras mensagens são utilizadas pelo escalonamento dinâmico para a redistribuição da tarefas, mas somente são utilizado caso ocorra uma necessidade de redistribuição das tarefas [39].

Uma comunicação que acontece com frequência no SGA, é o envio de um sinal implementado como mensagem antes do envio de cada mensagem da aplicação, para verificar

se o gerenciador destino da mensagem está ativo. Para isso é executado um comando remoto *mpitask* na máquina destino da mensagem. Esse teste é necessário para evitar um *bug* da versão do MPI/LAM onde o remetente da mensagem termina a sua execução caso o destino da mensagem não esteja ativo. Caso não ocorra qualquer troca de mensagem entre um gerenciador de nível superior e do nível inferior em um intervalo de 5 segundos, também é executado um comando remoto *mpitask* para verificar se o gerenciador de nível inferior ainda está ativo. As rotinas de tolerância a falha terão que recriar o processo gerenciador caso identifique que ele não está mais ativo antes de um novo envio de mensagem [13].

4.2.3 O Problema

A utilização do *buffer* do MPI e do TCP permite a aplicação continuar a trocar mensagens sem a obrigação de utilizar a mensagem recém recebida no momento em que ela chega ao *host* de destino. O benefício citado, pode se torna em um problema quando o somatório dos tamanhos das mensagens pendentes é maior do que a capacidade disponível nos *buffers*. A falta de espaço nos *buffers* do TCP faz com que o TCP aplique as estratégias de controle de fluxo [3] para evitar o congestionamento e *overflow* de seus *buffers*.

O controle de fluxo do TCP, mesmo sendo uma importante ferramenta para manter uma comunicação numa velocidade satisfatória para os *hosts* envolvidos, mas pode consumir uma quantidade significativa de tempo de CPU e resultar no atraso na comunicação devido as interrupções na transferência de dados.

Normalmente o *Linux* não está configurada para transferir dados em alta velocidade, pois geralmente não é desejável ter um grande quantidade de memória dedicada apenas para a comunicação de dados, pois para aumentar a velocidade da conexão e permitir a existência de um número maior de pacotes pendentes, uma das alternativas é aumentar o tamanho do *buffer* do TCP de envio e recebimento de mensagens [3] e ativar o ACK seletivo [36].

Um tamanho apropriado dos *buffers* do TCP de forma a promover mais eficiência depende de vários fatores, como: taxa de processamento do destino, velocidade da rede, taxas de erro, topologia de rede, tamanho da memória e tamanho dos dados a serem transferidos. Tipicamente, quando os dados a serem transferidos são extremamente grande, pode se desejável configurar o tamanho dos *buffers* do TCP até o valor máximo, isso reduzir a ocorrência de controle de fluxo e o custo de CPU para controlar a troca de mensagens.

Em um ambiente compartilhado como as grades, pode não ser possível definir uma valor apropriado para as configurações da rede para as várias aplicações que podem ser executadas nos recursos do ambiente paralelo, obrigando ao gerenciador a realizar o seu próprio controle do fluxo de mensagens. Requerer que o administrador altere as configurações do TCP também não é uma boa alternativa, pois isso impactará todas as aplicações que executam nesse sistema.

A necessidade do controle de fluxo no sistema gerenciador pode ser melhor estudada nas aplicações BoT, onde logo no inicio da execução, a tarefa mestre envia uma mensagem para cada tarefa escrava. Essas mensagens são enviadas para permitir a execução de todas as tarefas escravas, pois nesse tipo de aplicação as tarefas escravas possuem dependência apenas com a tarefa mestre.

Quando é executada uma aplicação BoT no SGA, onde existe uma estrutura hierárquica de gerenciadores, a comunicação entre a tarefa mestre e o GM é o primeiro gargalho encontrado. A comunicação massiva entre a aplicação e o gerenciador global durante o inicio da execução faz com que o global dedique toda o seu processamento a receber as mensagens da aplicação e encaminhar para o gerenciador local. Caso a velocidade com que a aplicação envia as mensagens seja maior do que a velocidade do gerenciador global em recebê-la e encaminhá-la para o gerenciador local, acontecerá um acumular nos *buffers* entre a aplicação e o gerenciador global, fazendo com que a aplicação fique bloqueada no envio até que o global retire quantidade suficiente de dados do *buffer* do TCP para armazenar uma nova mensagem.

Como pode ser observado, a falta de sincronismo entre o envio e a velocidade de processamento das mensagens recebidas pela aplicação destino, causa a sobrecarga do *buffer* do TCP fazendo com que a aplicação fique proibida de enviar qualquer nova mensagens.

Cada mensagem recebida da aplicação é guardada no gerenciador global é encaminhada para o gerenciador local utilizando um comando *MPI_ Isend*. Conforme já explicado, o comando *MPI_ Isend* é não bloqueante o que permite a sobreposição dos tempo de envio com o de processamento de uma nova mensagem. Entretanto, uma mensagem enviada mas sem a confirmação de que o destino a recebeu fica pendente internamento no MPI e no SGA. O aumento do número de mensagens pendentes, causa um atraso no envio de novas mensagens utilizando o comando *MPI_ ISend*. Isso ocorre porque, ao utilizar esse modo de envio o MPI mantém o controle de todas as mensagens que a aplicação sem a confirmação do sucesso no envio. À medida que o número de mensagens pendentes crescem, é observado um aumento gradual no tempo para enviar um nova mensagem. No

próximo capítulo serão apresentados os experimentos realizados que mostram que dependendo do número de mensagens pendentes, o tempo para enviar uma única mensagem chegar a ser dezenas de vezes maior do que quando não tem mensagem pendente.

Controlar o número de mensagens pendentes além de aumentar a velocidade da comunicação, também é importante para conter o uso da memória. Uma aplicação que envia mensagens via o comando *MPI_ Isend* necessita guardar em memória todas os dados da mensagens pendentes, até que ocorra a confirmação do recebimento dessas mensagens. O MPI também necessita gerenciar o status das mensagens enviadas para que quando a aplicação executar o comando *MPI_ Test* ou *MPI_ Wait* seja possível informar se a mensagem ainda está pendente. Antes das alterações que propiciaram uma melhoria na administração das mensagens pendentes, o SGA verificava o sucesso do envio em dois momentos:

- Logo após enviar uma mensagem *MPI_ Isend*, onde normalmente a mensagem continuava pendente, ou
- caso o gerenciador fique sem receber qualquer mensagem de comunicação, nesse caso o SGA retirar o gerenciador da CPU executando um comando *sleep* de 1 microssegundo e após isso testa as mensagens pendentes. O previsto é que esse cenário não aconteça enquanto a aplicação do usuário estiver enviando mensagens e sobrecarregando os *buffers* de recebimento do global, pois sempre existirá uma mensagem a ser recebida pelo global.

Pelas razões citadas, o número de mensagens pendentes na versão inicial do SGA era crescente fazendo com que o tempo para completar um comando de envio não bloqueante aumentasse a cada envio de uma nova mensagem. O tempo gasto para executar cada comando *MPI_ Isend* impacta diretamente na vazão de direcionamento das mensagens do global para o local. Quando a vazão de direcionamento do global é menor do que a vazão de execução das tarefas, as máquinas do ambiente compartilhado ficam ociosos aguardando a chegada das mensagens que permitirão as tarefas ficarem disponíveis para a execução.

O congestionamento de mensagens nos *buffers* do TCP também impactam diretamente no monitoramento da aplicação e nos mecanismos de tolerância a falhas. Duas importantes verificações de monitoramento são afetadas em caso de congestionamento:

- O envio do *heartbeat* - ocorre quando o SGA verifica que nenhuma mensagem foi trocada entre dois gerenciadores em um intervalo de 5 segundo. O gerenciador que

enviou essa mensagem espera um intervalo de 5 segundos, se não receber uma resposta supõe que o gerenciador destino não está mais ativo, podendo nesse momento tentar recriá-lo ou retirando-o da lista de disponíveis.

- O envio do sinal - ocorre sempre antes de qualquer envio de mensagem para verificar se o destino está em execução, conforme citado na seção 4.2.2. A execução desse comando remoto, é bastante sensível a sobrecarga dos *buffers* do TCP, ocasionando em atraso na resposta de um sinal enviado e reduzindo a vazão de envio de mensagens do global.

  Uma otimização feita no envio do sinal permitiu uma melhora no número de mensagens direcionadas pelo gerenciador global para o local. Essa otimização consistiu em criar um *thread* que testa se o sinal foi respondido, em caso negativo o processo dorme, e um novo teste é feito assim que o gerenciador consegue a CPU. Em média na configuração 2, a resposta da *thread* ocorre em 4000 microssegundos em uma rede onde os *buffer* do TCP não estão cheios. Na versão anterior, o teste dessa *thread* acontecia apenas após 10000 microssegundos após a execução do sinal, caso não obtivesse resposta o reteste acontecia após aguarda pelo menos 100000 microssegundos, reduzindo a vazão de direcionamento para cerca de 10 mensagens por segundo. A utilização do sinal otimizado permitiu ao SGA aumentar a sua vazão para direcionar cerca de 200 mensagens por segundo. A verificação após o gerenciador ganhar o processador, permite ao SGA aguardar um tempo mais próximo do mínimo necessário para a resposta desse comando remoto, aumentando a velocidade de comunicação e atrasando a sobrecarga dos *buffers* do TCP e o congestionamento da rede. Por essa razão, alguns experimentos serão feito no próximo capítulo utilizando a versão sem a otimização do sinal, para mostrar o impacto do congestionamento do *buffer* no escalonamento da aplicação e os últimos experimentos apresentarão os resultados com a otimização do sinal. Não foi observado aumento expressivo na utilização da CPU por testar o sinal imediatamente após o processo gerenciador retomar a CPU.

4.2.4 O Controle da Utilização da Memória

Para a execução de aplicações com grande número de tarefas ou que enviam mensagens grandes é fundamental que se controle a quantidade de memória utilizada. A quantidade de mensagens enviadas pela aplicação não pode ocasionar em uma utilização maior de memória do que a quantidade definida pelo usuário/administrador para o gerenciador global.

Para a análise do uso da memória são avaliados 3 parâmetros:

- O tamanho das mensagens enviadas via *MPI_ Isend* mas ainda pendentes - As mensagens enviadas via o comando *MPI_ ISend*, mas que ainda não foram recebidas pela aplicação de destino e ficam pendentes na aplicação remetente até que seja executado um comando *MPI_ Test* com sucesso. A contabilidade dessa mensagens é feita totalizando o somatório das mensagens pendentes. Foi definido para os experimentos dessa dissertação que o somatório dos tamanhos das mensagens pendentes não pode ultrapassar 300 *Mbytes*. Quando o tamanho máximo de mensagens pendentes é atingido, o gerenciador global ou local testa todas as mensagens pendentes até que o somatório dessas mensagens seja inferior a 300 *Mbytes*.

- As mensagens enviadas mas sem a confirmação de que o destino concluiu a sua execução - Essas mensagens ficam guardadas nas estruturas da tolerância a falhas para garantir que caso ocorra alguma falha no gerenciador destino seja possível criar um novo gerenciador e reenviar as mensagens enviadas para o gerenciador que falhou. O controle da quantidade de memória utilizada por essas mensagens é feito através de um *vetor* circular com tamanho de 2x*NCtrl*. Nesse vetor é guardando o *rank* da tarefa que possui a mensagem pendente e a quantidade de *bytes* enviados na mensagem e que ficarão guardados nos gerenciadores até que a aplicação escrava conclua a execução dessa tarefa. Isso permite um controle exato da quantidade de *bytes* que estão sendo guardados no gerenciador que possui a mensagem pendente.

- O tamanho da tabela *Hash* - A quantidade de memória utilizada pela tabela *Hash* é contabilizada junto com os dois itens citados acima e representa a alocação em memória das mensagens da aplicação.

O somatório das três utilizações de memória citadas não pode ultrapassar o limite estabelecido pelo usuário nos arquivos de entrada do SGA. A versão inicialmente criada prevê que o usuário somente poderá controlar a memória total em uso no gerenciador global. Na próxima seção será mostrada a interação entre a quantidade de memória utiliza e o controle do fluxo de mensagem.

4.2.5 O Controle das Mensagens

Para resolver o problema de comunicação, foi desenvolvida uma nova camada de controle entre a aplicação é o SGA. Essa camada tem por objetivo controlar toda a troca de

mensagem entre a aplicação e o gerenciador global. O conceito básico é sincronizar a vazão do remetente com a vazão do destinatário reduzindo o congestionamento de mensagens entre a aplicação e o SGA.

O controle do fluxo de mensagem tenta gerenciar o número máximo de mensagens que a configuração paralela utilizada pode suportar, evitando congestionamento da rede devido ao grande número de mensagens pendentes de serem transmitidas. O número de mensagens gerenciadas pelo controle de fluxo será denominado por *NCtrl*, cujo valor será definido em função do número máximo de processos que podem estar em execução concorrentemente nos gerenciadores das máquinas.

Nenhuma mensagem pode ser enviada pela tarefa mestre se a quantidade de memória utilizada pelo gerenciador global irá ultrapassar o limite máximo disponibilizado pelo usuário. Como o controle de fluxo é feito entre a aplicação do usuário e o global é possível prevê se um novo envio de mensagem utilizará mais memória do que a quantidade máxima permitida. Nesse momento o controle de fluxo é acionado não permitindo um novo envio até que seja liberado quantidade suficiente de memória para o envio da nova mensagem.

Cada mensagem da aplicação mestre enviada é contabilizada como mais uma mensagem pendente, e cada mensagem recebida pela rotina de controle de fluxo com o resultado da execução de uma tarefa, debita uma mensagem pendente. Quando o número de mensagens pendentes é igual a duas vezes *NCtrl*, o controle de fluxo entra em ação recebendo mensagens dos gerenciadores de nível inferior até que sejam recebidas a confirmação da execução de *NCtrl* mensagens, para assim permitir o envio de novas *NCtrl* mensagens serem enviadas. As mensagens recebidas, são guardadas em uma lista para serem enviadas para a aplicação quando as forem solicitadas.

Para cada mensagem enviada é contabilizada a quantidade de memória utilizada através dos três parâmetros citados na seção 4.2.4, caso o uso de memória utilizada pelo gerenciador global seja maior do que a quantidade máxima de memória disponível, a aplicação deixa de enviar mensagens mesmo antes do número máximo de mensagens definido pelo controle de fluxo de mensagens (NCtrl), evitando erro na execução da aplicação por falta de memória. A aplicação mestre nesse caso fica bloqueada de enviar novas mensagens, até que seja liberada memória suficiente para o envio da próxima mensagem.

A rotina de controle de fluxo fica ciente da conclusão da execução das tarefas utilizando as mensagem da aplicação escrava que segue a hierarquia de gerenciadores até chegar a camada de abstração MEGmpi, onde o controle fluxo está implementado. Como a aplicação mestre estará bloqueada em um envio, ela não conseguirá receber as mensagens

das aplicações escravas que informam o resultado da execução. Por essa razão, a rotina de controle de fluxo armazena todas as mensagens da aplicação escrava em uma lista até que a tarefa mestre as solicite ao SGA.

Utilizar a estratégia de reter todas as chamadas até que n mensagens sejam recebidas se mostrou eficaz para evitar que a rotina de controle de fluxo seja chamada a cada envio de mensagem. O modelo proposto já está adequado para a execução paralela em máquinas com mais de um núcleo, pois considera o número de processos executando concorrentemente além de considerar o número de máquinas paralelas.

4.3 Resumo

Este capítulo descreveu o modelo de comunicação do SGA, onde através da utilização da estrutura hierárquica a troca de mensagens entre as tarefas mestre e escravas são mediadas pelo gerenciadores do *SGA EasyGrid*. Foram apresentadas as principais causas do problema de congestionamento de mensagens na execução de aplicações paralelas, apresentando a necessidade do controle de fluxo para a execução da aplicação. Foi proposto um controle de fluxo de mensagens e a necessidade controlar a quantidade de memória utilizada para a execução de aplicações grandes. O próximo capítulo apresentará os experimentos realizados nessa dissertação que comprovam a necessidade desses controles para a execução de aplicações BoTs.

Capítulo 5

Análise Experimental

Este capítulo tem como objetivo analisar os resultados dos experimentos que comprovam a necessidade da utilização da estratégia de controle de fluxo na execução de aplicações *BoT*. Serão apresentados também os resultados dos testes das otimizações realizadas sobre o gerenciador de aplicação SGA *EasyGrid* melhorando o seu desempenho.

5.1 A Aplicação *BoT* Utilizada

Para avaliar a necessidade do controle de fluxo de mensagens foram realizados experimentos utilizando aplicações Mestre/Escravo com grande quantidade de tarefas e/ou que enviam mensagens grandes. Além da sua importância e aplicação para a solução de uma grande gama de aplicações científicas, as aplicações BoTs são também importantes para o estudo do congestionamento de mensagens e dimensionamento de recursos devido ao seu padrão intenso de comunicação, onde cada tarefa escrava necessita receber apenas dados da tarefa mestre para iniciar a sua execução. Por isso, as tarefas escravas são quase independentes fazendo com que o grau de paralelismo seja limitado pela capacidade de envio de mensagens do processo mestre e pela quantidade de recursos de processamentos utilizados na grade.

Outra vantagem, é a simplicidade dessas aplicações, onde facilmente pode ser obtido um bom escalonamento estático. Isso permite avaliar o controle de fluxo de forma isolada, evitando que as precedências entre as tarefas limite a taxa de envio das mensagens.

5.2 Ambiente de Avaliação

Os experimentos foram executados no *Smart Grid Computing Lab*(SGCLab), localizado no Instituto de Computação da Universidade Federal Fluminense. Os recursos do ambiente utilizado era de uso exclusivo para os testes para evitar que outras aplicações executando em paralelo influencie os resultados. No entanto, na primeira configuração descrito na subseção 5.2.1, não é possível afirmar que o ambiente utilizado é totalmente controlado, pois como a rede é compartilhada com os usuários do laboratório a utilização da rede pode introduzir uma sobrecarga na comunicação. A rede existente no laboratório é basicamente formada por duas configurações do SGCLab que serão descritas nas subseções 5.2.1 e 5.2.2.

5.2.1 Configuração 1

A primeira é composta de 32 processadores Pentium IV 2.6 GHz com 512Mb de RAM, utilizando Linux Fedora Core 2 e LAM/MPI 7.1.4 de 32 bits. Estas máquinas estão divididas em três sites interconectados por switches e Gigabit Ethernet, conforme apresentado na figura 5.1. Essa configuração apenas permite a execução sem erro de aplicações BoT, gerenciada pela versão inicial do SGA, com tamanho de até 50 mil tarefas, devido a quantidade de memória principal existente em cada máquina.

5.2.2 Configuração 2

A segunda configuração é formada por 8 máquinas onde cada uma delas possui 2 processadores Intel Xeon E5419 de 2.33 GHz com 4 núcleos e 16 GBytes de RAM, utilizando um Linux Centos 5.3 e LAM/MPI 7.1.4 de 32 bits, que estão interconectadas por um switch Gigabit Ethernet, conforme figura 5.2. Esse ambiente é totalmente isolado, pois o roteador é dedicado e exclusivo para as maquinas desse cluster.

5.3 Métricas de Avaliação e Parâmetros de Execução

Para a avaliação das otimizações feitas sobre o SGA e a eficiência do controle de fluxo, foi utilizado como principal parâmetro o tempo de execução da aplicação na grade computacional, também chamado de *makespan*. Para avaliar a eficiência das soluções o *makespan* obtido sempre será comparado com um limite inferior do *makespan* ótimo. É considerado

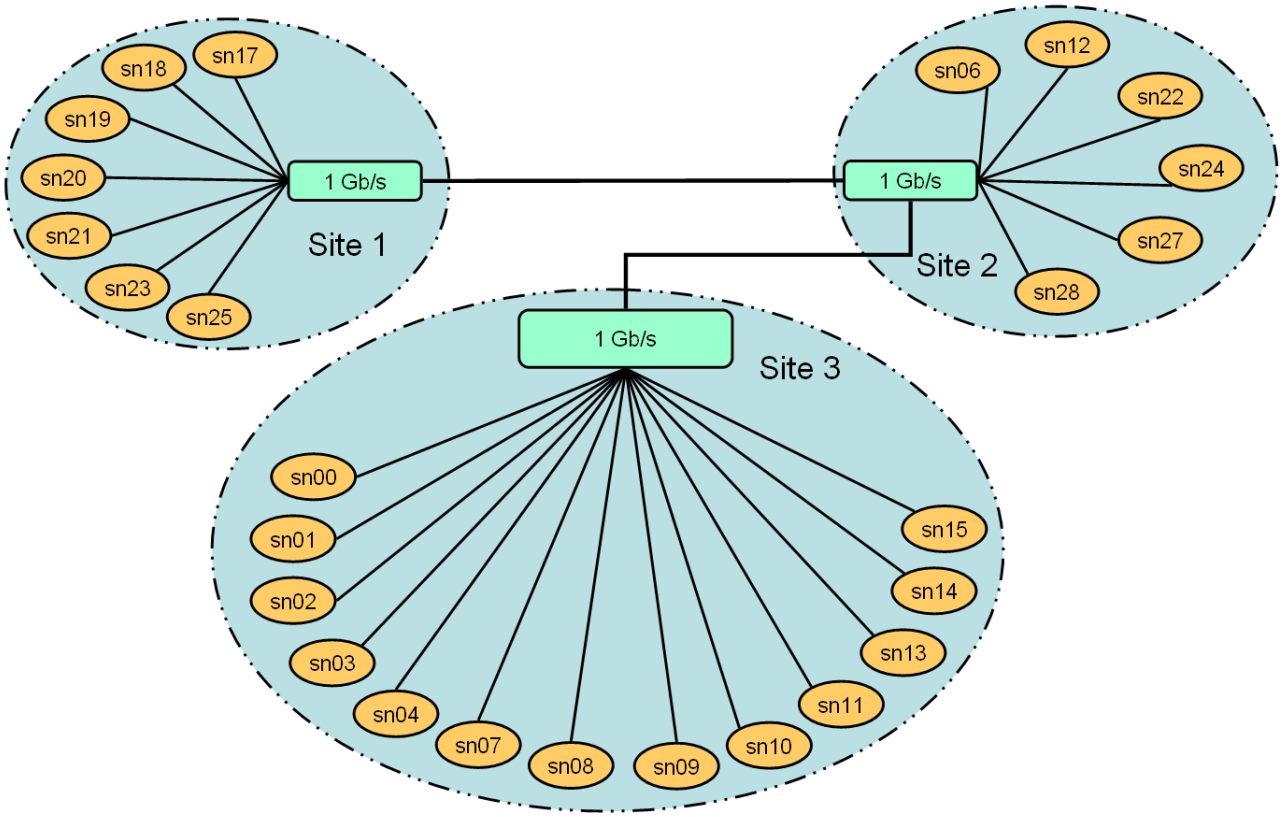

Figura 5.1: Configuração 1 da Grade utilizada para a execução das aplicações

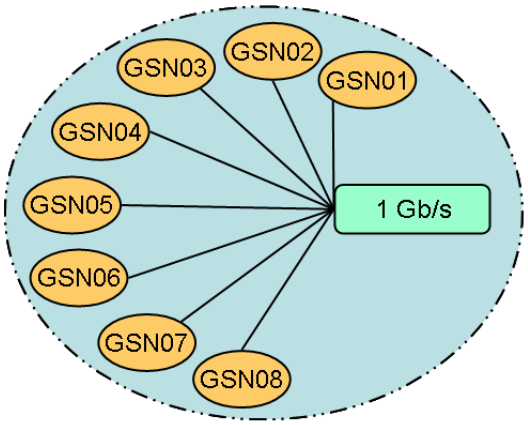

Figura 5.2: Configuração 2 da Grade utilizada para a execução das aplicações

como o limite inferior porque supõe uma distribuição das tarefas de forma perfeitamente balanceada entre os processadores da grade utilizada, e não leva em consideração os tempos gastos com trocas de contexto, criação dos processos da aplicação e a comunicação entre os processos da aplicação.

Cada resultado apresentado representa a média de no mínimo 3 execuções sucessivas e o tempo de execução de cada tarefa da aplicação Mestre/Escravo será denotado por *Texec* e será apresentado em segundos. Será denotado por N o número de processos da aplicação, por M o tempo de execução da aplicação e o *makespan* sempre será apresentado em segundos. Já o número de mensagens pendentes que o controle de fluxo irá gerenciar por vez será denotado por *NCtrl*.

5.4 Otimizações Realizadas no SGA

O principal objetivo das otimizações e permitir a execução eficiente para aplicações BoT grandes. Nesse caso, as aplicações que mais demandam cuidados são aquelas que possuem um grande número de tarefas nessa dissertação serão as aplicações com mais de 10 mil tarefas ou as que enviam mensagens grandes, as aplicação que enviam mensagens maiores que 1Kbyte.

Para garantir a escalabilidade foram implementadas algumas estratégias de programação, que melhoraram a gerência de recursos, permitindo ao SGA executar aplicações grandes sem apresentar grandes impactos no *makespan* ou no uso de memória principal.

5.4.1 A Utilização da Tabela *Hash*

A utilização da tabela Hash é importante porque permite ao SGA a execução de uma quantidade maior de aplicações, sem que ocorra falha na execução devido a falta de memória. Nesse contexto foi proposta a utilização de uma tabela *Hash* por ser uma estrutura de dados mais robusta e adequada para guardar grande quantidade de informações sem atrasar em demasia a busca, conforme foi explicado com detalhes no tópico 4.1.2.

O número de entradas da tabela *Hash* utilizada em todos os testes dessa dissertação estava configurada com o tamanho de 1024 posições, o que necessita de aproximadamente $680 Kbytes$ da memória principal para endereçar as listas que contêm as tarefas da aplicação.

Nos experimentos que seguem a *Versão Vetor* será a versão do SGA onde as tarefas são

guardadas em uma estrutura de dados do tipo vetorial e a *Versão Hash* denotará a versão onde as tarefas são guardadas na tabela *Hash*. Essas duas versões do SGA executaram com tolerância a falhas e escalonamento dinâmico habilitado.

O experimento 5.1 foi feito na configuração 1 utilizando 23 máquinas dentre as 26 disponíveis e o tempo apresentado representa a média de 3 execuções. O tempo de execução de cada tarefa da aplicação foi de 5 segundos e o tamanho de cada mensagem enviada foi de 24 *bytes*.

Tabela 5.1: Comparação entre a Versão *Vetor* e Versão *Hash*.

	Tempo de execução (segundos)				
N	10000	20000	30000	40000	50000
Versão Hash	2455	4847	7647	10906	14268
Versão Vetor	2410	4782	7555	10748	14203
Ótimo Teórico	2175	4350	6520	8695	10870
Versão Hash X *Versão Vetor*	101,88%	101,37%	101,22%	101,47%	100,46%
Versão Vetor X *Ótimo Teórico*	110,80%	109,93%	115,87%	123,61%	130,66%
Versão Hash X *Ótimo Teórico*	112,89%	111,43%	117,29%	125,43%	131,26%

A utilização da tabela *Hash* ao invés do vetor de tarefas ocasionou em um incremento de cerca de 1% no tempo de execução da aplicação. Esse custo adicional se deve ao esforço para obter a informação de uma tarefa na nova estrutura de dados *Hash* em relação a uma simples consulta a uma estrutura vetorial. Por essa razão, inicialmente se obtou pela utilização do vetor de tarefa na versão original do SGA [16]. A comparação do *makespan* da *Versão Hash* com o Ótimo teórico, mostra que a medida em que aumenta o tamanho da aplicação a *Versão Hash* e *Versão Vetor* começam a obter piores resultados em relação ao Ótimo Teórico. Isso não foi ocasionado pela inclusão da tabela Hash, mas devido ao congestionamento de mensagens, que será evitado utilizando o controle de fluxo. O tempo para a criação do vetor de tarefas e da tabela Hash são bastante similares, conforme pode ser verificar na tabela 5.2.

Tabela 5.2: Tempo para gerar o *Vetor* de tarefa e tabela *Hash*.

	Tempo de criação da estrutura de dados (em segundos)						
N	10000	20000	30000	40000	50000	100000	200000
Versão Hash	0,083	0,087	0,112	0,14	0,17	0,34	0,71
Versão Vetor	0,064	0,081	0,106	0,12	0,18	0,33	0,69

Outro aspecto analisado foi o incremento do uso da memória principal devido a utilização da tabela *Hash*. Os resultados da tabela 5.3 mostram que não houve um aumento expressivo na utilização de memória para inicializar o gerenciador global empregando a

Versão Hash quando comparado a *Versão Vetor*. Como a alocação de memória para a *Versão Vetor* é estática, os valores mostrados na tabela 5.3 representam a quantidade de memória utilizar do inicio até o fim da execução. Para a *Versão Hash*, esses valores representam o valor máximo de memória que será utilizada pela tabela *Hash*, pois as tarefas executadas são excluídas da tabela *Hash*. O gerenciador global como controla todas as tarefas da aplicação, é o gerenciador que mais consome recursos de armazenamento, por isso, sempre será verificada a memória utilizada pelo global como critério de análise sobre o uso de memória do SGA. Os experimento descritos na tabela 5.3 foram realizados na configuração 2, para que fosse possível comparar a execução de aplicação com tamanho de 200 mil tarefas. Os resultados apresentados representam a média de 3 execuções.

Tabela 5.3: Comparação do uso de memória entre a *Versão Vetor* e *Versão Hash*.

	Quantidade memoria utilizada (MBytes)			
Numero de tarefas	10000	20000	100000	200000
Versão Hash	8,70	15,36	71,68	139,00
Versão Vetor	8,60	15,19	70,9	137,40

Observe que a *Versão Hash* utiliza mais memória do que a *Versão Vetor*, por dois motivos, a necessidade de guardar o vetor de apontadores para as listas e devido ao apontador de cada nó da lista para o próximo elemento.

5.4.2 Carga Local dos Dados da Aplicação

Na versão inicial do SGA onde os dados da aplicação eram lidos pelo gerenciador global e enviados para os gerenciadores do site, para que posteriormente sejam encaminhados para o gerenciador da máquina, será denotado por *Versão Rede*. A nova versão onde os dados da aplicação são carregados localmente pelos gerenciador do *site* e da máquina será denotado por *Versão Local*.

A tolerância a falha é uma das funcionalidades existentes nas duas versões testadas, no entanto, somente a *Versão Local* permite ao usuário desabilitá-la em tempo de compilação, a *Versão Local* sem a tolerância a falhas será denotada por *Versão Local sem TF*, e será importante para mostrar o custo computacional dessa funcionalidade. A *Versão Local* e a *Versão Local sem TF* já possuem a tabela *Hash* implementada.

A *Versão Local* já possui implementada a otimização dos apontadores para o final da lista de tarefas e do último elemento acessado. Ambos buscam reduzir o tempo de acesso a tabela *hash* considerando as características das aplicações BoTs, conforme foi explicado em 4.1.1. Os experimentos a seguir foram executados na configuração 2 e tem como

objetivo apresentar os resultados obtidos com a alteração no processo de carga inicial das tarefas da aplicação, assim com a criação dos apontadores para o final da lista de tarefas e do último elemento acessado.

Tabela 5.4: Comparação do tempo de inicialização entre as versões (em segundos).

	Versão Local				Versão Rede					
N	1	2	3	Média	1	2	3	Média	RedeX Local	Local Sem TF
10000	0,41	0,41	0,41	0,41	0,50	0,51	0,51	0,51	123%	0,12
20000	1,28	1,28	1,29	1,28	1,57	1,64	1,59	1,60	125%	0,24
30000	2,78	2,74	2,73	2,75	3,32	3,34	3,35	3,33	121%	0,33
40000	5,04	5,11	5,08	5,07	5,80	5,79	5,83	5,81	114%	0,43
50000	8,37	8,24	8,25	8,28	9,26	9,24	8,94	9,15	110%	0,64
60000	12,23	12,24	12,15	12,21	13,67	13,07	12,86	13,20	108%	0,79
70000	17,27	17,29	17,28	17,28	21,15	21,33	21,68	21,39	124%	0,93
80000	22,84	22,84	22,78	22,82	32,73	31,46	31,68	31,96	140%	1,21
90000	29,65	30,58	30,51	30,24	52,24	52,71	51,67	52,21	173%	1,63
100000	39,11	38,98	37,68	38,59	76,02	74,69	77,02	75,91	197%	2,01

A *Versão Local* manteve o seu tempo de inicialização sempre menor do que a *Versão Rede*, no entanto, observa-se que para as aplicações com tamanho inferior a 60 mil tarefas a diferença percentual entre os tempos de inicialização das duas versões tende a diminuir a medida em que a aplicação cresce. Já para aplicações maiores que 60 mil tarefas observa-se um aumento na diferença do tempo de inicialização. Isso ocorre porque, a medida em que a aplicação cresce o envio das tarefas da aplicação via a rede de comunicação deixa de ser o principal ofensor para a inicialização, fazendo com que outras estruturas de dados e funções que são manipuladas localmente impactem mais fortemente no tempo de inicialização.

O grande ofensor para o tempo de inicialização de aplicações grandes são as estruturas de dados mantidas para a tolerância a falha e monitoramento. Mesmo após as otimizações nas estruturas de armazenamento e nas lista utilizada pelo tolerância a falha e monitoramento, como a inclusão de apontadores que agilizam as inserções e buscas, mas a tolerância a falha ainda traz um impacto negativo na inicializar de aplicações grandes devido a necessidade gerenciar grande quantidade de informações sobre o escalonamento e as tarefas.

Uma opção para o usuário do SGA que utiliza a *Versão Local* é desabilitar a tolerância a falha para obter a máxima performance e redução no tempo de inicialização. Isso é feito utilizando um parâmetro de compilação que automaticamente cria um executável sem as funcionalidade de tolerância a falhas e monitoramento.

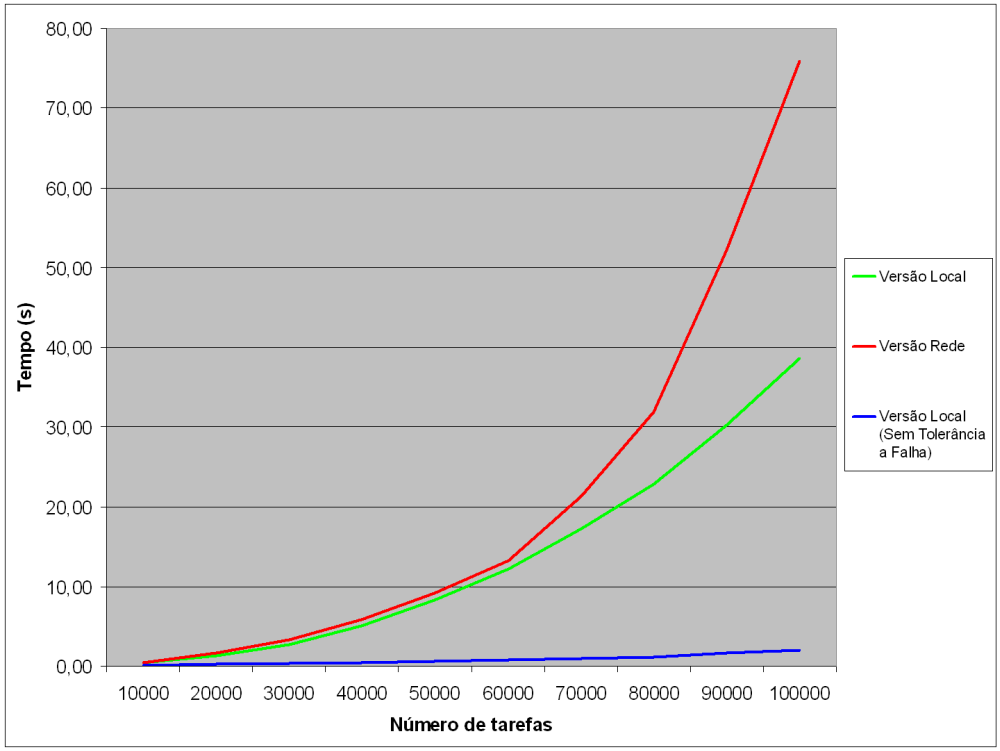

Figura 5.3: Tempo de inicialização - Versão Rede X Versão Local X Versão sem TF

5.5 O Envio de Mensagens utilizando o MPI

O tempo de envio de uma mensagem na rede depende fundamentalmente do tamanho da mensagem, da velocidade da conexão, da distância entre a origem e o destino da mensagem, do atraso de fila nos roteadores e do atraso no processamento. Sendo que o atraso de processamento em rede comutadas, costuma se desprezíveis.

Os experimentos dessa seção tem como objetivo verificar a variação no tempo de comunicação dependendo do comando MPI utilizado e da forma como a aplicação ou o sistema gerenciador se comporta para o envio e recebimento de mensagens. Será possível verificar que, dependendo do comportamento da aplicação para tratar as mensagens recebidas e enviadas, pode ser gasto um tempo além do previsto na comunicação entre processos. Outra contribuição desse seção é avaliar os impactos do tamanho da mensagem enviada no tempo de comunicação entre processos.

Para a obtenção dos tempos que serão informados na próxima seção, foram construídos 4 protótipos de aplicação, os 2 primeiros apenas trocam mensagens e os dois últimos a tarefa escrava dorme por 100 microssegundos antes de responder para a tarefa mestre sobre o sucesso da sua execução. Todos os protótipos são aplicações mestre-escravo, que enviam 10 mil tarefas. A diferença entre os protótipos é feita através da alteração do comando

MPI utilizado pela tarefa mestre para enviar as mensagens e o comportamento da tarefa escrava para receber as mensagens. Os detalhes de cada aplicação serão explicados a cada teste. Nos testes que seguem foram utilizadas apenas as máquinas *gsn01* e *gsn02* da configuração 2 informada na seção 5.2.

5.5.1 O Primeiro Protótipo - O Comportamento do Send com Ping-Pong

O primeiro protótipo é uma aplicação mestra-escravo onde o mestre envia uma mensagem para o escravo, utilizando o comando *MPI_Send*, e aguarda o recebimento de uma mensagem antes que uma nova seja enviada. A tarefa escrava ao receber a mensagem retorna uma nova mensagem de mesmo tamanho da recebida para quem o enviou. Esse funcionamento é conhecido como *ping e pong* e será utilizado para verificar o tempo de rede gasto para o envio e recebimento de uma mensagem. Neste caso não existe a possibilidade de qualquer atraso na rede relacionado a falta de espaço no *buffer* do TCP ou congestionamento da rede, visto que a comunicação é síncrona e a rede dedicada. O gráfico 5.4 mostra o tempo de envio e recebimento de 10 mil mensagens de mesmo tamanho.

Tabela 5.5: Tempo de RTT para uma aplicação de 10 mil mensagens sincronizadas

Tamanho da mensagem (bytes)	Tempo total(em segundos)
24	0,99
512	1,00
1K	1,42
2K	1,63
4K	2,03
8K	2,89
16K	4,99
32K	9,26
64K	14,60
1M	187,32

5.5.2 O Segundo Protótipo - O Comportamento do Send sem Ping-Pong

No segundo protótipo, o mestre envia uma mensagem para o escravo utilizando também o comando *MPI_Send*, mas pode recebe uma mensagem do escravo se existir alguma disponível em seu *buffer* de recebimento, caso contrário mais de um envio pode ocorrer, sem que qualquer mensagem seja recebida. Comparando os resultados mostrado na tabela 5.5 com os resultados do experimento da tabela 5.6 e figura 5.5 observa-se uma

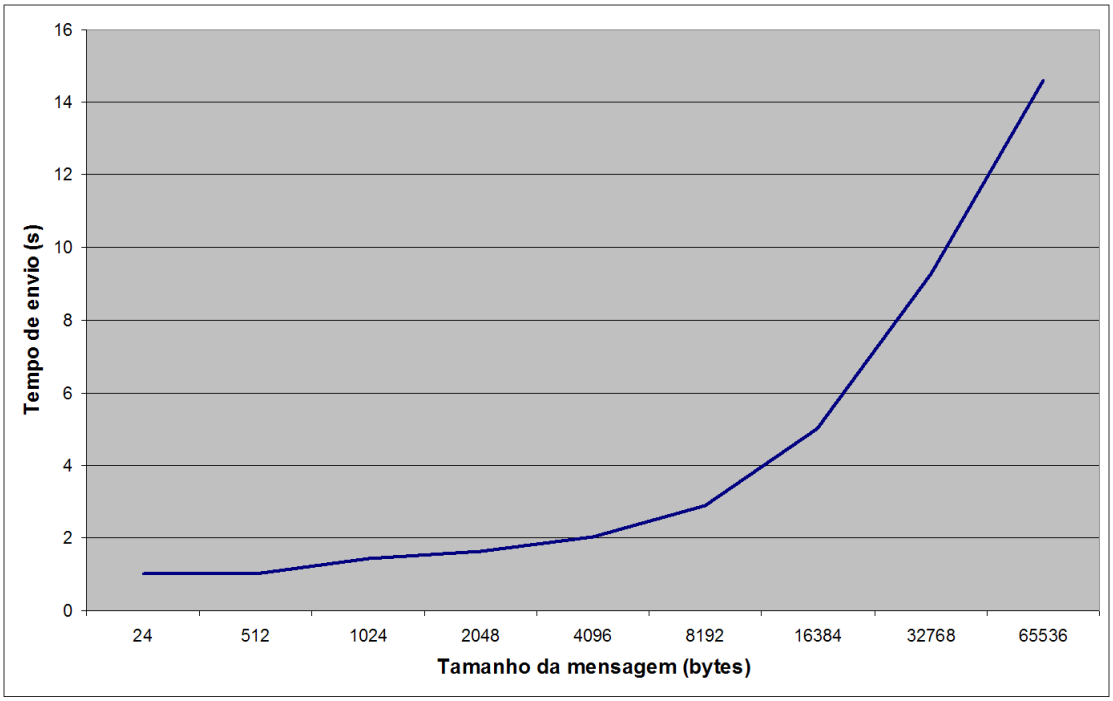

Figura 5.4: Tempo de envio e recebimento para uma aplicação que envia 10 mil mensagens sincronizadas

redução substâncial no tempo total para o envio e recebimento de 10 mil mensagens. Isso ocorre através da sobreposição dos tempos de envios de mensagens, pois as mensagens com tamanho de até 64 Kbytes utilizando o comando MPI_Send não são bloqueantes permitindo que antes que uma mensagem chegue aos destino uma nova mensagem seja enviada. O processo de envio dessa aplicação é bem parecido com a aplicação BoT utilizada nos experimentos sobre o controle de fluxo de mensagem e também é similar a forma como o gerenciador da máquina se comunica com o gerenciador do *site*.

Tabela 5.6: Tempo de envio e recebimento de 10 mil mensagens sem sincronismo

Tamanho da mensagem (bytes)	Tempo total(s)
24	0,22
512	0,24
1K	0,25
2K	0,32
4K	0,35
8K	0,71
16K	1,42
32K	2,88
64K	7,70
1M	Não executou

É importante observar na tabela 5.6 um aumento no tempo de envio de 10 mil mensagens de 64 Kbytes sem sincronismo em relação ao aumento observado entre 16 Kbytes

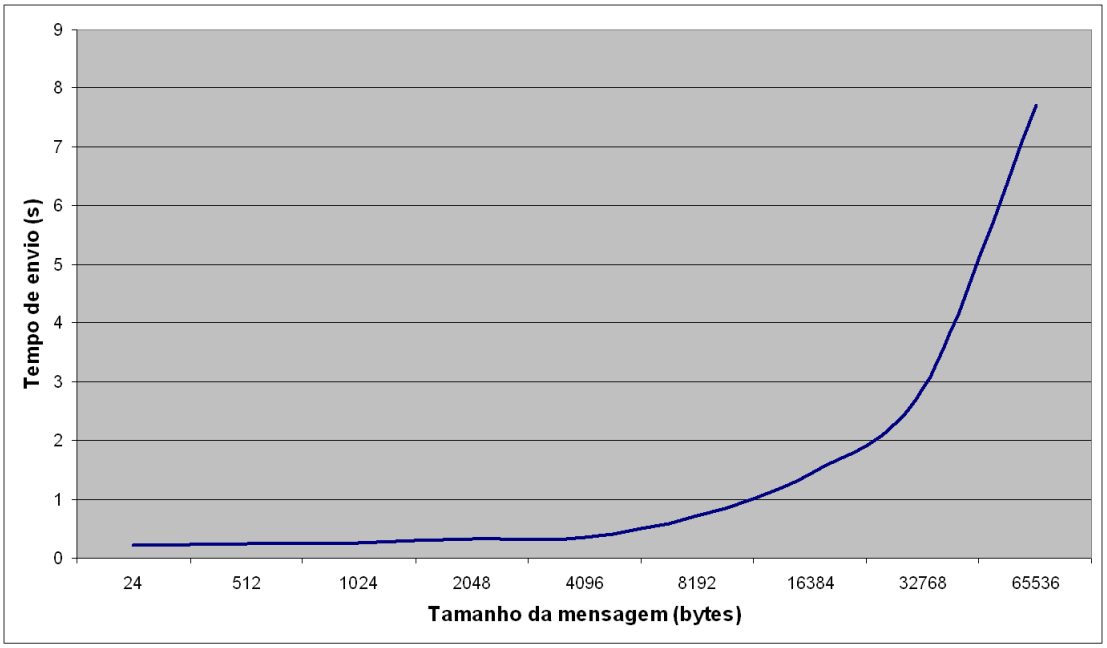

Figura 5.5: Tempo de envio e recebimento de 10 mil mensagens sem sincronismo

e 32Kbytes, por exemplo. A principal razão desse aumento é devido a mudança de protocolo do *Short* para o *Eager* onde é transferido o corpo e o cabeçalho da mensagem em pacotes distintos. Esse modelo de envio ocasiona em atrasos na comunicação devido a necessidade de enviar dois pacotes para a mesma mensagem. O protocolo *Eager*, foi descrito na seção 2.4 do capítulo 2.

A razão para o envio de mensagens de 1 Mbyte não ter sucesso é porque ambas as aplicações ficam em *Deadlock* tentando executar o comando *MPI_Send*. Após a tarefa mestre enviar a primeira mensagem utilizando o comando *MPI_Send* verifica que não existe qualquer mensagem a ser recebida e tenta enviar um nova mensagem. Já a tarefa escrava, recebe a mensagem enviada pelo mestre e tenta enviar um nova. Nesse momento, as duas aplicações ficam em *Deadlock* aguardando do comando *MPI_Send* bloqueante. As mensagens com tamanho superior a 64Kbytes utilizam o protocolo *Rendezvous* que é bloqueante, conforme explicado na seção 2.4 do capítulo 2.

5.5.3 O Terceiro Protótipo - O Comportamento do Isend sem MPI_Test

O terceiro protótipo é similar a aplicação anterior, mas é utilizado o comando *MPI_Isend* ao invés do *MPI_Send* e a tarefa escrava executa um comando *sleep* por 100 microssegundos até responder para a tarefa mestre. A grande vantagem do *MPI_Isend* é que a

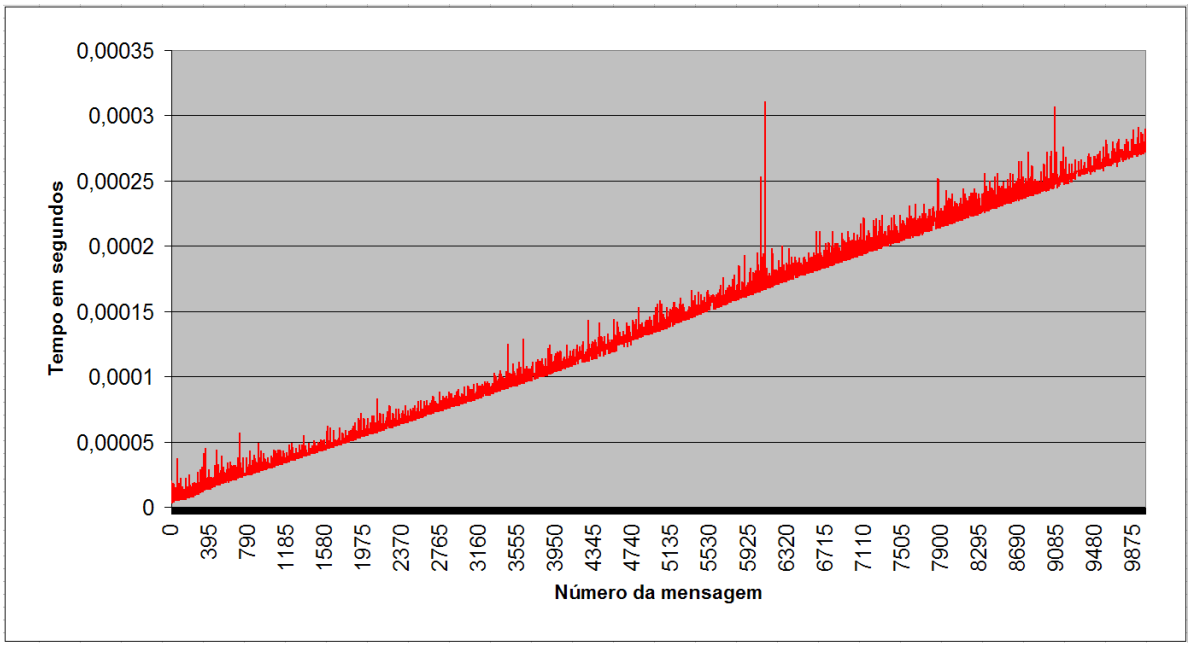

Figura 5.6: Tempo médio para cada operação de envio de mensagens de 8Kbytes - *MPI_ Isend* sem sincronismo

aplicação que enviou a mensagem não espera a confirmação do recebimento pelo destino para continuar a sua computação, independente do tamanho da mensagem. O SGA utiliza o comando *MPI_ Isend* na comunicação entre todos os gerenciadores, com exceção a comunicação entre o gerenciador da máquina e do *site*, onde é utilizado o comando *MPI_ Send* pois são enviadas apenas mensagens pequenas que não são bloqueantes.

A utilização do comando *MPI_ Isend* é muito importante para promover a sobreposição dos tempos de envio com o processamento da máquina local, pois o processo que enviou a mensagem não aguarda o término do envio para iniciar uma novo processamento.

Para verificar o sucesso do envio, é necessário que a aplicação que enviou a mensagem utilize a função *MPI_ Test* ou *MPI_ Wait*. O gráfico 5.6 simula uma comunicação onde o destino não está conseguindo processar as mensagens recebidas com a mesma vazão que elas são enviadas. O remetente, como utiliza o comando o *MPI_ Isend*, que não é bloqueante, não aguarda as confirmações do sucesso do envio e apenas após o envio de todas as mensagens começa a receber as confirmações. Cada mensagem dessa simulação tinha o tamanho de 8Kbytes, e os *buffers* do TCP de envio e recebimento estavam configurado com o seu valor padrão de 128 Kbytes, permitindo guardar até 32 mensagens.

Pela análise do gráfico 5.6, o tempo gasto pela aplicação para executar o comando *MPI_ Isend* aumente a cada envio. Esse tempo não é devido ao recebimento da confirmação do envio, mas exclusivamente para executar o comando *MPI_ Isend*. Por exemplo, o

tempo para enviar a mensagem 10 mil (0,000272s) supere em mais de 50 vezes o tempo para enviar a primeira mensagem (0,000005s). Isso ocorre em razão do controle de fluxo do TCP não permitir o envio de novas mensagens quando o *buffer* do TCP está cheio e pela administração feita pelo MPI para as mensagens pendentes de envio.

Além do aumento do tempo de execução do comando *MPI_ Isend*, a necessidade de guardar em memória todos os dados das mensagens enviadas até a confirmação do recebimento pelo destino, pode gerar um grande consumo de memória principal e até erro na execução da aplicação. Como por exemplo, uma aplicação que envia mensagens de 1Mbyte e o sistema está com 5000 mil mensagens pendentes, neste caso será necessário cerca de 5 Gbytes em memória só para guardar as mensagens pendentes.

O principal objetivo desse experimento foi mostrar que a falta de harmonia entre o envio e o recebimento, pode ocasionar em atrasos na comunicação relacionados ao controle de fluxo do TCP e degradar fortemente a performance da comunicação entre os processos.

5.5.4 O Quarto Protótipo - O Comportamento do Isend com MPI_Test

O último protótipo é similar ao anterior e corresponde a uma aplicação com as mesmas característica de comunicação do SGA, onde após utilizar o comando *MPI_ Isend* é realizado um teste através do comando *MPI_ Test* para verifica se o *host* destino já recebeu a mensagem enviada. Em geral, executar um comando *MPI_ Test* logo após o envio via o comando *MPI_ Isend* não retorna sucesso, devido ao tempo de envio da rede normalmente se maior do que o tempo de execução local de algumas linhas de código. A utilização do comando *MPI_ Wait* é uma alternativa, mas ocasiona no sincronismos entre as aplicações. Os resultados abaixo apresentam a comparação quando utilizado o controle de fluxo proposto e sem utilizá-lo.

Como pode ser observado pelo gráfico 5.7, o tempo para o envio de mensagens utilizando o controle de fluxo se mantém praticamente constante. Ao contrário do que pode ser observado quando este controle não é utilizado. É importante observar que para aproximadamente as primeiras 800 mensagens, o envio sem o controle de fluxo é mais rápido do que com o controle, isso se deve a rotina de controle de fluxo incluir um pequeno *overhead* a cada envio, principalmente na simples aplicação utilizada que apenas envia ou recebe mensagens.

O aumento no tempo para executar o comando *MPI_ ISend* sem o controle de fluxo,

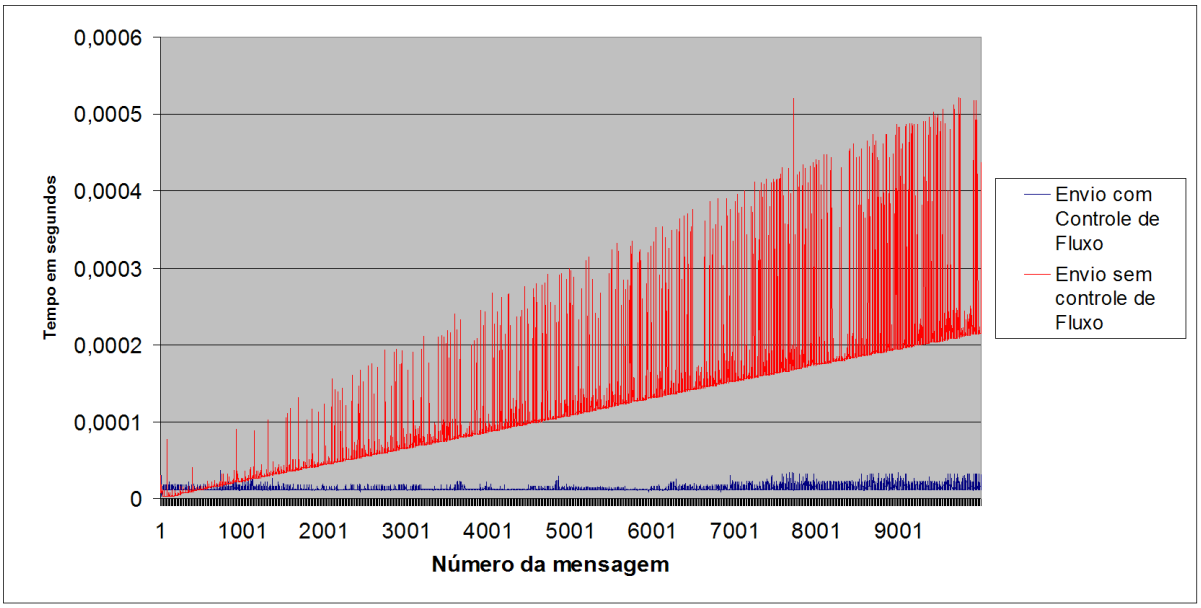

Figura 5.7: Comparação entre o tempo execução de cada operação de envio utilizando o comando *Isend* com e sem o controle de fluxo

cresce devido a tarefa mestre executar o comando *MPI_ Isend* mais rapidamente do que o tempo de envio na rede, formando fila de mensagens pendentes que o MPI precisará gerenciar. O aumento linear observado é devido ao esforço do MPI para administrar a fila de mensagens. Com o controle de fluxo, não existe a formação da fila de mensagens, pois somente são enviadas novas mensagens após a confirmação de que as mensagens anteriores já foram recebidas.

O tempo total de execução do experimento apresentado na figura 5.7 com o controle de fluxo foi de 20,24 segundos e sem o controle de fluxo foi de 22,40 segundos. A execução sem o controle de fluxo congestiona rapidamente o *buffer* do TCP fazendo com que a tarefa mestre fique bloqueada no envio sem conseguir fazer qualquer outra atividade. No entanto, quando o controle de fluxo é utilizado, enquanto a tarefa escrava ainda não tratou todas as mensagens enviadas o controle de fluxo recebe as mensagens da tarefa escrava, pois pode não ser possível enviar novas mensagens. Na aplicação utilizada essa previsão é feita considerando o tamanho do *buffers* do TCP do remetente e destinatário que estavam configurados com 128 Kbytes. Assim, como cada mensagem possuía o tamanho de 8 Kbytes o controle de fluxo permitia apenas o envio de 32 mensagens para evitar que os *buffers* do TCP sobrecarregassem. O ganho de cerca de 10% para a simples aplicação utilizada nesse experimento foi devido a evitar o congestionamento de mensagens, pois enquanto a rede estava congestionada a rotina de controle de fluxo recebeia mensagens da aplicação.

5.6 A Execução com Controle de Fluxo

O principal objetivo dessa seção é apresentar os resultados obtidos nos experimentos sobre os impactos da troca de mensagens em aplicações BoTs. Para analisar o comportamento do controle de fluxo foram realizados testes variando a quantidade de tarefas da aplicações e o tamanho das mensagens enviadas pelas aplicações.

Nos testes realizados, foram avaliadas aplicações BoTs onde o peso computacional de cada tarefa escrava variou entre 1 e 5 segundos e o tamanho das mensagens enviadas pela tarefa mestre irá varia de 24 bytes até 1 Mbyte.

O número máximo de mensagens pendentes no sistema, sem a confirmação de que já foram processadas pelos gerenciadores das máquinas, será denotado por *NCtrl*, que será calculada conforme apresentado na seção 4.2.5 do capítulo 4. O valor desse parâmetro é utilizado pelo controle de fluxo para restringir o número de envios da tarefa mestre. Para permitir uma sobreposição de envio de mensagens na execução da aplicação o controle de fluxo libera o envio de 2x(*NCtrl*) e assim que forem processadas *NCtrl* mensagens o controle de fluxo permite o envio de mais *NCtrl*, como já descrito anteriormente no Capitulo 4 dessa dissertação.

Nos experimentos que serão apresentados nessa seção foi utilizado um escalonamento estático *round robin* e o número de processos executados concorrentemente irá variar entre um e dois processos por núcleo, conforme será mostrado a cada teste.

5.6.1 Impacto do Congestionamento de Mensagens no *Makespan*

Este experimento mostrará que a utilização de um escalonador estático inadequado pode contribuir para o congestionamento de mensagens gerando impactos no *makespan* da aplicação. Nesse experimento foram utilizadas 5 máquinas da configuração 2, onde o GM e SM compartilham a mesma máquina na execução e as outras 4 máquinas executavam processos escravos da aplicação. Cada processo escravo executava por um segundo no processador para focar na comunicação e no *overhead* de mensagens do SGA.

Nos experimentos que seguem foram utilizados 3 escalonamentos estáticos diferentes:

- Sequencial - O escalonamento estático primeiro envia todas as mensagens alocadas para uma mesma máquina antes de enviar as mensagens da próxima máquina.
- Round Robin (RR) - Existe um rodízio no envio das mensagens, são feitas rodadas

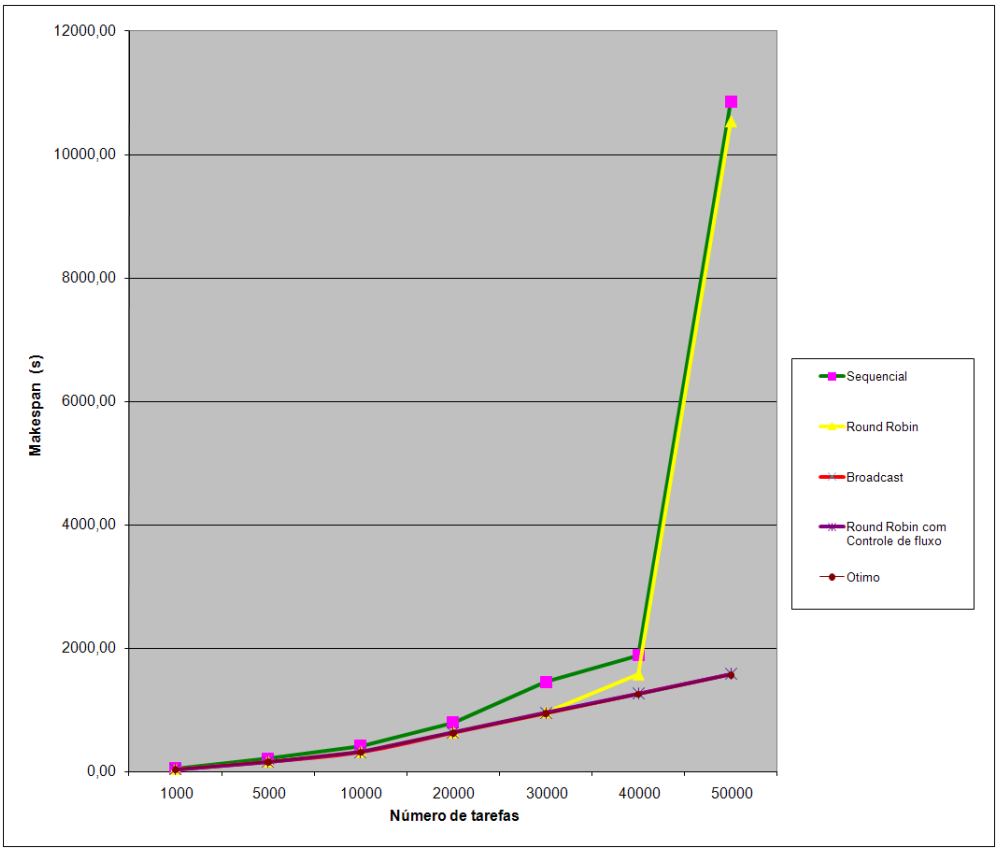

Figura 5.8: Impacto do congestionamento de mensagens no *makespan* da aplicação

enviando uma mensagem para cada máquina do ambiente paralelo até que não existam mais mensagem para ser enviadas.

- Round Robin (RR) com NCrtl = 256 - É utilizado o mesmo escalonamento Round Robin citado, mas com controle de fluxo.

- Broadcast - É enviada uma única mensagem da aplicação, os gerenciadores replicam a mensagem enviando uma cópia para cada processo gerenciador do nível inferior. Esse experimento representa o menor custo de comunicação possível, dado que só um única mensagem sai da tarefa mestre e somente uma única mensagem é recebida por cada gerenciador da máquina.

Tabela 5.7: Impacto do Congestionamento de Mensagens no *Makespan* da Aplicação

Estratégia	Sequencial	RR	Broadcast	RR com NCrtl = 256	Otimo
10000	410,91	315,99	315,91	315,93	312,5
20000	791,41	631,28	631,22	631,31	625
30000	1450,50	956,60	946,68	947,55	937,5
40000	1881,61	1569,15	1261,92	1262,66	1250
50000	>15000	10681,09	1577,63	1579,95	1562,5

A utilização do escalonamento sequencial faz com que um grande número de mensagem seja enviado para uma mesma máquina, criando um congestionamento de mensagem para a máquina escolhida e deixando as outras ociosas. O segundo escalonamento, o RR, já faz a distribuição das mensagens da aplicação de forma mais justa entre os recursos paralelos, no entanto, isso não é suficiente para aplicações grandes, onde mesmo utilizando um escalonamento RR, o número de mensagens enviados para uma mesma máquina pode gerar congestionamento.

Um melhor *makespan* é obtido através da utilização do controle do fluxo de mensagem, nesse caso como o ambiente paralelo possui 32 núcleos e o SGA estava configurado para permitir 2 processos por núcleo, o controle de fluxo no mínimo deveria ser configurado para permitir o envio de 64 mensagens. Como o tamanho das mensagens da aplicação nesse exemplo era de 24 *bytes* o controle de fluxo foi configurado para 256 mensagens, permitindo que cada núcleo recebesse 4x o número de mensagens necessárias para a sua execução. Para mensagens de 24 bytes o número de mensagens controle controladas não precisa ser muito rígido, pois como cada *buffer* do TCP possuía o tamanho de 128Kbytes seria necessário 10922 mensagens para o *buffer* do TCP ficar cheio e não permitir novo envios.

No SGA a função de comunicação *Broadcast* permite ao mestre transferir uma dado para todas as tarefas escravas através do envio de uma única mensagem. O gerenciador global recebe a mensagem e replica para cada um de seus gerenciadores de *site*. Cada *site* faz o mesmo, e cada gerenciador da máquina também replica a entrega para cada tarefa escrava. Por isso, a *Versão Broadcast* é importante para mostrar o impacto do envio das mensagens no *makespan* da aplicação. A *Versão Broadcast* representa o melhor *makespan* que pode ser obtido com o mínimo de comunicação. De forma análoga é apresentado o ótimo teórico, que considera que não ocorrerá o envio de qualquer mensagem.

Para aplicações com 50 mil tarefas utilizando o escalonador estático sequencial a execução não aconteceu após aguardar 15000 segundos para a *Versão Sequencial* e *Versão RR*. Isso ocorreu porque sem o controle de fluxo, os *buffer* do TCP são rapidamente sobrecarregados em aumento no tempo de comunicação entre máquinas.

As 50 mil tarefas de 24 *bytes* necessitavam de 1171 *Kbytes* e o *buffer* de envio e recebimento do MPI possuía 128 *Kbytes* cada um. A versão citada nesse experimento não possuía a otimização do sinal proposta em 4.2.3, e por isso executando na configuração 2 e sem o congestionamento de mensagens, o GM consegue tratar cerca de 83 mensagens por segundo. Quando os *buffers* do TCP estão cheio a sua capacidade de direcionar mensagens

para os SM reduz para cerca de 10 mensagens por segundo.

Em resumo foi observado que a utilização do controle de fluxo de mensagem propiciou uma redução no *Makespan* da aplicação evitando o congestionamento de mensagens e o atraso no envio do sinal.

5.6.2 A Utilização do Controle de Fluxo no SGA - Mensagens de 512 *bytes*

Nos experimentos que seguem mostram que o mecanismo de controle de fluxo traz bons resultados mesmo para aplicações que usam mensagem maiores do que as testadas até então, de 24 *bytes*. A simplicidade do controle de fluxo e a sua ação somente quando existem riscos de ocorrer um congestionamento, garante a estratégia implementada ter um impacto baixo na utilização da capacidade de processamento, fazendo com que o seu uso possa ocorrer mesmo para aplicação que não geram congestionamento na rede de comunicação. No experimento apresentado na tabela 5.8 e figura 5.9 a tarefa mestre envia uma mensagem de 512 *bytes* para cada tarefa escrava.

Tabela 5.8: Comparação do *makespan* entre execuções de aplicações BoTs com e sem controle de fluxo

N	Sem Controle de Fluxo	Com Controle de Fluxo	Ótimo Teórico
1000	32,40	32,25	31,25
5000	158,66	158,30	156,25
10000	315,91	315,94	312,5
20000	834,21	631,97	625
30000	1014,57	947,08	937,5
40000	1694,65	1262,24	1250
50000	10681,09	1579,44	1562,5

Observa-se que para as aplicações até 10 mil tarefas não existe a necessidade do controle de fluxo, mas também a sua utilização não impacta no *makespan* da aplicação. Para aplicações com 20 mil tarefas o controle de fluxo já reduz o *makespan* da aplicação, melhorando os resultado para as aplicações que enviam mensagens de 512 *bytes* e garantindo um *overhead* em relação ao ótimo de cerca de 1%.

5.6.3 A Utilização do Controle de Fluxo do SGA - Mensagens de 2K*bytes*

Os experimentos apresentados nas tabelas 5.9 e 5.10 e figuras 5.10, 5.11 e 5.12 foram realizados com o objetivo de mostrar os impactos do aumento do tamanho das mensagens

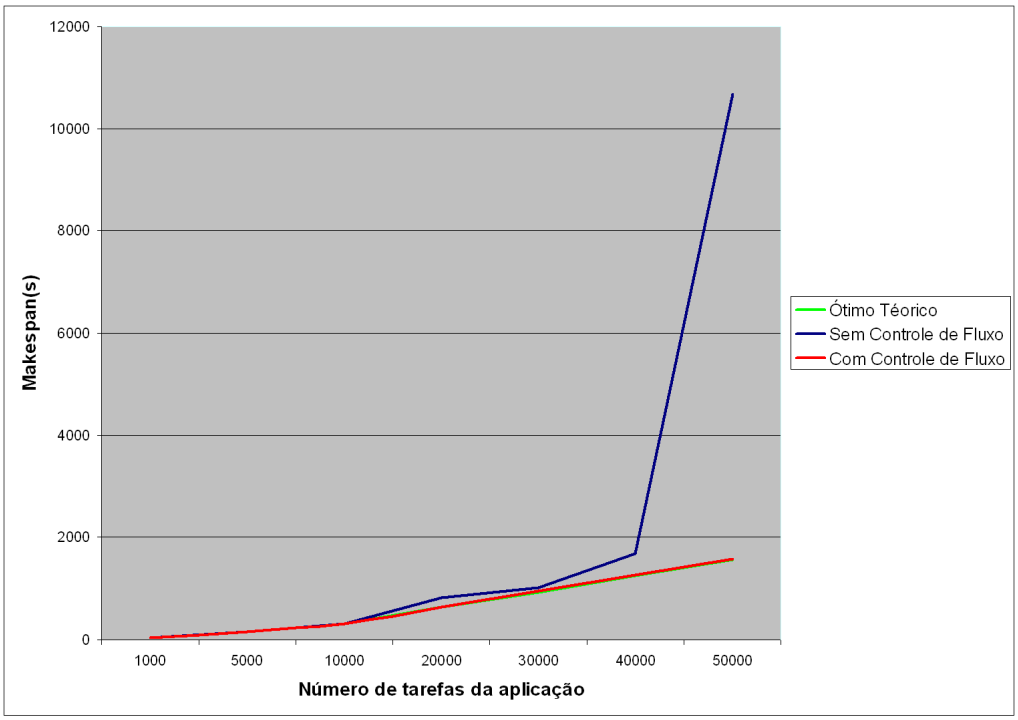

Figura 5.9: *Makespan* de Aplicações *BoTs* com tamanho de mensagem de 512 Bytes execuções com e sem controle de fluxo

enviadas no *makespan* da aplicação e no controle de fluxo de mensagens. Enquanto no experimento anterior, o controle de fluxo sempre garantia um bom resultado, neste veremos que isso não será sempre verdade. Para mensagens maiores, o controle de fluxo apenas garante um melhor resultado do que sem qualquer controle. Isso ocorre porque quanto maior o tamanho dos dados das mensagens mais rapidamente os *buffers* do TCP serão sobrecarregados gerando impactos na vazão de envio de mensagens do gerenciador global e aumentando o número de mensagens pendentes de confirmação do recebimento.

Para aplicações com 10 mil tarefas, mesmo sem o controle de fluxo o *makespan* não é impactado pelo congestionamento de mensagens. Já para aplicação com 20 mil tarefas, começa a ocorrer um congestionamento de mensagens ocasionando em um incremento do *makespan* a medida em que se tem um controle menos rígido do número de mensagens. O tempo de execução da aplicação sem o controle de fluxo chega próximo há duas vezes o tempo do ótimo teórico.

A execução de aplicações com 30 mil tarefas, já mostrou que mesmo com o controle de fluxo enviando um número reduzido de mensagens, o excesso de mensagens no *buffer* do TCP reduziu a vazão do redirecionamento do gerenciador global o que ocasiona em ociosidade nos gerenciadores de nível inferior impactando diretamente o envio do sinal que valida se a aplicação de destino está disponível para receber mensagens. Além disso, o

Tabela 5.9: *Makespan* de Aplicações *BoTs* variando o número de mensagens controladas

	Tempo de execução (seg)		
Controle de Fluxo	10 mil tarefas	20mil tarefas	30 mil tarefas
8	1265,04	2531,78	3797,87
16	632,57	1265,33	2137,33
32	319,69	642,16	1419,16
64	322,71	647,16	1407,59
128	316,52	638,86	1412,44
256	316,14	632,11	1406,42
512	316,27	632,41	1468,96
1024	315,86	631,37	1328,39
2048	316,17	631,63	1487,75
4096	316,12	631,53	1467,10
8192	316,01	968,59	1941,49
16384	316,10	1180,96	1997,14
Sem Controle de Fluxo	316,23	1180,96	1997,14
Ótimo Teórico	312,5	625	937,5

Tabela 5.10: Variação percentual do *makespan* das aplicações *BoTs* do experimento anterior em relação ao ótimo teórico

	Tempo de execução (seg)		
Controle de Fluxo	10 mil tarefas	20 mil tarefas	30 mil tarefas
8	404,8%	405,1%	405,1%
16	202,4%	202,5%	228,0%
32	102,3%	102,7%	151,4%
64	103,3%	103,5%	165,6%
128	101,3%	102,2%	150,7%
256	101,2%	101,1%	150,0%
512	101,2%	101,2%	156,7%
1024	101,1%	101%	141,7%
2048	101,2%	101,1%	158,7%
4096	101,2%	101,0%	156,5%
8192	101,1%	155,0%	207,1%
16384	101,2%	189,0%	213,0%
Sem Controle de Fluxo	101,2%	189%	213,0%

congestionamento de mensagens ocasiona no aumento do número de mensagens pendentes incrementado o tempo de cada envio, conforme foi mostrado através do experimento [**?**]. Os próximos testes desta seção mostrará que a otimização do envio sinal aliado ao controle de fluxo de mensagens e memória resolverá o problema de congestionamento mesmo para aplicação que enviam mensagens de 1 *Mbyte*.

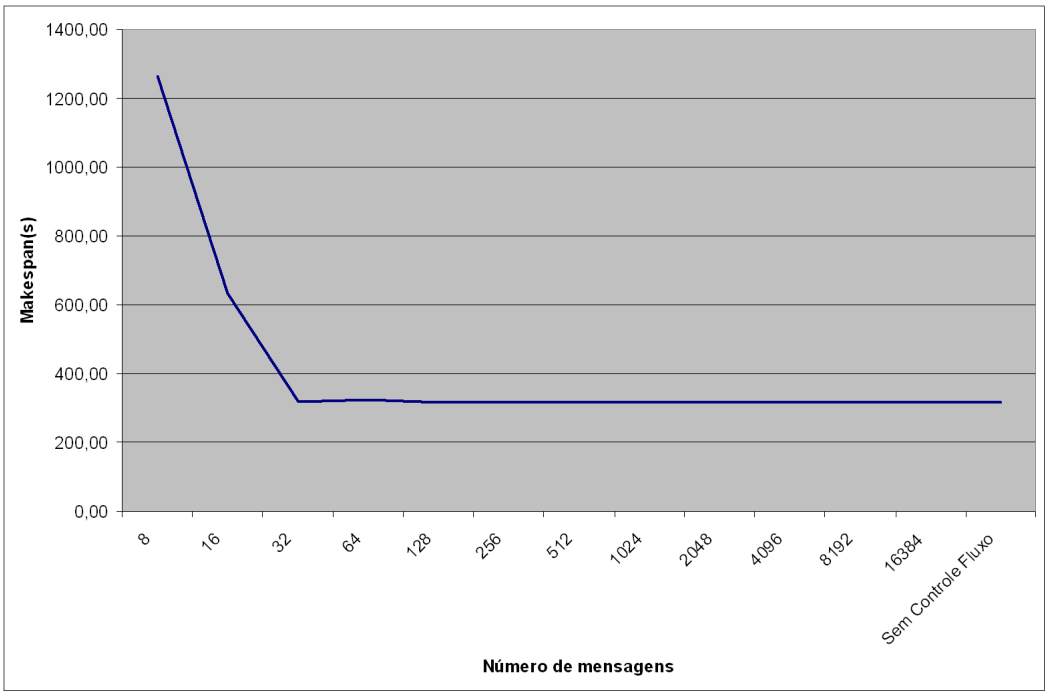

Figura 5.10: *Makespan* de uma aplicação *BoT* com 10 mil tarefas

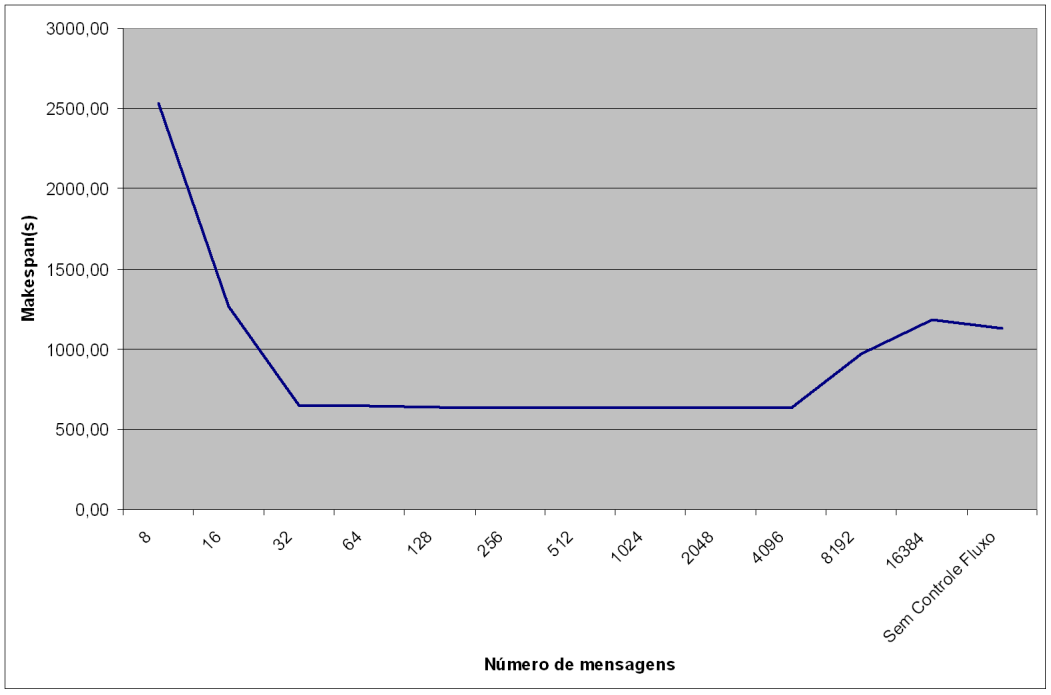

Figura 5.11: *Makespan* de uma aplicação *BoT* com 20 mil tarefas

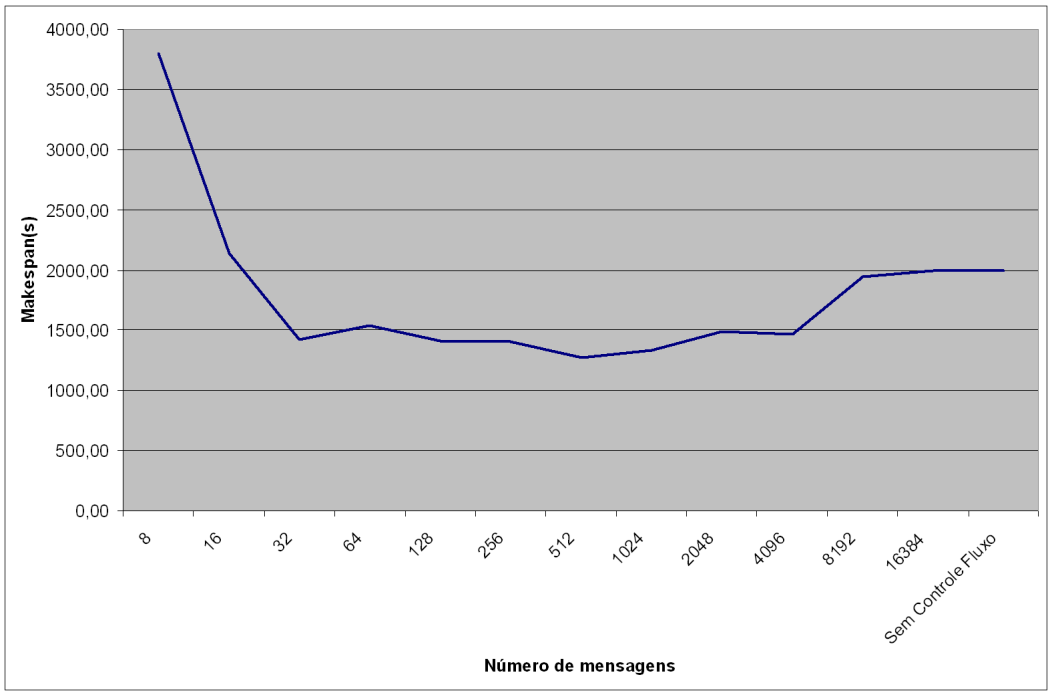

Figura 5.12: *Makespan* de uma aplicação *BoT* com 30 mil tarefas

5.7 A Nova Versão Com a Otimização da Memória e do Sinal

A nova versão do *EasyGrid SGA* que será testadas nesta seção possui a otimização do envio do sinal, o que permitiu ao SGA reduzir os tempos de envio das mensagens entre o GM e o SM, conforme apresentado no capítulo 4. Aliada a otimização do sinal a utilização de uma estratégia de controle de memória garante o sucesso da execução da aplicação por evitar o uso excessivo de memória principal. Todos os testes apresentados aqui foram executados na configuração 1, onde cada gerenciador da máquina executava um único processo da aplicação por processador e o custo computacional de cada processo da aplicação é de 1 segundo. O número de mensagens controladas pelo controle de fluxo já estava sendo calculado de forma automática conforme mostrado em 4.2.5, nestas execuções NCtrl utilizado foi de 64 mensagens, ou seja permitia a existência de pelo menso duas mensagens por núcleo, uma executando e outra na espera para ser executada.

5.7.1 Execução Sem o Controle da Memória

O resultado da execução do SGA quando não é feito qualquer controle de memória ou de fluxo de mensagens, mostra que o número excessivo de mensagens enviadas mas sem a confirmação do seu recebimento pela tarefa destino é a principal causa do uso excessivo da

memória. A tabela 5.11 mostra que sem qualquer controle de memória uma aplicação de 30 mil tarefas consome cerca de 1Gbyte de memória principal quando enviam mensagens de apenas 32 Kbytes. Vale lembrar, que como foi utilizada uma versão do MPI de 32 bits no máximo pode ser utilizado 4Gbytes da memória principal para cada gerenciador.

Tabela 5.11: Quantidade de memória utilizada sem controle de fluxo

	Quantidade de memória utilizada (Mbytes)		
Tamanho Mensagem/N	1 Kbyte	8 Kbytes	32 Kbytes
10000	20,0	91,5	331,5
20000	40,6	180,6	660,6
30000	59,6	269,6	989,0

5.7.2 Execução com o Controle da Memória

Para a execução de aplicações grande é necessária também a utilização de uma estratégia de controle de memória de forma a garantir a execução da aplicação sem falhas. Basicamente, no SGA foram mapeados 3 parâmetro para definir a quantidade de memória utilizada pela aplicação:

- A *Mem Isend* que representa a quantidade de memória utilizada pelas mensagens enviadas através do comando *MPI_ Isend*, mas ainda pendentes a confirmação de recebimento;
- A *Mem Ctrl* que define a quantidade de memória utilizada pelas mensagens que ainda aguardam a confirmação do sucesso da execução da tarefa escrava e
- A *Mem Hash* que caracteriza a quantidade de memória necessária para a tabela *Hash*.

Através do acompanhamento desses 3 parâmetros é possível obter um valor bastante aproximado da memória utilizado pelo SGA. A tabela 5.12 apresenta a quantidade de memória utilizada na execução de cada uma das instâncias da aplicação.

Observa-se que a quantidade de *Mem Isend* sempre fica próximo de *Mem Ctrl*, isso porque o controle de fluxo limita a quantidade de mensagens pendentes ao número de mensagens controladas, por isso, A quantidade de memória utilizada pelo *Mem Ctrl* sempre representa o somatório do tamanho das 128 mensagens permitidas pelo controle de fluxo de mensagem.

Tabela 5.12: Quantidade de memória utilizada com a utilização do controle de memória

		Quantidade de memória utilizada (Mbytes)			
N	Tam da Msg	Mem Isend	Mem Ctrl	Mem Hash	Total
10000	1 Kbyte	0,01	0,125	6,33	6,47
20000	1 Kbyte	0,01	0,125	12,67	12,81
30000	1 Kbyte	0,01	0,125	19	19,14
10000	8 Kbytes	0,84	1	6,33	8,16
20000	8 Kbytes	0,83	1	12,67	14,49
30000	8 Kbytes	0,81	1	19	20,8
10000	32 Kbytes	3,75	4	6,33	14,05
20000	32 Kbytes	3,81	4	12,67	20,45
30000	32 Kbytes	3,81	4	19	26,78
10000	1 Mbyte	103	128	6,33	236,33
20000	1 Mbyte	116	128	12,67	255,67
30000	1 Mbyte	117	128	19	263

Na tabela 5.13 segue uma comparação entre a versão com e sem o controle de memória, onde a versão sem o controle de memória chegou a consumir mais de 35 vezes mais memória principal do que a mesma aplicação sem esse controle.

Tabela 5.13: Comparação entre a versão sem e com o controle de memória

	Versão sem Ctrl /Versão com Ctrl(%)		
Tamanho Mensagem/N	1 Kbyte	8 Kbytes	32 Kbytes
10000	309%	1121%	2359%
20000	316%	1242%	3230%
30000	311%	1296%	3693%

Pelo resultado apresentado na tabela 5.11 é possível verificar que a versão com o controle de fluxo restringe a utilização da memória para apenas o subconjunto de mensagens mais prioritárias, as que o controle de fluxo permitiu a aplicação enviar sem a confirmação da utilização dessas mensagens. A execução das aplicações que enviam mensagens com mais do que 1 Mbyte não executou sem o controle de memória por necessitar de mais do que 4 Gbytes de memória principal, o limite de uso para uma aplicação de 32 bits.

5.7.3 O Controle de Mensagens e Memória com o Sinal Otimizado

Conforme mostrado na subseção 4.2.3 do capítulo 4, a otimização do sinal consistiu em buscar alternativas de validar mais rapidamente que o destino está ativo, permitindo ao gerenciador global aumentar a vazão de encaminhamento de mensagens. O experimento apresentado pela tabela 5.14 e figuras 5.13 e 5.14 foi realizado com o objetivo de mostrar

que a nova versão do SGA utilizando o controle de fluxo, o controle de memória e o sinal otimizado foi possível obter bons *makespan* mesmo para as aplicações que enviam mensagens maiores do que as utilizadas nos exemplos anteriores.

Tabela 5.14: *Makespan* das aplicações com controle de fluxo e com o sinal otimizado

		Tempo de execução (seg)					
N	Tam Msg	1	2	3	Média	Ótimo Teórico	% Ótimo
10000	1 Kbyte	182,3	182,03	182,44	182,26	179	101,8%
20000	1 Kbyte	363,19	363,09	363,18	363,15	358	101,4%
30000	1 Kbytes	544,53	544,59	544,43	544,52	536	101,6%
10000	8 Kbytes	182,29	182,28	182,23	182,27	179	101,8%
20000	8 Kbytes	362,97	363,68	363,57	363,41	358	101,5%
30000	8 Kbytes	544,03	544,67	544,73	544,48	536	101,6%
10000	32 Kbytes	182,22	182,28	182,11	182,20	179	101,8%
20000	32 Kbytes	363,7	363,54	363,44	363,56	358	101,6%
30000	32 Kbytes	545,07	544,73	545,04	544,95	536	101,7%
10000	1 Mbyte	185,54	183,35	182,76	183,88	179	102,7%
20000	1 Mbyte	369,75	370,29	369,29	369,78	358	103,3%
30000	1 Mbyte	554,5	554,15	554,81	554,49	536	103,4%
40000	1 Mbyte	739,17	738,29	738,57	738,68	715	103,3%
50000	1 Mbyte	930,25	922,86	922,58	925,23	893	103,6%
100000	1 Mbyte	1845,18	1846,44	1860,62	1850,75	1786	103,6%
200000	1 Mbyte	3700,13	3695,99	3690,58	3695,57	3572	103,5%
500000	1 Mbyte	9117,09	9351,01	9309,01	9259,04	8929	103,7%
1000000	1 Mbyte	18746,67	18741,39	18941,69	18809,92	17858	105,3%

Como ilustrado na figura 5.13, as aplicações que enviam mensagens de 1Kbyte até 32Kbytes não representaram impacto no *makespan* da aplicação, diferentemente do observado no experimento da tabela 5.10 onde o envio do sinal ainda não estava otimizado. O controle de fluxo de mensagens evitou o congestionamento na rede e o controle de memória evitou uma utilização excessiva da memória principal permitiu que a conclusão da execução. A otimização do sinal fez com que as mensagens da tarefa mestre chegassem para as tarefas escravas em uma vazão maior do que as tarefas escravas eram executadas evitando a ociosidade nos HM.

A figura 5.14 mostra que mesmo a tarefa mestre enviando mensagens de $1Mbyte$ também é obtido bons *makespans* com pouco *overhead* em relação ao ótimo teórico. Para comprovar que as otimizações de estrutura de dados e a utilização dos controles de fluxo e memória deixaram o *SGA EasyGrid* mais robusto, foram executadas aplicações com tamanho de até 1 milhão de tarefas. Conforme pode ser verificado na tabela 5.14 e figura 5.14, o *makespan* continuou apresentando valores dentro do previsto quando

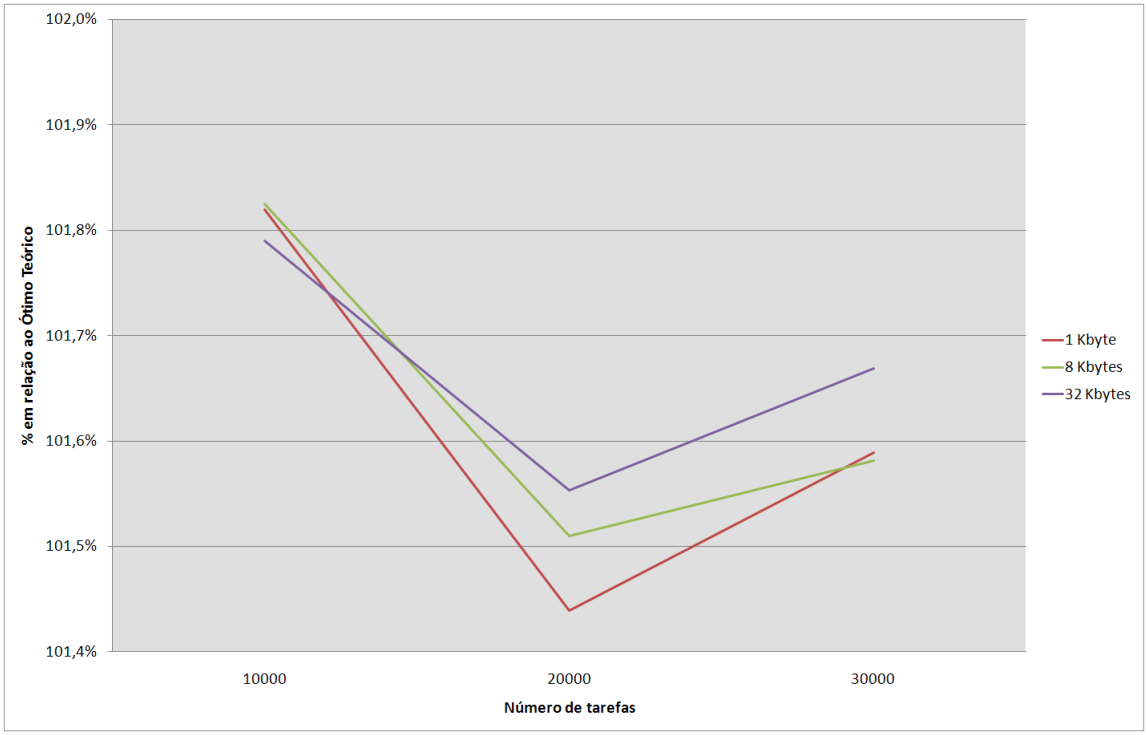

Figura 5.13: Comparação percentual entre o *makespan* obtido pelo SGA e o Ótimo Teórico

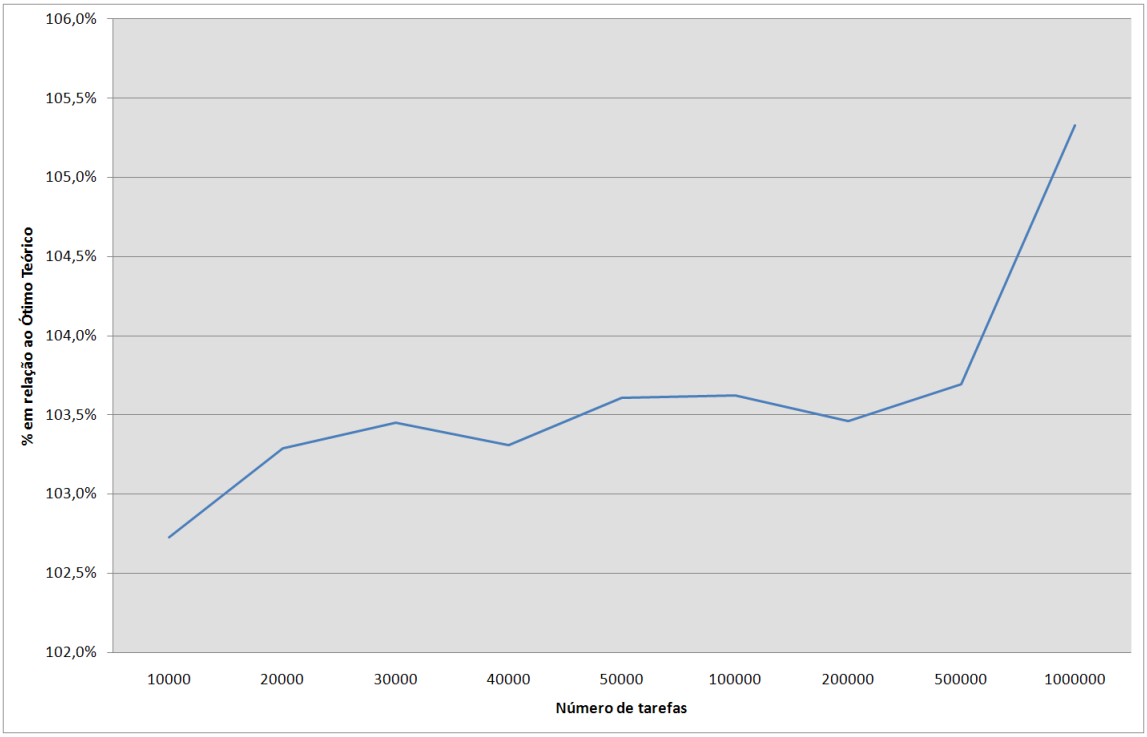

Figura 5.14: Comparação percentual entre o *makespan* obtido pelo SGA e o Ótimo Teórico - Apenas são enviadas mensagens de 1 Mbytes

comparado com o ótimo teórico. Esses testes com o sinal otimizado foram feitos sem as funcionalidade de tolerância a falhas, pois conforme apresentado pelo experimento 5.4 a utilização dessa funcionalidade gera um *overhead* para a inicialização de aplicações maiores, gerando ociosidade em todos os HM antes do inicio da execução. Os resultados mostrados comprovam que o controle de fluxo garante bons resultados mesmo onde o tamanho das mensagem é grande e a aplicação possui um número gigantesco de tarefas e mais de um Terabytes de dados foram trocados entre as aplicações.

5.8 Resumo

Esse capítulo apresentou uma análise dos resultados obtidos através da utilização do controle de fluxo de mensagens e memória é possível obter uma melhoria no *makespan* e ainda evitar erro de execução devido a congestionamento na rede ou falta de memória. Os *makespans* obtido mostram uma redução na intrusão dos gerenciadores do SGA em relação a versão inicial, principalmente através do uso das otimizações citadas no capítulo anterior, cujos resultados foram apresentados nesse capítulo.

Capítulo 6

Conclusões e Trabalhos Futuros

As grades computacionais se tornaram atraentes por oferecerem grande capacidade de processamento a um baixo custo e por isso é foco de pesquisa intensiva. Um dos principais objetivos dos pesquisadores é criar soluções para o gerenciamento das aplicações paralelas e distribuídas que compartilham os recursos de uma grade, de forma a proporcionar uma execução eficiente e segura.

Um ponto essencial para uma execução eficiente é que o sistema gerenciador de grades, além de controlar o escalonamento das tarefas entre os recursos de processamento existentes precisa também gerenciar os recursos de comunicação e armazenamento. Podendo assim, adaptar a execução da aplicação de acordo com o poder computacional da grade, a quantidade de memória e a capacidade de comunicação disponível. O estudo apresentado nessa dissertação se concentra no controle de mensagens e memória para as grades computacionais. Para melhor entendimento das necessidades de tais controles várias características relacionadas ao gerenciamento de uma aplicação na grade foram estudadas e avaliadas.

O SGA *EasyGrid* em desenvolvimento na Universidade Federal Fluminense [12, 13, 16, 39] é um sistema gerenciador de aplicações, responsável por gerenciar e automatizar aplicações para que estas executem de forma eficiente e segura nos vários recursos disponíveis de uma grade. A criação de ferramentas que permitem que a aplicação do usuário seja auto-gerenciável é fundamental para que se minimize o esforço de cientistas ao se utilizar uma grade computacional.

Apesar de existirem vários trabalhos que tratam o escalonamento de tarefas, em especial sobre as aplicações *BoT*, o estudo sobre o controle de fluxo de mensagens ainda não se tornou alvo de análises mais aprofundadas sobre como melhorar o escalonamento

de tarefas. Isso se torna em uma necessidade no SGA pois nesse sistema gerenciador não existe qualquer restrição do número de tarefas da aplicação do usuário.

As estratégias utilizadas nessa dissertação para aplicações do usuário do tipo BoT *bag-of-tasks* pode ser estendidas para outras classes de aplicação, mas as aplicações BoT foram as escolhidas por demandar mais recursos de memória e comunicação do que as outras existentes. Através da análise dos experimentos realizados em ambientes paralelos, pode-se afirmar que a simplicidade da política de controle de fluxo e memória implementadas garantem um baixo custo de processamento para a gerência da aplicação.

Um dos objetivos principais desta dissertação é prover o SGA*EasyGrid* com uma eficiente estrutura de gerenciamento de recursos de rede e memória, que possa garantir a execução de aplicação de larga escala, aquelas que possuem mais de 10 mil processos ou que trafegam mensagens maiores que $2Kbytes$. Assim, foi criada uma camada entre a aplicação do usuário e o gerenciador de nível mais alto que controla o fluxo de mensagens e memória utilizada pela aplicação. Conforme apresentado nos experimentos, o controle das mensagens garante uma melhor utilização dos recursos da grade ao permitir que um número controlado de tarefas estejam prontas e necessitem de uma análise mais apurada dos gerenciadores. A restrição do número de mensagens em aplicação BoTs mostrou-se eficiente para controlar a quantidade de memória principal e evitar congestionamentos de mensagens, isso permite um ganho indireto obtido por permitir que os gerenciadores administrem um número reduzido de tarefas prontas, trazendo uma maior agilidade na gerencia da aplicação e no escalonamento dinâmico da aplicação.

A camada de controle de fluxo incluída no SGA, assim como as outras funcionalidades já existentes nesse sistema gerenciador, são embutidas na própria aplicação MPI do usuário. São criadas, portanto, aplicações cientes do ambiente (*system aware*) que são capazes de se auto-otimizar de acordo com as variações do ambiente computacional. É importante ressaltar que não são feitas modificações no código da aplicação do usuário, as funcionalidades relativas ao gerenciamento da aplicação na grade são embutidas automaticamente no código, permitindo que as complexidades impostas pelo ambiente sejam transparentes ao usuário. Esta abordagem aumenta a quantidade de aplicações MPI pré-existentes que podem ser executadas de forma eficiente em um ambiente grade.

Neste trabalho, também foi realizado um estudo sobre o comportamento dos comandos de envio do MPI quando um remetente envia mensagens mais rapidamente do que o destino pode recebê-las. Os testes comprovam que restringir o envio de mensagens à velocidade em que o destino possa recebê-la aumenta a eficiência da comunicação e reduz

o *makespan* de uma aplicação ao evitar a sobrecarga na comunicação.

Foi mostrado também que o controle de fluxo do TCP mesmo sendo uma importante ferramenta de gerência da rede pode impactar nas mensagens de controle entre os gerenciadores, quando os *buffer* disponibilizados pelo TCP estão cheios. Isso ocorre porque o TCP não conhece o tipo das mensagens trafegado, bloqueando até as mensagens de monitoramento, como o envio do sinal que verificar se o destino está ativo. O principal impacto desse bloqueio é o aumento do tempo de resposta das mensagens de monitoramento, ocasionado em redução da vazão de envio de novas mensagens da tarefa mestre para as escravas e gerando ociosidade nos HMs.

Um outro importante estudo realizado nessa dissertação foi a implementação de estruturas de dados mais robustas para guardar os dados utilizados na gerência da aplicação. A utilização de tabela *Hash* ao invés de uma estrutura vetorial permite um controle mais fino do uso de memória sem trazer grandes impactos no tempo de execução do gerenciador. Assim como, a utilização de índices nas estruturas de listas de tarefas apontando para o provável elemento a ser buscado, pode torna as buscas mais eficientes contribuindo para reduzir a intrusão dos gerenciadores na execução da aplicação e permitindo uma melhor eficiência na execução de aplicação com grande número de tarefas.

Com relação aos trabalhos futuros, novas pesquisas pode ser feita considerando aplicações que possuam relações de precedência, mas espera-se que a relação de precedência entre as tarefas restrinja a utilização da rede reduzindo as falhas relacionadas ao uso excessivo da rede de comunicação. Essa foi a principal razão para que o estudo dessa dissertação concentra-se nas aplicações BoTs, onde o padrão de comunicação dessas aplicações as tornam mais propícias a gerar congestionamento de mensagens.

Outro ponto que poderá ser foco de pesquisas posteriores é o cálculo dinâmico do número de mensagens permitidas pelo controle de fluxo, isso trará benefício quando for utilizada uma grade real, onde máquinas podem deixar de pertencer a grade em tempo de execução. Além disso, pode ser expandida a verificação do uso de memória ao gerenciador da máquina e implementar estratégias que possa identificar outras aplicações executando concorrentemente e definir de forma dinâmica o total de memória permitida para cada gerenciador. Análises futuras também podem ser feitas sobre a utilização do controle de fluxo entre os SM e HM, assim, novos estudos podem ser feitos com sites que possuem latências diferentes onde o controle de fluxo deverá se comportar considerando a latência entre cada site.

Referências

[1] Abley, J., Savola, P., and Neville-Neil, G. Deprecation of Type 0 Routing Headers in IPv6. RFC 5095 (Proposed Standard), dezembro de 2007.

[2] Allman, M., Paxson, V., and Blanton, E. TCP Congestion Control. RFC 5681 (Draft Standard), setembro de 2009.

[3] Allman, M., Paxson, V., and Stevens, W. TCP Congestion Control. RFC 2581 (Proposed Standard), abril de 1999. Obsoleted by RFC 5681, updated by RFC 3390.

[4] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. Seti@home: an experiment in public-resource computing. *Communications of the ACM 45*, 11 (2002), 56–61.

[5] Boeres, C., Fonseca, A. A., Mendes, H. A., Menezes, L. T., Moura, N. T., Silva, J. A., Vianna, B. A., and Rebello, V. E. F. An EasyGrid Portal for scheduling system-aware applications on computational grids. *Concurrency and Computation: Practice and Experience 18*, 6 (2005), 553–566.

[6] Braden, R. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), outubro de 1989. Updated by RFCs 1349, 4379.

[7] Buyya, R., Ed. *High Performance Cluster Computing: Architectures and Systems.* Vol. 1. Prentice Hall, EUA, 1999.

[8] Buyya, R., Abramson, D., and Giddy, J. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *Proceedings of the 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region (HPC ASIA'2000)* (Beijing, China, Maio 2000), vol. 1, IEEE Computer Society, pp. 283–289.

[9] Casanova, H., Legrand, A., Zagorodnov, D., and Berman, F. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings 9th Heterogeneous Computing Workshop (HCW '00)* (Cancun, Mexico, Maio 2000), IEEE Computer Society, pp. 349–363.

[10] Chapin, S., Katramatos, D., Karpovich, J., and Grimshaw, A. S. The legion resource management system. In *Proceedings of the Job Scheduling Strategies for Parallel Processing* (Londres, UK, 1999), Springer Verlag, pp. 162–178.

[11] Cosnard, M., and Trystram, D. *Parallel Algorithms and Architectures.* PWS Publishing Co., Boston, EUA, 1995.

[12] da Costa Sena, A. *Um Modelo Alternativo para Execução Eficiente de Aplicações Paralelas MPI nas Grade Computacionais.* PhD thesis, Universidade Federal Fluminense - Instituto de Computação, 2008.

[13] da Silva, A., and Rebello, V. E. F. Low cost self-healing in MPI applications. In *Proceedings of the 14th European PVM/MPI User's Group Meeting, Paris, France* (Outubro 2007), Springer, pp. 144–152.

[14] da Silva, F. A. B., Carvalho, S., and Hruschka, E. R. A scheduling algorithm for running bag-of-tasks data mining applications on the grid. In *Euro-Par* (2004), pp. 254–262.

[15] de Ameida, E. S. Climatologia de mesoescala em grade computacional. Dissertação de Mestrado, Instituto Nacional de Pesquisas Espaciais, 2007.

[16] de Campos Vianna, D. Q. Um sistema de gerenciamento de aplicações mpi para ambientes grid. Dissertação de Mestrado, Universidade Federal Fluminense - Instituto de Computação, 2005.

[17] Dongarra, J. Dunigan, T. Messaging-passing performance of various computers. *Concurrency and Computation: Practice and Experience 9* (1997), 915–926.

[18] Forum, M. Message passing interfae forum:mpi 2 disponível em http://www.mpi-forum.org/, última visita em 11 de maio de 2009. Tech report, Oi, 2008.

[19] Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, Julho 1997. `http://www.mpi-forum.org/`, Último acesso 10/10/2008.

[20] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, junho 1995. `http://www.mpi-forum.org/`, Último acesso 10/10/2008.

[21] Foster, I., and Kesselman, C. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing 11*, 2 (Summer 1997), 115–128.

[22] Foster, I., and Kesselman, C., Eds. *The GRID 2: Blueprint for a New Computing Infrastructure.* 2^a edição. Morgan Kaufmann, 2004.

[23] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., Nova Iorque, EUA, 1979.

[24] Gooding, S., Arns, L., Smith, P., and Tillotson, J. Implementation of a distributed rendering environment for the teragrid. In *Challenges of Large Applications in Distributed Environments, 2006 IEEE* (0-0 2006), pp. 13–21.

[25] Gray, J. What next?: A dozen information-technology research goals. *Communications of the ACM 50*, 1 (2003), 41–57.

[26] Grossman, D. New Terminology and Clarifications for Diffserv. RFC 3260 (Informational), abril de 2002.

[27] Hinden, R., and Deering, S. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), fevereiro de 2006.

[28] Horowitz, E., and Sahni, S. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM 23*, 2 (1976), 317–327.

[29] Horowitz, E., and Sahni, S. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM 23*, 2 (1976), 317–327.

[30] Huedo, E., Montero, R. S., and Llorente, I. M. Experiences on adaptive grid scheduling of parameter sweep applications. In *Proceedings of the 12th Euromicro Conference of Parallel, Distributed and Network-Based Processing (PDP 2004)* (La Coruña, Spain, Fevereiro 2004), IEEE Computer Society, pp. 28–33.

[31] Ibarra, O. H., and Kim, C. E. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM 24*, 2 (1977), 280–289.

[32] Jacobson, V., Braden, R., and Borman, D. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), maio de 1992.

[33] Kruse, R., Leung, B., and Tondo, C. L. *Data Structures and Program Design In C*. Prentice Hall, 1996.

[34] Kruse, R., and Ryba, A. *Data Structures and Program Design in C++*. Prentice Hall, 1998.

[35] Kurose, J. F., and Ross, K. W. *Redes de Computadores e a Internet: Uma abordagem top-down*, trad. 3 ed. ed. Addison Wesley, São Paulo, 2006.

[36] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), outubro de 1996.

[37] Mogul, J., and Deering, S. Path MTU discovery. RFC 1191 (Draft Standard), novembro de 1990.

[38] Moy, J. Extending OSPF to Support Demand Circuits. RFC 1793 (Proposed Standard), abril de 1995. Updated by RFC 3883.

[39] Nascimento, A. P. *Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, Brasil, Maio 2008.

[40] Nascimento, A. P., Boeres, C., and Rebello, V. E. F. Dynamic self-scheduling for parallel applications with task dependencies. In *Proceedings of the 6th international Workshop on Middleware for Grid Computing* (Leuven, Belgium, December 2008), ACM Press.

[41] Nascimento, A. P., Sena, A. C., Boeres, C., and Rebello, V. E. F. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience 19*, 14 (2007), 1955–1974.

[42] Nascimento, A. P., Sena, A. C., da Silva, J. A., Vianna, D. Q. C., Boeres, C., and Rebello, V. Managing the execution of large scale MPI applications on computational grids. In *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)* (Rio de Janeiro, Brasil, Outubro 2005), IEEE Computer Society.

[43] Nascimento, A. P., Sena, A. C., Silva, A., Vianna, D. Q. C., Boeres, C., and Rebello, V. E. F. Autonomic application management for large scale mpi programs. *Int. J. High Perform. Comput. Netw. 5*, 4 (2008), 227–240.

[44] Papadimitriou, C., and Yannakakis, M. Towards an architecture independent analysis of parallel algorithms. *SIAM Journal on Computing 19*, 2 (Abril 1990), 322–328.

[45] Postel, J. TCP maximum segment size and related topics. RFC 879, novembro de 1983.

[46] Ramakrishnan, K., Floyd, S., and Black, D. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), setembro de 2001.

[47] Rodrigues, H., Trannin, H., Sena, A. C., and Rebello, V. Usando grades computacionais para avaliação de heurísticas de escalonamento de tarefas. In *Proceedings of the VI Workshop em Sistemas Computaionais de Alto Desempenho (WSCAD 2005)* (Outubro 2005), M. C. S. de Castro and E. Midorikawa, Eds.

[48] Sinnen, O. *Task Scheduling for Parallel Systems*. Wiley, 2007.

[49] Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.

[50] Ullman, J. D. NP-complete scheduling problems. *Journal of Computer and System Sciences 10*, 3 (1975), 384–393.