Universidade Federal Fluminense

Daniela Quitete de Campos Vianna

Um Sistema de Gerenciamento de Aplicações MPI para Ambientes Grid

NITERÓI

2005

Universidade Federal Fluminense

Daniela Quitete de Campos Vianna

Um Sistema de Gerenciamento de Aplicações MPI para Ambientes Grid

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. Área de concentração: Processamento Paralelo e Distribuído.

Orientador:
Prof. Eugene Francis Vinod Rebello. PhD.

NITERÓI

2005

Um Sistema de Gerenciamento de Aplicações MPI para Ambientes Grid

Daniela Quitete de Campos Vianna

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. Área de concentração: Processamento Paralelo e Distribuído.

Aprovada por:

Prof. Eugene Francis Vinod Rebello, PhD / IC-UFF
(Orientador)

Prof. Alexandre Plastino de Carvalho, DSc / IC-UFF

Profa. Maria Cristina Silva Boeres, PhD / IC-UFF

Profa. Noemi de La Rocque Rodriguez, DSc / PUC-Rio

Prof. Siang Wun Song, PhD / IME-USP

Niterói, 3 de Outubro de 2005.

Aos meus avós, pais e irmãos.

Agradecimentos

Foram muitas as pessoas que de forma direta ou indireta contribuíram para que fosse possível a realização de mais esse sonho: o mestrado. Essa é uma oportunidade de agradecer a cada uma dessas pessoas e dizer que sem elas nada disso seria possível.

Antes de mais nada, gostaria de agradecer a Deus por todas as maravilhas que a vida nos oferece e por nos dar a oportunidade de aproveitá-las ao máximo.

Papai (Antônio Claudio), mamãe (Lícia) e maninhos (Flávio e Luís Felipe), vocês são os principais responsáveis por tudo o que sou e tudo o que consegui até hoje. Vocês são o meu chão, minha força, minha inspiração e fazem cada segundo da minha vida valer a pena. Muito obrigada!!!!

Três grandes amigos merecem um agradecimento mais que especial nesse trabalho: grande Jack, Joni ("corretor ortográfico") e Vivizinha. Vocês são amigos para todas as horas, e qualquer coisa que eu diga não será suficiente para expressar a importância da amizade de vocês na minha vida. Também não posso deixar de agradecer as eternas amigas Laura e Renata.

A vida sempre nos reservas grandes surpresas, e no final do mestrado encontrei o que procurava há muito tempo: um amor. Uma pessoa para confiar, dividir, sonhar... encontrei você, Rodrigo. Você transformou os últimos dias do meu mestrado em momentos suaves e felizes; fez com que eu me sentisse mais confiante e orgulhosa do meu trabalho; foi paciente, amigo, incentivador... o que mais dizer???

O laboratório da Pós, além de ser um local sério de trabalho, também pode ser considerado um local estratégico para tratar de qualquer assunto que não seja trabalho. No laboratório acontecem as melhores discussões e são tramados os mais divertidos encontros: festas, churrascos, encontros no bar, esportes... rola de tudo!!!. Valeu galera do lab pelo incentivo, discussões e risadas. Um agradecimento especial à turma do SGCLab: Deolinda, Helder, Viterbo, Jacques, Idalmis, Camilo, Felipe, Henrique, Hildebrando, Bruno, Nilmax, Aline e Alexandre.

Aos amigos: Alex, Ary Henrique, Bruno ("Pessoa" – companheiro de batalhas e de caipirinhas), Cristiano, Eyder, Glauco, Ivairton, Johnny, Juliana, Kennedy, Luciana Brugiolo, Luciana e Robson, Luciano, Luis Valente, Marcelo Pires, Sandoval, Stênio, Tiago (Facada), Tiago (Jovem) e todos que de alguma forma estiveram presente nesses meses de muita luta, mas também muito divertimento.

Às meninas da república, com quem dividi e ainda divido momentos especiais e cotidianos da minha vida: Cris, Maysa, Renathinha e Vivi.

Agradeço aos funcionários e amigos Carlinhos, Izabela, Angela e Maria.

À todos os professores do IC, em especial aos professores Júlio Stacchini e Julius Leite, pelos esforços em conseguir uma bolsa emergencial que suprisse os cortes de bolsas sofridos pelo programa.

Aos professores Alexandre Plastino e Otton. Vocês foram mais do que professores, foram amigos, conselheiros, incentivadores e despertaram em mim o interesse pela pesquisa e pela vida acadêmica. Vocês são exemplos que levarei para sempre!

Aos membros da banca pelas contribuições apresentadas.

Agradeço à FAPERJ por financiar parte do meu mestrado.

E finalmente, ao meu orientador Vinod. Foram meses de convivência, discussões, troca de idéias, conselhos... Só tenho a agradecer e dizer que foi ótimo trabalhar com você. Com você aprendi cada vez mais a aprender, a pesquisar, a questionar; aprendi o valor de uma boa pergunta diante de mil respostas e acima de tudo passei a admirar ainda mais o profissional Vinod. Muito obrigada!!! Aí está o resultado do nosso tão sonhado trabalho!!!

Resumo

Os Grids Computacionais surgiram com o intuito de disponibilizar um ambiente para computação de alto desempenho, onde usuários que não possuem recursos locais suficientes possam executar suas aplicações. Entretanto, os recursos distribuídos do Grid são tipicamente heterogêneos, não dedicados e oferecidos sem nenhuma garantia de desempenho e disponibilidade. Como conseqüência, o desenvolvimento de aplicações eficientes para esse tipo de ambiente ainda é uma tarefa difícil. Este trabalho propõe uma nova abordagem, capaz de amenizar a dificuldade no desenvolvimento de aplicações para Grid, além de oferecer mecanismos de gerenciamento eficientes para execução das mesmas.

Atualmente, grande parte dos desenvolvedores de *middleware* Grid tomam por base um Sistema de Gerenciamento de Recursos que objetiva maximizar a utilização dos recursos disponíveis no ambiente Grid. Esses sistemas são capazes de fornecer serviços tais como escalonamento e tolerância a falhas, sendo todas as decisões tomadas com base apenas no estado dos recursos. A abordagem proposta neste trabalho, baseia-se na utilização de um Sistema de Gerenciamento da Aplicação (SGA) próprio para cada aplicação. O SGA utiliza o Grid de acordo com a disponibilidade dos recursos e com características específicas de cada aplicação. O SGA EasyGrid é embutido na aplicação em tempo de compilação, que resulta em uma aplicação *system-aware* (Smart G-App), sem que nenhuma alteração seja realizada no código fonte original do usuário. Além disso, o SGA EasyGrid foi desenvolvido utilizando uma implementação *padrão* do MPI, o que garante a portabilidade da aplicação gerada. Resultados experimentais mostram eficiência na execução das aplicações MPI *system-aware*, quando comparadas com as versões originais desenvolvidas para *clusters* de computadores.

Abstract

Computational Grids aim to make high performance computing available to users that do not have enough computational resources locally, by moulding geographically distributed resources into a single computing environment. However, these distributed resources are typically heterogeneous, non-dedicated, and are provided without any performance or availability guarantees. All of this makes the development of efficient Grid applications, capable of harnessing the benefits of such an ambient, a difficult and complex job. In this work, we propose a new approach that reduces the difficulty of Grid-enabling existing applications and which also provides a means for their efficient execution.

Nowadays, most developers of Grid middleware adopt a model based around a Resource Management System (RMS) whose objective is to maximize the utilization of the available Grid resources. One of the principal functions of the RMS is to provide services like scheduling and fault tolerance. Unfortunately, the decisions taken by the RMS are based solely on the state of the system. In the methodology proposed, each user application is fitted with an Application Management System (AMS) that provides management services specifically tuned to the needs of that individual application. The EasyGrid AMS is embedded into MPI applications at compile time, resulting in *system-aware* applications (Smart G-Apps). Results show that the new *system-aware* MPI applications are indeed faster than their conventional (cluster-based) counterparts. Also, the EasyGrid AMS has been developed in accordance with a standard implementation of MPI and, since it the AMS does not rely other system installed middleware, thus affords Smart G-Apps a greater degree of portability. Furthermore, no modifications to the user's source code are required.

Palavras-chave

1. Computação em Grid
2. Aplicações MPI
3. Sistema de Gerenciamento de Aplicações
4. Criação Dinâmica de Processos
5. Monitoramento de Aplicações
6. Escalonamento Dinâmico
7. Aplicações Adaptáveis

Abreviações

BoT	:	*Bag of Tasks*
GG	:	Gerenciador Global;
GM	:	Gerenciador Local da Máquina;
GS	:	Gerenciador do Site;
MPI	:	Message Passing Interface;
SGA	:	Sistema de Gerenciamento de Aplicações;
SGR	:	Sistema de Gerenciamento de Recursos;
Smart G-App	:	Smart Grid Application;

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A disponibilidade de sistemas computacionais cada vez mais velozes e robustos proporciona o crescimento do conhecimento em diversas áreas. Esse crescimento vem acompanhado por uma necessidade cada vez maior de poder computacional, capacidade de armazenamento de dados e rapidez no tempo de resposta para determinadas aplicações.

Como uma solução para atender à crescente demanda por poder computacional, surgiram os supercomputadores. Tais sistemas computacionais de alto desempenho são altamente especializados e possuem um custo muito elevado, podendo ultrapassar a quantia de milhões de dólares, o que torna o nível da computação de alto desempenho uma necessidade inalcançável para a maioria dos cientistas e companhias. Embora capazes de fornecer grande poder de processamento a um alto custo, os supercomputadores apresentam dificuldades de utilização ainda maiores que as máquinas seqüenciais tradicionais, exigindo dos desenvolvedores e cientistas um conhecimento profundo da arquitetura do sistema paralelo onde a aplicação será executada, além do desenvolvimento de novas metodologias de programação e alterações nas aplicações já existentes.

Motivados pela necessidade de uma base de dados confiável que permita a identificação de tendências em computação de alto desempenho, JD. Hans, W. Meuer e Erick Strohmaier criaram em junho de 1993 uma lista com os 500 sistemas de computadores mais rápidos do mundo. A lista, denominada *Top 500 Supercomputers* [11], é publicada duas vezes por ano com a ajuda de especialistas em computadores de alto desempenho, cientistas da computação, fabricantes e a comunidade da Internet. Os computadores do Top500 são classificados pelo seu melhor desempenho sobre o *benchmark* Linpack. O BlueGene/L, que foi construído pela IBM para atender a aplicações de Bioinformática, ocupa a primeira posição na última lista Top500 divulgada em junho de 2005, atingindo o desempenho de 136.800 GFlops e um custo de mais de 100 milhões de dólares. O sistema

que ocupou a primeira posição na primeira lista do Top 500 divulgada em junho de 1993, teve o desempenho máximo calculado em torno de 59,7 GFlops, aproximadamente 2.291,5 vezes menor que o BlueGene/L.

Apesar dos avanços tecnológicos e da oferta destes ambientes computacionais de alto desempenho, ainda existem problemas de difícil solução em diversas áreas, tais como ciência (bioinformática, biodiversidade, pesquisa climática, modelagem molecular, astronomia), engenharia e economia (modelagem da economia mundial), que necessitam de um poder computacional ainda maior do que aquele oferecido pelos supercomputadores atuais.

Com as melhorias apresentadas pelas redes locais (LANs - *Local Area Networks*) e a produção em massa de computadores pessoais e servidores básicos, novas alternativas para a construção de sistemas computacionais de alto desempenho têm surgido. Entre elas, a computação em *cluster* tem se desenvolvido como uma solução barata e de fácil acesso para muitos pesquisadores de diversas áreas, sendo esses sistemas construídos através da interligação de processadores (PC's ou estações) autônomos. Apesar do baixo custo, justificado pela utilização de componentes pré-existentes (*"off-the-shelf"*), esses sistemas são geralmente ineficientes tanto em desempenho quanto em consumo de energia e espaço físico. Esse conjunto de computadores comuns normalmente é dedicado a um número restrito de usuários, o que pode acarretar ociosidade durante grande parte do tempo. Devido à crescente demanda por poder computacional, os sistemas *cluster*s instalados atualmente já não são capazes de atender às necessidades apresentadas por diversas áreas do conhecimento.

Hoje, na maioria dessas áreas existem diversos grupos de pesquisas espalhados pelo mundo voltados para solução de um mesmo problema ou de problemas similares. A colaboração entre os pesquisadores envolvidos em tais projetos é fundamental para acelerar o processo de descoberta do conhecimento. Essa colaboração está sendo facilitada pelo desenvolvimento da Internet, por exemplo através da *World Wide Web*, que tornou possível o compartilhamento de dados e a troca de informações a nível global. Mais recentemente, junto com os avanços tecnológicos na área de redes WANs (*Wide Area Networks*), como por exemplo o desenvolvimento e implementação de redes de comunicação de alta velocidade baseadas em fibra ótica, surgiu a possibilidade de solucionar as necessidades computacionais apresentadas por pesquisadores utilizando-se ambientes de computação geograficamente distribuídos. Além de fornecer uma grande quantidade de poder computacional, tais ambientes possuem um custo muito inferior ao apresentado pelos supercomputadores atuais. Essa nova forma de colaboração é conhecida por diversos

nomes, sendo o mais recente Grids Computacionais.

Os Grids Computacionais surgem com o objetivo de fazer com que a computação de alto desempenho também seja acessível a usuários que não necessariamente possuem recursos disponíveis localmente, suficientes para resolver devidamente suas necessidades. Mais do que compartilhamento de dados e informações, como é possível através da *web*, com o surgimento dos Grids Computacionais é possível o compartilhamento de recursos, computação, instrumentos científicos, banco de dados e também conhecimento. Em um Grid Computacional, ao contrário dos supercomputadores, o custo é dividido entre os colaboradores, o que torna este tipo de computação acessível à maioria dos cientistas e companhias.

A existência de redes de longa distância de alta velocidade e baixo custo está sendo uma alavanca para o desenvolvimento de aplicações que tiram vantagem de recursos possivelmente distribuídos por uma grande área geográfica. Isso representa uma abertura para novos ramos de pesquisa que previamente encontravam-se limitados e sem exploração, por razões econômicas e práticas. Ainda assim, existe um número relativamente baixo de aplicações que exploram tal poder computacional de forma realmente vantajosa. Até hoje, a maioria das aplicações em Grid são escritas por especialistas da área e não por cientistas ou engenheiros. Isso na verdade mostra o grau de dificuldade encontrada por leigos em computação em propor versões de aplicações científicas e/ou comerciais, considerando os aspectos relevantes em Grids Computacionais. Como conseqüência, o desenvolvimento de aplicações eficientes para esse tipo de ambiente ainda é uma tarefa difícil, principalmente devido à sua natureza dinâmica, compartilhada e instável, ao contrário dos *cluster*s de computadores, que são na sua maioria estáveis e dedicados.

Então, é fundamental que os desenvolvedores de infra-estrutura Grid se preocupem em oferecer uma camada de *software*, chamada *middleware*, que tenta efetivamente esconder a complexidade desses sistemas, com o objetivo de simplificar a implementação de aplicações para Grids Computacionais. Em particular, nas camadas da arquitetura Grid identificadas em [44], o termo *middleware* indica a camada situada entre a aplicação e os recursos que fornece serviços tais como autenticação, autorização, descoberta e acesso a recursos, segurança, execução de tarefas remotas e movimentação de dados, conforme os oferecidos pelo Globus Toolkit [41].

Muito se tem pesquisado no que se refere ao *middleware* do Grid. Além do Globus Toolkit, que é uma ferramenta de gerenciamento de recursos do Grid, ambientes tais como os sistemas de gerenciamento de recursos Condor-G [46], Legion [49] [32], Nimrod/G [29] e

Unicore [38], e ferramentas de monitoramento do sistema como o NWS [68], encontram-se disponíveis já há alguns anos. No entanto, relativamente poucos usuários fora do círculo de projetistas desses ambientes vêm abraçando a nova tecnologia, devido às dificuldades de programação. Atualmente, o que se encontra disponível para o desenvolvimento de aplicações em Grid são *frameworks* que tentam esconder dificuldades de programação e oferecer abstrações quanto ao ambiente utilizado. Cactus [13], GrADS Project [52], AppLeS [23] (parte do projeto GrADS), GridLab [14], MyGrid [33] e Integrade [48] são exemplos de *frameworks* desenvolvidos pela comunidade científica que facilitam o uso e o controle dos recursos disponibilizados em ambientes Grids.

Existe uma grande expectativa na popularização de modelos de economia computacional voltados para computação em Grid. Desta forma, donos de recursos computacionais poderão alugá-los para diferentes usuários de forma que estes tenham acesso aos recursos mais adequados às suas necessidades a um baixo custo. A idéia é que o usuário do Grid pague pelos recursos da mesma forma que a população paga pelo uso da energia elétrica.

Com o esperado aumento na exploração dos Grids, a utilização eficiente dos recursos surge como uma necessidade para o sucesso dessa tecnologia. Para que uma aplicação em ambiente distribuído alcance um bom desempenho, ela deve ser escalonada de modo que os recursos disponíveis no sistema possam ser utilizados da forma mais eficiente possível. Na computação em Grid, o escalonamento é ainda mais complexo [67] [37], pois tipicamente envolve recursos com diferentes configurações, de diferentes domínios administrativos, geograficamente distribuídos e sujeitos à variação temporal de carga e disponibilidade. Além de obter um melhor aproveitamento dos recursos disponíveis, o escalonamento precisa também considerar as regras de utilização definidas para estes recursos pelos administradores dos diversos ambientes compartilhados.

Tipicamente, o gerenciamento dos recursos disponíveis em um Grid Computacional fica a cargo de um Sistema Gerenciador de Recursos (SGR), componente fundamental nesse tipo de ambiente. Alguns dos fatores mais relevantes que devem ser considerados pelo SGR é a autonomia dos *sites* que constituem o Grid Computacional, a heterogeneidade desses sistemas, as diferentes políticas de escalonamento adotadas e a existência de diversos mecanismos de segurança. Como conseqüência, o gerenciamento nesse ambiente torna-se ainda mais complexo do que em ambientes distribuídos dedicados, como os *clusters*.

Com o objetivo de utilizar de forma eficiente os recursos disponíveis, os SGRs tipicamente monitoram e analisam informações específicas do sistema, para que possam ser

selecionados os recursos mais adequados à execução de cada aplicação em cada instante. Entretanto, para aplicações paralelas, a escolha de recursos adequados à sua execução com base apenas em informações sobre o sistema pode não ser suficiente. Tarefas não podem simplesmente ser atribuídas aos recursos disponíveis sem se levar em consideração a disponibilidade dos dados em cada recurso, a dependência entre os processos e a necessidade de comunicação entre eles. É necessário considerar também características específicas de cada aplicação, para que se possa obter um bom desempenho na execução da mesma sobre os recursos selecionados. Devido à sua complexidade, SGR's são geralmente desenvolvidos para classes específicas de aplicações distribuídas, sendo as mais comuns aplicações do tipo *bag-of-tasks* (incluindo *parameter sweep* e mestre-escravo) ou aplicações *workflow*. Existem poucos SGR's para aplicações paralelas em geral [53].

Quando consideradas as características apresentadas não só por aplicações paralelas, mas também o dinamismo e heterogeneidade apresentada pelos ambientes Grids, fica clara a necessidade de um sistema capaz de adaptar a execução de uma aplicação às condições oferecidas pelo ambiente. Tal sistema precisa ser capaz de fornecer mecanismos de tolerância a falhas, escalonamento dinâmico de tarefas e de balanceamento de carga dinâmico apropriados para cada tipo de aplicação a ser executada em um Grid Computacional.

1.1 Contribuição

O projeto EasyGrid [26] questiona o uso de SGR's e adota a metodologia de um Sistema Gerenciador de Aplicações (SGA) para a execução eficiente e robusta de programas MPI em ambientes Grids. O *framework* EasyGrid permite que o programador se concentre em como explorar o paralelismo para resolver seu problema, ficando a cargo do *framework* gerar uma versão *system-aware* da aplicação capaz de utilizar os recursos disponíveis no Grid Computacional de uma maneira mais eficiente. Na aplicação *system-aware* (ou *Smart G-App - Smart Grid Application*) gerada pelo *framework* é embutido um *middleware* (SGA) ajustado especificamente às características da aplicação. Por fazer parte da aplicação *system-aware*, o SGA garente a portabilidade da mesma, eliminando as dependências em relação a determinados *middlewares* de serviços.

A escolha por aplicações paralelas que utilizam a biblioteca MPI para troca de mensagens foi motivada por sua larga utilização em programação paralela. Existe um consenso na comunidade científica de que aplicações paralelas não são consideradas adequadas para ambientes Grids, tipicamente por razões como o alto custo de comunicação nesses ambi-

entes, quando comparados com *cluster* de computadores. Grids são ambientes totalmente dinâmicos, instáveis e compartilhados, enquanto que a biblioteca MPI foi desenvolvida para uso em ambientes estáticos, seguros e dedicados.

Por ser o modelo de troca de mensagens MPI considerado de fácil utilização e por ele geralmente apresentar bom desempenho, acredita-se na existência de um grande número de cientistas de diversas áreas que possuem aplicações paralelas MPI projetadas para utilizar *cluster* de processadores. A dificuldade de reescrever tais aplicações para que estas se tornem adequadas à execução em ambientes Grids transformou-se numa grande motivação para o desenvolvimento de um sistema capaz de executar aplicações paralelas MPI em tais ambientes, sem que isso exija do desenvolvedor um esforço extra na reescrita da aplicação.

Este trabalho propõe uma implementação do *framework* EasyGrid através da utilização de uma hierarquia de processos gerenciadores capazes de administrar de forma eficiente e robusta a execução de aplicações MPI no ambiente descrito. Todas as funcionalidades oferecidas pelo sistema proposto, tais como monitoramento da aplicação, balanceamento de carga dinâmico e identificação de falhas são acrescentadas à aplicação do usuário, de maneira transparente, sem que para isso seja necessário qualquer tipo de alteração no código original da aplicação. Ou seja, aplicações paralelas são transformadas automaticamente em versões *system-aware* através da incorporação de um *middleware* de serviço específico à aplicação na forma de um SGA. Serviços básicos como informações sobre os recursos e segurança ficam a cargo de *middlewares* a nível do sistema, como o Globus Toolkit.

1.2 Apresentação da Dissertação

Esta dissertação está organizada da seguinte forma:

- Capítulo 1 - Introdução.
- Capítulo 2 - Computação em Grid: Neste capítulo, é apresentada uma visão geral sobre a computação em Grid dando-se destaque a diversos *middlewares* básicos e de serviços existentes atualmente. Aspectos de monitoramento, escalonamento dinâmico de tarefas e tolerância a falhas são rapidamente abordados.
- Capítulo 3 - Sistema de Gerenciamento da Aplicação: Propõe-se um sistema de gerenciamento distribuído e hierárquico para execução de aplicações MPI em ambientes Grid, constituindo a contribuição principal deste trabalho.

- Capítulo 4 - Experimentos Computacionais: Este capítulo é destinado à avaliação dos resultados obtidos com a realização de experimentos computacionais envolvendo diferentes cenários de execuções de aplicações paralelas MPI.

- Capítulo 5 - Conclusões e Trabalhos Futuros: Neste capítulo, são apresentadas as principais conclusões obtidas neste trabalho. Também são destacadas direções para possíveis trabalhos futuros.

Capítulo 2

Computação em Grid

Grids Computacionais têm o potencial de se tornarem plataformas poderosas a serem utilizadas pela comunidade de computação distribuída para executar aplicações de grande importância e alto teor computacional. No entanto, para se obter alto desempenho, é necessário especificar modelos de computação e de interface que capturem a natureza heterogênea dos recursos dos Grids, já que estes estão conectados por uma rede geralmente não muito segura e de forma distribuída. Também, como em qualquer tipo de sistema paralelo, executar aplicações em Grids requer escalonamento e monitoramento bastante cuidadosos, tanto da computação quanto da comunicação. Todavia, o dinamismo e heterogeneidade dos Grids limitam a aplicabilidade de ferramentas já disponíveis para sistemas paralelos. Tais considerações sugerem que, enquanto Grids podem ser construídos a partir da tecnologia de *softwares* distribuídos e paralelos (*middleware*), avanços significativos em mecanismos, técnicas e ferramentas são necessários.

Neste capítulo serão apresentados alguns *middlewares* básicos e de serviços existentes atualmente e questões relaciondas à necessidade de monitoramento, escalonamento de tarefas e tolerância a falhas para a execução eficiente de aplicações em Grids Computacionais.

2.1 Introdução

O nome *Computational Grid* (Grid Computacional) foi idealizado com base nas redes de interligação dos sistemas de energia elétrica (*power-grids*), em que um usuário utiliza a eletricidade sem ao menos saber em que local ela foi gerada, sendo a sua utilização totalmente transparente aos seus usuários [42]. Essa analogia retrata o objetivo de tornar o uso de recursos computacionais distribuídos tão simples quanto ligar um aparelho na

rede elétrica.

Os Grids Computacionais têm crescido com a necessidade da comunidade tanto científica como comercial, de se ter um sistema capaz de fornecer alto poder computacional a um custo razoável. O objetivo dos Grids Computacionais é o de agregar uma coleção de recursos distribuídos, heterogêneos e compartilhados, conectados por uma rede veloz, capaz de oferecer poder computacional para execução de aplicações já existentes e também de novas aplicações que possam utilizar este "novo" ambiente colaborativo. Entretanto, a exploração eficiente desse tipo de ambiente ainda é um desafio, principalmente devido ao seu comportamento dinâmico e instável. É necessário que a computação em Grid ofereça meios para que o desenvolvedor da aplicação possa escrever programas em linguagens de alto nível, sem que para isso maiores esforços sejam aplicados pelo próprio programador, a fim de que a aplicação possa acessar o Grid e utilizar seus recursos de forma eficiente.

Ao contrário dos sistemas paralelos e distribuídos tradicionais, Grids Computacionais precisam considerar questões como segurança, acesso uniforme aos recursos independentemente da localização dos mesmos, descoberta e agregação dinâmica de recursos e qualidade de serviço. Entre os grandes desafios [17] apresentados por esse novo ambiente estão:

- a heterogeneidade do ambiente Grid, resultado da quantidade de recursos diferentes que podem ser agregados e do grande número de tecnologias empregadas por tais recursos;
- a multiplicidade de domínios administrativos autônomos, cada qual com regras distintas para a utilização dos recursos;
- a escalabilidade, que pode acarretar em perda de desempenho com o aumento do número de recursos no Grid;
- a natureza dinâmica desses tipos de ambientes, onde a taxa de ocorrência de falhas na rede de interconexão e nas máquinas agregadas é alta;
- compartilhamento de recursos, o que leva a uma variação na quantidade de poder computacional disponível.

2.2 Arquitetura Grid

Como apresentado em [18], do ponto de vista do usuário final, Grids podem ser usados para fornecer os seguintes tipos de serviços: computacionais, de dados, da aplicação, de

informação e de conhecimento. Os **serviços computacionais** estão relacionados com o fornecimento de serviços seguros para execução de tarefas da aplicação sobre recursos computacionais distribuídos, de forma individual ou coletiva. Grids que oferecem tais tipos de serviços são chamados de **Grids Computacionais**. Os serviços capazes de prover acesso seguro a banco de dados distribuídos e gerenciamento dos mesmos são caracterizados como **serviços de dados**. A combinação de serviços computacionais e serviços de dados resultou nos chamados **Grids de Dados**. Os **serviços da aplicação** estão relacionados com o gerenciamento da aplicação e o fornecimento de acesso a *softwares* e bibliotecas remotas. A extração e apresentação de dados obtidos através do uso de serviços computacionais, de dados e da aplicação são chamados de **serviços de informação**, que associados à solução de problemas e à tomada de decisões são classificados como **serviços de conhecimento**. Este trabalho tem como foco os Grids Computacionais para execução de aplicações que apresentam grande quantidade de computação paralela e distribuída.

Com relação às características de *hardware* e *software*, os Grids Computacionais podem ser divididos basicamente em três níveis principais, apresentados na Figura 2.1. No primeiro nível é possível destacar a infra-estrutura Grid, formada por componentes de *hardware* e *software* integrados em uma única rede de recursos. Essa camada é representada por recursos como computadores, dispositivos de armazenamento e instrumentos científicos. No nível seguinte encontra-se o *middleware* Grid, composto de ferramentas e serviços responsáveis por disponibilizar os recursos à aplicação. O nível de *middleware* pode ser subdividido em mais dois outros níveis: o *middleware* básico e o *middleware* de serviços. O *middleware* básico é capaz de oferecer os serviços fundamentais para a operação de um Grid, como o gerenciamento remoto de processos, a co-alocação de recursos, o registro e a coleta de informações, segurança e aspectos de qualidade de serviço. O *middleware* de serviços provê um alto nível de abstração, incluindo ambientes de desenvolvimento de aplicações, ferramentas de programação, escalonamento de *jobs* (aplicação), tolerância a falhas, entre outros. O último nível refere-se a aplicações capazes de explorar os recursos disponíveis neste tipo de ambiente.

Na arquitetura Grid, é necessário garantir que as relações de compartilhamento possam ser iniciadas entre partes arbitrárias, acomodando novos participantes dinamicamente, através de diferentes plataformas, linguagens e ambientes de programação. A interoperabilidade é, então, fundamental na identificação dos componentes do sistema (usuários e recursos), na especificação do propósito e funções desses componentes e na indicação de como esses componentes devem interagir uns com os outros. Em ambientes de rede, interoperabilidade significa a existência de protocolos comuns entre tais compo-

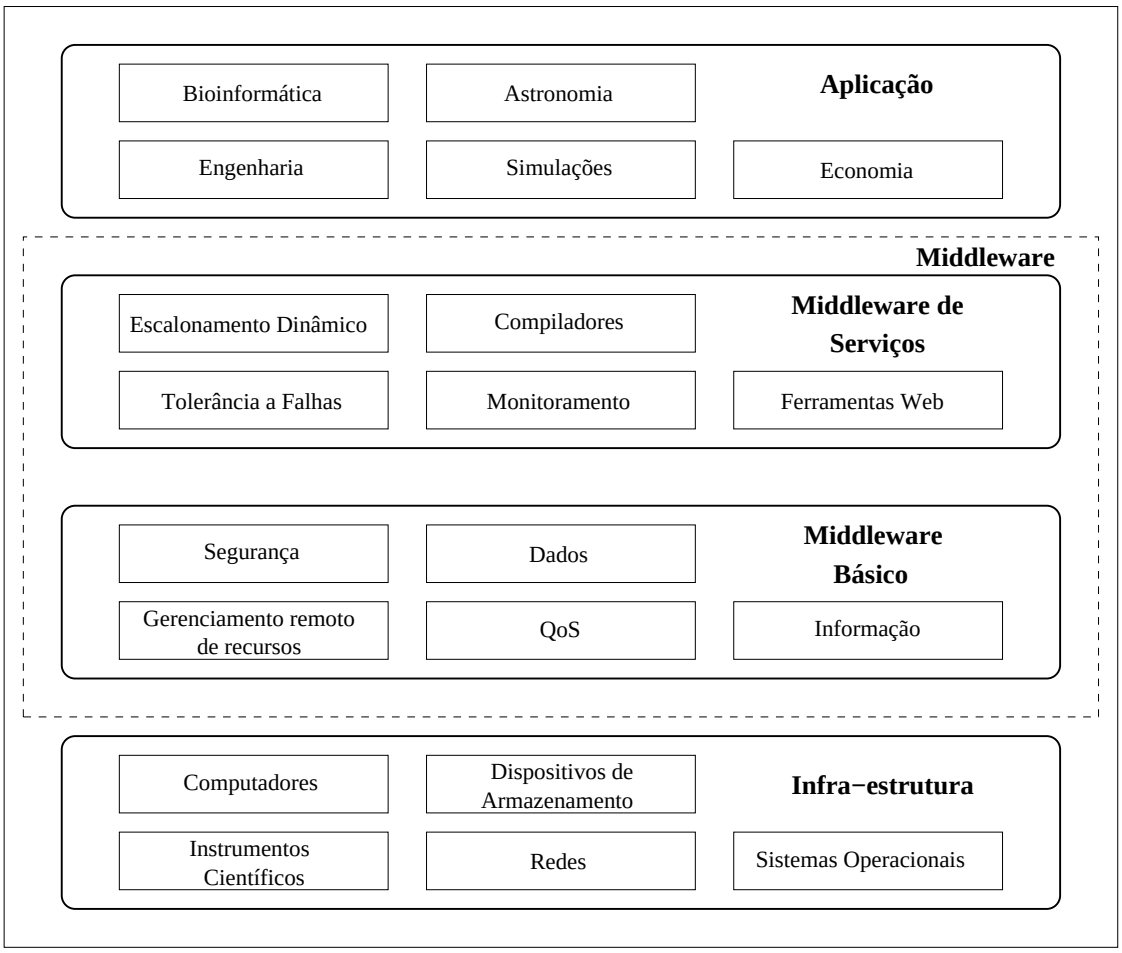


Figura 2.1: Níveis da arquitetura Grid.

nentes. A padronização de protocolos facilita a definição de serviços padrões capazes de aprimorar suas funcionalidades.

Foster, Kesselman e Tuecke apresentam em [44] uma descrição mais detalhada da arquitetura em termos de protocolos e serviços Grid através da organização de componentes em camadas. Componentes de uma mesma camada compartilham funcionalidades comuns à camada, mas podem ser construídos com base nas capacidades e comportamentos das camadas inferiores. A disposição das camadas na arquitetura Grid pode ser observada na Figura 2.2. A camada *Infra-estrutura* é responsável por fornecer os recursos (recursos computacionais, de armazenamento, de rede entre outros), para os quais o acesso compartilhado pode ser mediado pelos protocolos do Grid. No topo da hierarquia pode ser identificada a camada da *Aplicação*, que engloba a aplicação do usuário que será executada no ambiente Grid.

A camada acima da *Infra-estrutura* é a camada *Conectividade*, que define o núcleo de comunicação e protocolos de autenticação para transações em redes específicas. Os

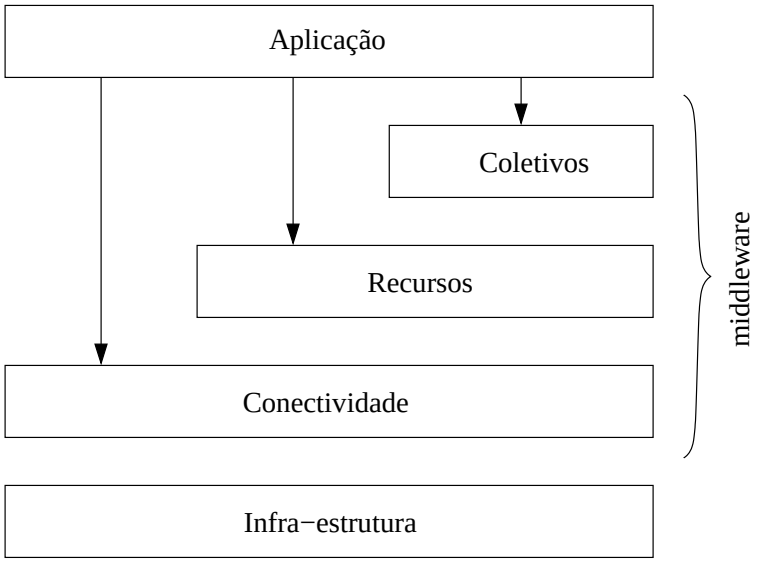


Figura 2.2: Camadas da arquitetura de protocolos e serviços Grid.

protocolos de comunicação do Grid Computacional tornam possível a troca de dados entre os recursos heterogêneos disponíveis na camada *Infra-estrutura*. Já os protocolos de autenticação, desenvolvidos sobre serviços de comunicação, fornecem mecanismos seguros para a verificação da identidade dos recursos e dos usuários.

Acima da camada de *Conectividade*, a camada de *Recursos* é responsável pela elaboração de protocolos capazes de inicializar e controlar o compartilhamento de recursos individuais. Os protocolos de informação pertencentes a essa camada são usados para obter informações sobre a estrutura e o estado dos recursos. Já os protocolos de gerenciamento são usados para negociar o acesso a recursos compartilhados, especificando, por exemplo, os recursos requeridos e as operações a serem desempenhadas.

A camada *Coletivos* contém protocolos e serviços que não são associados a nenhum recurso específico, possuindo uma natureza global capaz de capturar interações através de coleções de recursos. Como funções coletivas podem ser considerados os serviços de diretório para descoberta de recursos, os serviços de co-alocação e escalonamento, o monitoramento, a replicação de dados, os ambientes para solução de problemas (PSE - *Problem Solving Environments*), serviços de colaboração e diversas outras funções implementadas como serviços persistentes, com protocolos associados e bibliotecas projetadas para serem ligadas a aplicações.

Um grande esforço está sendo gasto em projetos e implementações de *middleware* para possibilitar o uso de Grids Computacionais. Tais sistemas *middleware* têm como objetivo facilitar a utilização eficiente e segura da infra-estrutura disponível em um Grid

Computacional, permitindo que aplicações aproveitem os benefícios deste novo ambiente sem o conhecimento das características específicas dos recursos distribuídos.

2.3 Middleware Básico

O objetivo de um *middleware* é aliviar a carga do programador na hora de projetar, programar e gerenciar aplicações distribuídas, fornecendo aos desenvolvedores um único ambiente de programação distribuída integrado e consistente [54]. Essencialmente, *middleware* é uma camada de *software* distribuída que fornece uma abstração para a complexidade e heterogeneidade dos ambientes distribuídos, com suas variedades de tecnologias de rede, arquitetura das máquinas, sistemas operacionais e linguagens de programação. Em Grids, o *middleware* é definido como uma camada de *software* que permite que usuários compartilhem aplicações, instrumentos científicos e dados.

Nas subseções seguintes serão apresentados alguns *middlewares* básicos, entre eles o Globus Toolkit, que já é considerado pela comunidade científica como um *middleware* básico padrão para ambientes Grids.

2.3.1 Globus Toolkit

No final de 1994, Rick Stevens (Laboratório Nacional de Argonne) e Tom DeFanti (Universidade de Illinois) propuseram estabelecer temporariamente ligações entre 11 redes de alta velocidade para criar um Grid nacional, denominado I-WAY (*Information Wide-Area Year*) [40], por duas semanas, antes e durante o congresso Supercomputing'95. Um pequeno grupo liderado por Ian Foster (Laboratório Nacional de Argonne) criou novos protocolos que permitiram aos usuários do I-WAY executarem um conjunto de mais de 60 aplicações sobre computadores de 17 centros de pesquisa, durante o evento. Este sucesso experimental motivou o desenvolvimento de projetos já existentes e a criação de novos projetos, como o Globus Toolkit.

O Globus Toolkit é um conjunto de serviços que facilita a computação em Grid. Esses serviços permitem a submissão e controle de aplicações, descoberta de recursos, movimentação de dados e segurança no ambiente do Grid. O Globus e os protocolos definidos em sua arquitetura são considerados um padrão como infra-estrutura para computação em Grid.

O Projeto Globus, que inicialmente foi desenvolvido por uma parceria entre o Lab-

oratório Nacional de Argonne, a Universidade de Chicago e a Universidade do Sul da Califórnia, conta atualmente com diversas instituições de pesquisa, agências federais e grandes empresas, como a IBM e a Microsoft, formando o chamado Globus Alliance. Entre as principais instituições de pesquisa que compõem o Globus Alliance está a Universidade de Edinburgh, o Royal Instituto de Tecnologia na Suécia e o NCSA (National Center for Supercomputing Applications). Globus é construído como uma camada da arquitetura na qual serviços de alto nível podem ser desenvolvidos usando serviços de mais baixo nível. O Globus Toolkit é modular e uma aplicação pode explorar características do Globus, tais como gerenciamento de recursos ou infra-estrutura de informação, sem usar as suas bibliotecas de comunicação.

Atualmente, existem três versões do *toolkit* Globus (versão 2, 3 e 4). Na versão 2, o Globus Toolkit fornece três elementos necessários para computação em um ambiente Grid. O primeiro é o *Resource Management* que está envolvido na alocação de recursos fornecidos pelo Grid. O segundo, o *Information Service*, fornece informações sobre os recursos Grid. Finalmente, o *Data Management* está envolvido no acesso e gerenciamento de dados no ambiente Grid.

A maior dificuldade identificada na construção de uma infra-estrutura como os Grids é a integração de recursos e serviços distribuídos de forma transparente e eficiente. Como conseqüência, os Grids Computacionais passaram a adotar uma abordagem orientada a serviços, resultado da integração das tecnologias e conceitos associados aos Grids com as tecnologias de *web services*. A partir daí, foi definida uma arquitetura de serviços básicos (*grid service*) denominada *Open Grid Service Architecture* (OGSA) [43], que engloba a idéia da interconexão de sistemas e a criação de ambientes virtuais multi-institucionais. Após definida a OGSA, foi especificada uma infra-estrutura básica de serviços denominada *Open Grid Services Infrastructure* (OGSI), com o objetivo de definir os comportamentos de um serviço Grid (*grid service*). A primeira implementação de um OGSI foi apresentada na versão 3 do Globus Toolkit e aprimorada na versão 4 [10]. O restante desta subseção está focada apenas nas funcionalidades da versão 2 (ainda disponíveis nas versões 3 e 4), por ser esta a mais utilizada pelos Grids em produção.

2.3.1.1 Segurança

Segurança é a base de qualquer ambiente Grid. Cada *site* ou provedor de recursos (*resource provider*) pode empregar qualquer uma das várias soluções de segurança local. O Globus Toolkit utiliza os protocolos GSI (*Grid Security Infrastructure*), baseados em uma chave

pública (*Public Key Infrastructure*), para fornecer autenticação, comunicação segura e autorização.

Provedores de recursos precisam definir que recursos serão compartilhados, quais usuários poderão compartilhar recursos e sobre que condições esse compartilhamento ocorrerá. O compartilhamento seguro dos recursos é garantido através da autenticação e autorização dos usuários. É através da autenticação que se estabelece a identificação de um usuário ou recurso. A autorização é o passo seguinte à autenticação. Nesse momento, são definidos os direitos do usuário sobre os recursos do Grid, de acordo com a relação de compartilhamento definida.

Os protocolos de autenticação fornecem mecanismos seguros de criptografia para verificação da identidade de usuários e recursos. Um *proxy* temporário é criado com base na chave privada do usuário, permitindo que este faça solicitações remotas. Usando esse *proxy*, não é necessária a intervenção do usuário para autenticação em cada acesso a um novo recurso do *Grid*.

Para permitir segurança na comunicação, o pacote OpenSSL é instalado como parte do Globus Toolkit. Ele é usado para criar um canal criptografado usando os protocolos de comunicação SSL/TLS (*Secure Socket Layer/Transport Layer Security*) entre clientes do Grid e servidores.

2.3.1.2 Gerenciamento de Recursos

O gerenciamento de um recurso local no Globus é realizado pelo GRAM (*Globus Resource Allocation Manager*), que atua como uma interface abstrata para recursos heterogêneos do Grid. Cada recurso é gerenciado por uma instância do GRAM, sendo esta responsável por instanciar, monitorar e reportar o estado das tarefas alocadas a tal recurso.

As requisições do cliente são recebidas pelo *Gatekeeper* que consulta o GSI (*Grid Security Infrastructure*). Este serviço permite uma autenticação única do usuário do Grid. A partir desta autenticação, o *Gatekeeper* verifica se o usuário é autorizado a executar sua tarefa no recurso em questão. Caso o usuário tenha o acesso permitido, é criado um *JobManager*, que é responsável por iniciar e monitorar a tarefa submetida. As informações sobre o estado do recurso são constantemente reportadas ao serviço de informação e diretório do Globus, o MDS (*Monitoring and Discovery System*).

Um processo pode ser submetido aos recursos do Grid através do comando *globusrun*. Esse comando é tipicamente executado tendo como parâmetro um *script* RSL (*Resource*

Specification Language). *Scripts* RSL identificam recursos (computadores), especificam requerimentos (número de CPUs, memória, tempo de execução, etc.) e parâmetros (localização dos executáveis, variáveis de ambiente entre outros).

Através do DUROC (*Dynamically-Updated Request Online Coallocator*) é possível que usuários submetam múltiplos processos para múltiplos GRAMs (alocação simultânea de um conjunto de recursos). DUROC usa um "*co-allocator*" para executar e gerenciar processos sobre os vários gerenciadores de recursos.

2.3.1.3 Gerenciamento de Dados

GridFTP fornece segurança e facilidade na transferência de dados entre *hosts* do Grid. O Globus Toolkit possui um módulo chamado GASS (*Global Access to Secondary Storage*) responsável por prover serviços de acesso a armazenamento secundário (arquivos em disco).

Juntamente com outros módulos do Globus Toolkit, é possível utilizar o GASS para manipulação de arquivos remotos, ou seja, disparar executáveis, redirecionar as saídas de um processo e ler e escrever em arquivos remotos utilizando-se as rotinas padrão de manipulação de arquivos. Para tornar o acesso aos dados mais rápido, é possível através do *Replica Management* a replicação de dados entre os *hosts* pertencentes ao Grid.

2.3.1.4 Serviços de Informação

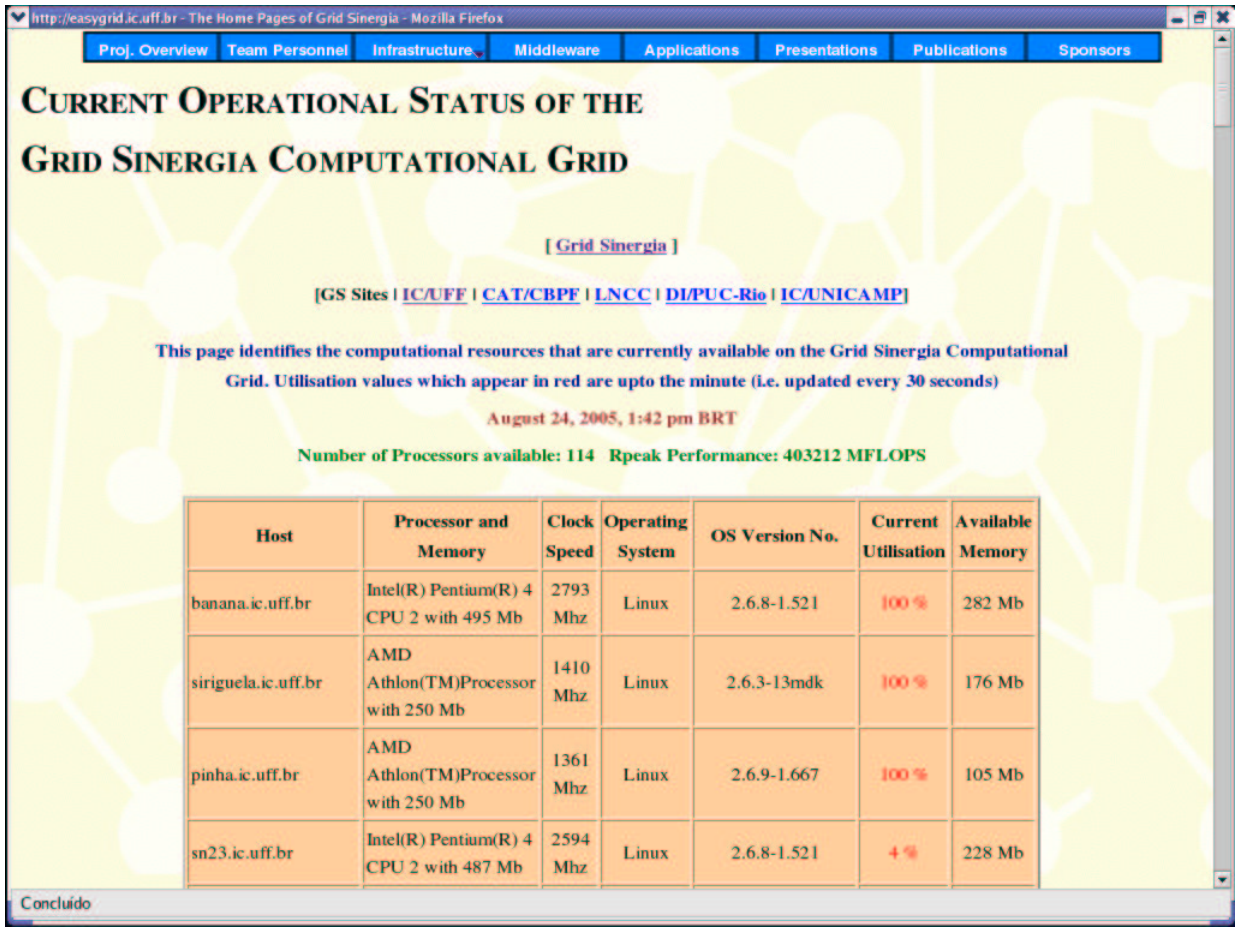
O MDS (*Monitoring and Discovery System*), também conhecido como GIS (*Grid Information Service*), fornece o serviço de informação em Globus (por exemplo, arquitetura de *hardware*: tipo de CPU, memória, sistema operacional, memória disponível, carga de CPU dos recursos).

O MDS usa o protocolo LDAP (*Lightweight Directory Access Protocol*) como uma interface para obter a informação sobre recursos. O modelo de serviços de diretórios LDAP é estruturado como um conjunto de entradas, onde cada entrada pode ter zero ou mais pares de atributos, e é referenciada através de um nome distinto (DN - Distinguished Name).

Os dois principais componentes do MDS são: *Grid Index Information Service* (GIIS) e *Grid Resource Information Service* (GRIS). GRIS é um repositório de informações de recursos locais derivados dos Provedores de Informação (*Information Providers*). GIIS é o repositório que contém índices de informações de recursos registrados pelos GRIS's

e outros GIIS's. Ele fornece uma visão global dos recursos do Grid. Os fornecedores de informações sobre recursos usam um protocolo *push* para alterar periodicamente o GIIS. Além do protocolo *push*, a existência de um protocolo *pull* permite que usuários obtenham informações do GIIS/GRIS. O Provedor de Informação traduz as propriedades dos recursos locais para o formato definido nos arquivos de configuração.

A Figura 2.3 apresenta uma ilustração com dados fornecidos pelo MDS para o Grid Computacional Grid Sinergia. Praticamente todas as informações disponibilizadas pelo MDS são estáticas, tais como quantidade total de memória, velocidade do processador, sistema operacional instalado, entre outras. Além de informações estáticas, algumas informações dinâmicas são também disponibilizadas, como quantidade de memória e a carga de CPU dos *hosts*. As informações dinâmicas fornecidas pelo MDS não retratam de maneira eficiente a dinamicidade dos ambientes Grids. O tempo de atualização dessas informações pode ser reduzido, entretanto a intrusão aumentará.



Host	Processor and Memory	Clock Speed	Operating System	OS Version No.	Current Utilisation	Available Memory
banana.ic.uff.br	Intel(R) Pentium(R) 4 CPU 2 with 495 Mb	2793 Mhz	Linux	2.6.8-1.521	100 %	282 Mb
siriguela.ic.uff.br	AMD Athlon(TM)Processor with 250 Mb	1410 Mhz	Linux	2.6.3-13mdk	100 %	176 Mb
pinha.ic.uff.br	AMD Athlon(TM)Processor with 250 Mb	1361 Mhz	Linux	2.6.9-1.667	100 %	105 Mb
sn23.ic.uff.br	Intel(R) Pentium(R) 4 CPU 2 with 487 Mb	2594 Mhz	Linux	2.6.8-1.521	4 %	228 Mb

Figura 2.3: Dados fornecidos pelo MDS para o Grid Computacional Grid Sinergia.

O Globus utiliza a biblioteca *Nexus* para comunicação. Essa biblioteca provê uma interface simplificada que opera sobre diversos protocolos de rede, como IP e ATM. Sobre

essa interface são implementados serviços de mais alto nível, como: MPI (*Message Passing Interface*), RPC (*Remote Procedure Call*) e entrada e saída remota.

2.3.2 Network Weather Service

O *Network Weather Service* (NWS) é um sistema distribuído que realiza monitoramento periódico e previsões dinâmicas sobre vários recursos computacionais e de rede, cujos dados podem ser capturados em um dado intervalo de tempo [68]. Tais previsões têm sido utilizadas com sucesso para implementar agentes de escalonamento dinâmico para aplicações de metacomputação. Seus desenvolvedores desejam estender a capacidade de monitoramento e previsão do sistema para satisfazer a necessidade de várias infra-estruturas de *software* distribuído, tais como Globus, Legion, Condor e Netsolve. A metodologia de escalonamento AppLeS tem feito uso intensivo das facilidades oferecidas pelo NWS [8].

O NWS apresenta sensores que monitoram as diferentes máquinas e pontos de uma rede, reportando as medidas obtidas para um servidor. Nesse servidor são processados e armazenados os dados provenientes de cada sensor. Os sensores podem atuar em diferentes campos, tais como medição de uso de CPU, uso de memória, etc. Além dos sensores que já acompanham o NWS, novos sensores podem ser programados para propriedades específicas de interesse pertinente à aplicação do usuário.

Entre os objetivos básicos do NWS estão: fornecer a previsão precisa dos dados, utilizar minimamente os recursos durante o processo de previsão (baixo grau de intrusão), estar sempre disponível para quando a aplicação do usuário solicitar dados de monitoramento e ser capaz de fornecer o máximo de informações possíveis sobre os recursos da rede onde o NWS está instalado. O NWS informa dados estatísticos de acordo com as variações que ocorrem em cada relatório de sensoriamento, o que permite a requisição de uma previsão estatística da possível tendência de uma determinada máquina, para uma característica específica em um dado intervalo de tempo.

O NWS atualmente apresenta três processos em sua execução:

- *Persistent State Process*, que grava e lê os dados do meio de armazenamento fixo (disco rígido);
- *Name Server Process*, que implementa os diretórios usados para criação de processos em recursos remotos e dados de baixo nível, tais como número de portas TCP/IP;
- *Sensor Process*, que é capaz de capturar o desempenho de um recurso específico.

Para se fazer previsões, é necessário que dados sejam coletados com uma freqüência apropriada. Para diminuir o tráfego com a informação de sensores, estes são organizados hierarquicamente em *cliques*. Membros de uma mesma *clique* podem se comunicar entre si, mas não podem se comunicar com membros de outras *cliques*. Sensores podem participar de mais de uma *clique*. Para ilustrar a utilização do NWS, no exemplo da Figura 2.4 é considerada a existência de 5 processos sensores executando sobre 5 máquinas do *site* UFF (clique UFF), 4 sensores rodando sobre 4 máquinas do *site* PUC (clique PUC) e 4 sensores rodando sobre 4 máquinas do *site* LNCC (clique LNCC). Para organizar o conjunto de sensores através de uma hierarquia de *cliques*, foram definidas duas *cliques* a mais, uma delas contendo 1 máquina do *site* UFF e 1 máquina do *site* PUC (*clique* UFF-PUC) e a outra contendo 1 máquina do *site* PUC e 1 máquina do *site* LNCC (*clique* PUC-LNCC). As cliques UFF-PUC e PUC-LNCC permitem a comunicação entre *sites*.

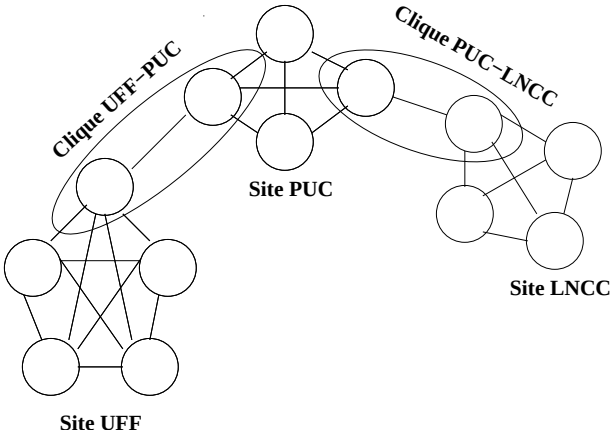


Figura 2.4: Exemplo de uma organização em *cliques*.

O NWS *Forecaster* trabalha apenas com pares de dados, representados como uma tupla (instante de tempo; valor da medida), e realiza uma previsão dinâmica desses dados. Ou seja, ele escolhe em tempo de execução um modelo matemático mais adequado dentre vários disponíveis para fazer previsões futuras sobre a utilização e desempenho dos recursos monitorados. No NWS, um conjunto de modelos de previsão é configurado em um sistema, cada um com sua própria parametrização. Quando um único *forecast* (previsão) é requerido, o sistema *forecast* NWS cria uma classificação entre os modelos de predição, de acordo com a exatidão dos resultados previamente obtidos por cada um deles. O modelo mais exato, no período de tempo anterior, é então escolhido para fazer a próxima previsão, caracterizando-se assim um método adaptativo.

O NWS apresenta uma interface de fácil utilização. Desta forma, qualquer módulo construído para sensoriamento de contexto pode facilmente utilizar o *Forecaster* para fazer

uma previsão de seus dados.

2.3.3 MPI - Message Passing Interface

O MPI e o PVM, os dois modelos de troca de mensagens mais empregados atualmente nas diversas áreas da computação, apresentam diferenças importantes nas suas formas de implementação. O PVM (*Parallel Virtual Machine*) é um pacote de *softwares* que permite que um grupo de computadores interconectados, possivelmente com diferentes arquiteturas, possa trabalhar cooperativamente formando uma máquina paralela virtual. PVM é de fácil utilização, mas não é tão poderoso quando comparado com o MPI.

O padrão MPI define uma biblioteca de funções que implementam o modelo de troca de mensagens padronizada e portável, e que foi desenvolvida por um fórum internacional [7] [39]. Essas rotinas incluem comunicação ponto-a-ponto, na qual uma operação de envio é usada para iniciar uma transferência de dados entre dois processos, e uma operação de recebimento correspondente é usada para extrair os dados da estrutura de dados no espaço de memória da aplicação, e operações coletivas tais como *broadcast* e reduções que implementam operações envolvendo múltiplos processos. Diversas outras funções tratam outros aspectos da troca de mensagens, incluindo no MPI-2 (uma extensão do MPI) a criação dinâmica de processos. O principal objetivo do MPI é otimizar a comunicação e aumentar o desempenho de aplicações paralelas ou distribuídas em máquinas paralelas, como supercomputadores ou *cluster* de processadores. A eficiência e generalidade do MPI é garantida através da disponibilização de um extenso padrão que oferece diversas implementações para uma mesma funcionalidade, como por exemplo existem diversas funções para o envio de mensagens. Outra característica dessa biblioteca, no que se refere à eficiência na comunicação, é a possibilidade de haver sobreposição de computação e comunicação.

A interface para troca de mensagens MPI tornou-se popular não apenas por ter sido desenvolvida para executar em uma variedade de sistemas, mas também por ser baseada na troca de mensagens, um dos paradigmas mais utilizados em programação paralela. Outras características têm contribuído para o MPI se tornar um modelo de programação. Entre elas estão o suporte ao gerenciamento de heterogeneidade (criação de novos tipos de dados), a construção de programas modulares (construção de comunicadores), gerenciamento da latência (operações assíncronas) e a representação de operações globais (operações coletivas).

Dois conceitos são fundamentais em MPI: grupos e comunicadores. Grupos em MPI

definem uma coleção ordenada de processos, cada um com um identificador (*rank*) distinto. Os grupos são responsáveis por definir os nomes de baixo nível para comunicação interprocesso (*ranks* são usados em operações de envio e recebimento, por exemplo). Eles podem ser manipulados separadamente dos comunicadores, mas somente comunicadores podem ser usados em operações de comunicação. Comunicadores são divididos em dois tipos: intracomunicadores e intercomunicadores. Intracomunicadores são usados para operações com um único grupo de processos, e intercomunicadores, para comunicação entre dois grupos de processos.

Após a inicialização de uma aplicação MPI, é criado automaticamente um comunicador denominado MPI_COMM_WORLD, contendo todos os processos pertencentes à aplicação. Os processos recebem dentro desse comunicador *ranks* com valores distintos, inteiros e crescentes a partir de zero. Através desse comunicador, os processos de uma aplicação podem se comunicar diretamente entre si.

Grupos em MPI são estáticos, ou seja, nenhum novo processo pode ser acrescentado a um grupo após a sua criação. Esse comportamento estático é justificado pelos desenvolvedores da biblioteca como uma conseqüência da necessidade de se aliar alto desempenho à corretude de operações coletivas. A biblioteca fornece operações capazes de unir e separar grupos, resultando sempre na criação de um novo grupo. Essa estabilidade é incompatível com a necessidade de gerenciamento dinâmico dos processos apresentadas pelos ambientes dinâmicos e heterogêneos como os Grids Computacionais, onde se faz necessária a utilização de mecanismos de tolerância a falhas e escalonamento dinâmico de processos de uma aplicação.

Usuários podem atingir um grau significativo de tolerância a falhas em suas aplicações [50]. O MPI padrão permite que *error handlers* (manipuladores de erro) sejam associados a comunicadores. Por *default*, é associado ao comunicador MPI_COMM_WORLD o *error handler* ERRORS_ARE_FATAL, que faz com que uma aplicação seja abortada em caso de falha de um dos processos. O desenvolvedor da aplicação poderia associar ao MPI_COMM_WORLD, o *error handler* MPI_ERRORS_RETURN, o qual especifica que funções MPI devem retornar um código de erro na ocorrência de falhas. *Error handlers* são associados a comunicadores. Um conjunto de erros pré-definidos são fornecidos pelo MPI e outros podem ser definidos pelo próprio desenvolvedor da aplicação. Um novo comunicador herda do comunicador pai, ou seja, comunicador a partir do qual foi criado, o seu *error handler*.

Uma outra opção apresentada pelo MPI para tentar fornecer tolerância a falhas está

relacionada ao uso de intercomunicadores. Programas que utilizam apenas o comunicador MPI_COMM_WORLD estão mais suscetíveis a falhas. Em casos de operações coletivas, uma falha em qualquer processo desse comunicador irá refletir nos demais processos desse mesmo comunicador. Um intercomunicador contém dois grupos de processos e toda a comunicação ocorre entre pares de processos de grupos distintos. Se cada processo pertencer a um grupo individual, nenhuma operação coletiva é necessária, e uma falha em um dos processos pode ser facilmente identificada pelo outro processo, que pode interromper a operação de comunicação. A única desvantagem dessa solução está no fato de que esse tipo de situação deve ser tratada pelo desenvolvedor da aplicação através da criação de intercomunicadores entre cada par de processos. Isso pode implicar em um esforço extra do programador para reescrever códigos já existentes.

O modelo de processos do padrão MPI-2 permite a criação e terminação cooperativa de processos após uma aplicação MPI ter iniciado. Ele fornece um mecanismo para estabelecer comunicação entre os novos processos criados e a aplicação MPI existente. A função MPI_Comm_spawn() cria dinamicamente processos MPI e estabelece comunicação com eles, retornando um intercomunicador contendo o processo pai no grupo local e o processo filho no grupo remoto. Esse intercomunicador pode ser usado para implementar tolerância a falhas, como citado anteriormente. Quando um dos processos disparados morre, os demais processos são capazes de continuar a execução. Além de facilitar a identificação e o tratamento de falhas, a função MPI_Comm_spawn() facilita a construção de mecanismos de balanceamento dinâmico de carga, permitindo a escolha do processador onde a tarefa deverá ser disparada.

O MPI padrão define quatro modos de comunicação:

- *Standard*: em operações de envio (*send*), dados locais devem ser enviados com sucesso ou copiados seguramente para espaço de *buffer* do sistema de maneira que o *buffer* da aplicação, que contém os dados, esteja disponível para reuso. Para operações de recebimento (*receive*), os dados devem ser seguramente armazenados em um *buffer* de recebimento para que esteja pronto para ser usado;

- *Buffered*: caso não tenha sido feita nenhuma chamada à função de recebimento correspondente, mensagens enviadas devem ser armazenadas em um *buffer*. Esse tipo de comunicação requer a alocação e o controle do usuário em relação ao espaço de *buffer* disponível;

- *Synchronous*: o processo que envia a mensagem não pode continuar sua execução

enquanto não houver um sinal de recebimento da mensagem pelo destinatário. Logo, uma operação de envio só será completa após o início da execução de uma operação de recebimento;

- *Ready*: o envio só pode começar após ter sido feita uma chamada à operação de recebimento. Se o recebimento ainda não teve a sua execução iniciada, a operação de envio retorna um erro e o comportamento da aplicação passa a ser indeterminado. Fica totalmente a cargo do programador garantir que a aplicação execute corretamente.

Para cada modo listado acima, existem funções capazes de desempenhar comunicação bloqueante e não-bloqueante. Funções de envio bloqueantes só retornam após a execução de operações de recebimento. Já as funções não-bloqueantes permitem que computações posteriores sejam realizadas antes que a operação de envio esteja completa. Para operações de recebimento existe apenas um tipo de operação bloqueante e um tipo de operação não-bloqueante. Qualquer uma das duas pode receber mensagens enviadas através de qualquer um dos modos de envio citados anteriormente.

As implementações convencionais do MPI costumam utilizar três diferentes protocolos para comunicação ponto-a-ponto: *Short*, *Eager* e *Rendezvous*. O Protocolo *Short* tem a menor latência, sendo o ideal para mensagens pequenas. Nesse protocolo, os dados são enviados juntamente com o cabeçalho (envelope) da mensagem.

No *Protocolo Eager*, os dados da mensagem são enviados logo após o cabeçalho. Nesse protocolo, o processo que está enviando a mensagem considera que esta pode ser armazenada pelo processo destino (processo de recebimento). Uma das desvantagens desse protocolo é que ciclos de CPU podem ser consumidos pelo processo receptor para retirar mensagens da rede e/ou copiar dados para um *buffer*. Isso acontece pois, ao ser enviada, a mensagem pode não ser escrita diretamente no *buffer* do processo de recebimento, mas ser armazenada temporariamente no *buffer* do sistema até que o processo destino execute uma chamada a função de recebimento. Outra desvantagem é a possível exaustão da memória e terminação do programa, caso o *buffer* do sistema do processo de recebimento tenha excedido. Porém, a vantagem deste protocolo está na redução dos atrasos de sincronização, uma vez que o processo de envio não precisa conhecer as condições do processo destino antes de enviar a mensagem.

O Protocolo *Rendezvous* é usado quando o espaço do *buffer* do processo de recebimento não pode ser estipulado ou quando os limites do *Protocolo Eager* são excedidos.

Ele requer um tipo de *handshaking* entre o processo de envio e de recebimento, causando um atraso de sincronização e evitando que ocorra exaustão de memória e terminação do processo destino. O *handshaking* ocorre da seguinte forma: o processo de envio manda uma mensagem (envelope) para o processo destino com informações a respeito da mensagem que precisa ser enviada. O processo destino, ao receber a mensagem, analisa se existe espaço suficiente para armazenar a mensagem em questão. Caso exista, ele envia para o processo de origem uma mensagem de confirmação. Ao receber esta mensagem, o processo de envio repassa os dados para o processo destino.

O protocolo a ser utilizado em cada transferência depende do tamanho da mensagem. Os limites de tamanho podem ser manipulados de forma estática ou dinâmica. Por exemplo, o valor *default* para o tamanho mínimo de mensagem que deve ser enviada usando o protocolo *Rendezvous* é 64 Kbytes. Porém, o MPI permite que o usuário manipule este valor através do comando "setenv MP_EAGER_LIMIT <valor em bytes>" antes da execução da aplicação, ou através de um parâmetro passado na linha de comando do *mpirun* "-ssi rpi_tcp_short <valor em bytes>".

Tipicamente, o MPI implementa sincronização entre os processos de envio e recebimento de duas maneiras: *polling* ou interrupção. No modo *polling*, a tarefa do usuário é interrompida pelo sistema em intervalos regulares para verificação de eventos de comunicação. Se um evento de comunicação ocorre enquanto a tarefa do usuário está ocupada, o evento deve esperar. No modo de interrupção, a tarefa MPI do usuário é interrompida pelo sistema sempre que eventos de comunicação ocorrem.

Serão apresentadas nas subseções seguintes as implementações LAM/MPI e MPICH, ressaltando suas características principais, e também como surgiu o suporte em tais versões para computação em Grid.

2.3.4 LAM/MPI

LAM/MPI é uma implementação *open source* do MPI padrão que é desenvolvido e mantido pelo *Open System Lab* da Universidade de Indiana. LAM/MPI suporta todo padrão MPI-1 e muito do padrão MPI-2. LAM/MPI não é somente uma biblioteca que implementa o MPI padrão, mas também o ambiente LAM em tempo de execução. Esse ambiente LAM é baseado em *daemons*, que fornecem muitos dos serviços requeridos por programas MPI.

LAM utiliza um *daemon* no nível do usuário (lamd) sobre cada máquina para criar

um ambiente de execução persistente. Uma coleção de lamds é iniciada pelo comando *lamboot*. Esse comando envia uma solicitação para cada máquina, tipicamente via rsh ou ssh, para que cada uma delas inicie um lamd. Quando cada lamd é criado, ele é preenchido com uma tabela de rotas contendo o endereço de todos os outros processos lamd. Os processos LAM *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 de todos os outros processos MPI, formando-se um comunicador consistente (MPI_COMM_WORLD). Todos os processos nesse comunicador precisam executar o MPI_Finalize() antes de terminar a execução. Pode-se considerar as chamadas às funções MPI_Init() como sincronizações ou barreiras naturais entre os processos MPI.

O SSI (*System Services Interface*) para LAM/MPI tem como objetivo habilitar múltiplas instâncias de interfaces do sistema, cada uma delas de sua respectiva coleção, para estarem disponíveis em tempo de execução. Cada coleção de interfaces é um tipo de componente. Os tipos de componentes SSI atuais são:

- boot: usado para iniciar processos sobre nós remotos sem o uso de LAM *daemons*;
- coll: usado para fornecer algoritmos *back-end* para operações coletivas MPI;
- cr: suporte para *checkpoint-restart*;
- rpi: RPI (*Request Progression Interface*) usado para tráfego de mensagens MPI ponto-a-ponto, sendo o responsável por mover *bytes* entre processos MPI.

LAM suporta seis diferentes módulos RPI SSI: gm, lamd, tcp, sysv, usysv e crtcp. O gm RPI é usado com o nativo *Myrinet networks*. O lamd RPI usa o mecanismo de comunicação *"out-of-band"* de LAM para troca de mensagens MPI. Nesse módulo, processos em máquinas diferentes se comunicam através do *daemon* de cada máquina utilizando o protocolo UDP. No módulo tcp RPI, *sockets* TCP são abertos entre os processos e são usados para todo tráfego MPI. Sysv RPI usa memória compartilhada para comunicação entre processos MPI que estão no mesmo nó e *sockets* TCP para comunicação entre processos MPI em diferentes nós. *"System V semaphores"* são usados para *lock* do *pool* de memória compartilhada. O módulo usysv RPI, assim como o sysv RPI, usa memória compartilhada para comunicação entre processos em um mesmo nó e *sockets* TCP para comunicação entre processos em máquinas diferentes, porém usa *spin locks* para *lock* do *pool* de memória compartilhada. O módulo crtcp é o mesmo que tcp, exceto pelo fato

de usar o mecanismo de *checkpoint-restart*. Em aplicações onde os processos são criados dinamicamente através da função MPI_Comm_spawn(), os tipos de comunicação que utilizam memória compartilhada (sysv e usysv) não podem ser empregados.

LAM permite a utilização de *appschemas* nas chamadas à função MPI_Comm_spawn(). O esquema de uma aplicação LAM/MPI (*appschema*) é um arquivo ASCII que permite ao programador especificar um conjunto arbitrário de CPU's ou nós onde serão disparados processos MPI. Caso o desenvolvedor da aplicação não especifique nenhum arquivo de esquema para a criação dinâmica de processos, o LAM escalona os processos da aplicação em nós LAM a partir da CPU 0 (ou CPU com menor identificador) e continua até a CPU com maior identificador, criando um processo em cada CPU. Caso o número de processos da aplicação seja maior que o número de CPU's, o procedimento de criação de processos se repete a partir da CPU 0 (escalonamento Round Robin).

2.3.5 MPICH

O projeto do MPICH (MPI *Chameleon*) teve início em 1993 como uma tentativa de fornecer uma implementação imediata do padrão MPI-1. Ele foi desenvolvido pelo Argonne National Laboratory como um projeto de pesquisa, com o objetivo de fornecer características que possibilitem uma única implementação de MPI para muitos tipos de *hardware*. Para que isto fosse possível, o MPICH implementou MPI sobre uma camada de comunicação de baixo nível chamada de Interface de Dispositivo Abstrata (ADI - *Abstract Device Interface*) independente da arquitetura. Essa ADI contém um conjunto de definições sobre as quais são construídas as funções padrão do MPI, fornecendo aos desenvolvedores um ponto de partida conveniente para implementações particulares.

O "CH" em MPICH tem o significado de *Chameleon* (Camaleão), como símbolo de adaptabilidade aos ambientes e conseqüente portabilidade. Além de adaptáveis aos ambientes, os camaleões são rápidos, tendo o projeto como segunda meta fornecer o máximo de eficiência possível a essa portabilidade. *Chameleon* é um pacote portável de alto desempenho que permite troca de mensagens em supercomputadores paralelos. Uma quantidade significante de tecnologias do "*Chameleon*" foram incorporadas na versão MPICH.

Atualmente, o MPICH é uma implementação completa da versão 1.2 do MPI padrão, com extensão para suportar as funcionalidades de E/S paralelo definidas no padrão MPI-2. Funcionalidades de gerenciamento de processos propostas pelo padrão MPI-2, entre elas a criação dinâmica de processos, ainda não foram incorporadas à versão do MPICH.

2.3.6 Versões da Biblioteca MPI Habilitadas para Grids Computacionais

Com o desenvolvimento dos Grids Computacionais, algumas versões da interface para troca de mensagens MPI têm surgido para atender aos requisitos da computação em Grid. A partir da versão 7.0, a implementação LAM do padrão MPI passou a fornecer suporte para execução de aplicações paralelas MPI em Grids Computacionais que utilizam o Globus Toolkit. Outra versão do MPI habilitada para Grids, o MPICH-G2 tem sido desenvolvida como uma extensão da implementação MPICH do MPI, para usar serviços fornecidos pelo Globus Toolkit para suportar a execução transparente e eficiente de aplicações em ambientes Grids heterogêneos.

2.3.6.1 MPICH-G2

O MPICH-G2 é uma implementação completa do padrão MPI-1 que usa serviços oferecidos pelo Globus Toolkit para estender a implementação MPICH do MPI para execução nos Grids Computacionais. Ela representa uma completa reimplementação do primeiro sistema proposto, MPICH-G. MPICH-G2 mascara a heterogeneidade através dos serviços fornecidos pelo Globus Toolkit para propósitos como autenticação, autorização, criação de processos, monitoramento e controle de processos, comunicação e acesso a arquivos remotos.

Para iniciar uma aplicação MPICH-G2, o usuário emprega o *Grid Security Infrastructure* (GSI) para obter uma credencial *proxy* usada na autenticação do usuário em cada um dos *sites* pertencentes ao ambiente Grid. O usuário pode também usar o *Monitoring and Discovery System* (MDS) para selecionar computadores com base, por exemplo, em aspectos de configuração, disponibilidade e conectividade de rede.

Uma vez autenticado, o usuário usa o comando padrão *mpirun* para solicitar a criação de uma computação MPI. O MPICH-G2 implementa este comando usando o RSL (*Resource Specification Language*) para descrever o processo a ser executado. Com base nas informações encontradas em um *script* RSL, MPICH-G2 chama a biblioteca de coalocação de processos distribuída com o Globus Toolkit, o *Dynamically-Update Request Online Coallocator* (DUROC), para escalonar e iniciar a aplicação através de vários computadores especificados pelo usuário.

Após ter sido iniciada a aplicação, MPICH-G2 seleciona o mais eficiente método de comunicação possível entre quaisquer dois processos, usando *vendor-supplied* MPI (vMPI) se

disponível, ou *Globus Communication* (Globus IO) com *Globus Data Conversion* (Globus DC) para TCP, caso contrário.

Muitas melhorias têm sido observadas no desempenho do MPICH-G2 quando comparado com a versão anterior, MPICH-G. Dentre essas melhorias estão o aumento da largura de banda, a redução da latência para mensagens utilizando o *vendor* MPI intramáquina, o uso eficiente de *sockets* e o uso de operações multi-níveis que são cientes da topologia para comunicações coletivas.

Como no MPICH, o MPICH-G2 não oferece suporte a todas as características do MPI-2, em particular às funções de gerenciamento dinâmico de processos, essenciais para ambientes heterogêneos e dinâmicos como os Grids Computacionais. Este suporte será fornecido assim que for incorporado na versão MPICH.

2.3.7 Versões Especializadas da Biblioteca MPI com Tolerância a Falhas

Pesquisadores têm explorado diversas abordagens para fornecer tolerância a falhas em programas MPI. Um dos mais recentes e completos sistemas implementados, é o MPICH-V [28]. Ele é uma versão tolerante a falhas do MPICH que fornece *checkpointing* e *logging* de mensagens, para permitir que processos abortados sejam substituídos. Cada mensagem é transferida para um servidor remoto CM (*Channel Memory*), que faz o *log* e reenvia mensagens quando necessário. CM's são considerados estáveis e persistentes (isto é, nunca falham), podendo os processos reiniciados recuperarem seus dados através de uma reconexão com os CM's. Conseqüentemente, o custo para recuperar os dados de um processo reiniciado equivale ao dobro do tempo de comunicação.

Uma abordagem similar ao MPICH-V é o MPI-FT [55], baseado na versão LAM da biblioteca MPI. Essa versão também utiliza o *log* de mensagens para a recuperação de processos. Um processo é responsável por copiar todas as mensagens e reproduzí-las para o processo de recuperação (*recovery*). Ao iniciar, o MPI-FT cria processos extras capazes de substituir um dos processos da aplicação em caso de falha. Este sistema possui um alto *overhead* associado ao armazenamento de todas as mensagens.

Egida [60] é outro sistema baseado no MPICH que usa *log* de mensagens para uma recuperação (*recovery*) transparente. Para suportar a atomicidade nas trocas de mensagens, ele substitui operações não bloqueantes por operações bloqueantes. O processo mestre, o qual é considerado livre de falhas, é responsável por alterar informações do canal

de comunicação. A versão atual é capaz de detectar apenas falha de processos, sendo a recuperação de um processo feita sobre o mesmo processador onde ele estava executando antes da ocorrência de falha.

O MPI/FT [22] adota a redundância de tarefas para fornecer tolerância a falhas. Este sistema possui um coordenador central que repassa mensagens para todos os processos redundantes.

CoCheck [65] é um sistema de *checkpointing* coordenado para PVM e tuMPI. Este sistema é uma camada situada acima do PVM (ou MPI), que acrescenta funcionalidades (*wrapper*) à API original. As mensagens de controle dos processos são implementadas no mesmo nível das mensagens da aplicação.

Starfish [12] é um ambiente para execução de programas MPI que suporta tolerância a falhas. Cada nó executa um *daemon* starfish, responsável pela interação com clientes, criação de programas MPI (processos da aplicação), recuperação de falhas e por manter a configuração do sistema. Starfish apresenta uma arquitetura flexível e portável, permitindo diferentes implementações de protocolos de *checkpoint/restart*.

Recentemente proposto, o MPICH-GF é um sistema tolerante a falhas construído com base no MPICH-G2, implementação MPI habilitada para execução em ambientes Grids. MPICH-GF é totalmente transparente para o desenvolvedor da aplicação e não requer nenhuma alteração no código fonte. Uma das desvantagens dessa abordagem é o fato de terem sido feitas alterações no Globus *JobManager* para suportar *rollback-recovery* e para controlar processos distribuídos pelo ambiente Grid. O MPICH-GF fornece gerenciamento de processo dinâmico, o que não está implementado na versão padrão MPI-1, permitindo a criação de novos processos e a junção desses processos a grupos já existentes. O MPICH-GF ainda não é considerado um padrão.

2.4 Middleware de Serviços

Uma aplicação para Grid precisa ser projetada para explorar a capacidade heterogênea dos recursos disponíveis e superar os efeitos negativos causados pela flutuação de desempenho e da disponibilidade dos recursos compartilhados.

Muito se tem pesquisado sobre o desenvolvimento de *middlewares* capazes de criar abstrações sobre a complexidade dos ambientes Grids. Grande parte desses serviços *middlewares* são relativamente novos, e os serviços disponíveis podem variar de *site* para

site. Para tirar vantagens desses serviços, o programador precisa conhecer a fundo não somente o problema a ser solucionado, mas também as funcionalidades disponíveis no *middleware* instalado sobre os recursos nos quais a aplicação será executada. Como existe uma grande variedade de *middlewares* e as aplicações Grids são desenvolvidas com o objetivo de serem executadas em ambientes distintos, isso obriga que o desenvolvedor da aplicação gere diferentes versões da aplicação para cada ambiente de execução. O grande desafio é obter aplicações eficientes e robustas capazes de executar em Grids, sem que isso gere uma carga ainda maior para o programador ou usuário. O ideal é que a mesma aplicação seja capaz de executar com eficiência tanto em máquinas paralelas convencionais (por exemplo, *clusters* de PC's ou supercomputadores) como sobre Grids, sem a necessidade de modificações.

Esta seção enfoca principalmente em *middlewares* tipicamente conhecidos como sistemas de gerenciamento de recursos (SGR's) ou que fornecem funcionalidades ou serviços comumente utilizados como parte de um SGR. Sistemas gerenciadores de recursos são capazes de fornecer serviços tais como escalonamento de processos de uma aplicação, sendo todas as decisões tomadas com base apenas no estado do sistema. Grande parte dos SGR's existentes são dependentes de outros *middlewares* básicos. Para que aplicações utilizem tais sistemas, estes precisam estar instalados em todos os *sites* do Grid onde serão executados os processos da aplicação. Um exemplo de SGR é o Condor-G que será apresentado na Subseção 2.4.1.

Uma outra alternativa para execução de aplicações paralelas em ambientes Grids, também apresentada nesta seção, baseia-se na utilização de um sistema de gerenciamento da aplicação (SGA) próprio para cada aplicação. O SGA utiliza o Grid de acordo com a disponibilidade dos recursos e com características específicas de cada aplicação. Quando o SGA é parte de uma aplicação (*system-aware*), diz-se que o SGA está embutido na aplicação e a portabilidade é garantida através da eliminação das dependências em relação a *middlewares* de serviços específicos. Como conseqüência, aumenta-se o número de recursos disponíveis para a aplicação e a possibilidade de execuções mais rápidas. Os SGA's precisam ser estendidos de maneira que a aplicação seja capaz de mudar o seu comportamento de acordo com as disponibilidades dos recursos. Um exemplo de SGA não embutido na aplicação é o MyGrid, apresentado na Subseção 2.4.10. O *middleware* proposto neste trabalho, parte do *Framework* EasyGrid apresentado na Subseção 2.4.12, é um exemplo de SGA embutido na aplicação.

2.4.1 Condor-G

Condor é um sistema gerenciador de recursos projetado para suportar computação de alta-vazão (*high throughput*) através da descoberta de recursos ociosos e da alocação de *jobs* da aplicação a esses recursos. O conjunto de estações de trabalho gerenciados pelo Condor é chamado de Condor *pool*. Tarefas submetidas pelo usuário são enfileiradas por agentes Condor até que sejam encontrados os recursos adequados a sua execução.

Com o surgimento dos Grids Computacionais, em especial do Globus Toolkit, o agente Condor foi adaptado para se comunicar com o GRAM, e foi chamado de Condor-G. O Condor-G combina os protocolos de gerenciamento de recursos interdomínio do Globus Toolkit com os métodos de gerenciamento de recursos intradomínio do Condor, para permitir ao usuário utilizar recursos de múltiplos domínios como se todos eles pertencessem a um domínio pessoal [46]. Condor-G oferece uma série de funcionalidades presentes em Condor como a gravação do estado da aplicação e a atribuição de prioridades às tarefas a serem submetidas. Porém, Condor-G não oferece facilidades como o *checkpointing* e a migração presentes em Condor.

A decisão de onde executar *jobs* do usuário é tomada pelo agente Condor-G. Várias estratégias são possíveis. A mais simples delas, usada na implementação inicial do Condor-G, utiliza uma lista de servidores GRAM definidos pelo usuário. Outras estratégias mais sofisticadas combinam informações sobre a autorização do usuário, necessidades da aplicação e *status* dos recursos (obtidos do MDS) para construir uma lista de recursos candidatos. A partir dos resultados obtidos através de verificações no *status* atual dos recursos, *jobs* do usuário serão submetidos aos recursos apropriados.

A tecnologia Condor *flocking* permite que múltiplos computadores com o Condor instalado trabalhem juntos, criando um ambiente computacional com o estilo de um ambiente Grid. A grande diferença entre Condor *flocking* e Condor-G está no fato de que Condor-G permite operação interdomínio sobre recursos remotos que necessitam de autenticação e utilizam protocolos padrões que fornecem acesso a recursos controlados por outros sistemas gerenciadores de recursos.

O agente Condor-G permite que o usuário trate o Grid como um recurso inteiramente local. A Figura 2.5 mostra como é estruturada a implementação desse agente. O escalonador responde a uma solicitação do usuário para submeter processos para execução em recursos Grid, através da criação de um novo *GridManager daemon* que irá submeter e gerenciar estes processos. Um processo *GridManager* trata todos os processos para um

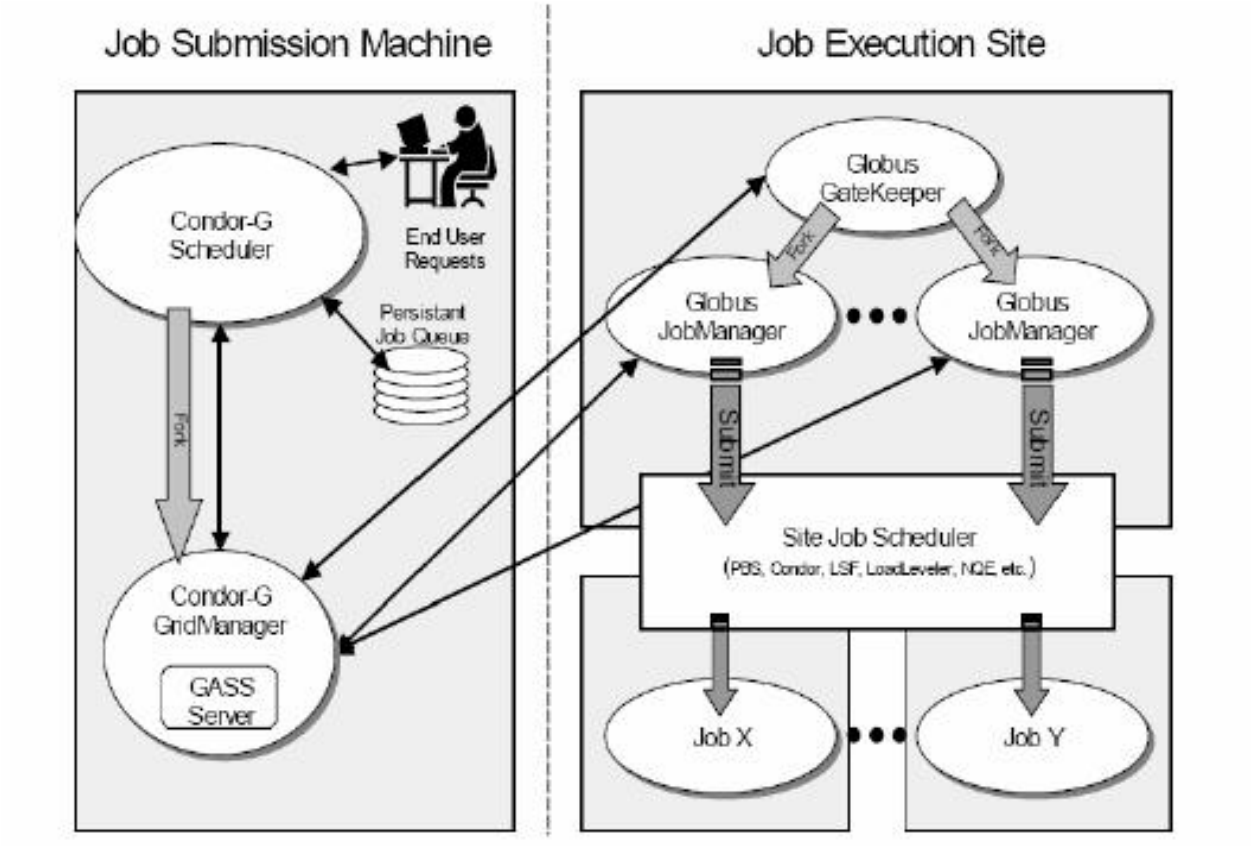


Figura 2.5: Execução remota pelo Condor-G [3].

único usuário e termina quando todos estes estiverem completos. Cada submissão de uma solicitação do *GridManager* resulta na criação de um *daemon* Globus *JobManager*, pelo Globus *Gatekeeper*. Este *daemon* se conecta ao *GridManager* usando o GASS para transferir os executáveis e os arquivos de entrada padrão, e subseqüentemente fornecer *streaming* em tempo real de saída padrão e erro. Em seguida, o *JobManager* submete processos para a execução do sistema de escalonamento do *site* local. Alterações no *status* desses processos são enviadas pelo *JobManager* para o *GridManager*, o qual altera o *Scheduler* onde o *status* é armazenado de forma segura.

O Condor-G foi construído para tolerar quatro tipos de falhas: falha do Globus *JobManager*, falha na máquina que gerencia os recursos remotos (*hosts Gatekeeper e JobManager*), falha na máquina onde o *GridManager* está executando (falha apenas no *GridManager*) e falha na conexão de rede entre duas máquinas.

O *GridManager* detecta falhas remotas através da verificação periódica do *JobManager* de todos os processos que ele gerencia. Se o *JobManager* não responde, o *GridManager* então verifica o *Gatekeeper* para aquela máquina. Se o *Gatekeeper* responde, então o

GridManager sabe que um *JobManager* falhou. Caso o *Gatekeeper* não responda, existem duas possibilidades: todos os recursos de gerenciamento da máquina falharam ou existe uma falha na rede. O *GridManager* não consegue distingüir esses dois casos. Se somente o *JobManager* falhou, o *GridManager* inicia um novo *JobManager*, o qual prosseguirá assistindo o processo ou avisando ao *GridManager* que o processo terminou. Caso contrário, o *GridManager* espera até que ele possa estabelecer contato com a máquina remota. Quando isso acontece, ele tenta se reconectar com o *JobManager*. Isso pode falhar por duas razões: ou o *JobManager* falhou (porque toda máquina falhou), ou o *JobManager* saiu normalmente (porque o processo terminou sua execução durante uma falha na rede).

Para proteção em caso de falha local, todos os estados relevantes para cada processo submetido são armazenados de forma segura e estável na fila de processos do *Scheduler*. Essa informação persistente permite que o *GridManager* recupere-se de uma falha local. Quando reiniciado, o *GridManager* lê a informação e reconecta-se para qualquer *JobManager* que estava rodando no momento da falha. Se um *JobManager* falhar, o *GridManager* inicia um novo *JobManager* para acompanhar aquele processo.

Para lidar com a disponibilidade variável de *sites* de um Grid Computacional, Condor-G utiliza uma técnica poderosa de gerenciamento denominada *Glide In*. Através do mecanismo *Glide In*, recursos do Grid (máquinas remotas) podem se unir temporariamente ao *pool* local do Condor, permitindo que o usuário submeta *jobs* para execução sobre os recursos adicionados da mesma forma que todos os *jobs* Condor são submetidos. Para que recursos remotos possam ser acrescentados ao *pool*, o Condor-G dispara inicialmente *daemons* Condor em computadores remotos para que estes possam informar a disponibilidade dos recursos nas máquinas em que estão executando. A partir desse momento, está criado um *pool* Condor capaz de oferecer as funcionalidades de um *pool* Condor padrão.

Os componentes de escalonamento no Condor-G possuem uma organização centralizada, ou seja, um único controlador de escalonamento, o *Scheduler*, é responsável pelas decisões do sistema. Esse tipo de organização apresenta algumas desvantagens como a falta de escalabilidade e a ausência de tolerância a falhas no *Scheduler*. Várias estratégias são possíveis para que o agente Condor-G determine onde executar os processos do usuário. A aproximação mais sofisticada combina informações sobre a autorização do usuário, os requerimentos da aplicação e o *status* dos recursos para construção de uma lista de recursos candidatos. Nesse caso, a escolha dos recursos para execução da aplicação requer que o usuário informe dados sobre a sua aplicação. O ideal seria que a escolha dos

recursos fosse o mais independente possível de informações fornecidas pelo usuário.

Condor-G pode ser utilizado para execução de aplicações paralelas MPI e PVM. Entretanto, necessita que os recursos utilizados para execução desse tipo de aplicação sejam dedicados, ou seja, após iniciada a execução, a aplicação não pode ser interrompida nem suspensa. Essa limitação não está de acordo com a natureza compartilhada dos Grids Computacionais.

2.4.2 Legion

O projeto Legion iniciado em 1993, é um ambiente de metacomputação orientado a objetos que fornece uma estrutura de *software* para Grid. Sua metodologia permite que usuários e administradores expressem seus desejos (tempo de resposta, utilização adequada dos recursos, entre outros), com o sistema agindo como mediador para encontrar uma alocação de recursos que seja ideal tanto para os usuários como para os administradores. Entre os objetivos do Legion estão a autonomia do *site*, suporte para heterogeneidade, capacidade de expansão, facilidade de uso, processamento paralelo para alcançar desempenho, tolerância a falhas, escalabilidade, segurança e suporte a multi-linguagens [49] [32] [6].

Em Legion, todos os componentes de *software* e *hardware* são representados por objetos. Cada objeto é um processo ativo que responde à solicitação de outros objetos do sistema. Legion define uma API para interação entre objetos, mas não especifica uma linguagem de programação ou protocolo de comunicação. Objetos comunicam-se através de chamadas de métodos assíncronos e as interfaces são definidas por um tipo de IDL (*Interface Description Languages*). O sistema não funciona sem o conjunto de *core objects*, que são essenciais para o gerenciamento de recursos. Entre os *core objects* e os *user objects* existem os *services objects*, que melhoram o desempenho do sistema, mas não são essenciais para sua operação.

Cada objeto Legion é definido e gerenciado pela sua própria classe de objetos ativa. Classes podem criar novas instâncias, escalonar a execução, ativar ou desativar um objeto, além de fornecer informação de estados para os objetos do cliente. As classes também podem agendar execuções. Em sistemas orientados a objetos, usuários podem herdar ou redefinir funcionalidades das classes. Esta característica permite que funcionalidades sejam adicionadas ou removidas para satisfazer as necessidades do usuário. Os dois principais *core objects* que representam os tipos de recursos básicos em Legion são: *Hosts* e *Vaults*. Objetos *Hosts* encapsulam a capacidade das máquinas e são responsáveis por instanciar objetos nos processadores. Objetos *Hosts* são implementados para interagir com sistemas

de gerenciamento de fila, tais como LoadLeveler e Condor. Objetos *Vaults* representam armazenamento persistente, mas somente para manter o estado de OPR (*Object Persistent Representation*). O OPR é usado para migração e em casos de *shutdown/restart*. Todos os objetos Legion automaticamente suportam *shutdown* e *restart*.

Os escalonadores (*Schedulers*) em Legion podem empregar diversos algoritmos de escalonamento, desde algoritmos simples até técnicas mais avançadas com algoritmos especializados ou baseados no conhecimento da aplicação. O modelo de gerenciamento do Legion permite que escalonadores definidos pelo usuário interajam com a infra-estrutura. Além de um escalonamento principal, o escalonador pode produzir um ou mais escalonamentos opcionais, que representam alternativas em caso de falha no escalonamento principal. Esses escalonamentos são construídos pelo *Scheduler* a partir das solicitações do usuário, preservando a autonomia dos recursos. Os componentes do modelo são: os recursos básicos (*Hosts e Vaults*), o banco de dados de informação (*Collection*), o componente que irá implementar o escalonamento (*Enactor*) e o monitor da execução (*Monitor*).

O *Scheduler* pode obter o estado do sistema, através de uma pesquisa ao *Collection* ou através de uma interação direta com os recursos. Em seguida, o *Scheduler* faz o mapeamento de objetos para recursos e o repassa para o *Enactor*, que irá negociar a utilização dos recursos, que podem estar em diferentes domínios administrativos. Após os objetos estarem executando, o *Monitor* pode solicitar um escalonamento, se necessário. Se durante a execução um recurso decide que o objeto precisa migrar, ele faz uma chamada ao *Monitor*, o qual notifica o *Scheduler* e *Enactor* que o re-escalonamento deve ser desempenhado.

Os *Hosts Objects* oferecem mecanismos de reserva de recursos. A reserva possui um instante de tempo inicial, uma duração e um período de *timeout* opcional. Os *Hosts* também suportam um banco de dados de atributos que permite que um rico conjunto de informações seja exportado.

O *Collection* permite que novas funções sejam inseridas, habilitando usuários a instalarem códigos para computar dinamicamente novas informações e integrá-las com as descrições de informações de recursos já existentes. Esta capacidade é especialmente importante para usuários do NWS (*Network Weather Service*), o qual prevê a disponibilidade futura dos recursos com base em estatísticas do passado.

O Legion possui uma implementação das bibliotecas MPI e PVM. Portanto, aplicações escritas utilizando estas bibliotecas precisam ser recompiladas e religadas para que possam se beneficiar das facilidades disponibilizadas pelo sistema. Uma aplicação pré-existente

pode ser utilizada sem ser recompilada, porém facilidades como *checkpointing* não estarão disponíveis à aplicação. Para obter os benefícios oferecidos em relação à Entrada/Saída remota, algumas alterações precisam ser feitas no código fonte da aplicação, o que não é recomendável, considerando que a maioria das aplicações para Grid é desenvolvida por cientistas de outras áreas e não por programadores.

O Legion oferece ainda suporte nativo a algumas linguagens de programação paralela, tais como MPL, BFS e Java. Aplicações que não utilizam nenhuma dessas linguagens podem ser encapsuladas dentro de objetos Legion. Se o programa realiza qualquer tipo de comunicação, basta que o programador escreva um adaptador para o programa já existente que converta as chamadas de métodos de comunicação do Legion.

No Legion, a organização dos componentes de escalonamento é hierárquica, o que permite escalabilidade e tolerância a falhas. Uma das desvantagens desse tipo de organização é que ela não fornece autonomia aos *sites*, o que é um problema em ambientes como os Grids, onde os diferentes *sites* não querem perder o controle sobre o uso dos seus recursos.

2.4.3 Nimrod/G

O sistema Nimrod pode ser usado para submeter, executar e coletar os resultados de múltiplos computadores [29]. Ele tem sido bem sucedido com um conjunto estático de recursos computacionais, mas não pode ser utilizado no contexto de Grids Computacionais, onde recursos estão espalhados por vários domínios administrativos. Esta restrição levou ao desenvolvimento de um sistema chamado Nimrod/G.

Nimrod/G é um sistema de gerenciamento de recursos e escalonamento baseado no sistema Nimrod, e que segue uma arquitetura modular baseada em componentes, possibilitando extensão, portabilidade, facilidade de uso e interoperabilidade entre componentes desenvolvidos independentemente. Ele usa os serviços MDS do Globus Toolkit para descoberta de recursos e a submissão de processos nos Grids Computacionais, e pode ser facilmente estendido para operar com qualquer outro serviço do *middleware* Grid, tal como NetSolve [30]. O objetivo do Nimrod/G é automatizar a modelagem e execução de aplicações *parameter sweep*, que são constituídas de tarefas independentes (com pouca ou nenhuma comunicação) executadas sobre diversos parâmetros.

Os componentes chave da arquitetura do Nimrod/G são: *Client* ou *User Station*, *Parametric Engine*, *Scheduler*, *Dispatcher* e o *Job-Wrapper*. O *Client* ou *User Station* atua como uma interface do usuário, controlando e supervisionando um experimento. Ele

também serve como um console de monitoramento e lista de *status* de todos os processos. Outra característica do Nimrod/G *Client* é permitir a execução de múltiplas instâncias do mesmo cliente em diferentes localizações. Isto significa que o experimento pode ser iniciado em uma máquina, monitorado em outra máquina pelo mesmo ou por outro usuário e ainda pode ser controlado de outra localização.

O *Parametric Engine* atua como um agente de controle de processos e é o componente central do sistema, responsável pelo gerenciamento e manutenção do experimento. Ele mantém o estado do mesmo e garante que este será gravado em um dispositivo de armazenamento persistente. Isso permite que o experimento seja reiniciado se um nó em que o Nimrod esteja executando falhar. O *Scheduler* é responsável pela descoberta e seleção de recursos e a submissão de processos. O algoritmo para descoberta de recursos interage com o diretório de serviço de informação no Grid (o MDS em Globus), identifica a lista de máquinas autorizadas e guarda o *status* de informação dos recursos. O *Dispatcher* inicia a execução do *job* no recurso selecionado e periodicamente atualiza o *status* da sua execução para o *Parametric Engine*. Finalmente, o *Job-Wrapper* interpreta um *script* contendo instruções para transferência de arquivos e execução de subtarefas. Ele é basicamente um intermediário entre o *Parametric Engine* e a máquina atual na qual a tarefa está executando.

Nimrod/G introduz o conceito de economia computacional como parte de um sistema de escalonamento para garantir que os desejos do usuário (tempo de resposta e custo) sejam atendidos. Isso torna o sistema mais complexo, sendo necessário utilizar serviços do *middleware* Grid para reserva de recursos e negociação de custo.

2.4.4 UNICORE

UNICORE (*Uniform Interface to Computer REsources*) fornece segurança e acesso intuitivo em um ambiente distribuído, oferece sólido mecanismo de autenticação integrado em seus procedimentos de administração, reduz esforço de treinamento e requerimentos de suporte e facilita a re-alocação de processos para diferentes plataformas [17].

O cliente UNICORE permite que o usuário crie, submeta e controle *jobs* a partir de qualquer *workstation* ou PC sobre a Internet. O cliente conecta-se a um *gateway* UNICORE o qual autentica o cliente e o usuário, antes de entrar em contato com o servidor UNICORE, que torna-se o responsável por gerenciar a submissão de *jobs* UNICORE.

Como parte do projeto UNICORE, um conjunto de funções tem sido desenvolvido para

permitir que usuários criem e gerenciem complexos *jobs batch* que possam ser executados sobre diferentes sistemas em diversos *sites*. Abaixo são listadas algumas das funções do UNICORE [38]:

- Criação e submissão de *jobs*: um *job* UNICORE (mais precisamente um grupo de *jobs*) pode recursivamente conter outros grupos de *jobs* e/ou tarefas. Ao ser submetido um grupo de *jobs* ao *site* UNICORE, o cliente UNICORE cria uma representação abstrata desse grupo (AJO - *Abstract Job Object*) armazenada como um objeto Java serializado ou em formato XML. O cliente UNICORE auxilia na criação, manipulação e complexo gerenciamento, interdependência entre processos de multi-sistemas, processos *multi-sites*, sincronização de processos e movimentação de dados entre sistemas, *sites* e espaços de armazenamento;

- Gerenciamento de *jobs*: o usuário tem total controle sobre os *jobs* e dados. É disponibilizado para o usuário o *status* de cada uma das tarefas, além de informação detalhada de *log* para análise de condições de erro. *Jobs* podem ser terminados e removidos do Grid UNICORE pelo usuário;

- Gerenciamento de dados: *jobs* UNICORE contêm tarefas dependentes que podem ser executas em diferentes centros computacionais. Um espaço temporário UNICORE é criado para cada grupo de *jobs* de maneira que a saída gerada por uma tarefa possa ser consumida por qualquer um dos seus sucessores. Em tempo de execução, o UNICORE realiza as movimentações de dados necessárias sem a intervenção do usuário;

- Controle de fluxo: o modelo de *job* adotado pelo UNICORE pode ser descrito como um conjunto de um ou mais Grafos Acíclicos Direcionados (GADs) que definem a ordem de execução. A versão 4 do UNICORE inclui execução condicional e repetitiva de um grupo de *jobs*. Este modelo permite que experimentos computacionais possam ser repetidos um número fixo de vezes ou até que uma condição definida seja atingida;

- Metacomputação: UNICORE é estendido para metacomputação, isto é, uso simultâneo de dois ou mais sistemas por uma única aplicação, tipicamente aplicações paralelas baseadas em MPI;

- Identificação única: a autenticação é desempenhada de uma maneira consistente e transparente e as diferenças entre plataformas são escondidas do usuário através

da criação de um portal que permite acesso aos supercomputadores, compilação e execução da aplicação e transferência de dados de entrada e saída;

- Gerenciamento de recursos: a disponibilidade de recursos de cada *site* (computação, dados e recursos de *software*) torna-se conhecida do UNICORE através dos administradores locais. O processo é completamente descentralizado. A informação sobre os recursos está disponível para todos os usuários UNICORE autenticados, durante o momento de criação e submissão de *jobs*. Após o usuário selecionar o sistema alvo, o cliente UNICORE verifica se a atribuição de *jobs* aos recursos selecionados é adequada. Caso não seja, o usuário será informado de que o *job* não pode ser executado como especificado.

Aplicações distribuídas com UNICORE são definidas como aplicações multiparte onde as diferentes partes podem rodar sobre diferentes sistemas de computadores de forma assíncrona ou podem ser sincronizadas seqüencialmente. Um processo UNICORE contém uma aplicação multiparte adicionada da informação sobre os sistemas destino, os recursos requeridos e as dependências entre as diferentes partes.

Durante o GGF7 (*Global Grid Forum*) o ministro Japonês da educação, cultura, esporte, ciência e tecnologia selecionou o UNICORE como o *middleware* Grid para uma nova iniciativa de pesquisa, denominada NAREGI (*National Research Grid Initiative*), comandada pelo Dr. Kenichi Miura do Laboratório Fujitsu.

2.4.5 Cactus

Cactus é um ambiente para solução de problemas (*PSE - Problem Solving Environments*) *open source* projetado por cientistas e engenheiros para atender a uma variedade de aplicações, incluindo astrofísica, relatividade e engenharia química [2]. Ele possui uma estrutura modular que facilita a computação através de diferentes arquiteturas e o desenvolvimento de código colaborativo entre diferentes grupos. Cactus teve origem na comunidade de pesquisa acadêmica, onde ele foi desenvolvido e usado durante muitos anos por um número grande de colaboradores internacionais, físicos e cientistas da computação. Atualmente, Cactus está associado a muitos projetos de pesquisa em ciência da computação, particularmente em visualização, gerenciamento de dados e camadas dos Grids Computacionais [13].

O nome Cactus veio do projeto de um núcleo central (ou *flesh*), o qual se conecta a um módulo da aplicação (ou *thorns*) através de uma interface extensível. *Thorns* podem

implementar aplicações científicas ou de engenharia, tais como fluidos dinâmicos. Outros *thorns* de um *toolkit* computacional padrão fornecem uma variedade de capacidades computacionais, tais como E/S paralela, distribuição de dados ou *checkpointing*.

Desenvolvedores constroem aplicações Cactus dinamicamente usando um meta-código com uma nova linguagem orientada a objeto. Esta descreve como pedaços diferentes de código, escritos em qualquer linguagem computacional comum tais como C, C++ e Fortran, podem interagir. Cactus executa sobre várias arquiteturas e fornece fácil acesso a muitas tecnologias de *software* que estão sendo desenvolvidas pela comunidade de pesquisa acadêmica. Cactus suporta muitas implementações de MPI, entre elas o MPICH, MPICH-G2 e LAM/MPI.

2.4.6 NetSolve

NetSolve é uma aplicação cliente-servidor projetada para solucionar problemas científicos complexos sobre Grid. Interfaces em Fortran, C e Matlab têm sido projetadas e implementadas possibilitando que usuários acessem e utilizem NetSolve mais facilmente [30]. Um projeto baseado em agentes tem sido implementado para garantir um uso eficiente dos recursos do sistema.

Balanceamento de carga é uma das características do projeto NetSolve. Dados todos os recursos computacionais disponíveis, NetSolve fornece ao usuário uma estratégia de "melhor esforço" para identificar o recurso adequado para a solução de um determinado problema.

Falhas podem ocorrer em diferentes níveis do protocolo NetSolve. Geralmente elas ocorrem devido ao mau funcionamento da rede, desaparecimento de um servidor ou falha no servidor. Um processo NetSolve detecta falha quando ele tenta estabelecer uma conexão TCP com um servidor e a conexão falha ou atinge *timeout* antes da conclusão. Neste caso, o processo NetSolve reporta o erro para o agente NetSolve, que marca o servidor que falhou, mas não o remove. O servidor será removido apenas após um determinado tempo e somente se ele não tiver sido reativado.

Outro aspecto de tolerância a falhas adotado pelo NetSolve é a tentativa de se minimizar os efeitos de uma possível falha. Quando o agente NetSolve recebe uma solicitação para um problema a ser solucionado, ele retorna uma lista ordenada de servidores adequados para resolver o problema em questão. O cliente irá tentar submeter seu problema a todos os servidores da lista, até que algum deles assuma o problema. Se nenhum servidor

da lista aceitar o problema, uma nova lista é gerada e assim sucessivamente, até que o problema seja solucionado.

O processo computacional sobre um *host* remoto pode morrer por alguma razão. Nesse caso, a falha é detectada pelo cliente NetSolve e o problema é enviado para outro servidor computacional disponível. Este processo faz com que o tempo de execução seja maior. O problema migra entre os possíveis servidores computacionais, até que ele seja solucionado ou não exista mais nenhum servidor.

Várias instâncias do agente NetSolve podem existir na rede. Uma estratégia global é ter uma instância do agente em cada rede local onde existem clientes NetSolve. Cada *host* em um sistema NetSolve executa um servidor "computacional" NetSolve (também chamado de recurso). Os recursos NetSolve possuem acesso a pacotes científicos (bibliotecas ou sistemas *stand-alone*). Um importante aspecto deste sistema baseado em servidor é que cada instância do agente tem sua própria visão do sistema. Sendo assim, algumas instâncias podem estar cientes de mais detalhes do que outras instâncias, dependendo da sua localização. Porém, eventualmente o sistema atinge um estado estável no qual cada instância possui todas as informações disponíveis no sistema.

Um grande esforço tem sido feito para combinar facilidade de uso, generalidade e desempenho, que são os principais objetivos do projeto NetSolve.

2.4.7 GridLab

Financiado pela comissão Européia, GridLab produz um conjunto de serviços orientados a aplicação e *toolkits*, fornecendo capacidades tais como: mediação de recursos, monitoramento, gerenciamento de dados, segurança, informação, serviços adaptativos, entre outras. Serviços são acessados usando o *Grid Application Toolkit* (GAT), que fornece aplicações com acesso a vários serviços GridLab, recursos, bibliotecas específicas, ferramentas, etc., de maneira que os usuários finais e especialmente desenvolvedores de aplicações possam construir e rodar aplicações sobre o Grid sem precisar conhecer detalhes sobre o ambiente de execução [4].

Toda tecnologia GridLab é embutida em uma arquitetura GridLab definida em um ambiente em camadas. Na camada mais alta (chamada de *User Space*) encontra-se o GAT (*Grid Application Toolkit*) e o *GridSphere* (*framework* para o desenvolvimento de um portal Grid). GAT é um conjunto de API's flexíveis, genéricas e coordenadas para acessar serviços Grid de códigos de aplicações genéricas, portais, sistemas de gerenciamento de

dados e outros, juntamente com implementações fornecidas por ferramentas desenvolvidas no projeto GridLab. GAT fornece aos programadores de aplicações Grid uma interface uniforme para vários tipos de *middleware* Grid. O objetivo do *GridSphere* é ser um *toolkit* genérico capaz de fornecer aos programadores mecanismos que permitam a agregação de suas próprias tecnologias de Grid.

A camada de *middleware* (chamada de *Capability Space*) cobre todas as capacidades do Grid requeridas pela aplicação, por usuários e administradores. Tais capacidades são:

- GRMS (*Grid Resource Management and Brokering Service*);
- Acesso e gerenciamento de dados (serviços Grid para gerenciamento e acesso de dados);
- GAS (*Grid Authorization Service*);
- iGrid (serviços de informação do GridLab);
- Delphoi (monitoramento da rede e serviço de predição de desempenho);
- Mercury (infra-estrutura de monitoramento do Grid);
- Visualização (serviço de visualização do Grid);
- Serviços Móveis (serviços Grid com suporte a tecnologia *wireless*).

Testes têm sido realizados com Cactus e Triana. O CGAT (*Cactus Grid Application Toolkit*) fornecerá uma extensão da interface GAT para Cactus. Cactus é uma das primeiras aplicações voltadas para GAT. O TGAT (*Work-Flow Application Toolkit*), com base no pacote Triana, desenvolvido originalmente para análise de dados de ondas gravitacionais, fornecerá um segundo exemplo de *toolkit* para GridLab.

2.4.8 GrADS

O objetivo do projeto GrADS [67], ainda em desenvolvimento, é simplificar computação sobre recursos heterogêneos e distribuídos, da mesma maneira que o *World Wide Web* simplificou o compartilhamento de informações sobre a Internet. GrADS fornece um novo serviço externo para ser acessado pelo usuário do Grid e principalmente por desenvolvedores de aplicações *grid-aware*.

O sistema GrADS tem como base três componentes: *Configurable Object Program*, que contém o código da aplicação e as estratégias para o mapeamento da mesma, o *Resource Selection Model*, que fornece a estimativa de desempenho da aplicação em relação a recursos específicos e o *Contract Monitor*, responsável por interromper e remapear processos quando uma degradação de desempenho é detectada. O *status* do sistema pode ser monitorado através de consultas ao MDS ou pela verificação de dados do NWS.

O principal objetivo desse *framework* é melhorar o tempo de resposta de uma aplicação individual. Um mecanismo de migração de tarefas é implementado, sendo este dependente da carga dos recursos, do tempo de execução da aplicação quando a carga é introduzida no sistema e dos benefícios que podem ser obtidos com a migração da aplicação.

2.4.9 AppLeS

Alcançar bom desempenho em sistemas de metacomputação pode ser difícil. Para utilizar de forma mais eficiente as plataformas heterogêneas distribuídas, uma aplicação deve ser escalonada de maneira que seja otimizado o uso dos recursos do sistema. Qualquer aproximação bem sucedida para escalonamento em metacomputação deve incorporar uma estratégia que leve em consideração heterogeneidade e contenção. Mecanismos de escalonamento que tomam decisões com base apenas nas características do ambiente de execução geralmente não são eficientes nesses tipos de sistemas o que leva desenvolvedores de aplicações a focalizar no desenvolvimento de escalonadores customizados para suas aplicações individuais.

A Universidade da Califórnia tem desenvolvido uma metodologia baseada em agentes para escalonamento no nível da aplicação chamada AppLeS (*Applications Level Scheduler*), agora parte do sistema GrADS. Agentes AppLeS são baseados no paradigma de escalonamento no nível da aplicação - tudo sobre o sistema é avaliado em termos de seus impactos sobre a aplicação. Cada aplicação terá seu próprio AppLeS e cada AppLeS irá combinar informações estáticas e dinâmicas para determinar um escalonamento customizado específico da aplicação e implementar esse escalonamento sobre os recursos distribuídos da metacomputação.

Existem várias conseqüências importantes do paradigma de escalonamento no nível da aplicação:

- Informações específicas da aplicação e informações específicas do sistema são necessárias para um bom escalonamento;

- Desempenho depende do critério de desempenho da própria aplicação;
- A distância entre os recursos é dependente de como eles serão usados pela aplicação;
- Informações dinâmicas são necessárias para que o estado do sistema seja avaliado corretamente;
- Previsões são válidas apenas dentro de um determinado intervalo de tempo;
- Um escalonamento é somente tão bom quanto a sua previsão base.

O objetivo do projeto AppLeS é desenvolver *software* para auxiliar e realçar as atividades de escalonamento do desenvolvedor da aplicação sobre um sistema de metacomputação distribuído [23]. Agentes AppLeS não são sistemas gerenciadores de recursos; eles fiam-se em sistemas como o Globus, Legion e outros, para desempenhar essa função. Cada agente AppLeS serve como um *middleware* que fornece um sistema de gerenciamento da aplicação, o qual dinamicamente deduz e coordena um escalonamento customizado para as aplicações de metacomputação.

O *AppLeS Parameter Sweep Template* (APST) utiliza técnicas de escalonamento de aplicações para obter melhora no desempenho de aplicações do tipo *parameter sweep* em Grids Computacionais. Esse tipo de aplicação apresenta pouca ou nenhuma comunicação e sua execução é geralmente repetida com freqüência, com alterações apenas nos seus parâmetros de entrada.

Um agente AppLeS é organizado em termos de quatro subsistemas e um único agente ativo chamado de *Coordinator*, como pode ser observado na Figura 2.6. Os quatro subsistemas são: *Resource Selector* que escolhe e filtra diferentes combinações de recursos para execução da aplicação, o *Planner*, que gera um escalonamento dependente dos recursos para uma dada combinação de recursos, o *Performance Estimator*, que gera um desempenho estimado para o escalonamento candidato de acordo com as métricas de desempenho do usuário e o *Actuator*, que implementa o melhor escalonamento sobre o alvo dos sistemas de gerenciamento de recursos.

O *pool* de informações é alimentado por três origens distintas: *Network Weather Service* (NWS), *User Interface* (UI) e *Models*. O NWS fornece informações dinâmicas sobre o sistema e faz uma previsão da carga dos recursos em um período de tempo no qual a aplicação será escalonada. A *User Interface* fornece informação específica sobre estrutura, características e corrente implementação da aplicação e suas tarefas, assim como informações sobre o critério do usuário para o desempenho, restrições de execução,

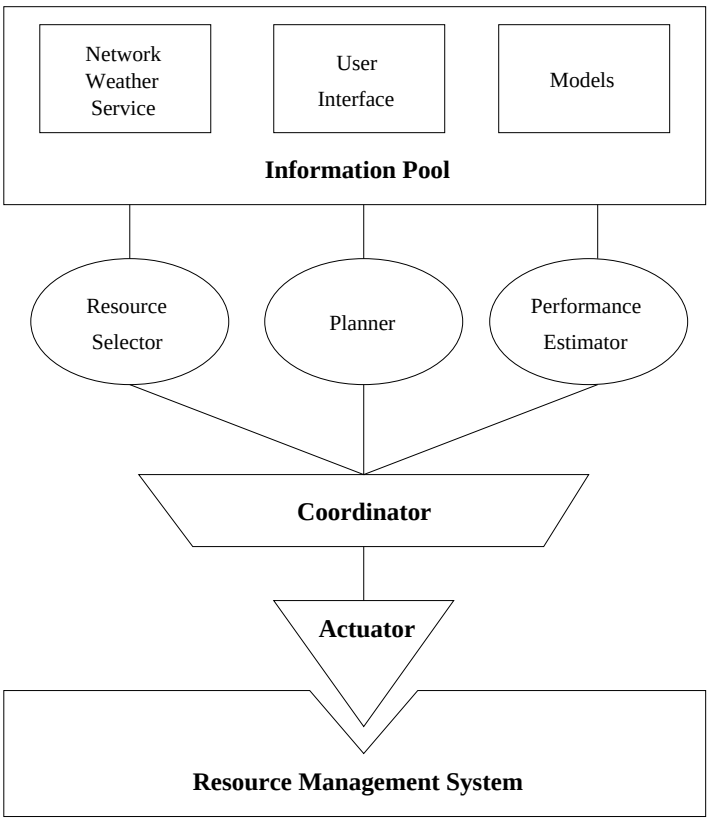


Figura 2.6: Organização de um agente AppLeS.

informações de login, etc. Finalmente, o *Models* fornece um repositório de modelos de classes de aplicações de metacomputação *default* e modelos específicos da aplicação para serem usados para estimativa de desempenho, planejamento e seleção de recursos.

A utilização do AppLeS ocorre da seguinte forma: inicialmente, o usuário fornece informações para o agente AppLeS através da UI. Caso o usuário não tenha estas informações, valores *default* adequados podem ser usados ou obtidos de análises automáticas. A UI irá informar ao *Coordinator* que máquina o usuário pode acessar, qual o *login* que o identifica, etc. Usando as informações do UI para guiar o processo de seleção, o *Resource Selector* identifica o conjunto de recursos que o *Coordinator* deve considerar. De acordo com a aplicação, o *Resource Selector* usa a noção de distância lógica entre recursos para priorizá-los. Para cada configuração viável de recursos, a função do *Planner* é calcular um escalonamento potencial. O *Coordinator* então usa o *Performance Estimator* para avaliar cada escalonamento de acordo com os objetivos de desempenho do usuário. O escalonamento que melhor otimizar os objetivos do usuário é escolhido para ser implementado pelo *Actuator*.

Pode-se concluir que, o AppLeS está mais preocupado em promover o desempenho de uma aplicação individual do que em otimizar o uso dos recursos do sistema ou maximizar

o *throughput* para uma coleção de processos. Uma das suas desvantagens está no fato de que, para se conseguir um bom escalonamento é preciso que o usuário forneça informações específicas da aplicação, o que não é uma tarefa fácil para usuários comuns. As previsões de desempenho fornecidas pelo NWS são fundamentais em AppLeS, o que o torna dependente deste.

Atualmente, AppLeS está sendo usado em aplicações de renderização volumétrica na vizualização de dados em imagens, em aplicações que simulam a microfisiologia molecular através do uso de difusão de Monte Carlo e de algoritmos de reações químicas e em problemas de modelagem de fluidos mecânicos [1].

2.4.10 OurGrid e MyGrid

OurGrid [9] [15] é um projeto de Grid Computacional desenvolvido em conjunto pela Universidade Federal de Campina Grande e Hewlett-Packard. O objetivo do OurGrid é pesquisar e desenvolver soluções para uso e gerenciamento de Grids Computacionais. O projeto OurGrid é fundamentado no projeto MyGrid, desenvolvido também na Universidade Federal de Campina Grande, que propôs e desenvolveu uma implementação de um sistema de Grid Computacional. O sistema de compartilhamento de recursos OurGrid é uma rede ponto-a-ponto de *sites* que compartilham recursos, capaz de garantir que os participantes da rede que doaram mais recursos sejam priorizados quando estes solicitarem tais recursos.

O MyGrid é um sistema de Grid Computacional para aplicações do tipo *Bag-of-Tasks* (BoT), ou seja, para aplicações que sejam compostas por um conjunto de tarefas independentes entre si [33]. Para utilizar a infra-estrutura do MyGrid, o desenvolvedor necessita informar os dados de entrada de cada tarefa e onde as informações de saída serão armazenadas. Não é necessário utilizar nenhuma biblioteca específica nem se preocupar com questões como o escalonamento das tarefas ou configuração do sistema Grid. Também não é necessário instalar nenhum programa nos computadores que formarão a rede. É necessário apenas que o usuário tenha algum tipo de acesso remoto aos outros nós que compõe o Grid, por exemplo, via SSH (*Secure SHell*).

Na arquitetura do MyGrid há apenas um escalonador centralizado para todos os nós que compõem o Grid. Toda tarefa a ser executada no Grid deve ser enviada para este escalonador que, após analisar a carga de trabalho em todos os nós do Grid, decide qual nó executará cada tarefa. Está prevista na arquitetura do MyGrid a capacidade de interoperabilidade com outros sistemas de Grid Computacional.

Entre as desvantagens desse sistema está a centralização do escalonador, o que pode acarretar em perda de escalabilidade e dificuldade em se prover tolerância a falhas. A vantagem do MyGrid está na sua simplicidade, porém fica este sistema limitado apenas à execução de tarefas BoT.

2.4.11 InteGrade

O InteGrade é um *middleware* Grid desenvolvido em conjunto por pesquisadores do Departamento de Ciência da Computação da USP (IME-USP), Departamento de Informática da PUC-Rio e Departamento de Computação e Estatística da UFMS com o objetivo de aproveitar o poder computacional ocioso de máquinas compartilhadas [48].

O InteGrade possui uma arquitetura orientada a objetos, onde cada módulo do sistema se comunica com os demais a partir de chamadas remotas. Este projeto utiliza CORBA como sua infra-estrutura de objetos distribuídos. Para que o usuário que compartilha a máquina com o Grid não perceba qualquer queda de desempenho nos serviços fornecidos pela sua aplicação, é instalada na máquina cliente uma implementação leve da infra-estrutura CORBA e o acesso aos seus recursos de *hardware* é controlado por um escalonador a nível do usuário.

Em ambientes como os Grids, a disponibilidade de recursos pode variar ao longo do tempo e um recurso pode retornar para o seu proprietário a qualquer momento. Essa variação na disponibilidade de recursos dificulta o escalonamento de aplicações, o que levou o projeto InteGrade a adotar um mecanismo de análise e padrões de uso, com o objetivo de, após diversas coletas de informações, ser capaz de fazer uma previsão probabilística da disponibilidade dos recursos compartilhados. Através das previsões é possível estimar por quanto tempo um recurso ficará ocioso refletindo em melhores decisões de escalonamento.

O InteGrade tem como objetivo oferecer suporte a uma grande variedade de aplicações paralelas. Em ambientes dinâmicos como os Grids Computacionais, garantir a execução de uma aplicação paralela é uma tarefa ainda mais difícil. É necessária a existência de mecanismos como o *checkpointing*, capazes de obter o estado de um processo em execução para que este possa ser reiniciado em caso de falha. Entretanto, em aplicações paralelas o *checkpointing* pode se tornar proibitivo, devido a possíveis mensagens da aplicação circulando pela rede. Para atenuar este problema, o InteGrade adotou o modelo BSP (*Bulk Synchronous Parallel*) como modelo de computação paralela. Tal modelo impõe freqüentes sincronismos, permitindo que o estado de uma computação seja salvo periodicamente para que possa ser usado em caso de falhas ou necessidade de migração. Um

pré-compilador automaticamente instrumenta o código fonte de uma aplicação C/C++, adicionando código capaz de salvar e recuperar o estado da aplicação. Em caso de falhas, a aplicação é reiniciada a partir do último estado global armazenado [35].

2.4.12 EasyGrid

O projeto EasyGrid tem como objetivo desenvolver um *framework* para a transformação automática dos programas paralelos baseados na biblioteca MPI em aplicações *system-aware*. O projeto leva em consideração a facilidade de uso e a portabilidade, questões fundamentais em ambientes Grids. Aplicações *system-aware* são aquelas capazes de se auto-adaptar às mudanças ocorridas no ambiente computacional.

A metodologia EasyGrid objetiva permitir aos programadores se concentrar em como explorar paralelismo para resolver o problema e deixar que o *framework* EasyGrid gere uma versão *system-aware* da aplicação capaz de utilizar de uma maneira apropriada os recursos Grid disponíveis ao usuário [26]. A maioria dos projetos de *middlewares* atuais adotam uma visão centrada nos recursos disponíveis no ambiente Grid para garantir uma utilização eficiente dos mesmos, o que pode ser observado na Figura 2.7. Tal gerenciamento é feito tipicamente por um Sistema Gerenciador de Recursos (SGR) com base em monitoramento e análise das informações específicas do sistema. Em ambientes Grids, esse tipo de abordagem pode não ser suficiente para que uma aplicação obtenha um bom aproveitamento dos recursos. É interessante considerar as características de cada aplicação para que possam ser feitos ajustes adequados à sua execução.

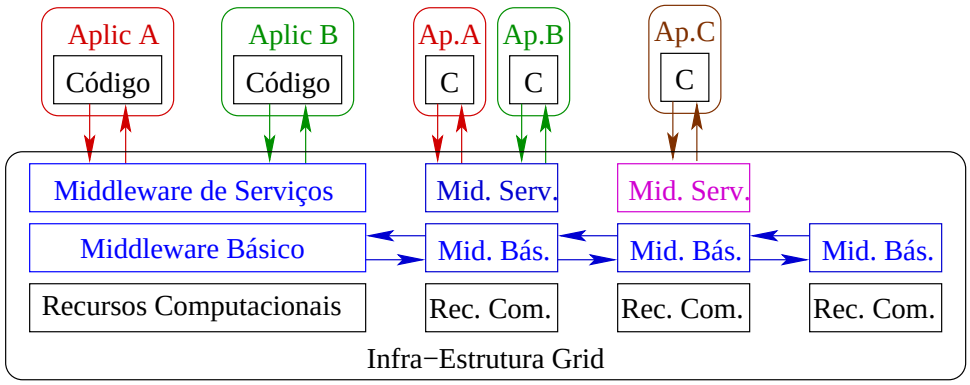


Figura 2.7: Visão centrada no sistema.

A metodologia EasyGrid é centrada na aplicação, ou seja, o *middleware* de serviço é parte de cada aplicação Grid individual. A utilização eficiente dos recursos é obtida por um Sistema Gerenciador da Aplicação (SGA) distribuído com cada aplicação *system-aware*, e não com a utilização de um Sistema Gerenciador de Recursos (SGR). A visão

centrada na aplicação é ilustrada na Figura 2.8, onde é possível observar que em cada aplicação é embutido um *middleware* de serviço específico na forma de um SGA. Como conseqüência, a aplicação torna-se portável, pois elimina a dependência de *middlewares* de serviços instalados nos recursos do Grid. Na visão centrada no sistema apresentada na Figura 2.7, o *middleware* de serviço é parte da infra-estrutura Grid, ou seja, ele precisa estar instalado em todos os recursos do Grid onde serão executados processos da aplicação. Esse tipo de visão limita o número de recursos disponíveis para uma aplicação em um ambiente Grid.

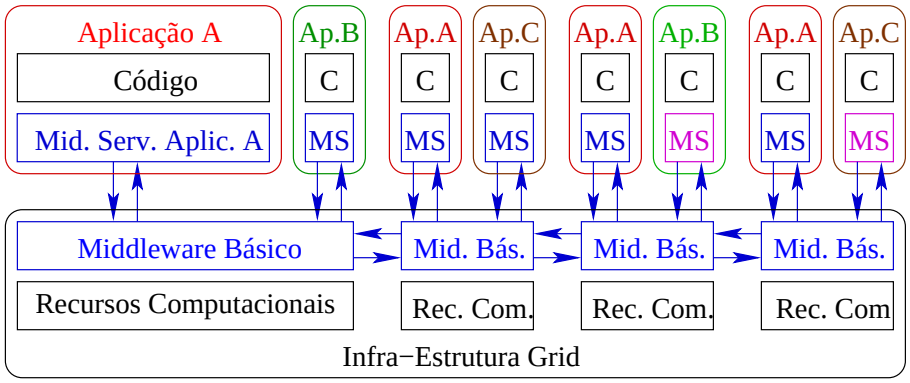


Figura 2.8: Visão centrada na aplicação.

No exemplo da Figura 2.7, as aplicações A e B (Ap.A e Ap.B) só podem executar nas duas primeiras máquinas do Grid, uma vez que a terceira máquina não dispõe do *middleware* de serviço necessário à execução dessas aplicações e a quarta máquina não possui nenhum tipo de *middleware* de serviço instalado. Apenas o terceiro recurso do Grid tem o *middleware* de serviço necessário para execução da aplicação C (Ap.C). Na Figura 2.8, as três aplicações (Ap.A, Ap.B e Ap.C) podem executar em qualquer recurso deste ambiente, devido ao fato do *middleware* de serviço estar embutido na própria aplicação.

Como a intenção do Grid é fornecer poder computacional para um grande número de usuários, o projeto EasyGrid tem como foco aplicações paralelas escritas em MPI, por ser este muito utilizado em programação paralela. Para que uma aplicação em um ambiente heterogêneo distribuído tenha bom desempenho, ela deve ser escalonada apropriadamente, para que os recursos do sistema possam ser utilizados de uma maneira mais eficiente. O projeto EasyGrid tem se dedicado no estudo de heurísticas e ferramentas de escalonamento estático e dinâmico. A necessidade de adaptar a execução de aplicações às características do ambiente Grid levou o projeto EasyGrid a investir em um modelo de escalonamento em duas etapas. Inicialmente, um escalonamento estático tenta obter um melhor escalonamento possível para o conjunto inicial de recursos. Com base em dados iniciais fornecidos pelo escalonamento estático e em informações obtidas durante

a execução, um escalonamento dinâmico é capaz de ajustar a execução da aplicação a mudanças ocorridas no meio. O monitoramento de aplicações foi uma das necessidades encontradas pelo projeto EasyGrid para ser capaz de prover escalonamento dinâmico de processos da aplicação e tolerância a falhas. Apesar da carga embutida com o gerenciamento da aplicação, o ideal é que o ambiente EasyGrid seja leve, ou seja, tenha um baixo grau de interferência e intrusão. O *framework* EasyGrid também será usado para estudar o problema de escalonamento estático e dinâmico, a integração de tolerância a falhas e as estratégias de escalonamento para aplicações *system-aware* em Grids Computacionais.

A Figura 2.9 apresenta um modelo do *framework* EasyGrid para gerar aplicações *system-aware* a partir do código MPI do usuário. Nesse modelo, os retângulos com os cantos arredondados representam as funções enquanto os retângulos comuns representam os arquivos. Além do programa MPI, uma lista com os recursos do Grid para os quais o usuário tem acesso (*Grid Access File*) é também necessária.

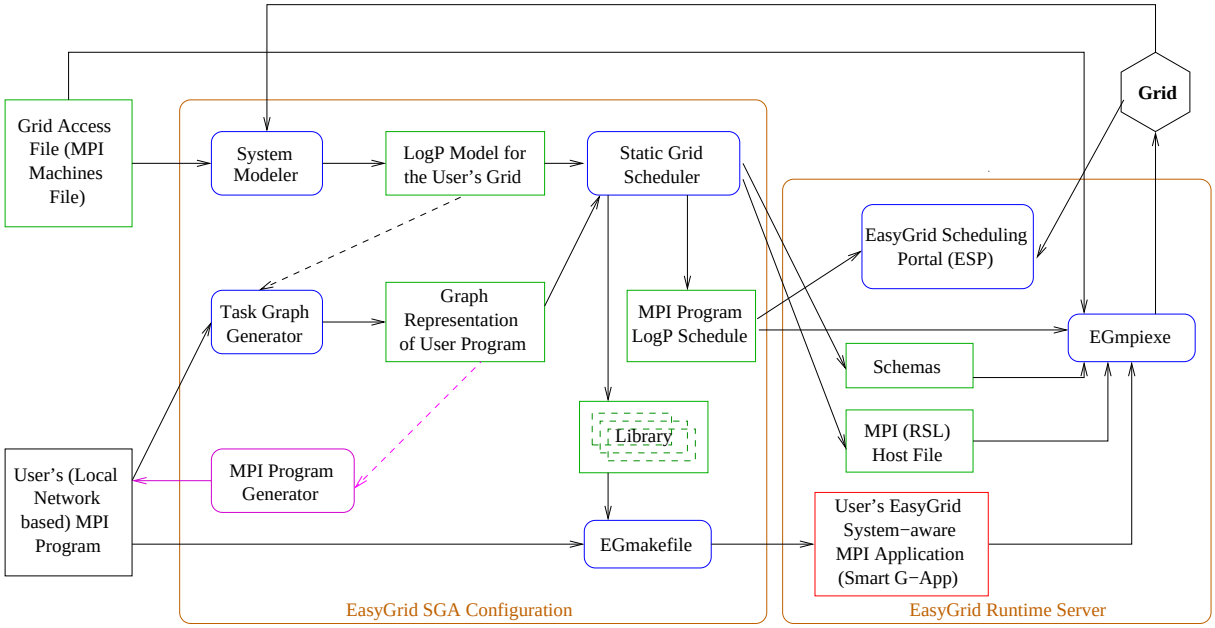


Figura 2.9: *Framework* EasyGrid.

A partir da lista de recursos Grid, o *System Modeler* cria um modelo arquitetural HLogP [57] através da obtenção de informações (por exemplo, velocidade de processador, carga da máquina, latência de comunicação entre outros) ou de serviços de diretórios (tal como MDS Globus). Com base nas características disponíveis dos recursos Grid e em características da aplicação, um escalonamento inicial é produzido pelo *Static Grid Scheduler* para guiar o início da execução da aplicação, através da identificação dos recursos para os quais cada processo MPI deve ser alocado. Em escalonamento de tarefas, uma aplicação paralela pode ser representada por um Grafo Acíclico Direcionado (GAD), onde

nós denotam tarefas e arestas representam as dependências de dados (comunicações MPI). Enquanto o *Task Graph Generator* pode ser usado para criar uma representação GAD do programa do usuário, o *MPI Program Generator* cria aplicações MPI sintéticas a partir de representações GAD. A utilização de aplicações sintéticas facilita a investigação dos efeitos da granularidade e estrutura do programa em relação à eficiência do escalonamento gerado e da sua execução em ambientes Grids. Além de ser uma ferramenta educacional para analisar e comparar os escalonamentos produzidos pelas diferentes implementações do *Static Grid Scheduler*, o *EasyGrid Scheduling Portal* pode ser usado como um portal para acessar e executar aplicações em um Grid.

O *Static Grid Scheduler*, além de gerar um escalonamento inicial das tarefas da aplicação, é também responsável por definir as máquinas onde serão disparados os processos gerenciadores do SGA EasyGrid. Essa informação, assim como dados da aplicação (nome do executável e parâmetros) e parâmetros do SGA (definidos no Capítulo 3) são armazenados em arquivos de esquema (*Schema*) utilizados para disparar a aplicação *system-aware*.

O *EGmakefile* gera uma aplicação MPI *system-aware* incorporando o *middleware* apropriado a cada aplicação (o SGA), sem alterar uma única linha do código fonte do usuário. As funções do SGA são embutidas nos processos MPI através de uma camada de abstração (*wrapper*) desenvolvida para agregar funcionalidades às chamadas às funções padrão do MPI. O *script EGmpiexe* é responsável por iniciar a execução da aplicação após verificar a validade do *proxy* Globus do usuário e realizar a distribuição inicial dos dados necessários (por exemplo, executável da aplicação) para execução da aplicação em tais ambientes.

Como será discutido no Capítulo 3, o SGA de uma aplicação EasyGrid é distribuído em três níveis: o Gerenciador Global, responsável pelo controle da execução da aplicação em todos os *sites* do Grid; os Gerenciadores dos Sites, responsáveis pela execução da aplicação nas máquinas dos *sites*; e os Gerenciadores Locais das Máquinas, responsáveis por gerenciar a execução de processos da aplicação atribuídos a cada máquina local. Nas seções seguintes, serão abordados aspectos de monitoramento, escalonamento de processos e tolerância a falhas, questões fundamentais em ambientes dinâmicos, compartilhados e heterogêneos como os Grids Computacionais.

2.5 Monitoramento

O monitoramento tem a sua importância focada no levantamento de informações relativas às características e às condições do sistema, bem como as informações ligadas ao comportamento da execução da aplicação. Mais do que um simples serviço de informação, o processo de monitoramento é fundamental em uma variedade de serviços, tais como escalonamento dinâmico de tarefas de uma aplicação, tolerância a falhas, análise de desempenho, replicação de dados e muitos outros.

Em ambientes Grids, devido ao dinamismo, heterogeneidade e compartilhamento dos recursos, o processo de monitoramento é ainda mais importante do que em sistemas distribuídos comuns. É importante considerar aspectos como: a quantidade de dados envolvidos em processos de monitoramento, que é ainda maior quando se trata de ambientes Grids; local para armazenamento de tais informações; de quanto em quanto tempo o monitoramento deve ocorrer; o grau de intrusão causado por esse processo, entre outros.

Sistemas de monitoramento para Grids Computacionais devem ser escaláveis, o que pode ser obtido com o uso de ferramentas de monitoramento com baixo grau de intrusão e alto desempenho. A portabilidade também é uma questão importante nesse tipo de ambiente heterogêneo, além da necessidade de um nível de segurança que permita o transporte seguro das informações monitoradas. Para uma precisão das informações obtidas durante o processo de monitoramento, é importante que os componentes do sistema e os seus usuários tenham uma noção de tempo global. A sincronização de relógios pode ser obtida através de tecnologias já estabelecidas, como por exemplo o NTP (*Network Time Protocol*) [58].

Sistemas distribuídos de monitoramento incluem tipicamente quatro estágios: geração de eventos de monitoramento, processamento das informações obtidas em tais eventos, distribuição dos dados para as partes interessadas e, finalmente, a apresentação dessas informações. Em seguida serão apresentados alguns sistemas de monitoramento existentes atualmente, em adição ao MDS e NWS apresentados nas Subseções 2.3.1.4 e 2.3.2. Grande parte das ferramentas de monitoramento que serão apresentadas nesta seção tem como objetivo obter informações que facilitem a administração dos recursos do Grid monitorado.

MapCenter [27], desenvolvido como parte do projeto EU DataGrid, é um monitor da aplicação que fornece aos usuários uma visualização (interface *web*) da disponibilidade e distribuição de serviços através do Grid. A intenção é que o MapCenter funcione como uma ferramenta de administração do Grid capaz de obter informações sobre a disponibil-

idade dos recursos.

GridICE [16], também conhecido como um InterGrid Monitor Map e EDT-Monitor, foi desenvolvido como parte do projeto DataTag para facilitar o trabalho dos administradores de ambientes Grids. Ele fornece informações sobre o *status* e utilização das Organizações Virtuais, *sites* e recursos. A partir de dados históricos e de informações em tempo real, esse sistema é capaz de gerar estatísticas básicas sobre a utilização dos recursos e apresentá-las ao usuário através de uma interface *web*.

Autopilot [61] é um *framework* que permite que aplicações se adaptem dinamicamente a alterações sofridas pelo ambiente de execução. Ele implementa uma série de sensores configuráveis que podem ser ativados dinamicamente, gerenciados e desativados, além de permitir que clientes adquiram sensores remotamente sem conhecimento da sua localização lógica e física. O mecanismo Autopilot é usado pelo projeto GrADS para monitorar o tempo de execução alcançado por diferentes partes de uma aplicação.

O CODE-based monitoring system [62] é um sistema de monitoramento e gerenciamento usado no NASA Information Power Grid (IPG). Este sistema é baseado no *framework* CODE (Control and Observation in Distributed Environments), tendo a intenção inicial de facilitar a admistração dos sistemas computacionais. O sistema de monitoramento CODE está preparado para monitorar máquinas, redes e serviços.

O GridRM (Grid Resource Monitoring) [19] é um projeto de pesquisa que tem como objetivo fornecer um "caminho único" a um conjunto diversificado de origens de dados de monitoramento, como os que são tipicamente encontrados em ambientes Grids (por exemplo, Ganglia, NetLogger, Network Weather Service, etc). Em GridRM, cada organização irá possuir um *gateway* baseado em Java, responsável por coletar e normalizar eventos ocasionados por sistemas de monitoramento local.

Hawkeye [5] é uma ferramenta de gerenciamento e monitoramento para *cluster* de computadores que utiliza parte da tecnologia de Condor e é disponibilizado como uma distribuição *stand-alone* para Linux e Solaris.

HBM (Globus Heartbeat Monitor) [64] é uma implementação de um serviço não-seguro de detecção de falhas de processos e máquinas. HBM foi empregado nas primeiras versões do Globus para verificar a disponibilidade de serviços Grid.

Mercury [20] é um sistema de monitoramento genérico construído como parte do projeto GridLab. Mercury é uma versão habilitada para Grids do monitor GRM, parte do ambiente de desenvolvimento de programas paralelos P-GRADE. GRM é uma biblioteca

de instrumentação para aplicações que utilizam troca de mensagens em ambientes paralelos tradicionais.

NetLogger (Network Application Logger Toolkit) [51] é usado para realizar análises de desempenho de sistemas complexos, tais como aplicações cliente-servidor e aplicações *multi-threads*. NetLogger combina eventos de monitoramento associados a máquinas, redes e aplicações, oferecendo uma visão geral do sistema para que possam ser identificados gargalos de desempenho.

OMIS Compliant Monitor (OCM-G) [21] é um sistema de monitoramento para aplicações Grids interativas, desenvolvido como parte do projeto EU CrossGrid. OCM-G é uma implementação habilitada para Grids do On-line Monitoring Interface Specification (OMIS). O sistema OCM-G procura equilibrar o *overhead* do processo de monitoramento de acordo com as características da aplicação, uma vez que ele trata de aplicações interativas que necessitam de respostas em tempo real.

Ganglia [56] é um sistema de monitoramento hierárquico inicialmente projetado para *clusters* de computadores, mas também usado em ambientes Grids. Ganglia introduz um considerável *overhead* tanto nas máquinas monitoradas como na rede, devido a constantes distribuições das informações coletadas.

RGMA (Relational Grid Information Service) [34] está sendo construído como parte do projeto EU DataGrid, um *framework* que combina monitoramento Grid e serviços de informação baseados em modelo de relacionamento.

2.6 Escalonamento de Tarefas

Uma das maiores questões em sistemas distribuídos é o desenvolvimento de técnicas eficazes para distribuição dos processos de aplicações paralelas sobre múltiplos processadores. O problema está em como distribuir (ou escalonar) processos entre elementos de processamento de forma a atingir um determinado desempenho alvo, considerando-se minimização do tempo de execução, minimização dos atrasos de comunicação e/ou maximização da utilização dos recursos [31].

O escalonamento local desempenhado pelo sistema operacional de um processador consiste em atribuir processos para fatias de tempo do processador (*time slice*). O escalonamento global é o processo de decisão de onde executar um processo em um sistema com vários processadores. O escalonamento global pode ser classificado em estático ou

dinâmico.

Tipicamente, o objetivo do escalonamento estático é minimizar o tempo total de execução de um programa concorrente, enquanto minimiza os atrasos provocados pela comunicação entre os processos da aplicação. A maior vantagem do escalonamento estático é que toda a sobrecarga (*overhead*) do processo de escalonamento ocorre em tempo de compilação. Entretanto, é difícil estimar em tempo de compilação a quantidade de tempo que um processo gastará na sua execução. A mesma dificuldade ocorre em relação às estimativas de atrasos de comunicação, pois em tempo de execução, atrasos podem ocorrer devido ao congestionamento na rede. Além dos problemas já citados, o algoritmo de escalonamento requer ferramentas para modelagem arquitetural capazes de prever a sua execução sobre uma dada arquitetura. Devido à grande variação no custo computacional e no custo de comunicação, é difícil identificar um modelo arquitetural suficientemente preciso e que não seja muito complexo, tornando-se extremamente difícil o desenvolvimento de algoritmos de escalonamento estático.

O escalonamento dinâmico é baseado na redistribuição de processos entre processadores durante o tempo de execução. A distribuição é desempenhada pela transferência de tarefas de um processador para outro mais apropriado (por exemplo, menos carregado no caso de balanceamento de carga) com o objetivo de melhorar o desempenho da aplicação.

A vantagem do escalonamento dinâmico em relação ao estático é que o sistema não precisa conhecer todo o comportamento da execução da aplicação antes que ela ocorra. A maior desvantagem do escalonamento dinâmico está na sobrecarga obtida durante a execução, devido à coleta e transferência de informações entre processadores, ao processo de tomada de decisão para seleção do processador adequado para possíveis transferências de tarefas e ao atraso decorrido da realocação das mesmas.

Em relação ao escalonamento dinâmico, uma pequena distinção é feita entre os conceitos de redistribuição de tarefas e migração de tarefas. Neste trabalho, considera-se que o processo de redistribuição de tarefas se refere à realocação de processos que ainda não estão em execução entre os recursos disponíveis no ambiente. A migração de tarefas envolve a atribuição de processos já em execução a recursos mais adequados. Os dois mecanismos têm como objetivo a adaptação de tarefas de uma aplicação a mudanças sofridas pelos ambientes de execução, sendo os dois mecanismos acionados pela ocorrência de eventos dinâmicos gerados tanto pelo Grid Computacional como pela aplicação em execução.

O problema de escalonamento de tarefas já é considerado complexo (NP-Completo

[47]) quando relacionado a sistemas homogêneos. Em sistemas como os Grids Computacionais, onde os recursos são heterogêneos e dinâmicos, a dificuldade em se atribuir tarefas a recursos se torna ainda mais acentuada. Entretanto, esse tipo de mecanismo é fundamental para o bom desempenho da aplicação em execução nos Grids, uma vez que aplicações precisam ser adaptadas à variação de carga e disponibilidade dos recursos existentes.

2.6.1 Escalonamento de Tarefas em Grids Computacionais

Os componentes de escalonamento de um SGR (Sistema Gerenciador de Recursos) podem ser organizados de forma centralizada, hierárquica ou descentralizada [53]. O escalonamento centralizado envolve apenas um controlador, que é responsável pelas decisões do sistema. Entre as vantagens dessa organização estão a facilidade de gerenciamento, facilidade de uso e a habilidade na co-alocação de recursos. Em SGRs para ambientes Grids, essa organização apresenta algumas desvantagens, tais como falta de escalabilidade, susceptibilidade a falhas e a dificuldade de acomodar múltiplas políticas decorrentes de recursos pertencentes a domínios administrativos diversos. O Condor é um exemplo de *middleware* que apresenta escalonamento centralizado.

No outro tipo de organização, o hierárquico, os controladores são organizados em uma hierarquia disponibilizando escalabilidade e tolerância a falhas. Em compensação, este tipo de organização não fornece autonomia para os *sites* e múltiplas políticas de escalonamento. Muitos sistemas Grid, tais como o Legion utilizam escalonadores hierárquicos. A organização descentralizada é uma outra alternativa capaz de fornecer escalabilidade, tolerância a falhas e ainda autonomia dos *sites* e múltiplas políticas de escalonamento. Um exemplo da organização descentralizada é o Ninf [59], um sistema de computação remoto orientado para fornecer computação numérica.

Para que o *overhead* envolvido no escalonamento dinâmico não torne inviável a execução da aplicação em ambientes Grids, é preciso que sejam tomadas algumas decisões com relação a questões como: em que momento deve ser ativado um escalonamento dinâmico, as condições sob as quais ele deve ocorrer e a política adotada para redistribuição de tarefas entre os recursos.

Um grande número de ativações do processo de escalonamento dinâmico pode acarretar em perda de desempenho, particularmente quando a aplicação é composta por um grande número de tarefas e diferentes aspectos do sistema alvo precisam ser levados em consideração. Por outro lado, se o número de ativações for muito pequeno, o escalonador dinâmico pode não ser capaz de ajustar a execução da aplicação às mudanças ocorridas no

sistema. Como conseqüência, fica clara a necessidade de se avaliar o efeito de diferentes eventos de escalonamento sobre o desempenho do escalonamento dinâmico realizado, assim como a política adotada para escolha dos recursos envolvidos na realocação de tarefas durante o processo de re-escalonamento.

Em aplicações paralelas que envolvem troca de mensagens, e conseqüentemente podem possuir relações de dependência, apenas o conhecimento dos custos computacionais de cada tarefa não é suficiente para que sejam tomadas decisões eficientes de escalonamento. É necessário também, considerar os atrasos decorridos do envio e recebimento de mensagens entre cada tarefa. Isso exige que as ferramentas de escalonamento sejam auxiliadas por mecanismos capazes de modelar a computação de uma aplicação através da obtenção de todas as características que envolvem o sistema alvo.

Heurísticas de escalonamento híbrido (empregam uma fase de escalonamento estático e uma fase de escalonamento dinâmico) [24] têm sido propostas com o intuito de atingir um melhor desempenho na execução de aplicações em ambientes dinâmicos e compartilhados, envolvendo o mínimo de *overhead* possível. Informações obtidas pelo escalonamento estático são fornecidas para as heurísticas dinâmicas como dados iniciais para suas tomadas de decisão.

Adaptar a execução de aplicações às mudanças de condições é essencial para se atingir bom desempenho e solucionar problemas ocasionados por falhas em recursos ou processos.

2.7 Tolerância a Falhas

Grande parte das aplicações paralelas e distribuídas atuais envolvem uma enorme quantidade de processamento, e são geralmente executadas em sistemas formados por centenas ou milhares de processadores interconectados entre si. Tais tipos de sistemas são totalmente vulneráveis a falhas, tanto nas redes de interconexão como nos processadores que compõem o ambiente de execução. A ausência de tolerância a falhas em tais ambientes obriga que aplicações sejam reiniciadas sempre que uma situação indesejada ocorra. Como conseqüência, aplicações são concluídas apenas se um intervalo longo livre de falhas ocorrer. Isso faz com que a média do tempo de execução de um programa cresça exponencialmente com o tamanho da aplicação.

Em particular, devido à sua natureza instável, Grids Computacionais estão mais sujeitos a falhas do que plataformas computacionais tradicionais. Máquinas podem ser desconectadas do Grid devido a falhas de *hardware*, problemas na rede e a terminação

forçada de processos em máquinas remotas para priorizar a computação local.

Esquemas de tolerância a falhas tipicamente consistem dos seguintes passos [69]:

- Detecção da Falha: processo de reconhecimento da ocorrência de erro em um sistema;
- Localização da Falha: após a detecção de erro no sistema, a parte do sistema que causou o erro é identificada;
- Contenção da Falha: após a falha ser localizada, a parte que falhou é isolada para prevenir uma possível propagação da falha para o resto do sistema;
- Recuperação da Falha: é o processo de recuperação do *status* do sistema para um estado consistente. Este passo pode incluir reconfiguração do sistema.

Uma das formas mais básicas de tolerânica a falhas para aplicações paralelas consiste de *Checkpointing e Rollback Recovery* [66]. Para garantir a confiabilidade de aplicações, é extremamente importante preservar o estado de uma aplicação, com o objetivo de preservar toda a computação já realizada. *Rollback Recovery* é a técnica mais comum utilizada para preservar o estado de uma aplicação, e por isso tem recebido grande atenção da comunidade científica.

A técnica de *Rollback Recovery* modela uma aplicação paralela baseada em troca de mensagens como um número fixo de processos em um sistema distribuído, que se comunicam sobre a rede através do envio de mensagens. Assume-se que os processos têm acesso a algum tipo de armazenamento seguro capaz de garantir que estes possam ser recriados em caso de falhas. Periodicamente, durante a execução de uma aplicação, o sistema grava um *snapshot* dos processos da aplicação. Caso um processo falhe, o estado da aplicação pode ser recuperado pela troca de estado para o mais recente estado do *checkpoint*. A técnica de *Rollback Recovery* pode ser subdividida em duas categorias: técnicas baseadas em *checkpoint* e técnicas baseadas em *log*.

2.7.1 Rollback Recovery Baseado em Checkpointing

Geralmente, protocolos de *Rollback Recovery* baseados em *checkpoint* salvam periodicamente o estado corrente de cada processo envolvido na computação. Existem três subcategorias de protocolos *Rollback e Recovery* basedos em *checkpoint*:

- *Checkpoint* Não-Coordenado (assíncrono): fica a cargo de cada processo decidir em que momento será realizado seu próprio *checkpoint*. Embora não sofra com a complexidade de sincronização, no *checkpoint* não-coordenado o processo pode decidir fazer um *checkpoint* que não pode ou não será usado em um estado global consistente, resultando em um *overhead* desnecessário na criação do *checkpoint*. É muito difícil determinar quais *checkpoints* fazem parte de um estado global consistente mais recente;

- *Checkpoint* Coordenado (síncrono): todos os processos da aplicação são forçados a realizar *checkpoints* ao mesmo tempo, gerando um único *checkpoint* consistente da aplicação. Esta técnica aumenta a complexidade na geração de *checkpoints*, mas reduz a complexidade na recuperação do estado da aplicação, uma vez que não é necessário pesquisar um conjunto de *checkpoints* consistentes;

- *Checkpoint* Induzido a Comunicação (parcialmente assíncrono): cada processo faz *checkpoint* localmente, como em um *checkpoint* não-coordenado. Entretanto, esta técnica permite que os processos sejam forçados a fazer um *checkpoint* de maneira a gerar um estado global consistente.

2.7.2 Rollback Recovery Baseado em Log de Mensagens

Protocolos de *Rollback Recovery* baseados em *log* realizam *checkpoint* com a gravação de mensagens enviadas e recebidas por cada processo da aplicação. Se o processo falha, o *log* pode ser usado para reiniciar o processo após o mais recente *checkpoint*, de maneira a reconstruir seu estado anterior. Existem três principais técnicas usadas pelos protocolos de recuperação baseados em *log* para garantir que todos os processos possam ser recuperados para um estado consistente em um evento de falha:

- *Log* Pessimista (síncrono): protocolos de *log* pessimista são projetados para situações em que se assume que uma falha pode ocorrer após qualquer evento não-determinístico. Supõe-se que um evento não-determinístico é modelado como um envio/recebimento de mensagem. Protocolos pessimistas fazem um *log* de cada evento não-determinístico, em um local de armazenamento seguro, antes que o evento afete a computação. O determinante de um evento não-determinístico engloba as informações necessárias para a reexecução do evento. No *log* pessimista, a aplicação fica bloqueada até a gravação do determinante (*log* síncrono). Implementações de *log* pessimista devem procurar utilizar técnicas especiais para reduzir os efeitos do *log*

síncrono sobre o desempenho da aplicação;

- *Log* Otimista (assíncrono): esta técnica grava *logs* em locais de armazenamento não seguros, sendo estes *logs* periodicamente escritos em um local de armazenamento estável. Esta técnica reduz o *overhead* sobre a aplicação, uma vez que o processo não precisa ficar bloqueado enquanto espera por cada mensagem ser escrita em disco. Conseqüentemente, a recuperação do sistema em caso de falha é muito complexa;

- *Log* Causal: os protocolos de *log* causal mantêm a vantagem apresentada pelos protocolos de *log* pessimista e otimista, mas em relação ao custo, necessitam de técnicas de recuperação muito mais complexas. Similar ao *log* otimista, o *log* causal grava *logs* em locais de armazenamento não seguros. Além disso, os protocolos de *log* causal asseguram que o determinante de cada evento não-determinístico, que precede de forma causal o estado de um processo, está armazenado em lugar estável ou localmente para aquele processo.

Apesar de ser uma área muita pesquisada, tolerância a falhas para ambientes Grids ainda é um grande desafio, principalmente quando são tratadas aplicações paralelas envolvendo troca de mensagens entre processos da aplicação.

2.8 Resumo

Este capítulo apresentou uma visão geral sobre a computação em Grid, abordando a sua estrutura em camadas e os componentes que as constituem. Três *middlewares* básicos foram destacados, entre eles o Globus Toolkit, fornecendo serviços tais como segurança e acesso a recursos e a biblioteca para troca de mensagens MPI. Entre os *middlewares* de serviços apresentados, é possível observar que a abordagem padrão adotada pela comunidade científica é orientada ao sistema (*System Centric View*), sendo grande parte dos projetos citados baseados em Sistemas Gerenciadores de Recursos. Uma visão alternativa é adotada pela metodologia EasyGrid, que baseia-se na utilização de um sistema de gerenciamento da aplicação para execução eficiente e robusta de aplicações em ambientes Grids. Aspectos de monitoramento, escalonamento de processos e tolerância a falhas são brevemente apresentados por serem estas questões fundamentais em Grids Computacionais. No próximo capítulo será descrito o Sistema de Gerenciamento de Aplicações MPI para ambientes Grids (SGA EasyGrid), contribuição central deste trabalho, projetado para oferecer suporte para cada um destes aspectos.

Capítulo 3

Sistema de Gerenciamento da Aplicação

Para que aplicações paralelas executem de forma eficiente em ambientes heterogêneos e distribuídos como os Grids Computacionais, mecanismos de gerenciamento, tais como escalonamento dinâmico de processos ou balanceamento de carga dinâmico e tolerância a falhas devem ser fornecidos pelos *middlewares* disponíveis em tais ambientes.

Neste capítulo, apresenta-se a estrutura do Sistema de Gerenciamento de Aplicações (SGA) para programas MPI executando em ambientes Grids, no contexto do *middleware* EasyGrid. O sistema proposto é capaz de controlar a execução de processos de uma aplicação MPI através de uma estrutura hierárquica de processos gerenciadores. A distribuição hierárquica foi adotada com o intuito de adequar os processos gerenciadores à organização dos recursos pelo Grid, minimizando o custo embutido pelo gerenciamento da aplicação.

O escalonamento dinâmico e a identificação de falhas são possíveis devido à existência de um mecanismo de monitoramento capaz de coletar dados durante a execução da aplicação MPI. Com base nesses dados, é possível tomar decisões com relação à redistribuição de processos pelos recursos disponíveis no Grid, de forma a obter uma melhora no desempenho da aplicação e solucionar problemas decorrentes de falhas em recursos ou processos.

A implementação do sistema de gerenciamento proposto procurou criar uma solução para execução de aplicações MPI, em ambientes Grid, sem a necessidade de modificar a implementação padrão do LAM/MPI. Isso garante a portabilidade da aplicação e conseqüentemente maximiza o número de máquinas do ambiente Grid que podem ser utilizadas na execução da aplicação.

3.1 Arquitetura Proposta

Conforme apresentado anteriormente, Grids Computacionais são constituídos de diversos recursos geograficamente distribuídos, heterogêneos e compartilhados, conectados por uma rede de alta velocidade. Esses recursos, geralmente pertencentes a domínios administrativos diferentes, são organizados em *sites* submetidos a diversas políticas de gerenciamento e diferentes configurações de *hardware* e *software*, tornando ainda mais difícil o controle da execução de uma aplicação nesse tipo de ambiente. Com base nessa disposição das máquinas nos ambientes Grids, está sendo apresentada nesta seção a proposta de uma hierarquia de gerenciadores distribuídos capazes de gerenciar de forma eficiente e robusta a execução de aplicações MPI no ambiente descrito.

A hierarquia proposta do SGA EasyGrid é formada de três níveis de processos gerenciadores, como pode ser observado na Figura 3.1. No topo da hierarquia (nível 0), aparece o Gerenciador Global (GG), encarregado de gerenciar todos os *sites* pertencentes ao Grid que estão envolvidos na execução da aplicação gerenciada. Além de gerenciar os *sites* do Grid, o processo GG funciona como uma interface para o usuário ou um portal Grid, fornecendo o *status* ou progresso da execução de uma aplicação, caso seja solicitado. O nível 1 corresponde aos processos Gerenciadores dos Sites (GS), que respondem pelo gerenciamento dos processos da aplicação atribuídos a cada *site*. E finalmente, o nível mais baixo da hierarquia (nível 2) é composto pelos Gerenciadores Locais das Máquinas (GM), responsáveis pelo escalonamento, criação e execução de processos da aplicação atribuídos à máquina (*host*) local. Tanto o processo GG como os processos GS's podem assumir as funções de um Gerenciador da Máquina, caso isso seja considerado vantajoso para execução da aplicação.

A Figura 3.1 exemplifica o SGA para uma única aplicação. A implementação do SGA proposto permite que para a execução de uma mesma aplicação, processos gerenciadores de níveis diferentes sejam atribuídos a uma mesma máquina ou a máquinas distintas do ambiente Grid.

Cada processo gerenciador do SGA EasyGrid, proposto nesta dissertação, é estruturado em camadas, como ilustrado na Figura 3.2. A funcionalidade de cada camada depende do nível do processo de gerenciamento na estrutura hierárquica. Essa estrutura hierárquica permite que a aplicação se adapte às mudanças sofridas pelo ambiente de execução, uma vez que cada processo gerenciador é capaz de adotar diferentes políticas de escalonamento dinâmico e tolerância a falhas específicas às suas necessidades.

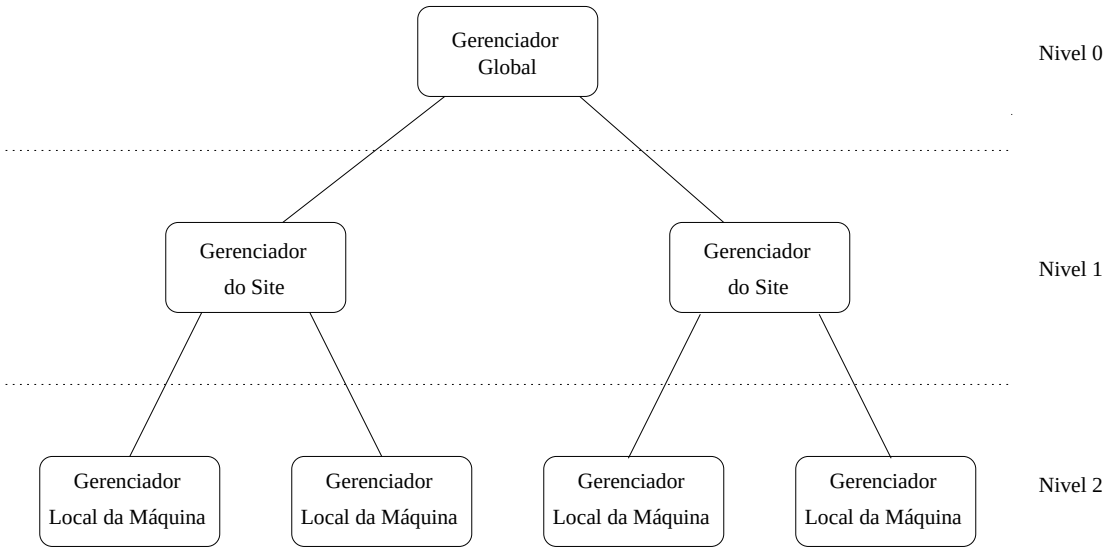


Figura 3.1: Hierarquia de gerenciadores criada com base na organização dos recursos no Grid.

A camada de gerenciamento de processos é responsável pela criação dinâmica de processos MPI, tanto da aplicação como gerenciadores, e pelo redirecionamento de mensagens trocadas entre os processos criados. A coleta de informações relevantes relacionadas à execução de uma aplicação MPI é desempenhada pela camada de monitoramento da aplicação. Os dados coletados pela camada de monitoramento alimentam as camadas de escalonamento dinâmico, responsável pela redistribuição de tarefas da aplicação e a camada de tolerância a falhas, responsável pela identificação e tratamento de falhas. No topo da estrutura em camadas do SGA EasyGrid está a aplicação MPI do usuário e a camada de abstração MPI (*wrapper*).

Outra possível abordagem para a arquitetura proposta do SGA EasyGrid está relacionada à construção da hierarquia de gerenciamento constituída por apenas dois níveis de processos gerenciadores. Nessa abordagem, o Gerenciador Global mantém as mesmas funcionalidades apresentadas na versão de três níveis. Porém, o Gerenciador do Site assume as funcionalidades dos Gerenciadores das Máquinas, passando a ser o responsável pela criação e gerenciamento de processos da aplicação disparados nas máquinas de nível 2.

Nas Subseções 3.1.1, 3.1.2 e 3.1.3 serão apresentados detalhes da criação e funcionalidade de cada processo gerenciador dentro da hierarquia proposta.

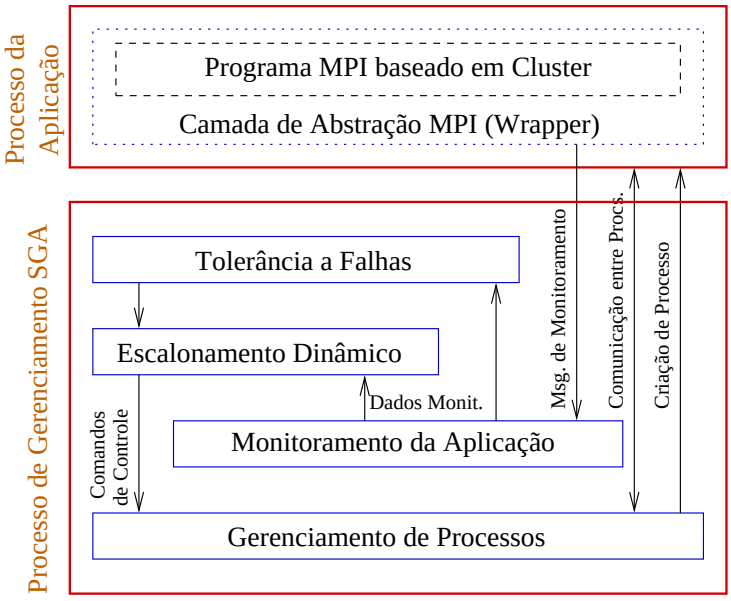


Figura 3.2: Estrutura em camadas do SGA EasyGrid.

3.1.1 Gerenciador Global

A aplicação *system-aware* gerada pelo *Framework* EasyGrid (Smart G-App) inicia a sua execução com a criação de um único processo, o Gerenciador Global do Grid. Apenas o Gerenciador Global tem conhecimento sobre todas as máquinas pertencentes ao Grid, selecionadas pelo usuário para execução da aplicação. Esse processo é o responsável por dar início à construção da hierarquia de gerenciadores, gerenciar mecanismos de monitoramento da aplicação e dos demais processos gerenciadores, identificar falhas nos *sites*, realizar trocas de mensagens entre *sites* da hierarquia e controlar o processo de escalonamento e re-escalonamento de tarefas da aplicação entre *sites* (re-escalonamento global). Mecanismos capazes de tratar falhas nos *sites* estão sendo estudados e serão acrescentados às funcionalidades do Gerenciador Global.

Uma falha no processo GG, do ponto de vista da aplicação, é considerada fatal e tem como conseqüência falhas em todos os processos da hierarquia. Para evitar que uma falha no processo GG comprometa toda a execução da aplicação, duas soluções são possíveis: o sistema que disparou a aplicação deve ser capaz de detectar a falha e recriar o processo GG; ou a aplicação (processo GG) deve ser disparada em uma máquina persistente, isto é, uma máquina confiável, estável e segura. Atualmente, praticamente todos os sistemas de tolerância a falhas propostos necessitam ao menos de uma máquina persistente capaz de armazenar informações sobre a execução da aplicação, para possíveis recuperações em caso de falhas nos recursos ou processos.

Ao ser disparado, o processo GG recebe como dados de entrada fornecidos pelo usuário do Portal EasyGrid informações que permitem a criação da hierarquia de processos gerenciadores e a criação e gerenciamento da aplicação a ser submetida ao Grid. Entre as informações fornecidas pelo Portal EasyGrid estão:

- nome da aplicação a ser executada e parâmetros associados a ela (o SGA e a aplicação são executáveis separados);
- um arquivo com uma lista de máquinas às quais o usuário tem acesso e que podem ser utilizadas para execução da aplicação;
- definição das máquinas onde processos gerenciadores específicos devem ser disparados, ou seja, máquinas onde serão criados os Gerenciadores Locais das Máquinas e os Gerenciadores dos Sites. A escolha das máquinas, assim como a atribuição de processos gerenciadores a estas máquinas será feita pelo Portal EasyGrid através do *Static Grid Scheduler*, apresentado na Seção 2.4.12 do Capítulo 2;
- um arquivo com o escalonamento estático produzido tipicamente por algum algoritmo de escalonamento. Esse escalonamento define uma prévia alocação dos processos da aplicação e é obtido na fase inicial de geração da aplicação *system-aware* pelo *Framework* EasyGrid, como apresentado na Seção 2.4.12 do Capítulo 2;
- número máximo de processos da aplicação que podem estar executando ao mesmo tempo na máquina em que foi disparado o GG. Para cada processo gerenciador pode ser atribuído um valor diferente para o número máximo de processos em execução;
- tipo de política de escalonamento local adotada pelo GG durante a execução da aplicação para escolha da próxima tarefa a ser disparada. Essa política só é utilizada pelo GG caso sejam atribuídas tarefas da aplicação à máquina gerenciada por ele, e este assuma as funcionalidades de um processo GM (As duas políticas existentes na versão atual serão explicadas com detalhes na Seção 3.6);
- grafo (GAD) que represente as dependências entre as tarefas da aplicação e os custos computacionais e de comunicação associados a elas;

Com base no seu conhecimento sobre a organização dos recursos no Grid, o Gerenciador Global inicia a criação da hierarquia de processos gerenciadores disparando dinamicamente em cada *site* um processo Gerenciador do Site (GS), responsável por gerenciar a execução de tarefas nas máquinas do *site* ao qual pertence. As funcionalidades e informações necessárias para execução de um processo GS são apresentadas na Subseção 3.1.2.

3.1.2 Gerenciador do Site

Gerenciadores dos Sites são criados dinamicamente pelo Gerenciador Global a partir de arquivos de esquema definidos pelo Portal EasyGrid. Este arquivo, como apresentado na Subseção 2.3.4 do Capítulo 2, define em que máquina de cada *site* deve ser disparado o processo GS e os parâmetros que estes devem receber ao serem criados. Os parâmetros recebidos pelo GS através do arquivo de esquema são:

- nome da aplicação do usuário e parâmetros associados a ela;
- número máximo de processos da aplicação que podem estar executando ao mesmo tempo na máquina em que foi disparado o GS;
- tipo de política de escalonamento local adotada pelo processo GS durante a execução da aplicação para escolha da próxima tarefa a ser disparada. A política de escalonamento local é utilizada pelo processo GS quando este assume as funcionalidades de um processo GM e executa tarefas da aplicação;
- grafo (GAD) que representa as dependências entre as tarefas da aplicação e os custos computacionais e de comunicação associados a elas;
- arquivos de esquema contendo informações sobre as máquinas do *site* selecionadas pelo usuário do Portal EasyGrid para execução da aplicação (máquinas em que serão disparados processos GM);

Partindo de informações recebidas no momento de sua criação, através de arquivos de esquema, cada GS identifica as máquinas (*hosts*) do *site* local que farão parte da hierarquia, sendo capaz de disparar processos gerenciadores, chamados de Gerenciadores Locais das Máquinas (GM), em cada um desses *hosts*. O processo GS, além de criar os GM's nas máquinas do *site*, é também responsável pelo seu gerenciamento, sendo um intermediário entre o GG e as demais máquinas do *site* gerenciadas por processos GM. Por cada GS são redirecionadas mensagens da aplicação, mensagens de monitoramento, mensagens referentes à identificação de falhas em máquinas do *site* e diversas outras mensagens de gerenciamento do *site* e de todo o Grid. Entre essas mensagens de gerenciamento estão as mensagens específicas do SGA EasyGrid, envolvidas no processo de escalonamento dinâmico de tarefas da aplicação.

3.1.3 Gerenciador Local da Máquina

Os processos GM's são responsáveis apenas pelo gerenciamento de tarefas da aplicação atribuídas à máquina local, não exercendo nenhuma influência direta em relação às demais máquinas envolvidas na execução. Ao serem disparados, os GM's recebem dos GS's, através de arquivos de esquemas, os seguintes parâmetros:

- nome da aplicação a ser executada e parâmetros associados a ela;
- número máximo de processos da aplicação que podem estar executando ao mesmo tempo na máquina em que foi disparado o GM;
- tipo de política de escalonamento local adotada pelo GM durante a execução da aplicação para escolha da próxima tarefa a ser disparada;

Um processo Gerenciador Local da Máquina pode ser disparado mesmo quando nenhum processo da aplicação é atribuído pelo escalonamento estático inicial à máquina gerenciada por ele. O comando *lamgrow* da biblioteca LAM/MPI, responsável pela adição de novos processadores a um ambiente LAM já criado, ainda não é suportado com o *toolkit* Globus. Logo, novos processadores não podem ser incluídos ao ambiente após a aplicação ter sido iniciada.

Uma visão da construção da hierarquia de gerenciadores (com 3 níveis de processos gerenciadores) em um Grid Computacional pode ser observado no exemplo de uma única aplicação ilustrado na Figura 3.3, onde é possível observar a existência de um processo GG executando em uma das máquinas do *site* 1. O processo GG se comunica diretamente com os GS's disparados em cada *site*, sendo estes os responsáveis por gerenciar as máquinas do *site* através de processos GM's criados em cada uma das máquinas.

3.2 Processos da Aplicação

Conforme citado na Subseção 3.1.1, o Gerenciador Global recebe através do Portal EasyGrid um arquivo contendo um escalonamento estático (tipicamente gerado por um algoritmo de escalonamento), enviando essas informações, através de mensagens MPI, para os Gerenciadores dos Sites que irão repassá-las para os Gerenciadores Locais das Máquinas. A partir do escalonamento recebido, cada processo gerenciador é capaz de identificar que tarefas da aplicação devem ser executadas no *host* gerenciado por ele. Apesar de não ser obrigatório, o escalonamento estático é fundamental para que seja feita uma distribuição

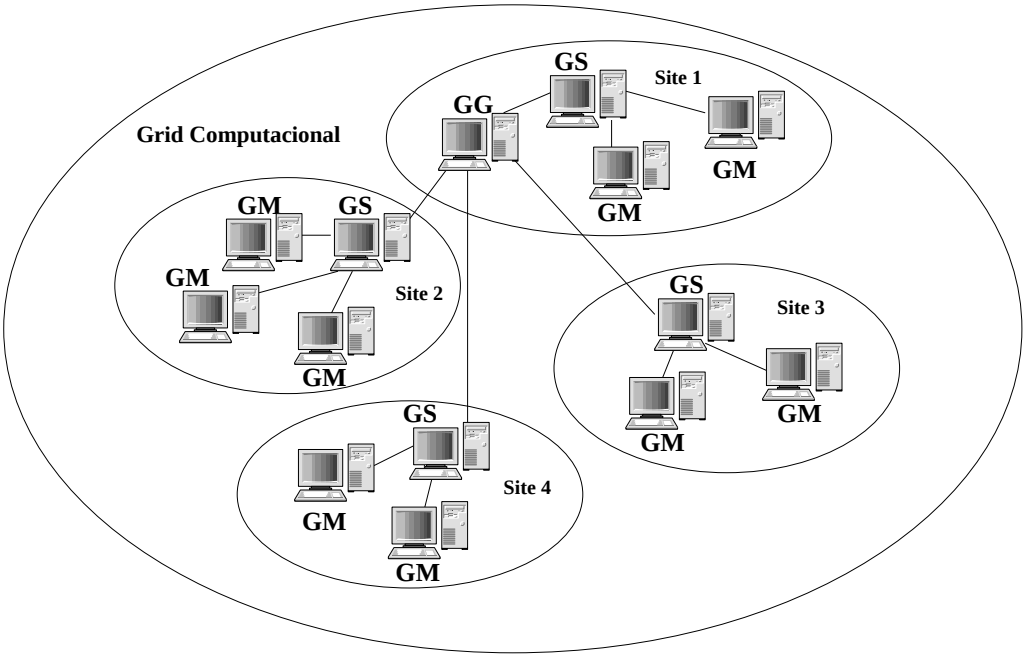


Figura 3.3: Criação da hierarquia de gerenciadores em um Grid Computacional.

inicial adequada de tarefas entre os recursos do Grid selecionados pelo usuário para execução da aplicação.

Inicialmente, todos os processos atribuídos aos *hosts* que não possuem dependência em relação a nenhuma outra tarefa da aplicação são inseridos em uma lista de tarefas prontas para execução. Porém, tarefas que possuem dependência em relação a alguma outra tarefa da aplicação não podem ser consideradas como tarefas prontas, sendo então inseridas em uma lista de tarefas pendentes.

Seguindo o modelo *dataflow*, cada tarefa atribuída ao *host* é criada dinamicamente pelos processos gerenciadores, desde que todos os dados de entrada necessários à sua execução estejam disponíveis. A decisão de só criar uma tarefa após todas as suas entradas de dados estarem prontas foi motivada pela definição adotada de tarefa. Considera-se que cada tarefa é formada pelo recebimento de dados de entrada, seguido da execução de determinadas operações e conseqüente envio de dados. Logo, para que uma tarefa possa realizar processamento, é necessário que todos os seus dados de entrada estejam disponíveis, motivando o atraso da sua criação por parte dos processos gerenciadores. Neste trabalho, uma tarefa é caracterizada como sendo um processo.

Além de respeitar as precedências entre as tarefas da aplicação, através do controle do número de mensagens iniciais (dados de entrada) necessárias à execução (criação) de um processo da aplicação, com a qualidade de serviço dos recursos em mente, faz-se necessária a adoção de políticas que controlem o número de processos em execução

concorrente em uma máquina do Grid. A falta de controle sobre o número de processos prontos disparados pode levar a máquina local a uma sobrecarga, uma vez que todos os processos prontos terão sido imediatamente criados, o que pode refletir em uma queda significativa no desempenho da aplicação. Além de causar sobrecarga, a falta de restrições ao se disparar processos prontos afeta diretamente as políticas de re-escalonamento de tarefas, ficando o processo de redistribuição de tarefas limitado às tarefas presentes na lista de pendentes. O problema da redistribuição de tarefas será explicado em detalhes na Seção 3.6.

A criação dinâmica apresenta inúmeras vantagens em relação à criação estática tradicional do MPI, entre elas o controle do número de processos concorrentes em execução em uma máquina, a facilidade do re-escalonamento dinâmico de tarefas que ainda não foram disparadas, a possibilidade de recriar um processo em caso de falha e finalmente, a escalabilidade e eficiência na execução de aplicações em grande escala, como será comprovado no Capítulo 4.

Na Seção 3.3, será apresentada a função dos processos gerenciadores na troca de mensagens entre tarefas da aplicação. Na Seçao 3.4, serão descritas as duas abordagens propostas do SGA EasyGrid na criação de processos gerenciadores e os modelos de comunicação adotados por elas.

3.3 Mecanismos de Comunicação entre Processos da Aplicação

Todas as tarefas da aplicação são disparadas de maneira independente das demais através de uma chamada ao comando MPI_Comm_spawn(), presente na versão LAM da biblioteca MPI. As tarefas são criadas como processos únicos com um mesmo identificador (*rank* 0), sendo diferenciadas pelos processos gerenciadores apenas pelo seu comunicador. Por não ser possível a comunicação direta entre esses processos, toda troca de mensagens entre tarefas da aplicação é realizada através dos processos da hierarquia de gerenciadores (GM, GS e GG). Para permitir o redirecionamento das mensagens da aplicação pelos processos gerenciadores, foi desenvolvida uma biblioteca denominada MEGmpi.h, que intercepta chamadas de envio e recebimento da aplicação substituindo-as por chamadas de envio e recebimento redefinidas pela biblioteca. A biblioteca MEGmpi.h funciona como uma camada de abstração (*wrapper*) que estende a capacidade das funções já existentes na biblioteca MPI. As funcionalidades oferecidas pela biblioteca MEGmpi.h são acrescen-

tadas à aplicação do usuário em tempo de compilação, sem que isso envolva alterações no código fonte.

Em uma chamada à função MPI_Send() da biblioteca MPI, o usuário precisa especificar seis parâmetros: a mensagem real a ser enviada, o tamanho dessa mensagem, o tipo de dados utilizado, o identificador da tarefa destino (*rank*), o identificador da mensagem (*tag*) e o comunicador ao qual pertencem os processos origem e destino. Quando uma tarefa da aplicação executa um comando de envio da biblioteca MPI (MPI_Send()), o comando é substituído pela chamada à função MEGMPI_Send() da biblioteca MEGmpi.h. Esta função acrescenta à mensagem original informações como: identificador da tarefa origem (*rank* origem), identificador da tarefa destino (*rank* destino), identificador da mensagem (*tag*) e quantidade de *bytes* da mensagem original. A Figura 3.4 apresenta a mensagem gerada pela chamada à função MEGMPI_Send() a partir da mensagem original da aplicação.

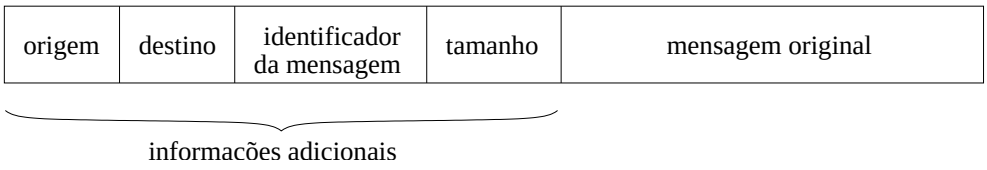


Figura 3.4: Mensagem gerada pelo MEGMPI_Send() após concatenar informações à mensagem original da aplicação.

Após concatenar informações à mensagem original da aplicação, o MEGMPI_Send() altera os parâmetros da função MPI_Send() referentes ao identificador do processo destino (*rank*) e ao comunicador, forçando o redirecionamento da mensagem para o processo gerenciador em execução na máquina local, que é o responsável por encaminhá-la para que esta alcance a tarefa destino.

Os processos gerenciadores precisam estar preparados para receber qualquer tipo de mensagem, em qualquer momento e de qualquer processo que tenha alguma ligação direta com eles como, por exemplo, tarefas da aplicação em execução no *host* local e gerenciadores dos níveis imediatamente acima e imediatamente abaixo do dele, caso estes existam. Para que isso seja possível, o papel das informações concatenadas à mensagem original é fundamental nos redirecionamentos de mensagens enviadas por uma tarefa da aplicação. Através dessas informações, os processos gerenciadores são capazes de identificar origem e destino da mensagem a ser redirecionada, além da quantidade de bytes envolvidos na comunicação, para que possa ser alocada uma área de memória suficiente para receber a mensagem original.

Sabendo que uma tarefa da aplicação só poderá ser disparada quando uma determinada condição for verdadeira (por exemplo, todas as entradas de dados necessárias à sua execução estiverem prontas), o processo gerenciador, ao receber uma mensagem da aplicação a ser redirecionada para uma tarefa destino atribuída à máquina local, verifica, antes de repassá-la, se a tarefa foi ou não disparada. Se a tarefa ainda não foi disparada, a mensagem é armazenada em uma lista de mensagens pendentes para ser enviada à tarefa destino somente após a sua criação. Caso contrário, a mensagem é imediatamente redirecionada para a tarefa destino.

Na chamada à função MPI_Recv() da biblioteca MPI, o usuário precisa especificar sete parâmetros: o *buffer* para armazenamento da mensagem, o tamanho do *buffer*, o tipo de dado esperado, o identificador do processo origem (*rank*), o identificador da mensagem (*tag*), o comunicador ao qual pertencem os processos origem e destino, e o *status* relacionado ao recebimento da mensagem. As chamadas a essas funções de recebimento dentro de uma tarefa da aplicação são substituídas por chamadas às funções MEGMPI_Recv() da biblioteca MEGmpi.h. Mensagens recebidas pelos processos gerenciadores podem ser de diversos tipos e tamanhos, e podem ter origens diversas. Ao receber uma mensagem da aplicação, a função MEGMPI_Recv() verifica se a mensagem repassada pelo processo gerenciador é a mesma que está sendo esperada pela tarefa destino naquele momento. Se não for, a mensagem é armazenada em uma lista de recebimentos pendentes até que o seu recebimento seja solicitado. Caso contrário, o recebimento da mensagem é realizado como se um simples comando MPI_Recv() tivesse sido executado.

O identificador atribuído a um processo pelo MPI no momento da sua criação é denominado *rank*. O identificador definido pelo *middleware* para cada processo gerenciador e processo da aplicação é denominado *rank do middleware*. Como citado no início desta seção, a criação individual de tarefas da aplicação atribui a todos os processos disparados pelo MPI_Comm_spawn() o mesmo *rank*. Para preservar o identificador original da tarefa (*rank do middleware*) definido pelo desenvolvedor da aplicação, sem que para isso fosse preciso realizar alterações no código do usuário, foram criadas funções auxiliares na biblioteca MEGmpi.h.

Através da função MEGMPI_Init(), executada a cada chamada da aplicação à função MPI_Init(), é possível obter o identificador original da tarefa (*rank do middleware*) e o número total de tarefas da aplicação. Esses dados são passados pelos processos gerenciadores para a aplicação durante a sua criação. O *rank do middleware* é retornado para a aplicação através de uma chamada a função MPI_Comm_rank(), que é substi-

tuída pela função MEGMPI_Comm_rank() da biblioteca MEGmpi.h. O número total de tarefas a serem executadas é passado para aplicação através de uma chamada a função MPI_Comm_size(), substituída pela função MEGMPI_Comm_size() também da biblioteca MEGmpi.h.

Com base no que foi colocado nesta seção, pode-se concluir que a biblioteca MEGmpi.h permite que mensagens da aplicação sejam redirecionadas através da hierarquia de gerenciadores, ficando a cargo dos processos gerenciadores a entrega segura das mensagens às tarefas destino, independentemente da máquina em que elas estejam executando. Isso é fundamental no sistema gerenciador da aplicação proposto, onde a localização das tarefas pode variar ao longo da execução da aplicação.

Além das funções apresentadas nesta seção, outras funções da biblioteca MPI foram reescritas na biblioteca MEGmpi.h, entre elas as funções MPI_Probe(), MPI_Iprobe(), MPI_Isend() e MPI_Irecv(). A função MPI_Probe() espera até que uma determinada mensagem esteja disponível na máquina destino. Esta função apenas verifica a chegada da mensagem, sendo o recebimento desempenhado por funções como MPI_Recv() e MPI_Irecv(). As funções MPI_Iprobe(), MPI_Isend() e MPI_Irecv() são versões não bloqueantes das funções MPI_Probe(), MPI_Send() e MPI_Recv(), respectivamente. Ou seja, os processos podem continuar a executar blocos de operações seguintes às chamadas a essas funções, mesmo que essas funções não tenham sido completadas. Tais funções foram reescritas de forma a serem capazes de identificar mensagens da aplicação redirecionadas pelos processos gerenciadores.

3.4 Criação de Processos Gerenciadores na Arquitetura Proposta

Toda comunicação entre os processos gerenciadores (GG, GS e GM) e tarefas da aplicação segue a organização hierárquica proposta. Assim, todos os envios e recebimentos de mensagens entre as tarefas são intermediados pelos processos gerenciadores, não existindo comunicação direta entre as tarefas da aplicação.

Estão sendo propostas neste trabalho duas possíveis abordagens para a criação de processos gerenciadores do SGA EasyGrid: a abordagem coletiva e a abordagem individual. Na abordagem coletiva, processos gerenciadores de um mesmo nível podem se comunicar diretamente, sem a necessidade de repassar a mensagem para um processo superior na hierarquia de gerenciadores. Por exemplo, Gerenciadores dos Sites podem

se comunicar diretamente sem precisar que a mensagem passe pelo Gerenciador Global. Sendo assim, toda comunicação entre os *sites* pode ser feita diretamente sem a necessidade de interferência do Gerenciador Global.

O mesmo não ocorre na abordagem individual, onde processos de um mesmo nível hierárquico não são capazes de se comunicar diretamente, sendo preciso a intervenção de um processo gerenciador do nível imediatamente acima para que a comunicação seja realizada. Nesse caso, para os Gerenciadores dos Sites se comunicarem, por exemplo, é preciso que a mensagem seja enviada do GS origem ao GG, para que este possa repassar a mensagem para o GS destino. Logo, na abordagem individual a comunicação entre *sites* é feita de forma indireta, através do GG.

Apesar da comunicação extra apresentada pela segunda abordagem em relação à primeira, sua utilização foi motivada pela facilidade que esta apresenta no que se refere a utilização de mecanismos eficientes de tolerância a falhas. Isso ficará mais claro nas subseções seguintes, onde serão detalhadas essas duas propostas de projeto.

3.4.1 Criação Coletiva de Processos Gerenciadores

A primeira abordagem do SGA EasyGrid tem como principal característica a comunicação direta entre processos gerenciadores pertencentes a um mesmo nível hierárquico. Isso é possível devido à criação coletiva de gerenciadores de um mesmo nível, ou seja, quando um gerenciador de um determinado nível na hierarquia é criado, ele dinamicamente dispara os processos gerenciadores do nível imediatamente abaixo de uma única vez. A criação coletiva é implementada através de uma única chamada à função MPI_Comm_spawn(), função disponível na versão LAM da biblioteca MPI que possibilita a criação dinâmica de processos MPI. As versões do MPI e suas funções utilizadas na implementação da ferramenta proposta foram apresentadas no Capítulo 2.

É possível observar o comportamento do Gerenciador Global na Figura 3.5. O GG, após ter sido disparado pelo usuário, cria dinamicamente e de uma única vez os processos gerenciadores de cada *site* (GS). Como a criação dos processos Gerenciadores dos Sites é feita através de uma única execução do comando MPI_Comm_spawn(), segundo as definições da biblioteca MPI, todos os Gerenciadores dos Sites pertencem a um mesmo grupo, têm um comunicador em comum (MPI_COMM_WORLD, criado automaticamente pelo MPI) e possuem identificadores (*ranks*) distintos. O comunicador comum e os identificadores distintos são os responsáveis por garantir a comunicação direta entre os Gerenciadores dos Sites. Sendo assim, por exemplo, o GS que possui *rank* 0 pode enviar

mensagens diretamente para os GS's de *rank* 1 e 2. O mesmo é possível a partir dos GS's de *rank* 1 e 2.

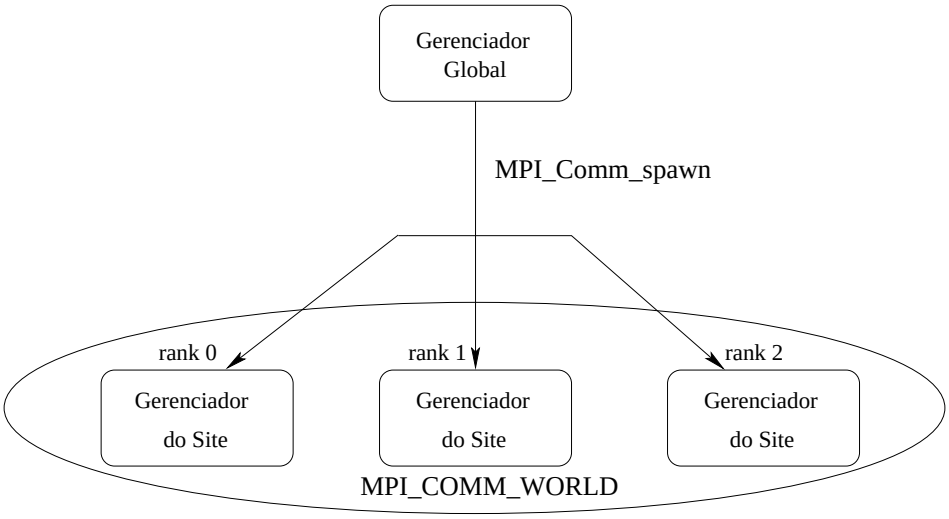


Figura 3.5: Criação coletiva dos Gerenciadores dos Sites pelo Gerenciador Global.

A existência de um comunicador único entre os processos gerenciadores de um mesmo nível hierárquico dificulta o desenvolvimento de mecanismos eficientes de tolerânica a falhas. Como apresentado na Seção 2.3.3 do Capítulo 2, em caso de operações coletivas uma falha envolvendo qualquer um dos processos desse comunicador reflete nos demais processos que fazem parte do mesmo comunicador. Esse tipo de problema pode ser evitado se cada processo for criado com um comunicador individual, sendo toda comunicação entre gerenciadores realizada através de intercomunicadores criados para cada par de processos dentro da hierarquia proposta.

Além de permitir que falhas se propaguem entre processos de um mesmo comunicador, a existência de um único comunicador torna ainda mais complexos os mecanismos de recuperação de falhas. Supondo-se que um dos Gerenciadores dos Sites tenha terminado inesperadamente a sua execução e que, após identificada a falha, o Gerenciador Global tenha decidido disparar um novo processo gerenciador para substituir o processo que falhou, para que este novo processo possa continuar a se comunicar diretamente com os demais processos do mesmo nível hierárquico, um novo comunicador (intracomunicador) deve ser criado englobando o novo processo GS e os processos (GS e GG) pertencentes ao intercomunicador já existente. A função MPI capaz de criar um novo intracomunicador a partir de um intercomunicador é conhecida como MPI_Intercomm_merge(). Essa função, por ser uma operação coletiva, deve ser chamada por todos os processos envolvidos, forçando um sincronismo entre eles, o que em ambientes como os Grids pode levar a uma queda significativa no desempenho da aplicação.

Observando a hierarquia de gerenciadores proposta, é possível identificar três casos distintos envolvendo processos gerenciadores de diferentes níveis no processo de comunicação. No primeiro caso, é realizada a troca de mensagens entre tarefas da aplicação designadas a executar em uma mesma máquina. No segundo caso, é considerada a situação onde duas tarefas atribuídas a máquinas distintas de um mesmo *site* estão trocando mensagens. E por último, no terceiro caso, é abordada a situação em que tarefas escalonadas são atribuídas a *sites* diferentes, sendo este último caso o que envolve maior custo de comunicação, uma vez que no pior dos casos a mensagem inicial é obrigada a percorrer dois níveis na hierarquia.

Na Figura 3.6 é apresentado o escalonamento estático inicial de dez tarefas da aplicação (t0, t1, t2, t3, t4, t5, t6, t7, t8 e t9) entre as máquinas dos níveis 0, 1 e 2 da hierarquia de gerenciadores. A hierarquia de gerenciadores ilustrada nesta figura é composta de dois processos Gerenciadores dos Sites e um Gerenciador Global. Cada *site* contém duas máquinas executando um processo Gerenciador Local da Máquina (GM).

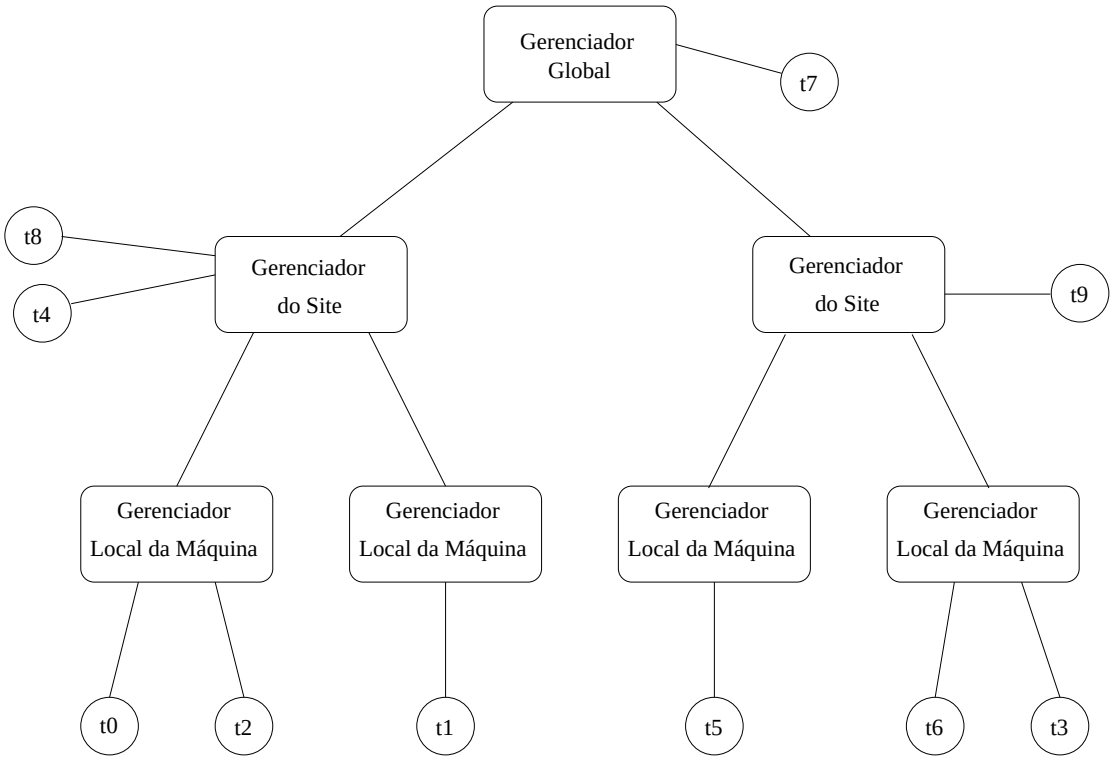


Figura 3.6: Escalonamento de uma aplicação entre as máquinas do Grid.

Para que tarefas atribuídas a uma mesma máquina possam se comunicar, é necessário apenas a intervenção do processo gerenciador em execução na máquina local. As tarefas t0 e t2, por exemplo, foram atribuídas a uma mesma máquina de nível 2, logo a comunicação entre elas é feita através do GM em execução na máquina local. A tarefa t0 envia uma

mensagem para tarefa t2, sendo esta interceptada pelo GM que, após identificar a localização da tarefa destino, repassa a mensagem enviada por t0. No exemplo apresentado, a tarefa t2 já havia sido disparada na máquina local, podendo a mensagem ser repassada diretamente pelo GM. Caso a tarefa t2 ainda não tivesse sido criada, a mensagem seria armazenada pelo GM na lista de mensagens pendentes da tarefa t2.

As tarefas t0 e t1, por exemplo, foram atribuídas a máquinas diferentes de um mesmo *site*. A mensagem enviada de t0 para t1 é interceptada pelo GM da máquina de origem, que verifica a máquina destino da tarefa t1. Inicialmente, o GM faz uma busca na lista de tarefas da aplicação verificando se a tarefa destino foi atribuída à máquina local. Caso a tarefa destino não seja identificada como pertencente à máquina local, o GM realiza uma nova busca na lista de tarefas verificando se a tarefa foi atribuída a alguma outra máquina do *site*. Ao descobrir que a tarefa destino foi atribuída a uma máquina do mesmo *site*, o GM de origem repassa a mensagem diretamente para o GM da máquina destino. Após o recebimento, o GM destino a envia para a tarefa destino t1.

As tarefas da aplicação t0 e t3, por exemplo, pertencem a *sites* diferentes e precisam trocar mensagens. A mensagem enviada de t0 para t3 é interceptada pelo GM em execução na máquina origem, que após identificar que a tarefa t3 não foi atribuída a nenhuma das máquinas do *site* local, a redireciona para o Gerenciador do Site. Ao receber a mensagem, o GS de origem identifica a que *site* foi atribuída a tarefa destino, para então redirecionar a mensagem para o Gerenciador do Site destino. A comunicação entre o GS do *site* origem e o GS do *site* destino é direta, devido à criação coletiva de processos gerenciadores de um mesmo nível hierárquico. Após receber a mensagem, o GS destino a repassa para o Gerenciador Local da Máquina que é o responsável por fazer a mensagem alcançar a tarefa destino (t3).

Para que os processos gerenciadores possam ser capazes de identificar a que máquina foi atribuída a tarefa destino envolvida na comunicação, é necessário que cada um desses processos possua uma lista atualizada das alocações de todas as tarefas da aplicação entre as máquinas do Grid. Essa lista é criada pelo Gerenciador Global com base no escalonamento estático inicial e repassada para as demais máquinas da hierarquia. Entretanto, a atualização dessas listas em todas as máquinas da hierarquia se torna uma operação cada vez mais complexa e custosa, à medida que o número de máquinas envolvidas na execução da aplicação cresce e que novos eventos de escalonamento dinâmico de tarefas são realizados ao longo da execução. A atualização dessas listas envolve um algoritmo de propagação de informação entre a hierarquia de gerenciadores.

3.4.2 Criação Individual de Processos Gerenciadores

Nesta seção será apresentada a segunda abordagem para a criação dos processos gerenciadores no sistema proposto. Entre as principais características dessa abordagem está a forma como as trocas de mensagens são intermediadas pelos gerenciadores.

Na primeira versão, apresentada na Seção 3.4.1, os gerenciadores de um mesmo nível hierárquico são criados ao mesmo tempo, tendo cada um deles um identificador único (*rank*) e um só comunicador, o que possibilita a comunicação direta entre eles. Nessa segunda abordagem do SGA, os processos gerenciadores são criados individualmente, ou seja, cada processo é criado a partir de uma chamada à função MPI_Comm_spawn(). Cada processo disparado recebe o mesmo identificador (*rank* 0) e um comunicador distinto dos demais, o que não permite a comunicação direta entre eles. Sendo assim, toda comunicação entre processos de um mesmo nível hierárquico precisa ser feita através de gerenciadores do nível imediatamente acima. Por exemplo, para que dois Gerenciadores dos Sites possam se comunicar, a troca de mensagens deve ser feita por intermédio do Gerenciador Global.

A principal motivação para o desenvolvimento dessa versão do SGA está relacionada diretamente a uma maior facilidade apresentada por ela no desenvolvimento de mecanismos eficientes de tolerância a falhas. A existência de um intercomunicador entre cada par de processos gerenciadores dentro da hierarquia garante que, em caso de falha de um dos processos, os demais processos do mesmo nível hierárquico não serão afetados por possíveis chamadas a funções coletivas. Além de evitar a propagação de falhas, outra vantagem de se usar comunicadores individuais está no mecanismo de recuperação de um processo que tenha terminado inesperadamente. Após identificada a falha, um novo processo pode ser criado dinamicamente sem haver a necessidade de execução da função MPI_Intercomm_merge() da biblioteca MPI para gerar um novo intracomunicador entre os processos do mesmo nível hierárquico. Ou seja, um novo processo pode ser criado para substituir o processo que falhou e nenhum sincronismo é imposto aos demais processos da hierarquia.

Na Figura 3.7 é apresentado um exemplo da criação individual de Gerenciadores dos Sites pelo Gerenciador Global, considerando a existência de três *sites* no ambiente Grid. Para disparar cada Gerenciador do Site, o GG realiza uma chamada à função MPI_Comm_spawn(), o que faz com que cada processo criado tenha o identificador zero (*rank* 0) e um comunicador distinto (comm1, comm2 e comm3). A existência de comunicadores diferentes obriga que a comunicação entre os três *sites* seja feita com a

intervenção do Gerenciador Global.

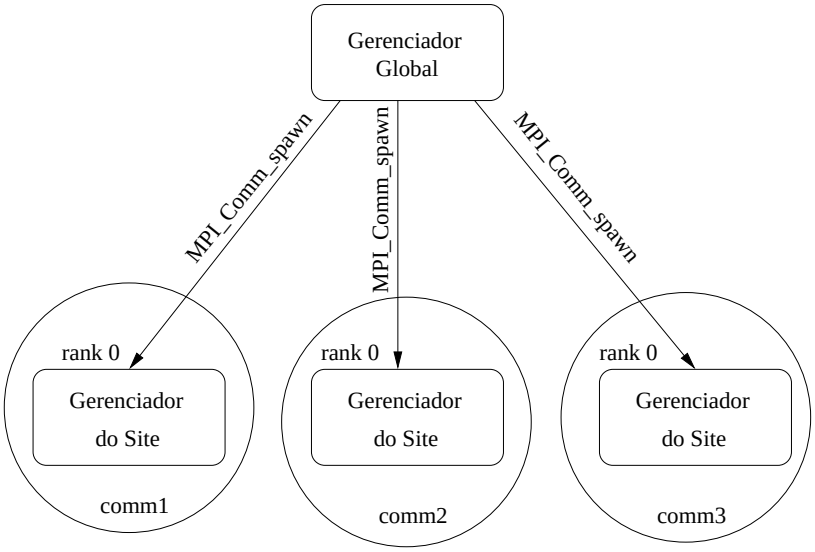


Figura 3.7: Criação individual dos Gerenciadores dos Sites pelo Gerenciador Global.

Assim como na abordagem coletiva, cada tarefa da aplicação é disparada individualmente pelo processo gerenciador através da chamada à função MPI_Comm_spawn(), sendo considerada como um processo independente, incapaz de se comunicar diretamente com as demais tarefas da aplicação. A única comunicação possível é a que é realizada com o processo gerenciador em execução na máquina local. A forma como as tarefas da aplicação são criadas independe da maneira como os processos gerenciadores foram disparados (de forma individual ou coletiva). Sendo assim, tarefas atribuídas a uma mesma máquina na abordagem individual comunicam-se da mesma forma que na abordagem coletiva, apresentada na Seção 3.4.1.

Na Figura 3.6, apresentada na Seção 3.4.1, as tarefas t0 e t2 têm suas mensagens redirecionadas pelo processo Gerenciador Local da Máquina tanto na abordagem coletiva quanto individual. Quando a troca de mensagens envolve tarefas atribuídas a máquinas distintas, o redirecionamento das mensagens pelos processos gerenciadores na abordagem individual é diferente da abordagem coletiva.

Considerando ainda o exemplo da Figura 3.6, a tarefa t0, ao enviar uma mensagem para t1, tem sua mensagem redirecionada para o Gerenciador Local da Máquina origem. Ao receber a mensagem, o GM verifica se a tarefa destino (t1) pertence à lista de tarefas escalonadas na máquina local. Sabendo que a tarefa destino não foi atribuída à máquina local, o GM redireciona a mensagem para o Gerenciador do Site local que é o responsável por identificar e repassar a mensagem para o GM da máquina destino. Ao receber a mensagem, o GM da máquina destino tem a responsabilidade de enviar a mensagem para

a tarefa destino t1. Ao contrário da abordagem coletiva, a comunicação entre máquinas distintas de um mesmo *site* só é possível através do Gerenciador do Site local.

A troca de mensagens entre as tarefas t0 e t3, envolve comunicação entre *sites*. A mensagem enviada por t0 é redirecionada para o Gerenciador Local da Máquina que, ao identificar que a tarefa destino não pertence à máquina local, repassa-a para o GS. Como a máquina destino não pertence ao mesmo *site* que a máquina de origem da mensagem, o GS deve repassá-la para o GG, que é o único processo da hierarquia de gerenciadores capaz de fazer a comunicação entre *sites* distintos. Ao receber a mensagem e verificar que a tarefa destino não pertence à máquina local, o GG a envia para o Gerenciador do Site destino, que é responsável por repassá-la para o GM destino. Este por sua vez, ao receber a mensagem, é capaz de redirecioná-la para a tarefa t3. Assim, a mensagem precisou passar por todos os níveis da hierarquia para alcançar a tarefa destino.

Ao contrário da abordagem coletiva, na abordagem individual todos os processos gerenciadores da hierarquia não precisam conhecer toda a lista de tarefas da aplicação com suas respectivas alocações. O Gerenciador da Máquina possui apenas a lista de tarefas atribuídas à máquina local, uma vez que comunicações envolvendo máquinas distintas do mesmo *site* são intermediadas pelo Gerenciador do Site. Logo, ao receber uma mensagem que não pertence a nenhuma das tarefas atribuídas à máquina local, o Gerenciador da Máquina irá repassar a mensagem para o Gerenciador do Site local. O Gerenciador do Site possui a lista de todas as tarefas da aplicação atribuídas a cada máquina pertencente ao *site* gerenciado por ele. Já o Gerenciador Global, por intermediar todo o tipo de comunicação entre *sites*, possui uma lista atualizada com todas as tarefas da aplicação e suas alocações entre as máquinas do Grid. Na abordagem individual, a atualização da lista de tarefas de cada processo gerenciador é uma operação mais simples e menos custosa que na abordagem coletiva, uma vez que a atualização é feita automaticamente durante o processo de redistribuição de tarefas, englobando apenas as máquinas envolvidas na redistribuição e o processo GG que recebe informações dos GS's.

Devido às facilidades apresentadas pela abordagem individual na criação da hierarquia de processos gerenciadores, todas as funcionalidades do SGA descritas nas seções seguintes terão como base esta abordagem. Na Seção 3.5 será apresentado o mecanismo de monitoramento adotado pela versão atual do SGA EasyGrid. Nas Seções 3.6 e 3.7 serão tratados aspectos de escalonamento dinâmico e tolerância a falhas também associados ao SGA.

3.5 Monitoramento Hierárquico

Muitas atividades que ocorrem de forma paralela à execução de uma aplicação, tais como escalonamento de tarefas, técnicas para recuperação de falhas, gerenciamento de recursos e análise de desempenho, necessitam de informações atualizadas, para que sejam tomadas decisões apropriadas. O monitoramento tem sua importância focada na obtenção de informações relativas às características e às condições do sistema, assim como às informações ligadas ao comportamento da execução das aplicações [45].

Para o escalonamento dinâmico, os dados coletados são fundamentais tanto para ativação do evento de re-escalonamento como para o novo escalonamento gerado. A partir dessas informações relacionadas à aplicação, tais como tempos de execução e custo de comunicação entre as tarefas, um algoritmo de escalonamento dinâmico é capaz de redistribuir tarefas da aplicação que ainda não foram executadas, entre os recursos adequados.

Conforme apresentado no Capítulo 2, neste trabalho são abordados os aspectos de escalonamento dinâmico referentes à distribuição de tarefas da aplicação que ainda não foram criadas, em vez da migração de tarefas em execução.

Nas subseções seguintes, serão apresentadas duas versões da ferramenta de monitoramento. A primeira é a versão original apresentada em [45]. A segunda versão é uma adaptação da primeira à estrutura hierárquica proposta na Seção 3.1.

3.5.1 Monitoramento para Aplicações MPI

Como citado em seções anteriores, existe uma diferença sutil entre as tarefas de uma aplicação paralela e os processos de uma aplicação MPI. Tarefas são compostas por possíveis recebimentos de mensagens, seguindo-se uma etapa de processamento e possíveis envios de mensagens para outras tarefas. É comum em escalonamento caracterizar uma tarefa como sendo um processo; porém, processos de uma aplicação podem ser compostos por várias etapas de processamento e comunicação, caracterizando-se a existência de mais de uma tarefa por processo. Em [45] foi proposta uma ferramenta de monitoramento incorporada ao *framework* EasyGrid. Nessa ferramenta, o monitoramento é realizado por processo, o que a torna mais flexível, uma vez que independe da abordagem adotada pelo programador.

O monitoramento da aplicação é feito através da substituição de funções da biblioteca MPI. Assim como na comunicação entre tarefas da aplicação, apresentada na Subseção 3.3,

para minimizar as alterações na biblioteca de troca de mensagens é introduzida uma camada que aumenta a capacidade das funções já existentes (*wrapper*). Para coletar e armazenar informações, foi criado um processo adicional na aplicação EasyGrid chamado de monitor. Tal processo possui o identificador (*rank*) igual ao número de processos da aplicação disparados. Ou seja, se na aplicação o número de processos é igual a dez, os *ranks* variam de 0 a 9 e o processo monitor terá *rank* 10. Esse processo pode ser disparado em qualquer uma das máquinas disponíveis no ambiente, sendo desconhecido para os demais processos da aplicação.

Sempre que operações como MPI_Init(), MPI_Send(), MPI_Recv() e MPI_Finalize() são chamadas na aplicação do usuário, elas são redefinidas pelo módulo monitor, que além de executar a função padrão, envia uma mensagem de *log* para o processo monitor contendo as seguintes informações:

- identificador (*rank*) da tarefa origem ou destino, dependendo da operação monitorada;
- tipo de operação realizada (inicialização, envio, recebimento, etc.);
- tempo de utilização da CPU no início e no final da função;
- tempo de parede de início e fim da função no processador de origem;
- no caso de operações de recebimento, o tempo de espera da função no processador de origem;

O módulo monitor utiliza um relógio global, para que os tempos de comunicação entre os processos do usuário possam ser obtidos. No início da execução, o processo monitor envia e recebe uma mensagem (método *ping-pong*) para determinar de forma aproximada a diferença entre o seu relógio e os relógios das máquinas a que foram atribuídos cada processo da aplicação. Assim, sempre que o processo monitor receber uma mensagem de *log*, ele irá calcular o tempo global em que a operação supostamente ocorreu.

Com base na estrutura proposta na Seção 3.1, foi remodelada a ferramenta de monitoramento apresentada em [45]. A nova estrutura de monitoramento será apresentada na subseção seguinte.

3.5.2 Monitoramento Hierárquico para Aplicações MPI

Na Seção 3.1 foi proposta uma das abordagens da hierarquia de processos gerenciadores com três níveis: Gerenciador Global (GG), responsável por gerenciar todos os *sites* pertencentes ao Grid Computacional, o Gerenciador do Site (GS), responsável por gerenciar a execução de tarefas da aplicação nas máquinas do *site* local e o Gerenciador Local da Máquina (GM), que controla a criação e execução de tarefas da aplicação na máquina local. Nesta seção, será levada em consideração essa abordagem de 3 níveis do SGA EasyGrid para criação da hierarquia de processos gerenciadores, apresentada na Subseção 3.4.2.

A ferramenta de monitoramento apresentada em [45] não prevê a criação dinâmica de processos e a distribuição hierárquica de processos gerenciadores no ambiente Grid. O primeiro ponto a ser discutido é a criação de um único processo monitor. Essa abordagem não se adapta à hierarquia apresentada, por dois motivos. O primeiro é porque não existe um comunicador único entre todos os processos gerenciadores e as tarefas da aplicação, o que impossibilita a comunicação direta entre eles. Em segundo lugar, o monitoramento tem que fornecer informações para os serviços de cada gerenciador e a funcionalidade desses serviços é diferente em cada um dos níveis da hierarquia. Para contornar tais problemas, é proposto neste trabalho um monitoramento hierárquico.

O monitoramento pode ser feito apenas nos processos da aplicação, ou apenas nos processos gerenciadores ou em ambos, processos gerenciadores e da aplicação. Essa decisão ficará a cargo do usuário e será feita em tempo de compilação. O usuário pode optar por não monitorar nem a aplicação e nem os gerenciadores. Nesse caso, apenas a função MPI_Finalize() dos processos da aplicação é monitorada.

O monitoramento dos processos gerenciadores é importante apenas para depuração e entendimento do comportamento dos processos gerenciadores durante a execução de tarefas de uma aplicação. Já o monitoramento dos processos da aplicação é fundamental para que possam ser obtidas informações necessárias aos mecanismos de escalonamento dinâmico da aplicação e mecanismos de tolerância a falhas.

A Figura 3.8 ilustra a versão original da ferramenta de monitoramento de aplicações MPI. É possível observar a existência de onze processos, sendo dez tarefas da aplicação (t0, t1, t2, t3, t4, t5, t6, t7, t8 e t9) e um processo monitor (M). A criação de um comunicador comum permite a comunicação direta entre os processos da aplicação e o processo monitor.

Na Figura 3.9 é apresentado um exemplo do monitoramento hierárquico sobre as mes-

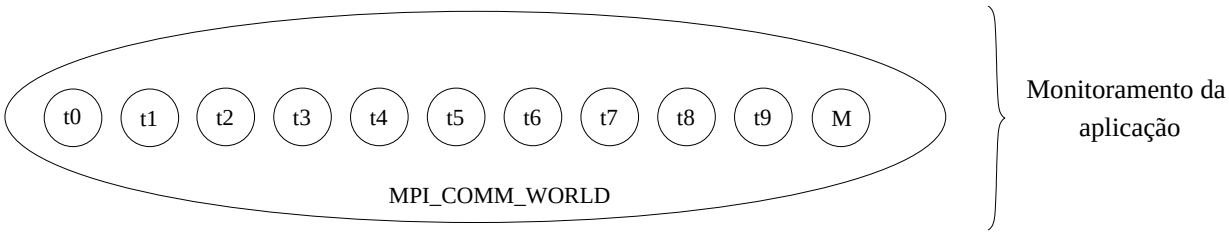


Figura 3.8: Versão original da ferramenta de monitoramento de aplicações MPI.

mas dez tarefas e sobre os processos gerenciadores. Ao contrário do que foi observado na Figura 3.8, não existe um processo monitor, sendo o monitoramento uma das funcionalidades do SGA EasyGrid, presente na segunda camada da arquitetura do SGA, como ilustrada na Figura 3.2. O monitoramento é administrado pelos processos gerenciadores, sendo o GG o responsável por controlar a coleta de informações em todo ambiente Grid. Este monitoramento pode ser realizado apenas sobre as tarefas da aplicação, ou sobre os gerenciadores ou em ambos.

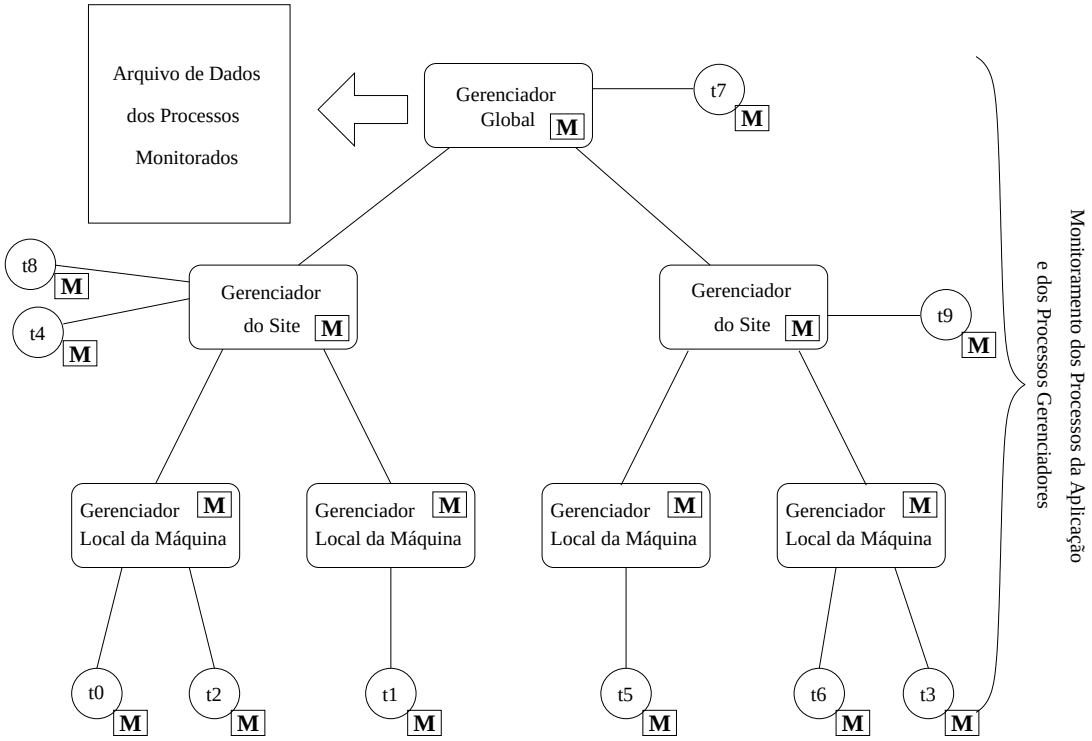


Figura 3.9: Monitoramento hierárquico.

Para a realização do monitoramento hierárquico, o módulo monitor sofreu algumas alterações em relação à sua versão original. Como as demais mensagens trocadas entre os processos pertencentes ao ambiente Grid, as mensagens de monitoramento devem obedecer

à hierarquia proposta. Os Gerenciadores dos Sites funcionam como intermediários entre os Gerenciadores Locais das Máquinas e o Gerenciador Global, assim como processos gerenciadores intermediam as comunicações entre as tarefas da aplicação.

A ativação do monitoramento nos processos gerenciadores e na aplicação é feita através de macros definidas em tempo de compilação. Apenas a função MPI_Finalize() é sempre monitorada, uma vez que os gerenciadores precisam estar cientes da finalização dos processos da aplicação.

O processo GG é responsável por criar e gerenciar estruturas para armazenamento de informações obtidas com o monitoramento de tarefas da aplicação e dos demais processos gerenciadores. O primeiro passo para o monitoramento é o cálculo da diferença de relógio entre a máquina que gerencia o processo de monitoramento (e conseqüentemente armazena as informações obtidas) e as demais máquinas do ambiente Grid. O cálculo do relógio global também deve obedecer à hierarquia apresentada.

O Gerenciador Global, responsável por gerenciar todo o monitoramento, troca mensagens com os Gerenciadores dos Sites, para que possa ser feito o cálculo da latência e da diferença de relógio entre eles, sendo esse cálculo feito pelo GG. O mesmo ocorre entre os processos GS's e os GM's, que trocam mensagens, sendo a latência e a diferença de relógio calculadas por cada Gerenciador do Site. Os valores calculados entre Gerenciadores Locais das Máquinas e os Gerenciadores dos Sites devem ser repassados para o Gerenciador Global, para que esse possa calcular a diferença de relógio entre ele e os Gerenciadores Locais das Máquinas (diferença de relógio entre GM e GG = diferença de relógio entre GM e GS + diferença de relógio entre GS e GG). Esse passo inicial é realizado para qualquer tipo de monitoramento ativado, seja ele o monitoramento da aplicação ou o monitoramento dos processos gerenciadores ou o monitoramento da aplicação e de processos gerenciadores.

Na versão hierárquica, o cálculo da diferença de relógio é disparado de forma diferente da versão original, não envolvendo as tarefas da aplicação. Na versão original, essa troca de mensagens é iniciada dentro da função MRMPI_Init(), função do módulo de monitoramento ativada a cada chamada à função MPI_Init() pelas tarefas da aplicação. Sendo assim, cada tarefa da aplicação precisa calcular a diferença entre o relógio da máquina em que está executando e o relógio do processo monitor. Isso faz com que o mesmo cálculo seja desempenhado por várias tarefas da aplicação, considerando a possibilidade de mais de um processo estar em execução em uma mesma máquina. Na versão hierárquica, a diferença de relógio é calculada pelos processos gerenciadores uma única vez, tornando-se

independente do número de tarefas da aplicação.

É importante ressaltar que em ambientes como os Grids Computacionais, uma noção de tempo global é fundamental não apenas para os serviços de monitoramento, mas também para outros serviços, tais como os de segurança. A sincronização dos relógios pode ser obtida através da utilização de tecnologias como o NTP (Network Time Protocol) [58]. Mesmo com a sincronização de relógios imposta pelos ambientes Grids, para garantir uma alta precisão nos tempo obtidos pelo monitoramento hierárquico, o SGA EasyGrid mantém o cálculo da diferença de relógio entre as máquinas pertencentes à hierarquia. Com relação às funções da biblioteca MPI que foram redefinidas, a principal diferença entre as duas versões é o destino das mensagens de *log*. Na versão original, todas as informações de monitoramento são enviadas para o processo monitor. Na versão hierárquica, o envio das informações coletadas deve obedecer à hierarquia definida. Dados coletados de tarefas da aplicação são enviados para o processo gerenciador em execução na máquina local que, por sua vez, redireciona esses dados recebidos para o processo gerenciador do nível acima, caso exista, até que essas informações cheguem ao Gerenciador Global. Todos os dados coletados do monitoramento dos processos gerenciadores dos níveis 1 e 2 são também enviados para o Gerenciador Global.

Quando os processos gerenciadores não estão sendo monitorados, somente as informações coletadas das tarefas da aplicação pelos processos gerenciadores dos níveis 1 e 2 precisam ser repassadas para o Gerenciador Global do Grid. Ao receber estas informações, o Gerenciador Global as armazena em uma estrutura, até que termine a execução da aplicação. Nesse momento, tais dados são gravados em um arquivo que contém todas as informações coletadas dos processos envolvidos no monitoramento. As informações obtidas durante o monitoramento também podem ser armazenadas em arquivos durante a execução da aplicação. A vantagem em se gravar as informações no final da execução da aplicação é que o processo de gravação não irá concorrer com a execução da aplicação.

3.5.3 Informações Disponibilizadas pelo Monitoramento Hierárquico

Assim como a ferramenta de monitoramento de aplicações MPI sofreu alterações para se adequar ao modelo hierárquico proposto, as informações coletadas dos processos monitorados também apresentam algumas modificações.

Conforme citado na Subseção 3.4.2, a criação individual de processos gerenciadores faz com que cada processo MPI disparado tenha seu *rank* (identificador) igual a 0 e comunicadores distintos. Para que seja possível a identificação das informações coletadas

dos processos monitorados, foram definidos pelo SGA identificadores denominados *ranks do middleware*, distintos para cada gerenciador de um mesmo nível hierárquico. É importante observar que existe uma exceção quanto aos Gerenciadores Locais das Máquinas, onde a atribuição de *ranks do middleware* é feita por *site* e não por nível hierárquico.

O GG terá sempre o *rank do middleware* igual ao *rank* definido pelo MPI durante a sua criação, que será sempre igual a zero, uma vez que existe apenas um Gerenciador Global. Ao disparar os Gerenciadores dos Sites, o Gerenciador Global atribui a cada um deles um *rank do middleware* distinto de acordo com sua ordem de criação. O mesmo é feito pelo GS, que ao criar os Gerenciadores Locais das Máquinas do *site* local define diferentes *ranks do middleware* para cada um deles.

Em um ambiente Grid com dois *sites*, por exemplo, o Gerenciador Global dispara dois processos Gerenciadores dos Sites com os respectivos *ranks do middleware*: *rank do middleware* 0 e *rank do middleware* 1, como pode ser observado na Figura 3.10. Na mesma figura, é observado que a distinção entre os *ranks do middleware* dos Gerenciadores Locais das Máquinas ocorre apenas dentro do *site*, ou seja, no primeiro *site* o Gerenciador do Site de *rank do middleware* 0 define para cada GM os *ranks do middleware* 0 e 1, respectivamente. O mesmo é feito pelo Gerenciador do Site de *rank do middleware* 1, que atribui a cada GM do seu *site* os *ranks do middleware* 0 e 1.

Os *ranks do middleware* das tarefas da aplicação são atribuídos pelos processos gerenciadores, que procuram manter a identificação de cada tarefa definida pelo desenvolvedor da aplicação. Ainda na Figura 3.10 é possível observar a existência de dez tarefas da aplicação (t0, t1, t2, t3, t4, t5, t6, t7, t8 e t9) disparadas pelos processos gerenciadores. Os *ranks do middleware* atribuídos a essas tarefas são distintos e variam de 0 a 9.

É através de tais informações (*rank do middleware* de cada tarefa e do *rank do middleware* do processo superior imediato na hierarquia de gerenciadores), além do tipo de processo monitorado, que se torna possível distinguir os processos durante a monitoração. Os tipos possíveis de processos monitorados foram assim definidos: AP - Processo da Aplicação, GM - Gerenciador Local da Máquina, GS - Gerenciador do Site e GG - Gerenciador Global.

Na versão hierárquica, algumas informações adicionais foram acrescentadas à mensagem de *log* para permitir a identificação dos processos monitorados e disponibilizar dados mais precisos para os mecanismos de escalonamento dinâmico e tolerância a falhas. Entre as informações acrescentadas está o número total de processos em execução no *host* local. As informações contidas na mensagem de *log* atual são as seguintes:

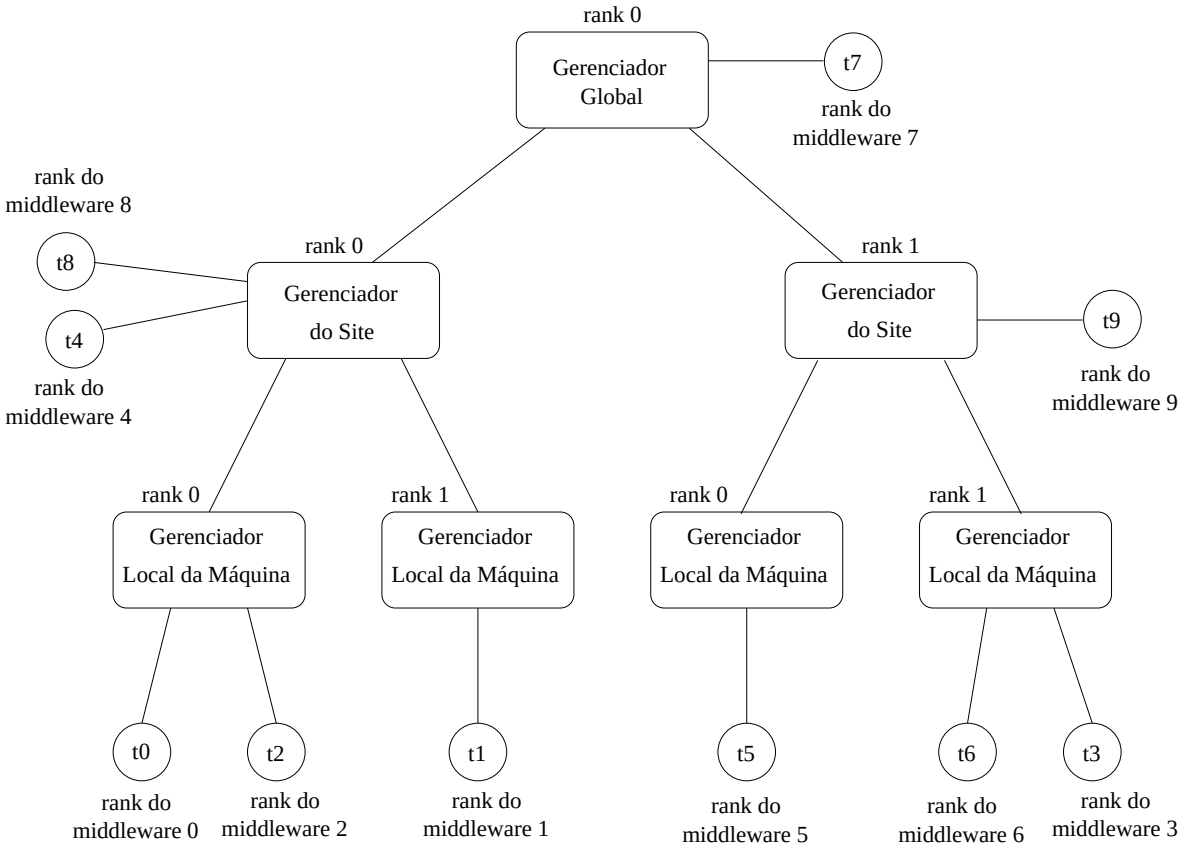


Figura 3.10: Atribuição de *ranks do middleware* distintos a cada processo da hierarquia de gerenciadores e da aplicação.

- tipo de operação realizada (inicialização, envio, recebimento, etc.);
- tipo de processo monitorado (AP, GG, GS ou GM);
- *rank do middleware* do processo origem e/ou destino envolvidos na operação monitorada. O processo pode ser um processo da aplicação ou um processo gerenciador;
- tempo de utilização da CPU no início e no final da função;
- tempo de parede de início e fim da função no processador de origem;
- no caso de operações de recebimento, o tempo de espera da função no processador de origem;
- número de processos em execução na máquina local no momento do monitoramento da operação;

Como ilustração, na Figura 3.11 são apresentados trechos de informações coletadas pelo monitoramento e armazenadas no arquivo monitor.txt. Nas linhas 0, 2, 4, 6 e 8 apresentam-se informações referentes aos processos monitorados, são elas:

- *rank do middleware* do processo;
- *rank do middleware* do processo superior imediato na hierarquia de gerenciadores;
- tipo do processo monitorado (AP, GG, GS ou GM);
- nome da máquina onde o processo executou;
- latência entre a máquina na qual foi executada o processo monitorado e a máquina a qual foi atribuído o processo Gerenciador Global;
- diferença de relógio em relação a máquina na qual foi executado o processo monitorado e a máquina que executou o Gerenciador Global;

Através dos três primeiros dados citados acima é possível identificar um processo dentro da hierarquia. A linha 6, por exemplo, refere-se ao Gerenciador Local da Máquina (GM), que possui *rank do middleware* 1 e pertence ao *site* gerenciado pelo processo Gerenciador do Site com *rank do middleware* 0. Analisando-se os demais dados apresentados ainda na linha 6, é possível identificar que o processo GM foi disparado no *host* "sn15.ic.uff.br", com valor da latência em relação à máquina "sn12.ic.uff.br" (gerenciada pelo GG) igual a 0.000ms e diferença de relógio em relação a essa mesma máquina igual a 16.409ms.

Quando um processo é monitorado, as linhas que aparecem abaixo da linha com os dados relacionados a este processo representam operações executadas pelo processo e os dados associados a tais operações. Para ilustrar essa situação, é possível observar o trecho do arquivo monitor.txt compreendido entre as linhas 8 e 40. A linha 8 apresenta dados do processo da aplicação (AP) com *rank do middleware* igual a 2. Esse processo da aplicação executou no *host* "sn14.ic.uff.br", logo foi disparado pelo Gerenciador Local da Máquina, referenciado na linha 4. Sua latência em relação à máquina "sn12.ic.uff.br" (gerenciada pelo GG) foi 0.000ms e a diferença de relógio 69.741ms. O *rank do middleware* do processo superior na hierarquia de gerenciadores não é necessário para identificar uma tarefa da aplicação, pois o tipo AP e o *rank do middleware* distinto apresentado por cada processo da aplicação são suficientes para identificar os processos da aplicação.

No intervalo entre as linhas 9 e 40 são apresentadas informações coletadas de operações realizadas pela tarefa com *rank do middleware* igual a 2. A linha 9 indica que a primeira operação MPI realizada por esta tarefa foi uma inicialização (MPI_Init()), seguida de uma operação de recebimento (MPI_Recv(), linha 14), três operações de envio (MPI_Send(), linhas 21, 26 e 31) e uma finalização (MPI_Finalize(), linha 36).

Os dados apresentados na linha 14 permitem a identificação de uma operação de recebimento (r) envolvendo as tarefas da aplicação 0 e 2. Neste caso, a tarefa com o *rank do middleware* igual a 2 está recebendo uma mensagem enviada pela tarefa de *rank do middleware* igual a 0. O identificador (*tag*) da mensagem da aplicação recebida é igual a 25 e o identificador (*tag*) da mensagem de *log* gerada pela biblioteca de monitoramento tem valor 46. Os valores seguintes representam: o comunicador entre o processo da aplicação e o Gerenciador Local da Máquina (163932392), tamanho da mensagem recebida em bytes (2400004), o tempo de CPU no início da operação de recebimento (0.000), o tempo de CPU no final desta operação (0.000) e o número de processos em execução na máquina local durante a coleta de informações.

O tempo de CPU consumido pela operação de recebimento é referente apenas ao momento em que a função de recebimento da biblioteca MPI (MPI_Recv()) é executada, sabendo que a mensagem da aplicação já chegou à máquina destino. Isso é possível através do uso de uma função da biblioteca MPI conhecida como MPI_Probe(). Essa função bloqueia a tarefa que a executou até que a mensagem esperada tenha chegado à máquina destino. Ao saber que a mensagem já está disponível, a operação MPI_Recv() é executada e o tempo de CPU referente apenas à execução dessa operação é calculado pelo monitor. O tempo que a tarefa destino leva do momento em que ela chamou a função MPI_Recv() até o momento em que a mensagem realmente chegou à máquina destino (obtida pelo MPI_Probe()) é chamado de tempo de espera da tarefa destino, e será exemplificado a seguir.

```
0       0 -1 GG sn12.ic.uff.br 0.000 0.000
1
2       0 0 GS sn13.ic.uff.br 0.000 82.087
3
4       0 0 GM sn14.ic.uff.br 0.000 69.741
5
6       1 0 GM sn15.ic.uff.br 0.000 16.409
7
8       2 -1 AP sn14.ic.uff.br 0.000 69.741
9       i AP 2 46 0.000 0.000
10      Sat Apr 9 16:56:36 2005 + 107ms
11      Sat Apr 9 16:57:45 2005 + 848ms
12      Sat Apr 9 16:56:36 2005 + 342ms
13      Sat Apr 9 16:57:46 2005 + 83ms
14      r AP 0 2 25 46 163932392 2400004 0.000 0.000 2
15      Sat Apr 9 16:56:36 2005 + 342ms
16      Sat Apr 9 16:57:46 2005 + 83ms
17      Sat Apr 9 16:56:36 2005 + 348ms
18      Sat Apr 9 16:57:46 2005 + 89ms
19      Sat Apr 9 16:56:36 2005 + 356ms
20      Sat Apr 9 16:57:46 2005 + 97ms
21      s AP 2 6 55 46 163932392 1600016 0.840 0.850 3
22      Sat Apr 9 16:56:41 2005 + 763ms
23      Sat Apr 9 16:57:51 2005 + 504ms
24      Sat Apr 9 16:56:41 2005 + 778ms
25      Sat Apr 9 16:57:51 2005 + 519ms
26      s AP 2 7 55 46 163932392 1600016 0.850 0.850 3
27      Sat Apr 9 16:56:41 2005 + 782ms
28      Sat Apr 9 16:57:51 2005 + 523ms
29      Sat Apr 9 16:56:41 2005 + 891ms
30      Sat Apr 9 16:57:51 2005 + 632ms
31      s AP 2 -1 52 46 163932392 4 0.850 0.850 2
32      Sat Apr 9 16:56:41 2005 + 891ms
33      Sat Apr 9 16:57:51 2005 + 632ms
34      Sat Apr 9 16:56:41 2005 + 891ms
35      Sat Apr 9 16:57:51 2005 + 632ms
36      f AP 2 46 0.850 0.850 2
37      Sat Apr 9 16:56:41 2005 + 891ms
38      Sat Apr 9 16:57:51 2005 + 632ms
39      Sat Apr 9 16:56:41 2005 + 891ms
40      Sat Apr 9 16:57:51 2005 + 632ms
```

Figura 3.11: Trechos do arquivo de monitoramento (monitor.txt).

A linha 15 representa a hora local, em milisegundos, do início da operação de recebimento pelo processo da aplicação com *rank do middleware* igual a 2 (Sat Apr 9 16:56:36 2005 + 342ms). A linha seguinte (linha 16) mostra o tempo global (tempo em relação ao processo GG) do início desta operação de recebimento (Sat Apr 9 16:57:46 2005 + 83ms). A linha 17 refere-se ao tempo local em que a tarefa destino leva esperando a chegada da mensagem após ter chamado a função MPI_Recv() (Sat Apr 9 16:56:36 2005 + 348ms). A linha 18, representa o tempo global de espera pela mensagem (Sat Apr 9 16:57:46 2005 + 89ms) e as linhas 19 e 20 representam o tempo local de finalização do recebimento (Sat Apr 9 16:56:36 2005 + 356ms) e o tempo global de finalização do recebimento (Sat Apr 9 16:57:46 2005 + 97ms), respectivamente.

Através dos dados apresentados na linha 21, é possível identificar uma operação de envio (s) envolvendo as tarefas da aplicação com *ranks do middleware* 2 e 6. Nesse caso, a tarefa 2 está enviando uma mensagem para a tarefa 6. A linha 22 representa a hora local, em milisegundos, do início da operação de envio pelo processo da aplicação com *rank do middleware* igual a 2 (Sat Apr 9 16:56:41 2005 + 763ms). A linha seguinte (linha 23) mostra o tempo global (tempo em relação ao processo GG) do início do envio (Sat Apr 9 16:57:51 2005 + 504ms). A linha 24 refere-se ao tempo local em que a tarefa origem leva esperando a operação de envio ser completada (Sat Apr 9 16:56:41 2005 + 778ms). A linha 25, representa o tempo global de finalização da operação de envio (Sat Apr 9 16:57:51 2005 + 519ms).

Os dados apresentados na linha 36 permitem a identificação de uma operação de finalização (f) realizada pela tarefa da aplicação com *rank do middleware* 2. A linha 37 representa a hora local, em milisegundos, do início da operação de finalização pelo processo 2 (Sat Apr 9 16:56:41 2005 + 891ms). A linha seguinte (linha 38) mostra o tempo global do início da operação de finalização (Sat Apr 9 16:57:51 2005 + 632ms) e as linhas 39 e 40 representam o tempo local e global em que a operação de finalização é completada, respectivamente.

3.6 Aspectos de Escalonamento Dinâmico

Uma das maiores necessidades apresentadas pela execução de aplicações em ambientes Grids é a capacidade de ajustar a execução dessas aplicações às mudanças sofridas pelo meio. Inicialmente, o SGA EasyGrid propõe um mecanismo de escalonamento dinâmico de aplicações MPI com base na redistribuição de tarefas da aplicação que ainda não

foram criadas. O re-escalonamento de processos em execução depende de mecanismos de *checkpointing*, o que será abordado em trabalhos futuros. Isto implica na integração dos mecanismos de re-escalonamento com as políticas de tolerância a falhas do SGA EasyGrid.

A metodologia adotada pelo SGA EasyGrid permite que processos gerenciadores do nível 0 e 1 (GG e GS's) assumam as funcionalidades apresentadas pelos processos Gerenciadores das Máquinas. Ou seja, o Gerenciador Global e os Gerenciadores dos Sites podem escalonar, criar e executar processos da aplicação atribuídos aos seus *hosts*. A decisão de criar ou não tarefas da aplicação prontas para execução deve ser feita pelos processos gerenciadores com base em critérios de escalonamento bem definidos.

Como apresentado na Seção 3.2, inicialmente, tarefas da aplicação que não possuem dependência em relação a nenhuma outra tarefa são inseridas em uma lista de tarefas prontas. Entretanto, tarefas que possuem alguma dependência são inseridas em uma lista de tarefas pendentes. Ao receber uma mensagem da aplicação, o processo gerenciador em execução na máquina local, verifica se a tarefa destino já recebeu todas as mensagens pendentes para sua execução podendo então passar para lista de tarefas prontas. Apenas as tarefas que já estão na lista de prontos podem ser disparadas pelos processos gerenciadores. É preciso que os processos gerenciadores respeitem o valor definido para o número máximo de processos em execução concorrente na máquina local. Ou seja, mesmo que existam mais processos prontos do que o número máximo definido, eles não poderão ser criados ao mesmo tempo. Valores diferentes para o número máximo de processos em execução podem ser atribuídos pelo usuário para cada máquina pertencente à hierarquia de gerenciadores.

Processos na lista de prontos e pendentes estão ordenados de acordo com o escalonamento estático inicial. Essa ordenação pode ser utilizada pelos processos gerenciadores no momento em que são tomadas as decisões em relação à criação dinâmica de tarefas da aplicação. O Portal EasyGrid, ao disparar o Sistema Gerenciador da Aplicação, define o tipo de política de escalonamento local que os processos gerenciadores devem adotar durante a seleção na lista de prontos de processos para execução. Duas opções estão disponíveis na versão atual do SGA. Na primeira, processos precisam ser disparados respeitando-se a ordem definida pelo escalonamento estático. Ou seja, se o primeiro processo da lista de prontos possui ordem de execução maior do que processos na lista de pendentes, ele só poderá ser disparado após a criação dos processos que o antecedem na ordem definida pelo escalonamento estático. Na segunda política, a ordem do escalonamento estático não é totalmente respeitada. Processos na lista de prontos são disparados mesmo que existam

processos na lista de pendentes com ordem de execução inferior a deles. Só é considerada a ordenação em relação aos processos prontos para execução.

Foram definidos dois tipos de eventos que podem levar os processos gerenciadores a disparar tarefas da aplicação. O primeiro evento está relacionado ao recebimento de uma mensagem da aplicação, o que pode fazer com que um processo passe do estado de pendente para o estado pronto (tarefa passa da lista de pendentes para a lista de prontos). O segundo evento está associado à conclusão de tarefas em execução no *host* local. Nota-se que quando esses eventos ocorrem, não necessariamente tarefas da aplicação são criadas. Após a ocorrência do primeiro evento, mesmo que o processo que recebeu a mensagem tenha passado para o estado pronto, o valor definido para o número máximo de processos em execução concorrente na máquina local já pode ter sido atingido, ou os processos presentes na lista de prontos podem não estar aptos à execução, devido à política de escalonamento local adotada, impedindo que novos processos sejam disparados. Em relação ao segundo evento, embora o número de processos em execução na máquina local possa ser menor do que o limite definido pelo usuário, pode não haver nenhum processo na lista de prontos aptos para execução, ou a lista pode estar vazia.

O número máximo de processos concorrentes e a ordem com que esses processos serão criados estão relacionados à política local de escalonamento adotada por cada processo gerenciador das máquinas em que foram atribuídas tarefas da aplicação. Além da política de escalonamento local da máquina, dois mecanismos de redistribuição de tarefas da aplicação baseados na hierarquia de processos gerenciadores proposta na Seção 3.4 estão disponíveis na versão atual do SGA EasyGrid. O mecanismo que considera apenas as máquinas pertencentes a um único *site* foi denominado *redistribuição local de tarefas do site*, e o mecanismo que leva em consideração todos os *sites* do Grid foi denominado *redistribuição global de tarefas*. Durante a execução de uma aplicação, os dois mecanismos de redistribuição citados podem ser ativados em diversos momentos.

É importante ressaltar que o objetivo deste trabalho não é propor nenhum algoritmo eficiente de escalonamento dinâmico, e sim propor mecanismos que permitam que tarefas da aplicação sejam re-alocadas durante a execução da aplicação. Os mecanismos de redistribuição local do *site* e redistribuição global de tarefas serão abordados nas Subseções 3.6.1 e 3.6.2.

3.6.1 Redistribuição Local de Tarefas do Site

A redistribuição local do *site*, como citado anteriormente, considera apenas as máquinas de um único *site* durante a re-alocação de tarefas. Através das informações obtidas durante o monitoramento da aplicação, o processo Gerenciador do Site é capaz de acompanhar a execução de tarefas da aplicação atribuídas às máquinas gerenciadas pelos Gerenciadores Locais das Máquinas, identificando e tratando possíveis situações que possam acarretar em uma queda no desempenho da aplicação. O Gerenciador do Site mantém uma lista com dados referentes a todas as tarefas atribuídas às máquinas do *site*. Entre esses dados estão *flags* que indicam se a tarefa já foi executada ou se está participando de um processo de redistribuição local.

Toda vez que uma tarefa da aplicação conclui a sua execução, uma mensagem de *log* é enviada pela biblioteca de monitoramento para o processo gerenciador da máquina local. Se o processo gerenciador pertencer ao nível 2 da hierarquia, a mensagem de *log* é repassada para o Gerenciador do Site e em seguida para o Gerenciador Global do Grid. Se a tarefa estava executando em uma máquina de nível 1, a mensagem de *log* é repassada pelo Gerenciador do Site para o Gerenciador Global. Ao receber a mensagem de *log*, o Gerenciador do Site marca a tarefa como concluída e verifica se é necessário ativar uma política local de redistribuição de tarefas. O término de uma tarefa em execução, na versão atual do SGA, pode levar à ativação da redistribuição de tarefas entre as máquinas do *site*. Porém, outras situações podem ser consideradas como possíveis eventos capazes de ativar uma política de redistribuição de tarefas.

Por exemplo, ao identificar que uma das máquinas do *site* está subcarregada, o Gerenciador do Site ativa a política local de redistribuição de tarefas. A ativação de uma política de redistribuição não garante que tarefas serão re-alocadas, uma vez que, após uma rápida análise, o processo GS pode concluir que a redistribuição proposta pela política ativa levará a um atraso no término da execução da aplicação. Então, finaliza-se a política ativada, sem que nenhuma mudança seja feita na distribuição atual das tarefas. Caso o Gerenciador do Site aprove a redistribuição proposta, um pedido de tarefas é enviado para máquinas do nível 2, solicitando um determinado número de tarefas da aplicação que ainda não foram disparadas pelo Gerenciador Local da Máquina. Ao receber um pedido de tarefas, o Gerenciador Local da Máquina analisa a sua lista de pendentes (tarefas que estão à espera de dados de entrada para a sua execução) e a lista de prontos (tarefas que já receberam todos os dados de entrada e estão esperando que determinadas condições sejam satisfeitas para a sua criação), verificando se é possível ceder tarefas para o Geren-

ciador do Site. Caso seja possível, o Gerenciador Local da Máquina envia para o GS uma mensagem de confirmação (*ack*) acompanhada de uma lista de tarefas cedidas para o processo de redistribuição local do *site*.

Após enviar a lista de tarefas cedidas, o GM redireciona todas as mensagens referentes a essas tarefas que estavam armazenadas na sua lista de mensagens pendentes esperando para serem enviadas para as tarefas destino, após estas serem criadas. Caso não possa atender ao pedido do Gerenciador do Site, o processo GM envia uma mensagem de rejeição (*nack*), informando que não é capaz de ceder tarefas para o processo de redistribuição local. Após receber *acks* e/ou *nacks* de todos os processos GM's do *site* aos quais foram feitos pedidos de tarefas, o Gerenciador do Site faz a redistribuição das tarefas recebidas de acordo com a necessidade de cada máquina.

3.6.1.1 Política de Escalonamento Dinâmico do Site

Para testar o mecanismo de redistribuição proposto, foi implementada uma política simples para o escalonamento dinâmico de tarefas do *site*. Os términos de tarefas da aplicação atribuídas ao *site* local são considerados eventos capazes de ativar a política de escalonamento dinâmico. Após identificar que uma máquina ficou ociosa (através de mensagens de monitoramento), o Gerenciador do Site identifica a máquina do nível 2 (gerenciada por um processo GM) que está mais carregada e envia uma mensagem para o processo gerenciador desta máquina, solicitando metade de suas tarefas para o processo de redistribuição. O número de tarefas solicitadas é calculado a partir da última informação de monitoramento recebida pelo processo GS. Ao receber o pedido, o processo GM verifica se na sua lista de tarefas pendentes ou prontas existe um número suficiente de tarefas da aplicação para atender ao pedido do GS. Caso exista, o processo GM envia para o GS uma mensagem de confirmação acompanhada da lista de tarefas que ainda não foram criadas, e que podem ser cedidas para o processo de redistribuição. Além de enviar as tarefas cedidas para o processo de redistribuição, o GM precisa redirecionar para o processo GS as mensagens destinadas às tarefas que estavam armazenadas na lista de mensagens pendentes. Ao receber a lista de tarefas e as mensagens redirecionadas, o GS termina o processo de redistribuição de tarefas enviando a lista de tarefas para o processador ocioso e repassando as mensagens destinadas as tarefas cedidas.

Caso o número de tarefas solicitado pelo GS seja maior do que o disponível na máquina gerenciada pelo GM, o processo GM envia uma mensagem de rejeição notificando ao Gerenciador do Site que o número de tarefas solicitadas não pode ser satisfeito pela

máquina local. Ao receber uma mensagem de rejeição (*nack*), o Gerenciador do Site termina o processo de redistribuição de tarefas sem alterar a distribuição atual de tarefas entre as máquinas do *site*.

A política de escalonamento dinâmico do *site* adotada na versão atual do SGA Easy-Grid não é eficiente, porém, conforme já citado anteriormente, a intenção deste trabalho é apenas validar o mecanismo de redistribuição implementado, e não propor algoritmos eficientes de escalonamento dinâmico de tarefas da aplicação.

3.6.2 Redistribuição Global de Tarefas

O processo de redistribuição global de tarefas da aplicação é semelhante ao processo de redistribuição local apresentado anteriormente. A ativação do mecanismo de redistribuição global é desempenhada pelo Gerenciador Global do Grid, com base nas informações obtidas pelo monitoramento da aplicação e nos dados associados a cada tarefa. Após ativada, a política de redistribuição precisa ser aprovada, da mesma forma como ocorre na redistribuição local, com o intuito de evitar que a re-alocação de tarefas leve a uma queda no desempenho da aplicação.

Se a política for aprovada, o Gerenciador Global envia uma mensagem para os Gerenciadores dos Sites envolvidos no processo de redistribuição, solicitando parte de suas tarefas para serem redistribuídas entre outros *sites* do Grid. Ao receber um pedido de tarefas enviado pelo GG, o processo GS analisa as demais máquinas do *site*, verificando se deve ou não dar continuidade ao processo de redistribuição global. A partir desse momento, o envolvimento dos processos GM's e GS é semelhante ao apresentado no processo de redistribuição local do *site*.

Caso o GS aprove o pedido de tarefas enviado pelo processo GG, uma análise é feita entre as máquinas do *site* e processos Gerenciadores Locais das Máquinas são escolhidos para participar do processo de redistribuição. Em seguida, o processo GS envia para os GM's escolhidos pedidos de tarefas ainda não executadas, de forma a atender à solicitação do processo GG. Ao receberem as solicitações, os processos GM's podem aceitar o pedido, através do envio de um *ack* acompanhado das tarefas cedidas, ou rejeitar, através do envio de uma mensagem *nack*.

A medida que recebe *acks* e/ou *nacks* dos processos GM's, o Gerenciador do Site repassa para o processo GG as mensagens recebidas, sejam elas confirmações acompanhadas pelas tarefas cedidas, ou rejeições. Após receber resposta de todos os Gerenciadores

dos Sites para os quais enviou pedido de tarefas, o processo GG analisa as tarefas recebidas (caso tenha recebido alguma) e as redistribui entre os demais *sites* envolvidos no processo de redistribuição.

3.6.2.1 Política de Escalonamento Dinâmico Global

Assim como na política de escalonamento dinâmico do *site*, apresentada na Subseção 3.6.1.1, a política de escalonamento dinâmico global implementada na versão atual do SGA EasyGrid não tem como objetivo a obtenção de eficiência na execução da aplicação, mas sim a validação do mecanismo de redistribuição de tarefas proposto. Sendo assim, como evento de ativação da política de escalonamneto dinâmico global foi escolhido o término da execução de todas as tarefas atribuídas a um *site*, ou seja, sempre que todas as máquinas de um *site* ficam ociosas, o mecanismo de redistribuição global de tarefas é ativada.

O Gerenciador Global seleciona o *site* com o maior número de tarefas ainda não disparadas para fazer o pedido de tarefas para o processo de redistribuição. Ao receber o pedido, o GS solicita a cada máquina do seu *site* uma determinada quantidade de tarefas, de forma a atender à solicitação do processo GS. *Acks* e *nacks* recebidos dos processos GM's são repassados para o Gerenciador Global pelo processo GS. Caso sejam cedidas tarefas, o Gerenciador Global as redirecionam para o *site* ocioso. Caso contrário, o processo de redistribuição é finalizado sem que o *site* ocioso receba tarefas do processo de redistribuição.

3.6.3 Redistribuição de Tarefas na Versão de 2 Níveis

Até o momento, todo o mecanismo de redistribuição de tarefas foi apresentado com base na versão da hierarquia com 3 níveis de processos gerenciadores. Apesar desta versão ter sido priorizada durante este trabalho, é importante ressaltar as diferenças apresentadas pelo mecanismo de redistribuição de tarefas na versão com 2 níveis.

Na versão de 2 níveis, por não existir processos GM, todo o mecanismo de redistribuição local de tarefas do *site* é realizado pelo processo GS, ou seja, não existe negociação de tarefas entre as máquinas do *site*. O próprio processo GS toma as decisões referentes à redistribuição, com base em informações obtidas com o monitoramento da aplicação e no seu conhecimento em relação às tarefas atribuídas a cada máquina do *site*. O processo de redistribuição envolve apenas a atualização das listas de tarefas escalonadas nas máquinas do *site*. Embora o processo de redistribuição de tarefas pareça ser bem

simples, a centralização das decisões por parte do Gerenciador do Site pode levar a uma sobrecarga desse processo, além de não permitir a adoção de políticas de escalonamento local nas máquinas do *site*. Acredita-se que uma falha no processo Gerenciador do Site possa ter conseqüências maiores que na versão de 3 níveis, uma vez que todas as informações sobre o *site* concentram-se no processo GS. Apenas estudos mais aprofundados podem levar a conclusões sobre as vantagens e desvantagens da abordagem do SGA com 2 níveis de processos gerenciadores.

3.7 Aspectos de Tolerância a Falhas

Em ambientes heterogêneos e dinâmicos como os Grids Computacionais, é muito comum a ocorrência de falhas nos recursos disponíveis.

A versão atual do SGA EasyGrid não oferece nenhuma estratégia de tolerância a falhas, mas considera na sua implementação a influência dessas estratégias. Ou seja, o SGA EasyGrid foi implementado com a intenção de fornecer facilidades para o tratamento de falhas, assim como para os mecanismos de redistribuição de tarefas da aplicação. A implementação de políticas de tolerância a falhas está fora do escopo dessa dissertação.

A primeira questão tratada pelo SGA no que se refere a tolerância a falhas é a criação individual de processos gerenciadores e processos da aplicação. A criação individual garante que toda a comunicação será feita entre pares de processos, sejam eles processos da aplicação ou processos gerenciadores, através de intercomunicadores. Sendo assim, em caso de falha de um dos processos, os demais processos envolvidos na execução da aplicação não são afetados.

Outra vantagem da criação individual de processos está na facilidade oferecida para a recuperação de processos em caso de falhas. Processos podem ser criados para substituir outros processos que tenham sofrido alguma falha, sem que para isso seja preciso informar aos demais processos envolvidos na execução da aplicação da existência desse novo processo criado. Isso evita a sincronização entre os processos espalhados pelas máquinas do Grid. Caso tivesse sido feita a opção pela versão coletiva, seria necessária a execução de uma operação MPI capaz de criar um novo intracomunicador entre o processo criado e os demais processos do mesmo nível hierárquico que o dele, impondo-se uma sincronização entre os processos envolvidos.

Determinadas funções da biblioteca MPI estão sendo usadas para identificar a ocorrência de erros em operações de envio e recebimento entre processos da aplicação e pro-

cessos gerenciadores. A identificação de falhas pelo MPI é habilitada através de *error handlers* associados aos comunicadores dos processos. Por *default*, o MPI associa aos comunicadores o *error handler* ERRORS_ARE_FATAL, que aborta a execução da aplicação em caso de falhas. Para que seja possível identificar e tratar os erros ocorridos durante a execução, é associado a cada comunicador criado pelo SGA o *error handler* MPI_ERRORS_RETURN, o qual especifica que funções MPI devem retornar um código de erro na ocorrência de falhas. Um conjunto de *error handler* pré-definidos é oferecido pelo MPI e novos tipos de erros podem ser definidos pelos desenvolvedores.

Após a execução de cada operação MPI, processos gerenciadores analisam o valor retornado por estas operações, identificando se ocorreu ou não alguma falha durante a comunicação. Caso sejam identificadas falhas, uma notificação é enviada para o Gerenciador Global, através dos demais processos da hierarquia.

Assim, se a falha for identificada pelo GM de uma das máquinas de nível 2, a notificação de erro é repassada para o processo Gerenciador do Site que se encarrega de enviá-la para o Gerenciador Global do Grid. Caso a falha seja identificada por um dos processos gerenciadores de nível 1, a notificação de erro é enviada pelo GS direto para o Gerenciador Global, que a partir da notificação recebida é capaz de ativar algum mecanismo de recuperação de falhas.

Mensagens de monitoramento podem ser utilizadas pelos processos gerenciadores como sinais de vida de processos da aplicação e de outros processos gerenciadores. Esses sinais indicam se um determinado processo em execução está ou não ativo. A partir dessa informação, decisões relacionadas ao tratamento de falhas podem ser tomadas pelos processos gerenciadores.

3.8 Resumo

Este capítulo descreveu a implementação do Sistema de Gerenciamento de Aplicações (SGA) para execução eficiente e robusta de aplicações MPI em ambientes Grids, no contexto do *framework* EasyGrid. Foi apresentada a construção da hierarquia de processos gerenciadores do SGA proposto, com 2 e 3 níveis de processos gerenciadores. Duas abordagens para criação dos processos gerenciadores foram apresentadas: abordagem individual e abordagem coletiva. A comunicação entre processos da aplicação e processos gerenciadores é possível devido à implementação de uma biblioteca (MEGmpi.h) capaz de adicionar funcionalidades a algumas funções já existentes na biblioteca MPI (*wrapper*).

A biblioteca MEGmpi.h também foi descrita neste capítulo. Aspectos de escalonamento dinâmico de tarefas e tolerância a falhas no contexto do SGA EasyGrid também foram apresentados, assim como o monitoramento hierárquico de aplicação MPI. No capítulo seguinte serão apresentados resultados obtidos com a utilização do SGA EasyGrid na execução de aplicações MPI sintéticas e reais.

Capítulo 4

Experimentos Computacionais

Neste capítulo são apresentados os resultados computacionais obtidos através da análise de características e funcionalidades do SGA EasyGrid para aplicações MPI sintéticas e reais. O objetivo principal dos experimentos realizados é comprovar a viabilidade, escalabilidade e qualidade dos resultados apresentados por execuções de aplicações paralelas MPI com o sistema de gerenciamento proposto. Como viabilidade, é considerada a capacidade do SGA EasyGrid de, a partir de uma aplicação MPI desenvolvida para *clusters* de computadores, gerar uma aplicação *system-aware* capaz de executar de forma eficiente e robusta em ambientes dinâmicos e heterogêneos como os Grids Computacionais. Em relação a escalabilidade, é comprovada a capacidade do SGA de executar grandes aplicações paralelas em Grids constituídos por um número considerável de processadores. A qualidade dos resultados obtidos com a execução de aplicações Smart G-Apps é avaliada através da análise do grau de intrusão imposto pelo gerenciamento da aplicação (criação dinâmica de processos, encaminhamento de mensagens, monitoramento, entre outros) e pela comparação dos tempos de execução da aplicação com o SGA e os modos estáticos padrão do MPI, tcp e lamd.

A maior parte dos experimentos apresentados neste capítulo foram realizados com a versão de 3 níveis do SGA EasyGrid. Em relação à versão de 2 níveis, foram analisadas apenas as questões relacionadas à viabilidade. A principal motivação para avaliação mais detalhada da versão de 3 níveis está relacionada com o fato desta versão permitir o monitoramento de todas as máquinas do Grid, além de evitar o surgimento de um gargalo nos processos gerenciadores de nível 1 (Gerenciador do Site), uma vez que, na versão de 2 níveis, tal processo é responsável pelo gerenciamento (criação dinâmica de processos, encaminhamento de mensagens, escalonamento dinâmico, tolerância a falhas, entre outros) de todas as máquinas do *site*. Uma análise mais detalhada das vantagens e desvantagens da versão de 2 níveis em relação à versão de 3 níveis fica como proposta

para trabalhos futuros.

4.1 Aplicações Paralelas

Como apresentado na Seção 2.6 do Capítulo 2, a eficiência da execução de aplicações paralelas depende das estratégias usadas para escalonar as tarefas entre os processadores disponíveis. A alocação de tarefas aos processadores pode ser realizada de forma estática (em tempo de compilação) ou dinâmica (em tempo de execução). Encontrar um modelo ideal para um programa é uma questão vital para o escalonamento de tarefas e a programação paralela em geral. Modelos de programas tais como Grafos Acíclicos Direcionados (GADs) dão suporte ao desenvolvimento de algoritmos capazes de gerar escalonamentos ótimos ou próximos do ótimo.

Os modos de execução estáticos do MPI criam todos os processos de uma aplicação de uma única vez. O mesmo não acontece na execução de aplicações Smart G-Apps. Nas aplicações *system-aware* geradas pelo *Framework* EasyGrid, processos da aplicação são criados dinamicamente ao longo da execução. Logo, ao se disparar um processo, é preciso levar em consideração as suas precedências, para se evitar situações como *deadlock* e *starvation*.

O projeto EasyGrid adotou a representação GAD para modelar a relação de precedência e as dependências de dados das tarefas de uma aplicação MPI. Grafos acíclicos direcionados (GADs) são denotados por $G = (T, E)$, onde $T = t_1, \ldots, t_n$ é o conjunto de vértices que representam as tarefas que compõem a aplicação e $E = (t_i, t_j)|t_i, t_j \in T$, o conjunto de arestas do grafo que definem as relações de precedência entre as tarefas. A cada tarefa $t_i \in T$ é associado um peso de computação $\varepsilon(t_i) \in N$ e a cada aresta $(t_i, t_j) \in E$, um peso de comunicação $\omega(t_i, t_j) \in N$ indicando o tamanho do dado que deve ser enviado de t_i para t_j. Se existe $(t_i, t_j) \in E$, então a tarefa t_j não pode ser iniciada até que receba os dados enviados por t_i.

Um dos problemas identificados em relação à utilização de GADs para modelar aplicações MPI está relacionado ao fato desse tipo de representação definir apenas a relação de precedência entre as tarefas. Em alguns tipos de aplicação, tarefas trocam dados entre si durante a execução, criando-se uma relação de co-dependência entre elas. Esse tipo de relação não é modelada pelo GAD. Surge então uma necessidade de se adaptar o modelo original, de forma que as relações de co-dependência sejam também representadas. Possivelmente, algumas alterações relacionadas ao escalonamento de tarefas terão de ser

feitas no sistema de gerenciamento proposto, de maneira que este seja capaz de considerar esse tipo de relação de co-dependência no processo de redistribuição de tarefas.

Para os experimentos computacionais apresentados neste capítulo, foram selecionadas quatro aplicações (sintéticas e reais) que podem ser representadas através de grafos. A utilização de aplicações sintéticas facilita o estudo dos efeitos das características de uma aplicação sobre a eficiência da sua execução no ambiente Grid, através da utilização do sistema de gerenciamento de aplicações proposto. Funcionalidades do SGA, tais como criação dinâmica de processos, escalonamento dinâmico de tarefas e tolerância a falhas, podem ser mais facilmente avaliadas através da manipulação de características das aplicações, tais como o custo computacional de cada tarefa e o custo de comunicação entre elas (caso exista comunicação entre as tarefas da aplicação). O *Framework* EasyGrid é capaz de gerar aplicações MPI sintéticas a partir de GADs, utilizando o *MPI Program Generator* apresentado na Subseção 2.4.12 do Capítulo 2.

Entre as classes de aplicações utilizadas estão duas das mais comumente gerenciadas pelos sistemas gerenciadores de recursos Grid atuais: aplicações *parameter sweep* e aplicações mestre-escravo. Aplicações *parameter sweep* executam várias instâncias (cada uma delas com diferentes valores para o conjunto de parâmetros) de um problema. Essas duas classes de aplicações podem ser representadas por um grafo (GAD) *fork-join*, uma vez que as tarefas escravas são geralmente independentes, com pouca ou nenhuma comunicação entre elas. Tanto as aplicações *parameter sweep* quanto as aplicações mestre-escravo são do tipo *Bag of Tasks* (BoT). Aplicações BoT são constituídas de tarefas totalmente independentes entre si, ou seja, tarefas que não necessitam se comunicar para continuar a computação. Além das aplicações *parameter sweep* e mestre-escravo, foi também utilizada nos experimentos realizados uma aplicação paralela, onde as tarefas dessa aplicação trocam mensagens ao longo da execução.

Abaixo são listadas características das 4 aplicações (3 sintéticas e 1 real) utilizadas:

- Aplicação Sintética sem Comunicação: essa aplicação sintética tem como característica tarefas totalmente independentes, ou seja, nenhum tipo de comunicação é realizada entre as tarefas da aplicação. Cada tarefa realiza um determinado cálculo matemático dentro de um intervalo de tempo manipulado de acordo com os objetivos dos experimentos a serem realizados. Além de controlar o custo computacional de cada tarefa, foram feitas variações no número de tarefas para cada intância dessa aplicação. Essa aplicação será referenciada ao longo do texto como Aplicação CPU *Bound*.

- Eliminação de Gauss: a aplicação sintética com comunicação adotada para testes, representa a paralelização do procedimento matemático conhecido como eliminação de Gauss. A eliminação de Gauss é o método mais utilizado para solucionar sistemas de equações lineares, que consiste em transformar um sistema de equações em uma matriz triangular superior equivalente. Uma característica das instâncias desta classe é que os GADs possuem granularidade grossa, uma vez que os pesos de execução são superiores aos pesos de comunicação, em média.

- Aplicação *Parameter Sweep*: a aplicação *parameter sweep* usada nos seguintes experimentos avalia a seleção de heurísticas de escalonamento de tarefas estáticas baseadas sobre um dado conjunto de *benchmarks* GADs, variando-se a granularidade da aplicação, diferentes arquiteturas e configurações de comunicação. Tipicamente, isso envolve mais de 100 mil instâncias de escalonamento. Neste trabalho, a implementação Smart G-App gerencia somente a execução; as heurísticas de escalonamento são códigos externos executados através de chamadas de sistemas de processos MPI do usuário. Para fins de análise, as tarefas foram modificadas de maneira a terem aproximadamente os mesmos tempos de execução. As tarefas continuam a receber dados de entrada de uma tarefa origem e retornam os resultados para essa mesma tarefa origem; entretanto, nenhuma chamada de sistema ou E/S é realizada. Esta versão sintética, denominada APS_s (Aplicação *Parameter Sweep* sintética), permite investigar como diferentes granularidades podem afetar o SGA.

- Aplicação dos Térmions: a aplicação dos térmions é uma implementação de uma aplicação paralela *Bag of Tasks* que calcula a dispersão térmica macroscópica em meios porosos [63]. Dado um meio poroso composto de elementos sólidos e fluidos, a dispersão térmica é avaliada pelo movimento de um grande número de partículas hipotéticas, chamadas térmions, a partir de um ponto fixo chamado meio. A distância percorrida a cada iteração por cada térmion é determinada pela sua posição, energia, um componente randômico e pelas propriedades térmicas dos sólidos ou velocidade do fluxo do fluido. Então, o custo computacional para calcular o caminho de cada térmion é variável e não pode ser determinado *a priori*, devido à natureza randômica do caminho dos térmions. A aplicação dos térmions pertence à classe de aplicações mestre-escravo, onde cada escravo representa um térmion. A aplicação é representada por um GAD *fork-join*, sendo atribuído aos pesos de cada tarefa valores estimados iguais, uma vez que o custo computacional exato de cada térmion não é conhecido antes da sua execução.

4.2 Ambiente Experimental

A maioria dos experimentos computacionais apresentados neste capítulo foram realizados sobre um *cluster* distribuído composto por 43 processadores montados de maneira a simular um Grid Computacional. A simulação de um Grid é motivada pela necessidade de um ambiente controlado que permita a medição e comparação de desempenho, assim como a comprovação de funcionalidades do sistema de gerenciamento proposto, como por exemplo o escalonamento dinâmico de tarefas. Vários *subclusters* foram interconectados por redes de velocidades diferentes, como Gigabit Ethernet, Fast Ethernet e Ethernet, e compostos por processadores heterogêneos, executando Linux Fedora Core 2, Globus Toolkit 2.4 e MPI LAM 7.0.6. Uma ilustração simplificada dessa configuração pode ser visualizada na Figura 4.1. Os *sites* 1, 2, 3 e 4 são compostos por processadores idênticos Pentium IV 2.6 GHz com 512 Mb de RAM, com exceção da máquina sinergia do *site* 1, que possui 1 Gb de RAM. Já o *site* 5 é formado por processadores AMD Athlon(TM) 1.3 GHz com 250 Mb de RAM e por processadores Pentium IV 2.8 GHz com 512 Mb de RAM.

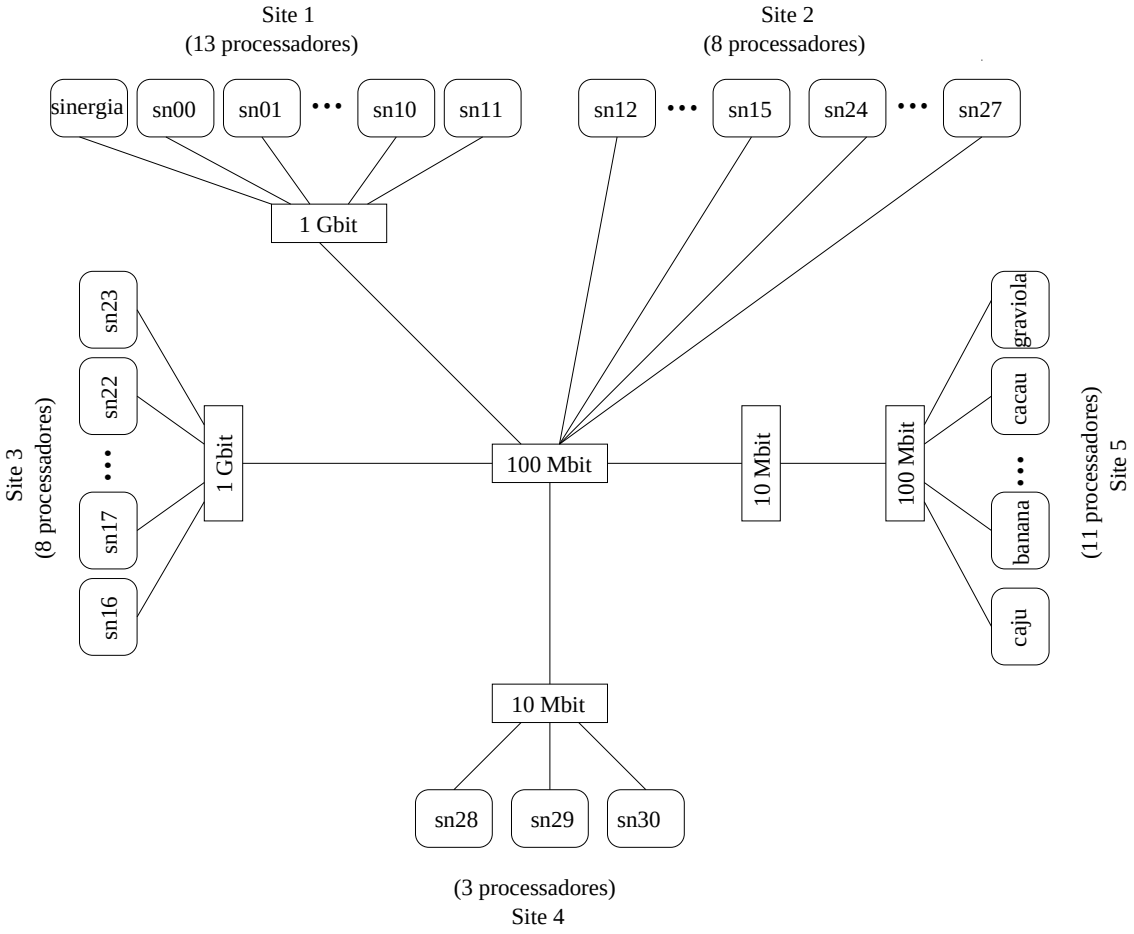


Figura 4.1: Arquitetura do Grid UFF.

Foram realizados alguns experimentos em um ambiente Grid real, envolvendo máquinas da UFF e da PUC-Rio. À arquitetura apresentada acima, foi acrescentado um *site* com 24 máquinas da PUC-Rio com a seguinte configuração: Pentium II 400 MHz com 280 Mb de RAM. As máquinas da PUC estão conectadas à UFF através da rede GIGA de 1 Gbps.

Cada resultado apresentado neste capítulo foi obtido através da média de 5 execuções consecutivas das aplicações e cenários considerados. Apesar da maioria dos experimentos terem sido realizados com o ambiente de execução em modo exclusivo, a média dos tempos de execução garante uma maior segurança nos resultados apresentados devido ao compartilhamento da rede.

4.3 Criação Dinâmica de Processos

O principal objetivo dos experimentos apresentados nesta seção é avaliar o custo da criação dinâmica e o número ideal de processos que devem ser disparados concorrentemente pela política de escalonamento local de cada gerenciador. Inicialmente, foram consideradas duas arquiteturas, ilustradas na Figura 4.2. Na primeira delas, é apresentada como um pior caso a situação onde existem apenas três máquinas (um único *site*), uma delas executando o Gerenciador Global, outra executando o Gerenciador do Site e finalmente uma para execução do Gerenciador Local da Máquina. A arquitetura com três máquinas foi considerada como um pior caso, uma vez que a criação de um processo gerenciador por máquina permite incluir nos testes realizados o custo de comunicação entre processos gerenciadores em máquinas distintas. Nessa primeira arquitetura, as máquinas a que foram atribuídas os processos GG e GS representam uma porcentagem alta dos recursos disponíveis no ambiente de execução. Na segunda arquitetura, são consideradas 11 máquinas distribuídas entre dois *sites*. Em cada um dos dois *sites* existem quatro máquinas executando um processo GM, sendo todas as 11 máquinas ligadas ao mesmo *switch* de 1 Gbit. Nas duas arquiteturas consideradas, cada máquina executa apenas um processo gerenciador. Entretanto, não existe nenhuma restrição na implementação do SGA que impeça a atribuição de mais de um processo gerenciador por máquina.

No sistema de gerenciamento proposto, cada processo da aplicação é criado com um único comunicador MPI. Sendo assim, processos da aplicação só podem ser disparados um de cada vez pelos processos gerenciadores do SGA. O *overhead* para criação dinâmica de processos (MPI_Comm_Spawn()) na plataforma utilizada é de aproximadamente 5ms

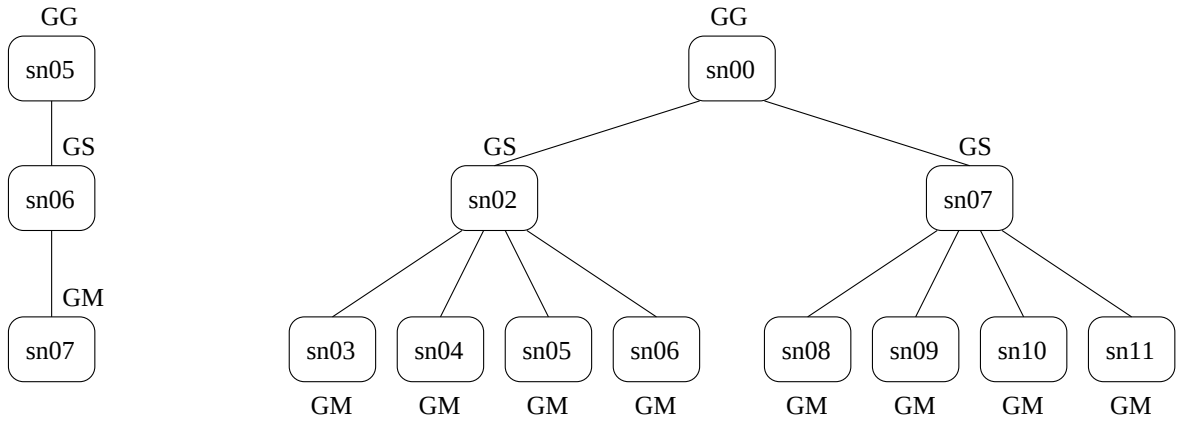


Figura 4.2: Arquiteturas com 1 e 2 *sites*.

para criação local e 15ms para criação remota.

Na abordagem do SGA EasyGrid com três níveis de processos gerenciadores, a criação de processos da aplicação é sempre local, ou seja, o processo gerenciador em execução na máquina local é responsável por criar dinamicamente as tarefas da aplicação atribuídas a esta máquina. Já na abordagem com apenas dois níveis de processos gerenciadores, todas as tarefas da aplicação atribuídas às máquinas do *site* gerenciadas pelo GS são disparadas remotamente por este processo gerenciador. Então, se o gerenciador da máquina está executando processos da aplicação seqüencialmente, o SGA vai acrescentar um custo (percentual) inversamente proporcional ao tempo de execução das tarefas da aplicação. Na abordagem de três níveis, onde a criação de processos é local, o *overhead* para criação de tarefas de 10ms e 1s, por exemplo, é de 50,0% e 0,5%, respectivamente. Na abordagem de dois níveis, o *overhead* para criação dessas tarefas de 10ms e 1s é de 150,0% e 1,5%, respectivamente. Isso mostra que, em execuções de aplicações pesadas, que envolvem tarefas com altos custos computacionais, o *overhead* da criação dinâmica tende a não ser significativo. Todos os experimentos para avaliação do custo de criação dinâmica dos processos foram realizados com o monitoramento da aplicação e dos gerenciadores ativos.

No entanto, os processos gerenciadores podem esconder o *overhead* da criação dinâmica de processos a partir da execução concorrente de mais de um processo da aplicação. Sendo assim, a criação de um processo pode ser sobreposta pela execução de um outro. O número ideal de processos que devem estar em execução concorrentemente é avaliado através dos resultados (tempo total de processamento) apresentados na Tabela 4.1. Estes resultados foram obtidos através da execução da Aplicação CPU Bound envolvendo 100 tarefas totalmente independentes, variando-se a granularidade dessas tarefas em 0,01s, 0,10s, 0,50s, 1s, 5s e 10s. Essa variação da granularidade foi determinada com base no valor do *time*

slice, que é de aproximadamente 100ms. Os testes foram realizados utilizando a arquitetura com três máquinas apresentada na Figura 4.2 e com as funcionalidades da camada de monitoramento do SGA desativadas. Todas as três máquinas executaram tarefas da aplicação.

Tabela 4.1: Tempo total de processamento (em segundos) da Aplicação CPU Bound com 100 tarefas, variando o número máximo de processos concorrentes e a granularidade das tarefas.

	Granularidade da Tarefa (s)					
Processos Concorrentes	0,01	0,1	0,5	1,0	5,0	10,0
1	1,48	4,55	18,20	35,27	171,35	342,35
2	1,22	4,25	17,87	34,91	171,44	342,63
3	1,18	4,20	17,88	34,96	171,66	343,28
4	1,20	4,20	17,90	35,01	171,90	343,86
5	1,23	4,19	17,94	35,04	172,14	344,18
10	1,19	4,21	18,00	35,22	173,32	346,54
15	1,21	4,19	18,10	35,37	174,47	349,17
20	1,17	4,19	18,12	35,54	175,62	351,76
25	1,20	4,19	18,19	35,77	176,44	353,27
30	1,20	4,21	18,23	35,96	178,31	357,04

A partir dessa tabela, é possível concluir que, na prática, o número de processos concorrentes não deve ser muito grande, uma vez que, à medida que cresce o número de processos concorrentes o desempenho da aplicação tende a piorar, provavelmente devido às constantes trocas de contexto. Para tarefas que consomem muita CPU (*CPU bound*), a execução de 2 processos concorrentes é suficiente. Para tarefas que fazem muitas chamadas de sistema, o número de processos concorrentes pode variar entre 3 e 5. Os resultados computacionais apresentados nas seções e subseções seguintes foram obtidos através de execuções do SGA EasyGrid com uma política de escalonamento local capaz de disparar 2 processos concorrentes.

Como apresentado no Capítulo 3, os processos de uma aplicação são criados dinamicamente pelo SGA EasyGrid, sendo associado a cada um deles um comunicador distinto. Na prática, foi identificada uma limitação imposta pela plataforma MPI no que se refere ao número de comunicadores utilizados ao mesmo tempo por uma aplicação. Para solucionar essa questão, os processos gerenciadores liberam o comunicador associado a cada processo disparado após a conclusão do mesmo.

4.4 Comparação entre Execuções com os Modos Estáticos do MPI e com o SGA Proposto.

Para os experimentos realizados ao longo dessa seção, foi utilizada a abordagem estática do SGA EasyGrid. Essa versão é considerada estática por não possuir nenhum mecanismo de redistribuição de tarefas, ou seja, as camadas de escalonamento dinâmico e tolerância a falhas dos processos gerenciadores não estão ativas. A abordagem estática permite que seja avaliada apenas a intrusão do SGA EasyGrid no que se refere ao custo da criação dinâmica de processos e das mensagens de *log* geradas pelos mecanismos de monitoramento.

A Tabela 4.2 e a Figura 4.3 apresentam os resultados obtidos com a execução da Aplicação CPU *Bound* com o SGA e os mecanismos de comunicação estáticos do MPI, tcp e lamd, na arquitetura com três máquinas apresentada na Figura 4.2. Os mecanismos de comunicação estáticos tcp e lamd foram apresentados na Subseção 2.3.4 do Capítulo 2. O escalonamento estático inicial das tarefas da aplicação foi determinado pelo algoritmo *round-robin* tradicional do MPI, e as tarefas foram atribuídas às 3 máquinas da arquitetura. Os resultados apresentados mostram que, para todas as instâncias, variando-se o número de tarefas (50, 100, 376, 500, 738, 1000 e 10000 tarefas) e a granularidade (0,01s, 0,10s, 0,50s e 1s), as execuções com o SGA obtiveram os melhores tempos de resposta quando comparados com as execuções realizadas com as implementações estáticas tradicionais do MPI.

Em relação aos modos de comunicação estáticos do MPI, tcp e lamd, as entradas *não executou* (ne) na Tabela 4.2 mostram que o LAM/MPI não é capaz de executar aplicações com mais que 376 tarefas no modo tcp e 738 tarefas no modo lamd, sobre 3 processadores.

Não existe nenhuma limitação explícita imposta pelo MPI para o número de processos disparados. Porém, na prática foi possível observar a existência de um limite para o número de descritores de arquivos abertos. LAM abre um conjunto de conexões TCP e, para cada conexão, é associado um arquivo descritor. Isso explica o motivo pelo qual as execuções estáticas no modo tcp suportam uma aplicação com um número menor de tarefas que o modo estático lamd. No modo tcp, como apresentado na Subseção 2.3.4 do Capítulo 2, são abertos *sockets* TCP para cada par de processos da aplicação (atribuídos à mesma máquina e a máquinas distintas), o que envolve um grande número de arquivos descritores abertos. No modo lamd, também apresentado na Subseção 2.3.4 do Capítulo 2, essa limitação é identificada quando a aplicação envolve um número maior de processos e o número máximo de arquivos descritores é atingido ao se abrir conexões entre os processos

Tabela 4.2: Comparação dos tempos de execução (em segundos) da Aplicação CPU *Bound* nos modos SGA, tcp e lamd sobre a arquitetura com 3 processadores.

		Granularidade da Tarefa (s)			
N	Modo	0,01	0,1	0,5	1,0
50	tcp	1,30	2,83	9,59	18,05
	lamd	1,03	2,57	9,36	17,80
	SGA	0,90	2,42	9,30	17,77
100	tcp	2,83	5,82	19,37	36,24
	lamd	1,88	4,98	18,46	35,36
	SGA	1,19	4,27	18,00	34,94
376	tcp	47,97	58,08	108,33	170,48
	lamd	22,79	34,36	84,41	147,25
	SGA	2,87	14,21	65,11	127,95
500	tcp	ne	ne	ne	ne
	lamd	48,95	63,95	130,16	213,23
	SGA	3,65	18,62	86,10	169,42
738	tcp	ne	ne	ne	ne
	lamd	146,43	167,86	301,60	388,12
	SGA	5,25	27,20	126,73	249,77
1000	tcp	ne	ne	ne	ne
	lamd	ne	ne	ne	ne
	SGA	6,78	36,61	171,59	338,38
10000	tcp	ne	ne	ne	ne
	lamd	ne	ne	ne	ne
	SGA	64,05	361,42	1705,11	3391,13

da aplicação e o *daemon* MPI em execução na máquina local. No modo lamd, não é necessário abrir um *socket* TCP para cada par de processos da aplicação, uma vez que toda comunicação inter-máquina é feita através dos processos *daemons* disparados em cada uma das máquinas do ambiente LAM.

Pelos resultados apresentados na Tabela 4.2, é possível observar duas características importantes do SGA em relação aos modos de comunicação estáticos do MPI, tcp e lamd: bom desempenho e escalabilidade. Com um pequeno número de tarefas, o tempo de execução da aplicação usando o SGA é um pouco melhor que no modo estático. Quando o número de tarefas da aplicação aumenta, o desempenho relativo da Aplicação CPU *Bound* melhora significativamente. Em relação à escalabilidade, o SGA é capaz de gerenciar um grande número de processos. Um outro comportamento importante a ser observado é que, à medida que o número de tarefas aumenta, o tempo de execução da aplicação não degrada, como acontece com os modos estáticos tcp e lamd, mantendo-se proporcionalmente o mesmo desempenho da aplicação com um pequeno número de tarefas. Como apresentado na Tabela 4.2, as versões estáticas são limitadas para um número relativamente pequeno

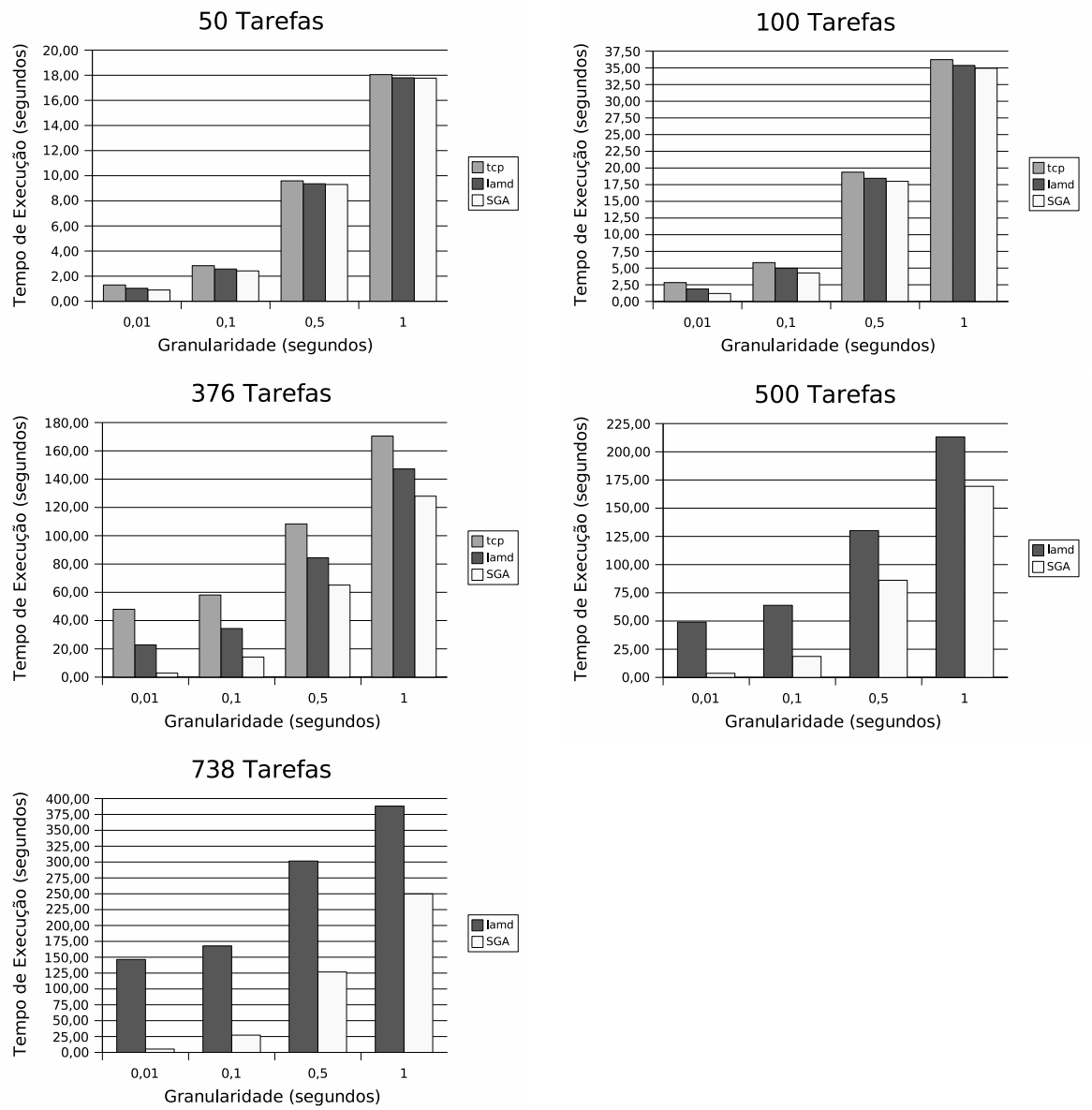


Figura 4.3: Comparação do tempo de execução da Aplicação CPU *Bound* com o SGA e os modos estáticos tcp e lamd sobre a arquitetura de 3 processadores.

de tarefas e, com o aumento do número de tarefas, o desempenho da aplicação cai.

A Figura 4.4 apresenta o desempenho da Aplicação CPU *Bound* variando-se o número de tarefas (50, 100, 376, 500, 738, 1000 e 10000 tarefas) e mantendo-se a granularidade em 0,01 segundos. Através desta figura, é possível comprovar a degradação no desempenho das execuções com os modos tcp e lamd, onde os tempos de execução da aplicação crescem exponencialmente à medida que se aumenta o número de tarefas da aplicação. Ao contrário do que ocorre com os modos tcp e lamd, nas execuções com o SGA o tempo de execução da aplicação cresce linearmente com o aumento do número de tarefas da aplicação.

A Figura 4.5 mostra a intrusão do SGA (criação dinâmica de processos) à medida

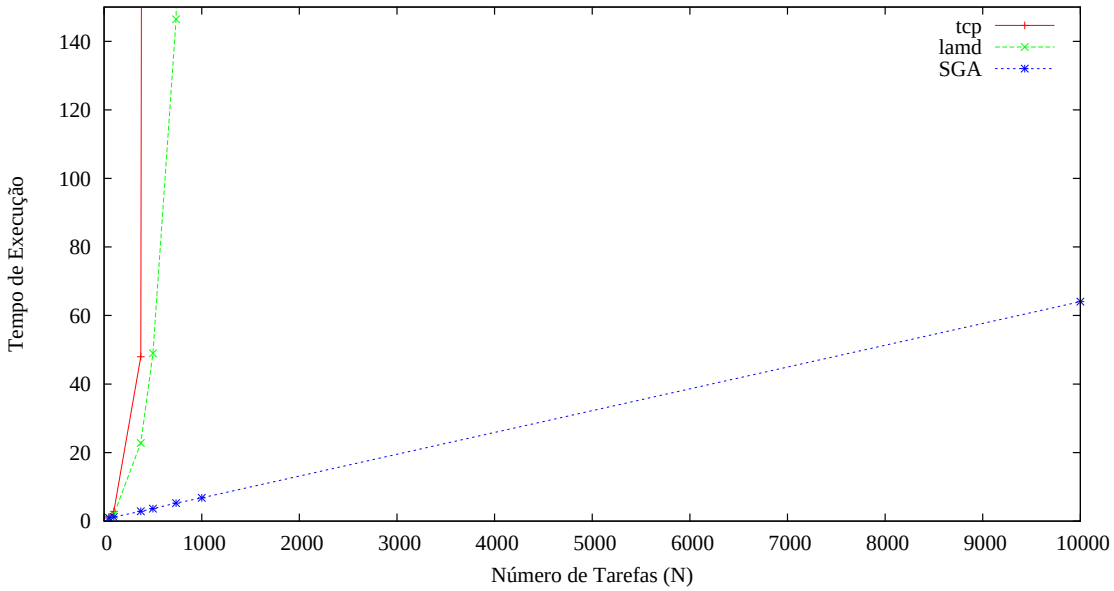


Figura 4.4: Desempenho da Aplicação CPU *Bound* na arquitetura com 3 processadores, variando-se o número de tarefas entre 50 e 10000 e mantendo a granularidade de 0,01 segundos.

que variam o custo computacional (granularidade) das tarefas da aplicação e o número de tarefas envolvidas na execução. A intrusão é calculada a partir dos resultados apresentados na Tabela 4.2 e do tempo de execução mínimo teórico, que é igual a $N \times$ (granularidade da tarefa) $\div$ (número de processadores). É possível observar que a intrusão do SGA para criação dinâmica de tarefas tende a tornar-se mínima, praticamente insignificante, para aplicações com um grande número de tarefas com custos computacionais consideráveis.

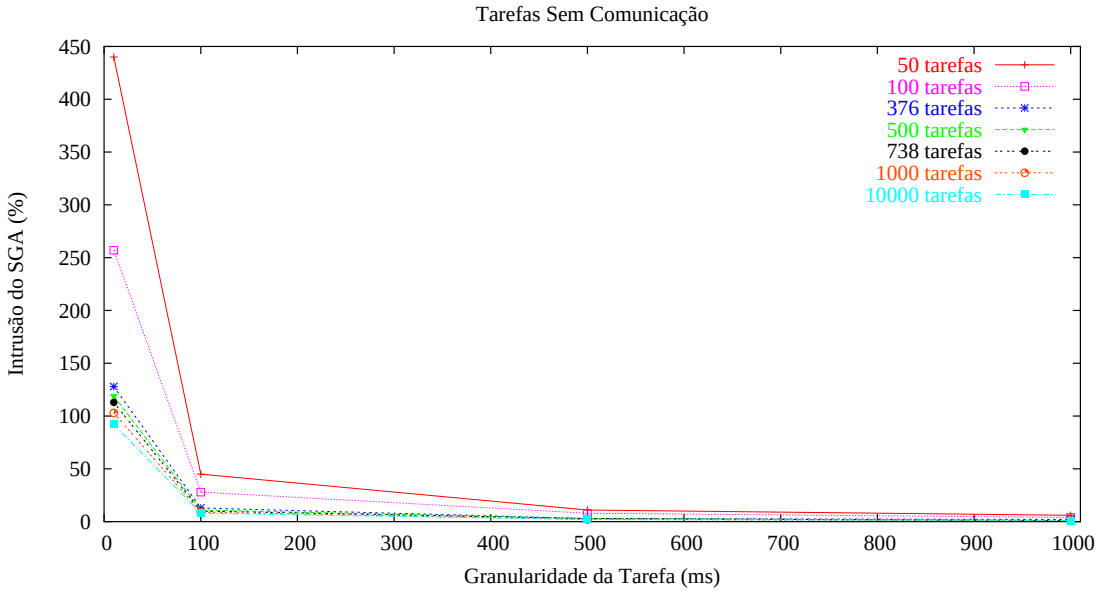


Figura 4.5: Intrusão do SGA EasyGrid em execuções da Aplicação CPU *Bound* na arquitetura com 3 máquinas.

As Figuras 4.6 e 4.7, também geradas a partir dos resultados apresentados na Tabela 4.2,

mostram o ganho do SGA em relação aos tempos de execução da Aplicação CPU *Bound* com os modos de comunicação tcp e lamd, respectivamente. Quanto maior o número de tarefas da aplicação, melhor é o desempenho da execução com o SGA em relação às versões estáticas. Isso se deve ao fato de que, nas versões estáticas do MPI, todas as tarefas da aplicação são disparadas ao mesmo tempo e conseqüentemente pertencem a um mesmo comunicador, o que faz com que chamadas à função MPI_Init() funcionem como uma barreira natural envolvendo todos os processos da aplicação. No SGA, processos são criados individualmente, existindo apenas um processo por comunicador, o que torna a execução dinâmica livre do atraso provocado pela sincronização da função MPI_Init().

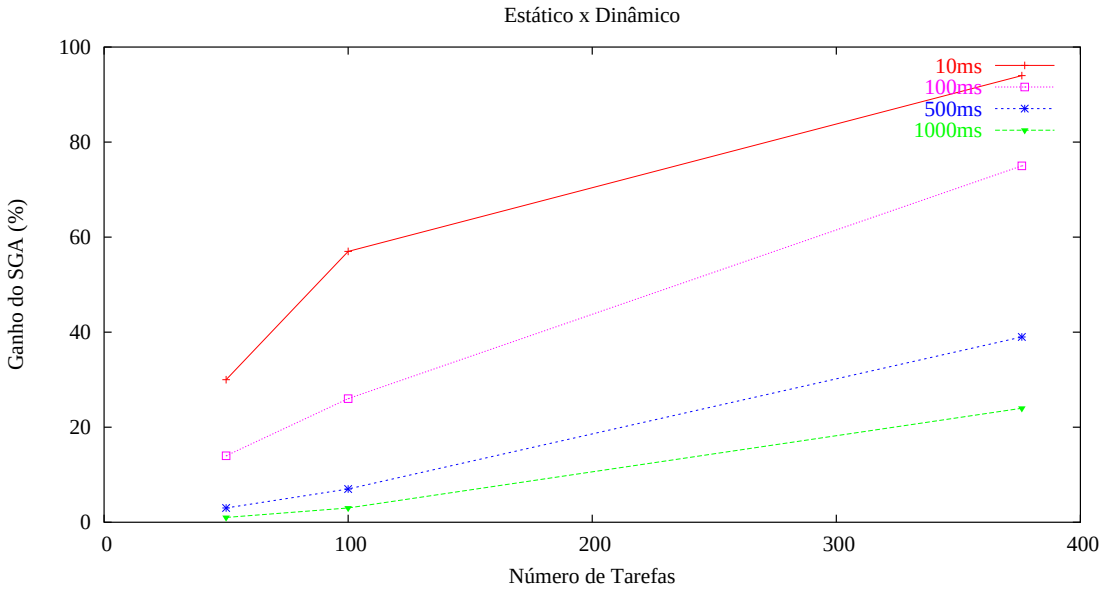


Figura 4.6: Desempenho do SGA EasyGrid em relação à execução estática no modo tcp. Experimentos realizados com a Aplicação CPU *Bound* na arquitetura com 3 processadores.

Com relação às Figuras 4.6 e 4.7, ainda é possível observar a tendência das execuções estáticas obterem uma melhora no tempo total de execução da aplicação à medida que se aumenta a granularidade das tarefas. O custo provocado pela sincronização imposta pela função MPI_Init() depende do número de tarefas da aplicação, e não da granularidade das mesmas. Logo, dado um número fixo de tarefas, quanto maior o tempo total de execução das tarefas, menos significativo é o atraso provocado pela chamada à função MPI_Init().

A Tabela 4.3 e a Figura 4.8 apresentam os resultados obtidos com a execução da Aplicação CPU *Bound* com a arquitetura de três máquinas. Nesse caso, nas execuções com o SGA, as tarefas da aplicação foram atribuídas a apenas duas máquinas da arquitetura, às máquinas em que estão executando o processo Gerenciador da Máquina e Gerenciador do Site. O objetivo desses experimentos, é analisar o quanto representa para o SGA a

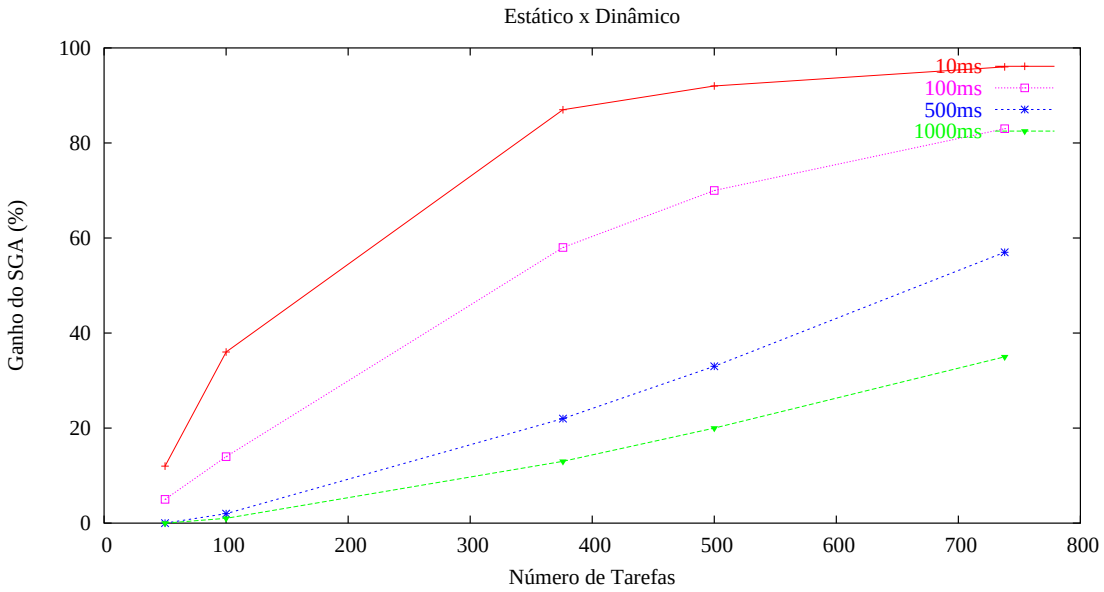


Figura 4.7: Desempenho do SGA EasyGrid em relação à execução estática no modo lamd. Experimentos realizados com a Aplicação CPU *Bound* na arquitetura com 3 processadores.

perda de poder computacional quando não são atribuídas tarefas da aplicação somente à máquina gerenciada pelo processo GG. Ao contrário dos resultados apresentados na Tabela 4.2, as execuções realizadas com o SGA não obtiveram o melhor desempenho para todas as variações de número de tarefas e granularidades apresentadas. Para a Aplicação CPU *Bound* com 50 tarefas, o SGA teve o tempo de execução um pouco maior que o modo estático tcp para as granularidades 0,1s, 0,5s e 1s, e maior que o modo estático lamd para todas as variações de granularidade (0,01s, 0,1s, 0,5s e 1s). Para a aplicação com 100 tarefas, o SGA só obteve os melhores resultados (menores tempos de execução) em relação aos dois modos de comunicação estáticos, para as tarefas com custo computacional de 0,01s. Para as execuções com 376, 500, 738, 1000 e 10000 tarefas, as execuções com o SGA apresentaram os melhores resultados para as quatro granularidades usadas nos experimentos, com exceção dos valores obtidos para algumas granularidades em instâncias com 376 e 500 tarefas. Entretanto, os resultados obtidos foram piores que os apresentados na Tabela 4.2, uma vez que, na arquitetura com três máquinas, a perda de uma das máquinas na execução de tarefas da aplicação representa uma queda significativa no poder computacional disponível.

Os resultados ilustrados na Figura 4.8 reforçam o fato de que, quanto maior o custo computacional das tarefas, menos significativo é o atraso provocado pelas chamadas à função MPI_Init nas execuções com os modos estáticos do MPI. Como nesse caso as execuções com o SGA utilizaram apenas 2 processadores da arquitetura, a perda de uma

Tabela 4.3: Comparação dos tempos de execução (em segundos) dos modos SGA, tcp e lamd. Experimentos realizados com a Aplicação CPU *Bound* na arquitetura com 3 processadores. As tarefas da aplicação foram atribuídas a apenas 2 máquinas da arquitetura.

		Granularidade da Tarefa (s)			
N	Modo	0,01	0,1	0,5	1,0
50	tcp	1,30	2,83	9,59	18,05
	lamd	1,03	2,57	9,36	17,80
	SGA	1,06	3,30	13,43	25,91
100	tcp	2,83	5,82	19,37	36,24
	lamd	1,88	4,98	18,46	35,36
	SGA	1,47	5,98	26,23	51,23
376	tcp	47,97	58,08	108,33	170,48
	lamd	22,79	34,36	84,41	147,25
	SGA	3,93	20,95	96,59	191,03
500	tcp	ne	ne	ne	ne
	lamd	48,95	63,95	130,16	213,23
	SGA	5,10	27,67	128,26	253,86
738	tcp	ne	ne	ne	ne
	lamd	146,43	167,86	301,60	388,12
	SGA	7,35	40,55	188,92	374,46
1000	tcp	ne	ne	ne	ne
	lamd	ne	ne	ne	ne
	SGA	9,62	54,74	255,70	507,12
10000	tcp	ne	ne	ne	ne
	lamd	ne	ne	ne	ne
	SGA	99,02	552,05	2552,81	5067,42

máquina na execução de tarefas da aplicação fez com que o SGA obtivesse pior desempenho quando comparado com os modos estáticos do MPI em execuções com um pequeno número de tarefas e com os maiores custos computacionais.

A Tabela 4.4 apresenta o percentual de aumento do tempo total de execução da Aplicação CPU *Bound* com o SGA na arquitetura com 3 processadores, quando as tarefas são atribuídas a apenas 2 máquinas da arquitetura, em vez de 3. O objetivo dessa análise foi verificar o quanto representa para o SGA a perda computacional da máquina gerenciada pelo processo GG, quando não são atribuídas a ela tarefas da aplicação. Os resultados mostram que, para a maioria das instâncias, as execuções utilizando apenas 2 máquinas da arquitetura obtiveram tempos de execução aproximadamente 50% maiores que as execuções com 3 processadores. As únicas exceções foram as execuções com as instâncias pequenas (poucas tarefas e baixa granularidade), onde a utilização de um processador a mais não representou um ganho significativo em relação à sobrecarga imposta pelo sistema de gerenciamento. Em execuções com instâncias pequenas, a sobrecarga do SGA torna-se

mais evidente.

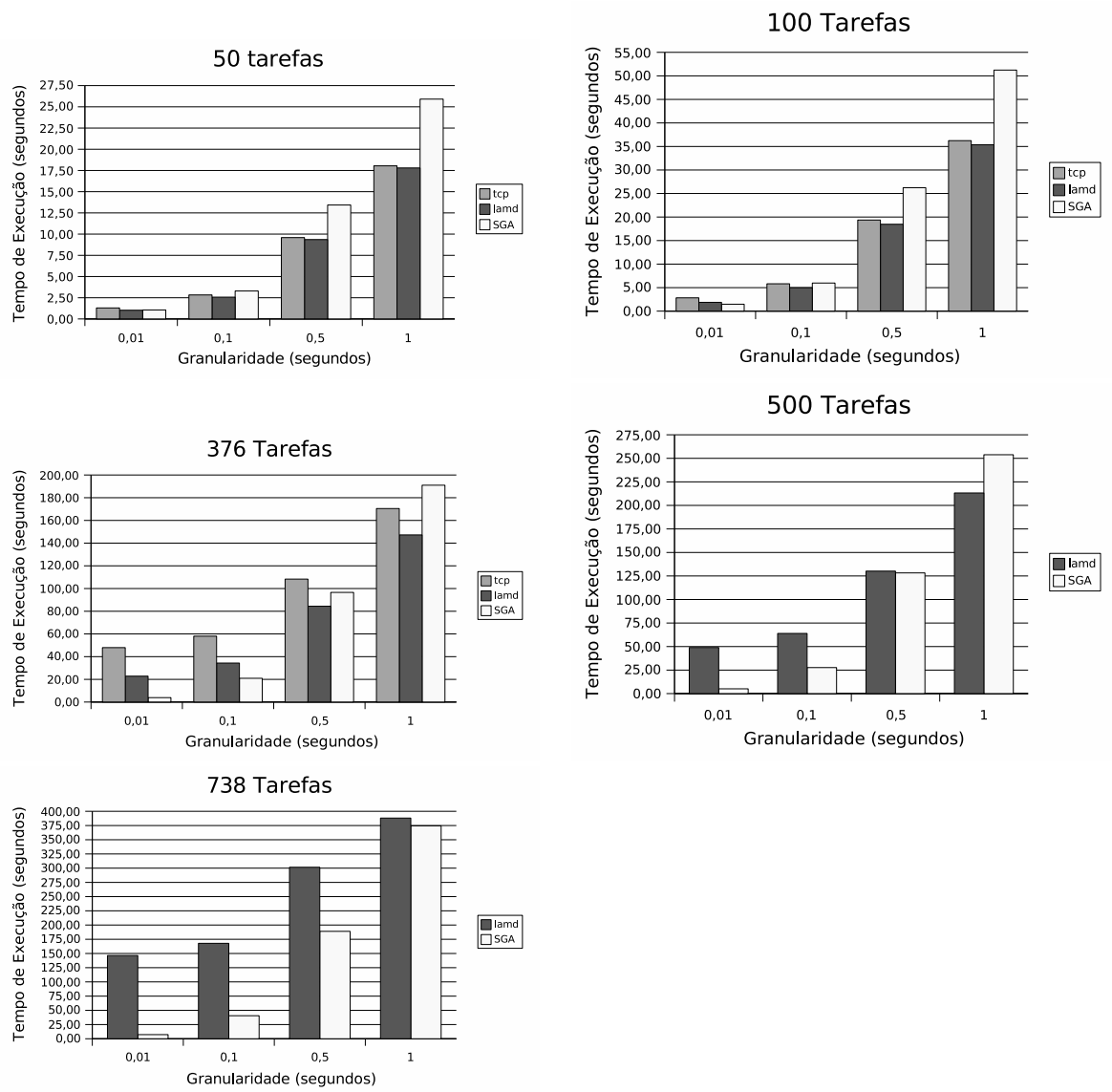


Figura 4.8: Comparação do tempo de execução da Aplicação CPU *Bound* com o SGA e os modos estáticos tcp e lamd. As tarefas foram escalonadas apenas entre dois dos três processadores que compõem a arquitetura utilizada.

A Tabela 4.5 apresenta os resultados obtidos com a execução da Aplicação CPU *Bound* na arquitetura com 11 máquinas. Tarefas da aplicação foram atribuídas às 11 máquinas disponíveis no ambiente de execução. O comportamento do SGA EasyGrid foi o mesmo verificado na execução dessa mesma aplicação na arquitetura com três máquinas, ou seja, o SGA apresentou bom desempenho e escalabilidade quando comparado com as implementações estáticas tradicionais do MPI. Com 11 máquinas, foi possível executar até 500 tarefas no modo estático tcp e até 1000 tarefas no modo estático lamd. Com o aumento do número de máquinas de nível 2 (máquinas que executam o processo GM), é provável que o Gerenciador do Site precise dedicar mais tempo para o gerenciamento

Tabela 4.4: Percentual de aumento no tempo total de execução da Aplicação CPU *Bound* com SGA na arquitetura com 3 processadores. Quando não executando tarefas da aplicação juntamente com o processo GG.

	Granularidade da Tarefa (s)			
N	0,01	0,1	0,5	1,0
50	17,78%	36,36%	44,41%	45,81%
100	23,53%	40,05%	45,72%	46,62%
376	36,93%	47,43%	48,35%	49,30%
500	39,73%	48,60%	48,97%	49,84%
738	40,00%	49,08%	49,07%	49,92%
1000	41,89%	49,52%	49,02%	49,87%
10000	54,60%	52,74%	49,72%	49,43%

das máquinas dos *sites* envolvidas na execução de tarefas da aplicação. Entretanto, essa diminuição de poder computacional na execução de processos da aplicação é compensada pelo fato da perda representar uma fração muito pequena do poder computacional total disponível. Isso é devido ao custo de gerenciamento das máquinas do *site* ser pequeno, principalmente por não haver troca de mensagens entre as tarefas da aplicação, não se justificando a não atribuição de tarefas às máquinas de nível 1.

A Figura 4.9 apresenta uma análise da intrusão do SGA nas execuções da Aplicação CPU *Bound* com 11 máquinas. Assim como nos experimentos realizados com a arquitetura de três máquinas, a intrusão do SGA EasyGrid diminui à medida que o custo computacional (granularidade) e o número de tarefas da aplicação aumentam.

A Tabela 4.6 apresenta a diferença (percentual) entre as intrusões obtidas nas execuções com as arquiteturas de 3 e 11 máquinas. Com um número muito pequeno de tarefas e com granularidades baixas, o SGA EasyGrid tende a ser mais intrusivo nas execuções com a arquitetura de 11 máquinas. Isso se deve ao fato de que o custo para a criação da hierarquia de gerenciadores, o aumento do número de mensagens iniciais de gerenciamento e a distribuição de poucas tarefas da aplicação entre as 11 máquinas disponíveis tem um peso considerável em relação ao tempo total de execução da aplicação. O número de níveis na hierarquia de gerenciadores (3 níveis) é o mesmo tanto nos experimentos com a arquitetura de 3 processadores como nos experimentos com a arquitetura de 11 processadores. Entretanto, como cada gerenciador do nível abaixo é disparado seqüencialmente pelo gerenciador do nível acima, o custo para criação de toda a hierarquia de processos gerenciadores tende a ser maior na arquitetura com 11 processadores.

Apesar do custo para criação da hierarquia de 11 processadores ser maior do que o

Tabela 4.5: Comparação dos tempos de execução (em segundos) dos modos SGA, tcp e lamd obtidos em experimentos com a Aplicação CPU *Bound* na arquitetura com 11 processadores.

		Granularidade da Tarefa (s)			
N	Modo	0,01	0,1	0,5	1,0
50	tcp	1,02	1,48	3,49	5,97
	lamd	0,90	1,35	3,35	5,85
	SGA	0,88	1,30	3,31	5,81
100	tcp	1,78	2,66	6,65	11,64
	lamd	1,45	2,33	6,36	11,28
	SGA	1,05	1,80	5,79	10,80
376	tcp	12,03	15,36	29,43	46,69
	lamd	7,54	10,86	25,41	42,23
	SGA	2,03	4,55	18,57	36,11
500	tcp	24,99	30,00	47,87	70,54
	lamd	15,33	19,65	38,72	61,67
	SGA	2,54	5,75	24,19	47,25
738	tcp	ne	ne	ne	ne
	lamd	43,71	51,77	77,28	109,27
	SGA	3,37	8,09	35,25	69,47
1000	tcp	ne	ne	ne	ne
	lamd	109,04	118,22	153,26	198,29
	SGA	4,33	10,73	47,61	93,72
10000	tcp	ne	ne	ne	ne
	lamd	ne	ne	ne	ne
	SGA	39,46	100,22	469,22	930,57

obtido pela arquitetura com 3 processadores, os altos valores apresentados no cálculo da intrusão se devem ao fato de que, na realidade, tais valores correspondem à intrusão do SGA mais a intrusão imposta pelo próprio MPI. Todos os valores de intrusão são calculados com base no tempo de execução esperado, o que nunca seria atingido pelas execuções estáticas tradicionais do MPI, que apresentaram resultados piores que as execuções com o SGA, como apresentado nas Tabelas 4.2 e 4.5. À medida que a granularidade e o número de tarefas da aplicação aumentam, o grau de intrusão do SGA EasyGrid passa a ser praticamente o mesmo nas arquiteturas de 3 e 11 máquinas.

A Tabela 4.7 apresenta os resultados obtidos com a execução da Aplicação CPU *Bound* na arquitetura com 11 máquinas, sendo que nessa situação, ao contrário do que foi apresentado na Tabela 4.5, tarefas da aplicação não foram atribuídas à máquina de nível 0, que estava executando o Gerenciador Global do Grid. Nesse caso, a perda de uma máquina na execução de tarefas da aplicação não fez com que nenhuma das execuções do SGA obtivessem tempos maiores (menor desempenho) que os obtidos com as execuções

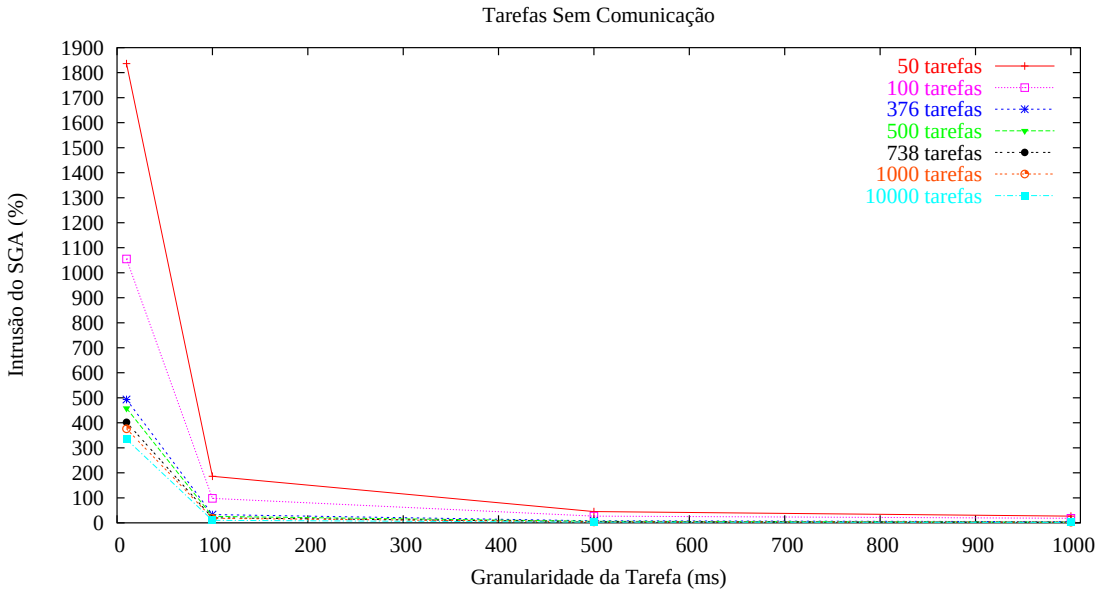


Figura 4.9: Intrusão do SGA EasyGrid à medida que varia a granularidade das tarefas em execuções com a aplicação CPU *Bound* na arquitetura com 11 processadores.

Tabela 4.6: Diferença (percentual) de intrusão entre as execuções com a arquitetura de 3 e 11 máquinas em execuções da Aplicação CPU *Bound* com o SGA.

	Granularidade da Tarefa (s)			
N	0,01	0,1	0,5	1,0
50	1396,00%	140,80%	34,04%	21,20%
100	798,00%	69,90%	19,38%	13,98%
376	364,89%	19,73%	4,75%	3,55%
500	339,80%	14,78%	3,12%	2,30%
738	288,89%	10,01%	2,05%	2,02%
1000	272,90%	8,20%	1,79%	1,58%
10000	241,91%	1,81%	0,92%	0,63%

estáticas lamd e tcp. A perda de uma máquina para a arquitetura com 11 processadores não ocasionou uma queda significativa no poder computacional disponível para execução de instâncias da aplicação com um pequeno número de tarefas ou com baixa granularidade.

Nos experimentos realizados com a aplicação *parameter sweep* APS_s nas arquiteturas de 3 e 11 máquinas, os resultados obtidos seguiram o mesmo comportamento das execuções realizadas com a Aplicação CPU *Bound*. Também para a aplicação APS_s, o SGA mostrou ter um bom desempenho e escalabilidade. A Figura 4.10 apresenta o grau de intrusão do SGA em execuções com a aplicação APS_s na arquitetura com 11 máquinas. As tarefas da aplicação foram escalonadas entre as 11 máquinas disponíveis pelo algoritmo de escalonamento estático *round robin*. Assim como nas execuções com a Aplicação CPU

Tabela 4.7: Comparação dos tempos de execução da Aplicação CPU *Bound* nos modos estáticos tcp e lamd e com o SGA EasyGrid na arquitetura de 11 máquinas.

		Granularidade da Tarefa (s)			
N	Modo	0,01	0,1	0,5	1,0
50	tcp	1,02	1,48	3,49	5,97
	lamd	0,90	1,35	3,35	5,85
	SGA	0,89	1,29	3,32	5,84
100	tcp	1,78	2,66	6,65	11,64
	lamd	1,45	2,33	6,36	11,28
	SGA	1,06	1,83	5,92	10,92
376	tcp	12,03	15,36	29,43	46,69
	lamd	7,54	10,86	25,41	42,23
	SGA	2,11	4,88	20,11	39,20
500	tcp	24,99	30,00	47,87	70,54
	lamd	15,33	19,65	38,72	61,67
	SGA	2,51	6,20	26,49	51,83
738	tcp	ne	ne	ne	ne
	lamd	43,71	51,77	77,28	109,27
	SGA	3,37	8,83	38,93	76,50
1000	tcp	ne	ne	ne	ne
	lamd	109,04	118,22	153,26	198,29
	SGA	4,39	11,67	52,29	102,94
10000	tcp	ne	ne	ne	ne
	lamd	ne	ne	ne	ne
	SGA	40,04	110,71	516,15	1023,43

Bound, o grau de intrusão do SGA EasyGrid nas execuções com a aplicação APS_s diminui à medida que a granularidade e o número de tarefas da aplicação aumentam. Esses resultados são considerados favoráveis, uma vez que o objetivo do SGA EasyGrid é gerenciar a execução de aplicações de grande escala, ou seja, aplicações que demandam grande quantidade de poder computacional.

O escalonamento *round robin* tradicional definido pela biblioteca MPI não considera as dependências entre as tarefas da aplicação. Ele simplesmente distribui as tarefas entre os processadores disponíveis de forma circular, como apresentado na Seção 2.3.4 do Capítulo 2. Com o escalonamento *round robin*, aplicações que envolvem trocas de mensagens ao longo da execução podem entrar em um estado de *starvation*. Isso é conseqüência do fato de que, no SGA EasyGrid, as tarefas da aplicação só podem ser disparadas após todos os dados de entrada estarem disponíveis, podendo levar as tarefas da aplicação a um estado de espera permanente. Sendo assim, para os experimentos computacionais realizados com a aplicação de Gauss, foi utilizado um algoritmo de escalonamento baseado

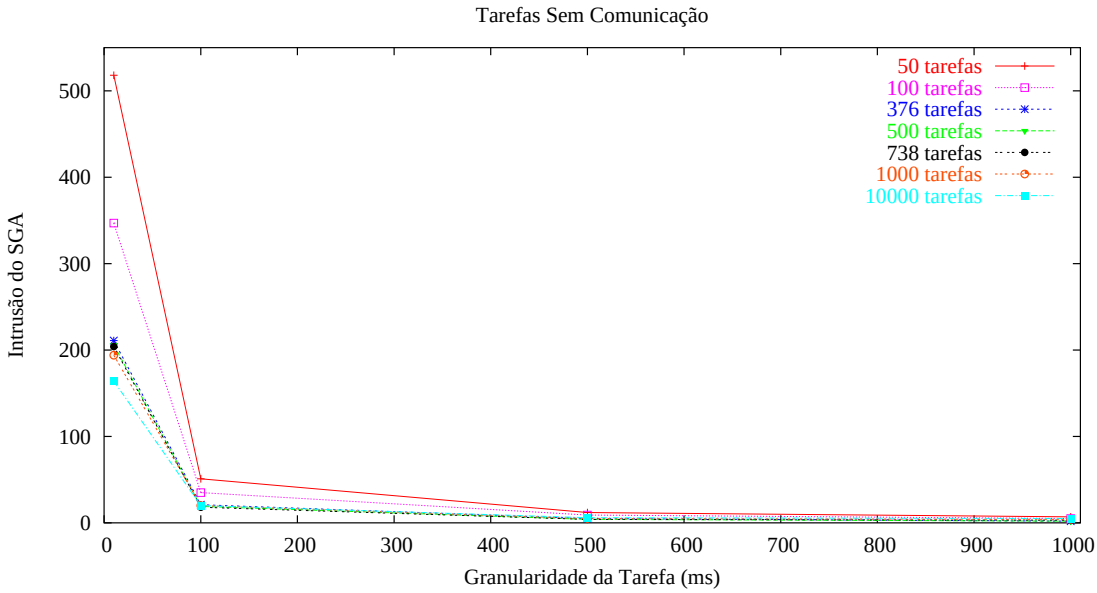


Figura 4.10: Intrusão do SGA EasyGrid a medida que varia a granularidade das tarefas em execuções com a aplicação APS_s na arquitetura com 11 processadores.

na heurística *list scheduling*. A idéia básica do algoritmo *list scheduling* é atribuir prioridades aos nós do GAD, criando-se uma lista ordenada de nós. O escalonamento é feito selecionando-se uma tarefa livre de mais alta prioridade da lista e um processador ocioso para alocar essa tarefa. Uma tarefa é considerada livre quando todos os seus predecessores imediatos já foram escalonados. O objetivo dos experimentos realizados não é avaliar nenhuma das heurísticas de escalonamento presentes no *framework* EasyGrid, e sim avaliar as funcionalidades do SGA proposto.

Experimentos foram realizados com a aplicação que simula o método de Eliminação de Gauss com 54, 104, 377, 495, 740, 989 e 10010 tarefas nas arquiteturas de 3 e 11 máquinas. A Tabela 4.8 apresenta os resultados obtidos para execução da aplicação na arquitetura com 11 máquinas. Foram feitas execuções com dois dos modos estáticos do MPI, tcp e lamd, e os resultados obtidos foram comparados com as execuções usando o SGA EasyGrid. O escalonamento estático *list scheduling* foi utilizado tanto nas execuções estáticas do MPI (tcp e lamd) como nas execuções com o SGA. Assim como nas execuções com a Aplicação CPU *Bound* e com a aplicação APS_s, as execuções da aplicação de Gauss com o SGA obtiveram melhores desempenhos quando comparadas com as execuções estáticas no modo tcp e lamd. Ao contrário da Aplicação CPU *Bound* e da aplicação APS_s, a aplicação de Gauss envolve troca de mensagens ao longo da execução, o que poderia ter acarretado em um custo maior do SGA para o encaminhamento de mensagens quando comparado com os modos estáticos, o que acabou não acontecendo.

Tabela 4.8: Comparação dos tempos de execução da aplicação de Gauss nos modos SGA, tcp e lamd na arquitetura com 11 processadores.

N	Modo	List Scheduling
54	tcp	5,03
	lamd	2,46
	SGA	1,97
104	tcp	11,87
	lamd	4,90
	SGA	3,98
377	tcp	151,41
	lamd	28,16
	SGA	16,11
495	tcp	254,58
	lamd	49,20
	SGA	22,60
740	tcp	ne
	lamd	93,25
	SGA	38,22
989	tcp	ne
	lamd	194,10
	SGA	57,29
10010	tcp	ne
	lamd	ne
	SGA	1761,48

Experimentos com as arquiteturas de 3 e 11 máquinas foram realizados com o objetivo de avaliar o grau de intrusão imposto pelo monitoramento da aplicação. Os resultados obtidos mostram que a intrusão do processo de monitoramento pode ser considerado desprezível. Isso se deve ao fato de que a camada de monitoramento (segunda camada) do SGA EasyGrid objetiva apenas coletar e fornecer dados necessários para um melhor comportamento das camadas acima. Os resultados apresentados na Tabela 4.9 referem-se à execução da Aplicação CPU *Bound* na arquitetura com 11 máquinas. Na Aplicação CPU *Bound* não ocorrem trocas de mensagens ao longo da execução; como conseqüência, o monitoramento refere-se apenas à coleta de informações referentes às operações básicas do MPI, tais como o MPI_Init() e o MPI_Finalize(). É importante ressaltar que o monitoramento de toda a aplicação, ou seja, monitoramento de funções como o MPI_Send() e MPI_Recv(), é opcional, assim como o monitoramento dos processos gerenciadores. Na segunda coluna da tabela, os campos Sem, AP, ALL referem-se respectivamente a execuções sem monitoramento (nenhuma função MPI é monitorada, com exceção do MPI_Finalize()), execuções com o monitoramento apenas dos processos da

aplicação (no caso da Aplicação CPU *Bound* apenas o MPI_Init() e o MPI_Finalize()) e execuções com o monitoramento de ambos os processos, da aplicação e gerenciadores.

Tabela 4.9: Grau de intrusão do monitor em execuções com a Aplicação CPU *Bound* na arquitetura com 11 processadores.

		Granularidade da Tarefa (s)			
N	Monitor	0,01	0,1	0,5	1,0
50	Sem	0,88	1,30	3,31	5,81
	AP	0,91	1,69	3,32	5,83
	ALL	0,94	1,73	3,32	5,83
100	Sem	1,05	1,80	5,79	10,80
	AP	1,11	1,84	5,85	10,91
	ALL	1,15	1,86	5,86	10,93
376	Sem	2,03	4,55	18,57	36,11
	AP	2,08	4,58	18,68	36,17
	ALL	2,17	4,59	18,64	36,17
500	Sem	2,54	5,75	24,19	47,25
	AP	2,55	5,77	24,25	47,32
	ALL	2,58	5,87	24,24	47,31
738	Sem	3,37	8,09	35,25	69,47
	AP	3,50	8,11	35,42	69,53
	ALL	3,58	8,36	35,46	69,52
1000	Sem	4,33	10,73	47,61	93,72
	AP	4,37	10,93	47,70	93,89
	ALL	4,62	11,03	47,88	93,89
10000	Sem	39,46	100,22	469,22	930,57
	AP	46,18	107,85	474,10	941,20
	ALL	48,12	110,45	477,19	933,92

A Tabela 4.10 apresenta o percentual de intrusão do monitoramento em execuções da Aplicação CPU *Bound* na arquitetura com 11 processadores. Assim como na Tabela 4.9, os campos AP e ALL representam respectivamente execuções com monitoramento apenas da aplicação e execuções com monitoramento da aplicação e dos processos gerenciadores. O percentual de intrusão é calculado com base nos valores obtidos com execuções sem monitoramento. A partir dos resultados apresentados, é possível concluir que a sobrecarga de monitoramento diminui à medida que o custo computacional (granularidade) das tarefas aumenta. Quanto maior o custo computacional da tarefa, menos significativa é a sobrecarga de monitoramento. Para as execuções com 10.000 tarefas e granularidade de 0,01s, a intrusão da camada de moniramento do SGA EasyGrid foi de aproximadamente 20%. O alto grau de intrusão nesse caso é justificado pelo grande número de tarefas com baixa granularidade, permitindo que a sobrecarga de monitoramento sobressaia em relação ao tempo total de execução. Entretanto, para o mesmo número de tarefas, a

intrusão cai consideravelmente à medida que o custo computacional das tarefas aumenta. Isso pode ser observado nas execuções com 10000 tarefas e granularidade 1s, onde o grau de intrusão não chegou a 0,4%.

Tabela 4.10: Percentual de intrusão do monitoramento em execuções da Aplicação CPU *Bound* na arquitetura com 11 máquinas.

		Granularidade da Tarefa (s)			
N	Monitor	0,01	0,1	0,5	1,0
50	AP	3,41%	1,56%	0,30%	0,34%
	ALL	6,82%	1,56%	0,30%	0,34%
100	AP	5,71%	2,22%	1,04%	1,02%
	ALL	9,52%	3,33%	1,21%	1,20%
376	AP	2,46%	0,66%	0,59%	0,17%
	ALL	6,90%	0,88%	0,38%	0,17%
500	AP	0,39%	0,35%	0,25%	0,15%
	ALL	1,57%	2,09%	0,21%	0,13%
738	AP	3,86%	0,25%	0,48%	0,09%
	ALL	6,23%	3,34%	0,60%	0,07%
1000	AP	0,92%	1,86%	0,19%	0,18%
	ALL	6,70%	2,80%	0,57%	0,18%
10000	AP	17,03%	7,61%	1,04%	1,14%
	ALL	21,95%	10,21%	1,70%	0,36%

Apesar de terem sido utilizadas nos experimentos apenas a Aplicação CPU *Bound*, onde as únicas funções monitoradas são o MPI_Init() e MPI_Finalize(), é possível tirar algumas conclusões em relação à utilidade e à intrusão da camada de monitoramento do SGA. Com o monitoramento apenas da função MPI_Finalize(), é possível obter informações sobre a máquina em que estava executando o processo monitorado no momento do término da sua execução. Entre as informações que podem ser coletadas estão a carga da máquina e o número de processos da aplicação em execução no momento. Logo, é possível distingüir a carga da aplicação da carga dos demais usuários. Essas informações podem ser utilizadas por políticas de escalonamento dinâmico para tomada de decisão em relação a possíveis redistribuições de tarefas. E o mais importante: a sobrecarga de monitoramento pode ser considerada baixa em relação ao tempo total de execução da aplicação.

4.5 Execuções em um Ambiente Grid Real

Nesta seção são apresentados resultados obtidos com execuções da aplicação dos térmions com 1.000 tarefas em uma arquitetura com 61 processadores, ilustrada na Figura 4.11.

A arquitetura utilizada envolve máquinas da UFF e PUC-Rio distribuídas em 5 *sites*. Os *sites* são compostos respectivamente por 12, 8, 8, 8 e 23 processadores. A divisão de processadores por *site* foi feita com base na velocidade da rede que interliga tais processadores. Os testes não foram realizados em modo exclusivo, ou seja, no momento das execuções o ambiente estava sendo compartilhado com outros usuários.

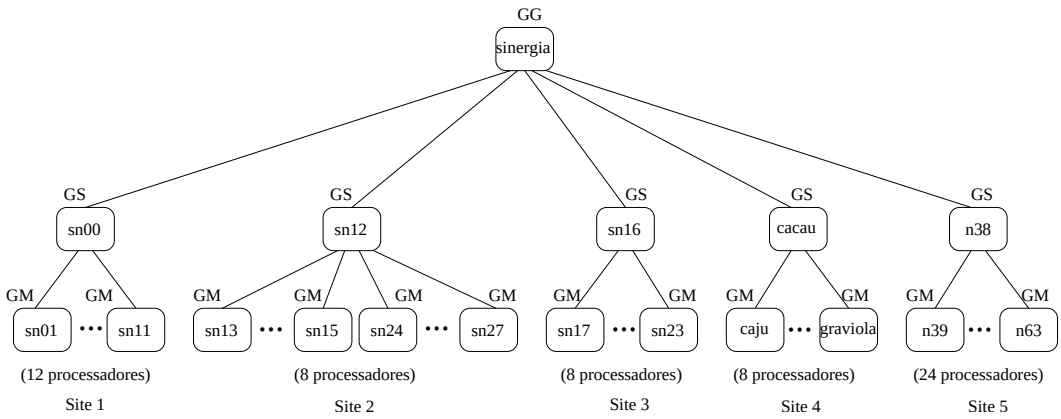


Figura 4.11: Arquitetura com 61 máquinas.

O objetivo dos experimentos realizados é mostrar o comportamento do SGA proposto em execuções envolvendo um ambiente Grid real. Os resultados apresentados na Figura 4.12 mostram a necessidade de algoritmos dinâmicos eficientes capazes de adaptar a execução da aplicação às mudanças ocorridas no ambiente. Esses resultados foram obtidos a partir de uma das execuções da aplicação dos térmions com 1.000 tarefas, com o escalonador estático utilizando o algoritmo *round robin*, na arquitetura descrita na Figura 4.11. A partir dos resultados apresentados, é possível observar a necessidade de um algoritmo dinâmico eficiente capaz de balancear a carga dos *sites*. A utilização de um algoritmo estático mais elaborado também poderia levar a uma queda no tempo total de execução da aplicação, porém apenas um escalonador dinâmico poderia ser capaz de, em tempo de execução, redistribuir as tarefas da aplicação de maneira a minimizar os efeitos das mudanças de carga ocorridas no ambiente Grid.

A Figura 4.13 ilustra uma arquitetura com 40 máquinas subdivididas em 5 *sites* com 12, 8, 8, 3 e 8 processadores, respectivamente. Todos os testes foram realizados com as máquinas em modo não-exclusivo, ou seja, aplicações de usuários distintos estavam em execução durante a realização dos experimentos. Assim como na arquitetura com 61 máquinas, os experimentos foram realizados com a versão do SGA apenas com o escalonador estático.

Os resultados apresentados na Figura 4.14 foram obtidos através de uma série de execuções da aplicação dos térmions com 1000 tarefas, utilizando o SGA com escalonador

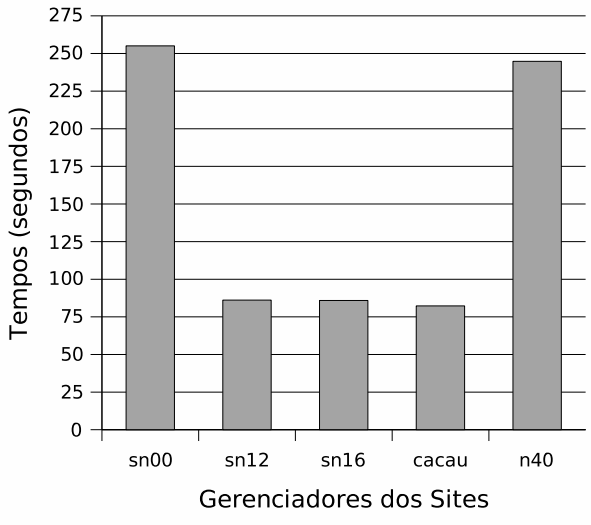


Figura 4.12: Tempos de execução de cada GS obtidos em um dos experimentos realizados com a aplicação dos térmions com 1000 tarefas na arquitetura com 61 processadores.

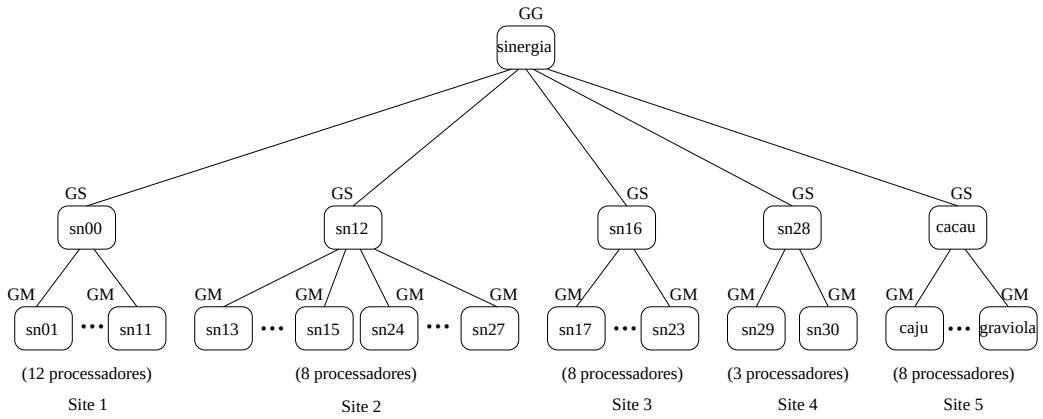


Figura 4.13: Arquitetura com 40 máquinas.

estático. O desbalanceamento ocorre entre *sites* e dentro de um mesmo *site*. O desbalanceamento entre *sites* pode ser observado na Figura 4.14 e é representado pelos tempos de execução dos gerenciadores de cada *site*. O desbalanceamento dentro de um mesmo *site* é identificado a partir do tempo total de execução de cada gerenciador da máquina, como ilustrado nas Figuras 4.15 e 4.16, que representam os tempos obtidos por cada gerenciador da máquina de dois *sites* da arquitetura utilizada. Esses resultados comprovam a necessidade de políticas de escalonamento dinâmico eficientes capazes de realizar redistribuição local (dentro do *site*) e global (entre *sites*) de tarefas, de maneira a ajustar a execução da aplicação às variações de carga ocorridas nas máquinas do Grid.

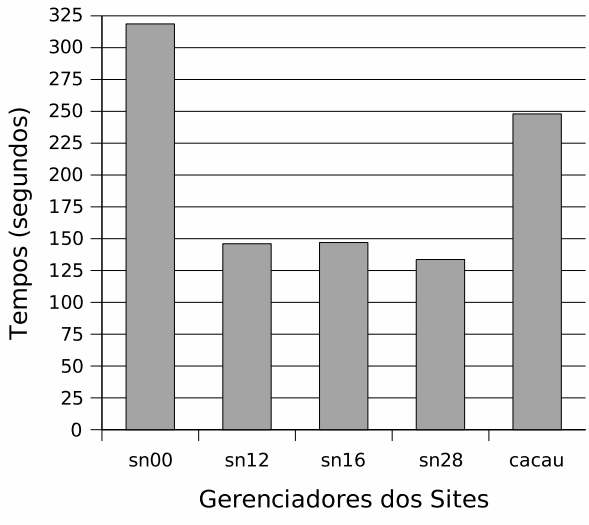


Figura 4.14: Tempos de execução (em segundos) dos gerenciadores dos *sites* em uma execução da aplicação dos térmions com 1000 na arquitetura com 40 processadores.

4.6 Execuções Concorrentes de Aplicações Smart G-App

O principal objetivo dos experimentos apresentados nesta seção é avaliar a execução concorrente de mais de uma aplicação Smart G-App em máquinas comuns do ambiente Grid. Nos testes realizados, foi utilizada a aplicação dos térmions com 1000 tarefas executando em duas arquiteturas com 9 e 10 máquinas, ilustradas na Figura 4.17. As duas arquiteturas apresentam máquinas comuns, sendo disparados em cada uma dessas máquinas processos gerenciadores diferentes. Inicialmente, foram analisadas execuções individuais da aplicação dos térmions em cada uma das arquiteturas, com o ambiente em modo exclusivo. Essas execuções são classificadas nessa seção como individuais, uma vez que, durante os experimentos, não havia nenhuma outra execução em andamento nas máquinas que compõem a arquitetura. Em seguida, foram realizadas execuções concorrentes da aplicação dos térmions nas duas arquiteturas apresentadas. Essas execuções concorrentes foram classificadas como coletivas, ou seja, durante os experimentos haviam duas aplicações Smart G-App em execução nas máquinas das arquiteturas apresentadas.

Para os experimentos individuais, apenas um processo gerenciador foi disparado em cada uma das máquinas da arquitetura. Entretanto, para os experimentos coletivos, por existirem máquinas comuns às duas arquiteturas, foi disparado em algumas máquinas mais de um processo gerenciador, de tipos diferentes. Por exemplo, na máquina sn11 foi disparado um processo GM para execução da aplicação Smart G-App na arquitetura com 9 máquinas e um processo GS para execução concorrente da aplicação na arquitetura com

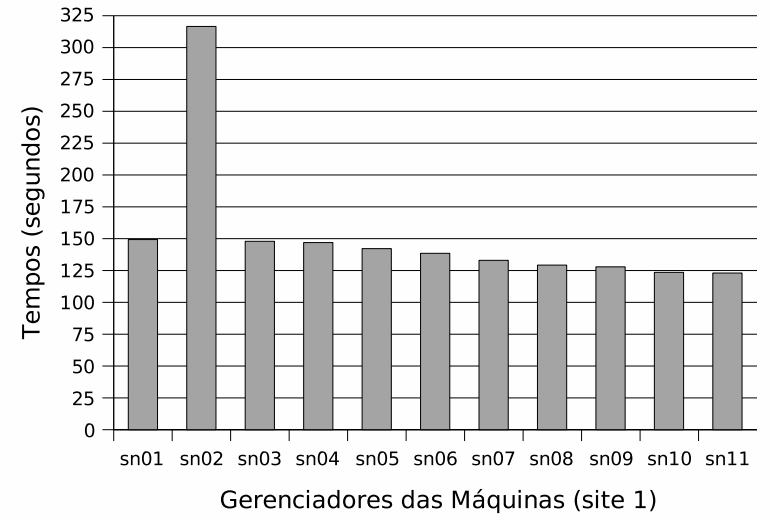


Figura 4.15: Tempos de execução (em segundos) dos gerenciadores das máquinas do *site* 1 em uma execução da aplicação dos térmions com 1000 na arquitetura com 40 processadores.

10 máquinas.

Os resultados obtidos com os experimentos individuais e coletivos são apresentados na Tabela 4.11, onde são comparados tempos de execuções do SGA com o escalonador estático e com o escalonador híbrido 1. O escalonador Híbrido 1 utiliza as políticas de escalonamento dinâmico de tarefas local e global apresentadas nas Seções 3.6.1.1 e 3.6.1.1 do Capítulo 3. Nos experimentos individuais, a execução Smart G-App com escalonador híbrido 1 apresentou resultados piores que as execuções com apenas o escalonador estático. Isso se deve ao fato de que, com o escalonador híbrido 1, o escalonador dinâmico realiza redistribuições desnecessárias de tarefas durante a execução da aplicação. É importante ressaltar que o escalonador dinâmico utilizado nesses experimentos não foi desenvolvido com o objetivo de atingir melhora nos tempos de execução de uma aplicação. Ele foi desenvolvido apenas com o intuito de validar o protocolo de redistribuição de tarefas desenvolvido para o SGA. Logo, as decisões referentes ao re-escalonamento de tarefas são tomadas sem levar em consideração características da aplicação e do ambiente. Apesar de não ser inteligente, o escalonador híbrido 1 apresentou resultados satisfatórios quando comparado com o escalonador estático, nas execuções coletivas. Esses resultados mostram a importância de algoritmos de escalonamento dinâmico em ambientes compartilhados como os Grids Computacionais.

A Tabela 4.12 apresenta o tempo de execução de cada processador durante experimentos coletivos com a aplicação dos térmions nas arquiteturas com 9 e 10 processadores. Através dos resultados obtidos, fica claro o desbalanceamento de carga apresentado pelas

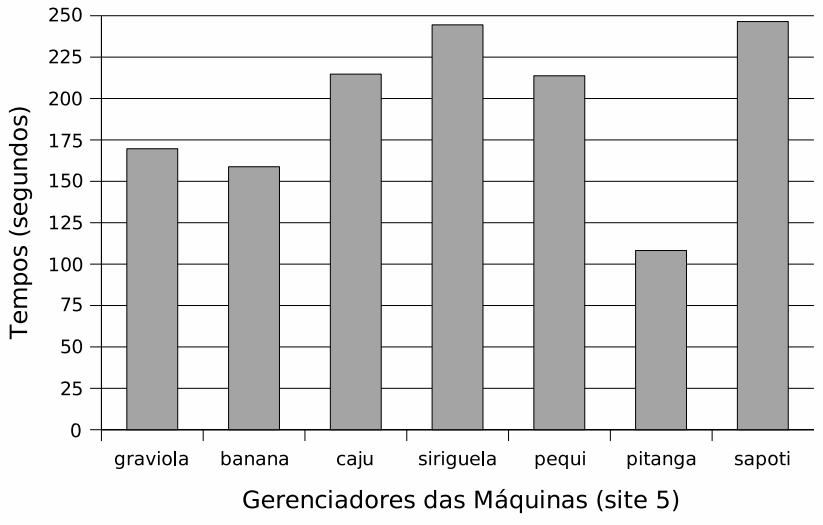


Figura 4.16: Tempos de execução (em segundos) dos gerenciadores das máquinas do *site* 5 em uma execução da aplicação dos térmions com 1000 na arquitetura com 40 processadores.

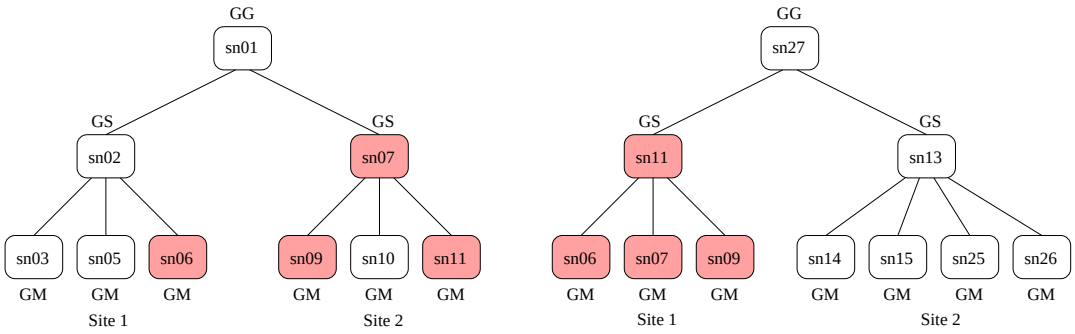


Figura 4.17: Arquitetura com 9 e 10 máquinas.

execuções com a versão do SGA utilizando apenas o escalonamento estático. Ou seja, as tarefas da aplicação são atribuídas aos processadores em tempo de compilação e nenhuma redistribuição de tarefas é realizada ao longo da execução. As execuções coletivas com o SGA utilizando o escalonamento híbrido 1 apresentaram resultados melhores, havendo uma tentativa do escalonador dinâmico de adaptar a execução da aplicação às mudanças de carga ocorridas no ambiente.

A Tabela 4.13 apresenta a distribuição de tarefas entre as máquinas das arquiteturas com 9 e 10 processadores em uma execução coletiva da aplicação dos térmions, com o SGA utilizando o escalonador Híbrido 1. A instância da aplicação dos térmions adotada nos experimentos apresentados possui um total de 1000 tarefas distribuídas uniformemente pelo escalonamento estático *round robin* entre os processadores da arquitetura. Com exceção das máquinas em que foram disparados os Gerenciadores Globais nas arquiteturas com 9 (sn01) e 10 (sn27) processadores, tendo sido escalonado apenas o processo 0 da

Tabela 4.11: Execução da aplicação dos térmions com 1000 tarefas nas arquiteturas com 9 e 10 processadores.

	1 Aplicação		2 Aplicações	
Escalonador	9 procs.	10 procs.	9 procs.	10 procs.
Estático	592,38	540,84	1133,34	1086,84
Híbrido 1	647,26	631,76	967,93	963,55

Tabela 4.12: Tempo (segundos) gasto por cada processador para execução de um conjunto de tarefas durante experimentos coletivos com a aplicação dos térmions nas arquiteturas com 9 e 10 processadores.

	Estático		Híbrido 1	
Máquinas	9	10	9	10
sn01	1132,36	-	967,86	-
sn02	1038,90	-	967,63	-
sn03	541,06	-	967,47	-
sn05	541,41	-	967,34	-
sn06	1038,82	982,49	967,07	962,21
sn07	1132,22	1065,19	967,59	962,34
sn09	1037,96	982,86	967,40	962,29
sn10	541,78	-	967,10	-
sn11	1120,69	1068,30	966,90	962,44
sn13	-	540,12	-	962,34
sn14	-	481,24	-	962,24
sn15	-	481,19	-	962,16
sn25	-	480,72	-	962,25
sn26	-	480,07	-	962,16
sn27	-	1068,41	-	962,68

aplicação dos térmions, responsável por distribuir os dados iniciais entre os demais processos da aplicação e coletar os resultados no final da execução. Os valores entre parênteses correspondem ao número inicial de tarefas atribuído a cada uma das máquinas pelo algoritmo de escalonamento estático. Já os valores fora dos parênteses representam o total de tarefas executadas por cada um desses processadores no final da execução da aplicação. Com base nos valores apresentados na tabela, é possível observar a redistribuição de tarefas entre os processadores da arquitetura ao longo da execução. Os processadores a que foram atribuídos processos gerenciadores das duas execuções tiveram algumas de suas tarefas redistribuídas entre máquinas menos sobrecarregadas. A política de escalonamento Híbrido 1, apesar de não ser muito eficiente, foi capaz de identificar a sobrecarga de alguns processadores da arquitetura e acionar os protocolos de redistribuição de tarefas local e global apresentados nas Seções 3.6.1 e 3.6.2 do Capítulo 3, ajustando a execução

das aplicações às mudanças sofridas pelo ambiente de execução.

Tabela 4.13: Distribuição de tarefas entre as máquinas das arquiteturas com 9 e 10 processadores nas execuções coletivas da aplicação dos térmions com o SGA utilizando o escalonador Híbrido 1.

	Híbrido 1	
Máquinas	9	10
sn01	1 (1)	-
sn02	125 (125)	-
sn03	198 (125)	-
sn05	162 (125)	-
sn06	94 (125)	78 (111)
sn07	125 (125)	76 (111)
sn09	90 (125)	78 (111)
sn10	118 (125)	-
sn11	88 (125)	112 (112)
sn13	-	111 (111)
sn14	-	111 (111)
sn15	-	144 (111)
sn25	-	111 (111)
sn26	-	179 (111)
sn27	-	1 (1)

4.7 Redistribuição de Tarefas

Os experimentos apresentados nesta seção mostram a capacidade do SGA EasyGrid de modificar em tempo de execução a alocação estática inicial das tarefas da aplicação, de maneira a melhorar o desempenho. No caso de aplicações *parameter sweep* e aplicações *bag of tasks*, onde as tarefas são geralmente independentes, as questões referentes à comunicação entre processos pode ser ignorada. Nesse caso, o problema de escalonamento pode ser visto como a redistribuição de tarefas, que foram alocadas a recursos que estão sobrecarregados, entre recursos menos carregados (balanceamento de carga). Entretanto, o processo mestre precisa continuar se comunicando com os processos escravos realocados, o que exige do SGA a capacidade de redirecionar mensagens referentes a tarefas da aplicação envolvidas em um processo de redistribuição.

Para avaliar os mecanismos de redistribuição de tarefas presentes no SGA EasyGrid, foi utilizada a aplicação dos térmions. Os experimentos apresentados a seguir foram realizados em uma arquitetura com 30 máquinas, ilustrada na Figura 4.18. As máquinas foram consideradas como pertencentes a um único *site*, constituído de 28 processadores

executando um processo Gerenciador da Máquina, uma máquina executando o Gerenciador do Site e outra executando o Gerenciador Global do Grid. Todos os experimentos foram realizados em modo exclusivo, ou seja, durante os testes não existiam aplicações de outros usuários em execução no ambiente Grid. Sendo assim, é possível controlar a execução de cargas externas, permitindo a avaliação de políticas de escalonamento dinâmico de tarefas.

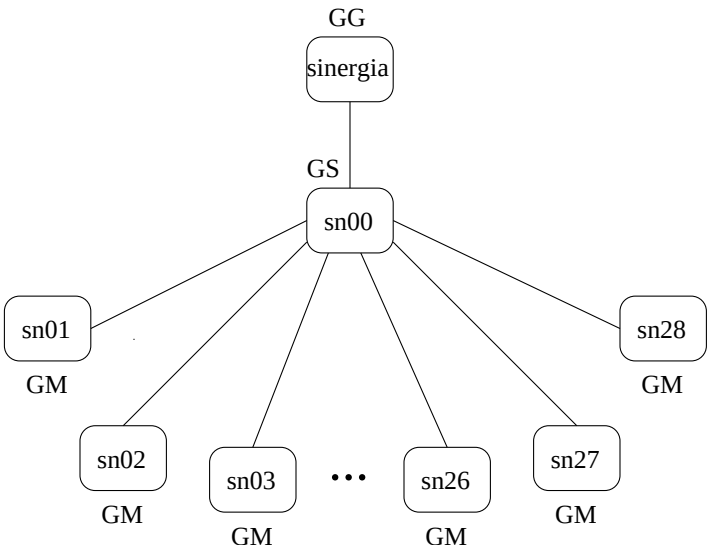


Figura 4.18: Arquitetura com 30 máquinas.

Para que fosse possível a ativação do escalonador dinâmico e a conseqüente redistribuição de tarefas, cargas externas adicionais foram disparadas em alguns dos processadores da arquitetura selecionados para execução da aplicação dos térmions. Foram considerados os seguintes cenários (C) de execução:

1. Nenhuma carga extra, ou seja, o ambiente Grid estava totalmente dedicado à execução da aplicação Smart G-App;

2. 25% dos processadores (7 máquinas) executando uma carga extra (um programa CPU-bound);

3. 25% dos processadores (7 máquinas) executando duas cargas extras (dois programas CPU-bound);

4. 50% dos processadores (14 máquinas) executando uma carga extra (um programa CPU-bound);

5. 75% dos processadores (21 máquinas) executando uma carga extra (um programa CPU-bound).

Tarefas da aplicação foram escalonadas inicialmente entre os 28 processadores gerenciados pelos processos GM. Apenas a tarefa mestre foi alocada ao processador gerenciado pelo GG. A Figura 4.19 apresenta os resultados obtidos com execuções da aplicação dos térmions com o escalonador estático, escalonador híbrido 1 e escalonador híbrido 2. Diz-se que a execução do SGA envolve apenas o escalonador estático quando a camada dos processos gerenciadores referente ao escalonamento dinâmico está inativa. O escalonador híbrido 1 envolve um escalonamento estático inicial e um escalonamento dinâmico baseado apenas no mecanismo de redistribuição apresentado na Seção 3.6 do Capítulo 2. Esse escalonador híbrido (híbrido 1) não foi desenvolvido com o objetivo de reduzir os tempos de execução de uma aplicação, e sim de validar os mecanismos de redistribuição de tarefas local e global propostos neste trabalho. O escalonador Híbrido 2, apresentado em [36], implementa uma política de escalonamento dinâmico, entre as máquinas de um *site*, mais eficiente do que a adotada pelo algoritmo Híbrido 1.

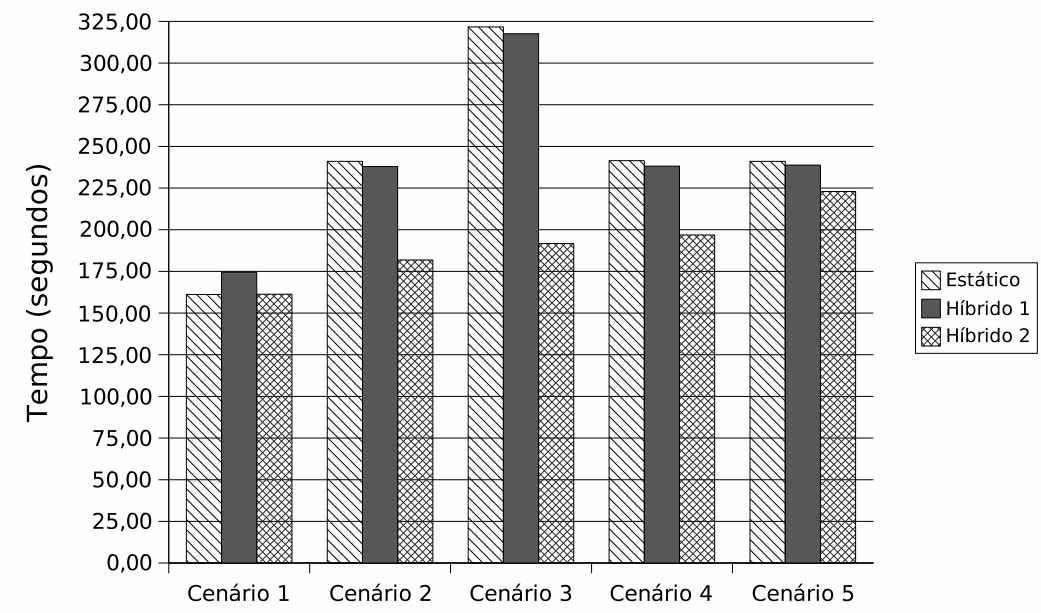


Figura 4.19: Comparação entre os escalonadores estático e híbridos (em segundos).

No escalonador Híbrido 2, todas as decisões são tomadas pelo GS com base em informações obtidas através de mensagens de monitoramento. Se o tempo desde a última mensagem de monitoramento exceder um limite de tempo pré-determinado (possivelmente devido ao longo tempo de execução de processos da aplicação ou ao fato de não existir nenhum processo em execução), o GM envia para o GS uma mensagem de *heartbeat* contendo informações obtidas do sistema operacional. Com base nas mensagens de monitoramento, o processo GS é capaz de estimar o tempo de execução restante das tarefas da aplicação que foram alocadas a cada uma das máquinas do *site* e que ainda não foram executadas.

Os tempos de execução apresentados na Figura 4.19 para o escalonador estático referem-se aos tempos do processador que levou mais tempo para executar as tarefas da aplicação. É possível observar a queda significativa no tempo de processamento (melhora no desempenho) da aplicação quando é utilizado o escalonador híbrido 2. No cenário 1, os tempos de execução da aplicação com o escalonador estático e com o escalonador híbrido 2 foram aproximadamente os mesmos, o que era esperado, uma vez que nenhuma mudança de carga ocorreu no ambiente de execução e as tarefas foram distribuídas igualmente entre os processadores homogêneos. O escalonador híbrido 1 mostrou-se pior que o escalonador estático no ambiente sem carga, devido a redistribuições desnecessárias de tarefas da aplicação realizadas pelo escalonador dinâmico.

No segundo cenário, tarefas da aplicação alocadas para o processador com carga extra perdem apenas 1/3 da CPU, uma vez que duas tarefas da aplicação são executadas concorrentemente. Entretanto, o escalonador dinâmico compensa tal perda pelo remapeamento de tarefas entre os processadores menos carregados, quando o escalonamento híbrido é utilizado. Nos cenários 3, 4 e 5 os tempos de execução resultantes são proporcionais à variação na carga dos processadores.

4.8 Viabilidade da Versão do SGA com Hierarquia de Gerenciadores de 2 Níveis

Como citado no início deste capítulo, apenas experimentos referentes à viabilidade foram realizados com a versão do SGA EasyGrid com 2 níveis de processos gerenciadores. Um estudo mais detalhado dessa abordagem fica como proposta para trabalhos futuros. Na versão de 2 níveis, todo o gerenciamento das máquinas de um *site* é realizado pelo Gerenciador do Site, inclusive a criação e gerenciamento de processos da aplicação atribuídos às máquinas de nível 2 (máquinas onde seriam disparados processos GM na versão de 3 níveis). Acredita-se que a sobrecarga de gerenciamento imposta ao Gerenciador do Site pode levar a aplicação Smart G-App a uma degradação no seu tempo total de execução. Nessa abordagem, todo o escalonamento dinâmico das tarefas atribuídas a um *site* passa a ser realizado pelo processo GS, diminuindo a flexibilidade do *middleware* em relação às políticas de escalonamento adotadas para cada máquina do *site*. Na versão de 3 níveis, diferentes políticas de escalonamento podem ser utilizadas por processos gerenciadores distintos. Como na versão de 2 níveis existe apenas um processo gerenciador por *site* (processo GS), apenas uma política de escalonamento dinâmico pode ser adotada na implementação atual.

Na Tabela 4.14 são apresentados resultados de execuções individuais da aplicação dos térmions com 1000 tarefas utilizando as abordagens do SGA com 2 e 3 níveis de processos gerenciadores. Os experimentos foram realizados em modo exclusivo nas arquiteturas com 9 e 10 processadores, ilustradas na Figura 4.17. Apenas o escalonador estático foi utilizado pelo SGA, ou seja, nenhuma política de escalonamento dinâmico estava ativa durante os experimentos. É possível observar que a abordagem de 2 níveis apresentou os melhores resultados em relação à abordagem de 3 níveis. Isso se deve ao fato de que, na versão de 2 níveis, a sobrecarga inicial para criação da hierarquia de processos gerenciadores é menor. Porém, sem um estudo mais aprofundado, não é possível afirmar qual seria o comportamento dessa versão em execuções de aplicações que envolvessem um custo mais alto de comunicação (maior número de mensagens). Acredita-se que a sobrecarga de mensagens poderia levar a uma degradação do desempenho do Gerenciador do Site. Também não foi possível avaliar o comportamento dessa abordagem com possíveis ativações do escalonador dinâmico. Os protocolos de redistribuição de tarefas local e global foram implementados, assim como na abordagem com 3 níveis. Entretanto, poucos experimentos foram realizados em ambientes não exclusivos, o que dificultou qualquer conclusão a respeito da eficiência dessa versão no processo de redistribuição de tarefas de uma aplicação.

Tabela 4.14: Tempos de execução (em segundos) da aplicação dos térmions com 1000 tarefas nas arquiteturas com 9 e 10 processadores utilizando as abordagens de 2 e 3 níveis do SGA.

Níveis	9 procs.	10 procs.
2	588,07	536,85
3	592,38	540,84

Execuções coletivas foram realizadas com a aplicação dos Térmions nas duas arquiteturas. Ou seja, duas aplicações Smart G-App estavam executando concorrentemente nas arquiteturas com 9 e 10 processadores. O objetivo desses experimentos foi avaliar a viabilidade de execuções concorrentes de aplicações Smart G-App utilizando a abordagem de 2 níveis do SGA. As Figuras 4.20 e 4.21 apresentam os tempos de execução (em segundos) dos Gerenciadores das Máquinas em um dos experimentos coletivos. Os resultados obtidos confirmam a necessidade de políticas de escalonamento dinâmico de tarefas capazes de ajustar os tempos de execução entre as máquinas de um *site* e entre *sites*.

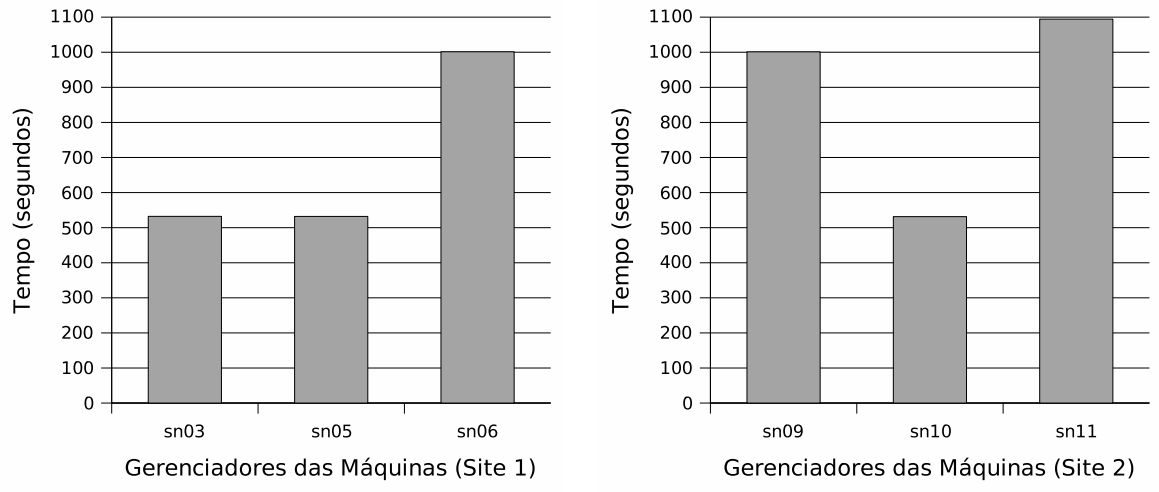


Figura 4.20: Tempos de execução (em segundos) dos Gerenciadores das Máquinas em um dos experimentos com a aplicação dos Térmions na arquitetura com 9 processadores.

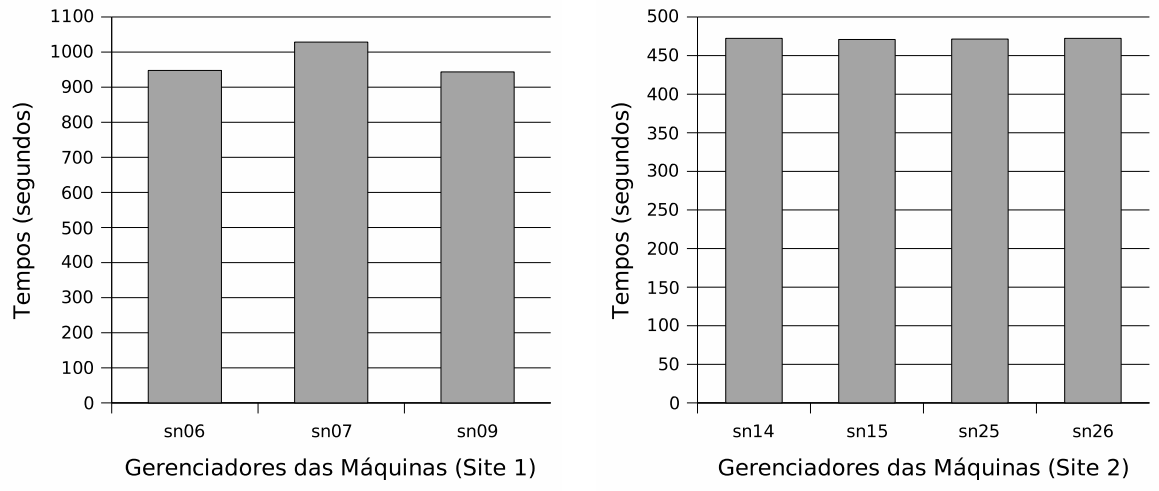


Figura 4.21: Tempos de execução (em segundos) dos Gerenciadores das Máquinas em um dos experimentos com a aplicação dos Térmions na arquitetura com 10 processadores.

4.9 Resumo

Neste capítulo foram apresentados resultados computacionais que mostram a viabilidade, escalabilidade e qualidade das execuções de aplicações Smart G-Apps em ambientes dinâmicos e heterogêneos como os Grids Computacionais. Resultados iniciais mostram a baixa intrusão imposta pelo sistema de gerenciamento proposto e sinalizam a necessidade do desenvolvimento de políticas de escalonamento dinâmico de tarefas específicas para cada classe de aplicação. A eficiência da criação dinâmica de processos MPI fica evidente nas comparações entre os tempos de execução das aplicações com o SGA proposto e os modos estáticos do MPI, tcp e lamd.

Políticas eficientes de escalonamento dinâmico de tarefas têm sido desenvolvidas e resultados satisfatórios têm sido obtidos. Além de escalonamento dinâmico, políticas de tolerância a falhas estão sendo empregadas no SGA EasyGrid, mostrando que a partir da infra-estrutura proposta neste trabalho é possível executar aplicações MPI de maneira eficiente e robusta. Resultados iniciais foram apresentados em [36] e [25].

Capítulo 5

Conclusões e Trabalhos Futuros

A principal contribuição deste trabalho foi o desenvolvimento de um sistema de gerenciamento hierárquico distribuído para execução de aplicações MPI de grande escala em Grids Computacionais de maneira eficiente e robusta.

Muito se tem pesquisado em relação a *middlewares* para Grids que facilitem o desenvolvimento e execução de aplicações em tais ambientes. Atualmente, a maioria dos sistemas propostos tem como base sistemas de gerenciamento de recursos (SGRs). Sistemas gerenciadores de recursos utilizam apenas informações sobre o sistema para alocação de processos de uma aplicação às máquinas do Grid. Para aplicações paralelas, apenas a utilização de informações do sistema pode não ser suficiente para que seja feita a melhor alocação das tarefas. Uma outra abordagem possível, é a utilização de sistemas de gerenciamento de aplicações (SGA) capazes de tomar decisões com base em características específicas de cada aplicação. Esses sistemas podem ser embutidos ou não na aplicação. Quando embutidos na aplicação, eles são capazes de fornecer portabilidade aumentando o número de recursos Grid que podem ser utilizados para execução da aplicação.

O sistema de gerenciamento proposto neste trabalho é específico para aplicações LAM/MPI. Essa escolha foi motivada pelo grande número de aplicações MPI já existentes em diversas áreas da ciência. O SGA EasyGrid é embutido na aplicação em tempo de compilação, sem que seja necessária nenhuma alteração no código original do usuário. As funções MPI utilizadas pela aplicação do usuário são substituídas por funções pertencentes à biblioteca MEGmpi.h (*wrapper*) também proposta neste trabalho. Essas funções permitem a troca de mensagens entre tarefas da aplicação, através de processos gerenciadores, independente das máquinas em que estas serão executadas.

Os processos gerenciadores foram desenvolvidos com base na visão hierárquica dos ambientes Grids, podendo o SGA adotar uma abordagem de 2 ou 3 níveis. Na abordagem

de dois níveis, são disparados um Gerenciador Global responsável por controlar toda execução no ambiente Grid e Gerenciadores dos Sites, um para cada *site*, responsáveis por controlar a execução de tarefas da aplicação atribuídas às máquinas do *site*. Na abordagem de três níveis, além do GG e dos GS's são disparados processos GM em cada máquina do *site*. Cada processo GM é responsável pela criação e gerenciamento da execução de processos da aplicação atribuídos à máquina local. A abordagem de 3 níveis, permite que informações sobre a aplicação e sobre os recursos sejam coletadas de todas as máquinas do Grid, devido à existência de processos gerenciadores executando em cada uma dessas máquinas.

O projeto de cada processo gerenciador tem como base uma arquitetura em camadas composta por uma camada de gerenciamento de processos, responsável pela criação dinâmica de processos MPI e pelo redirecionamento de mensagens; uma camada de monitoramento da aplicação, capaz de coletar e fornecer informações sobre o sistema e a aplicação em execução; uma camada de escalonamento dinâmico, responsável pelo re-escalonamento de tarefas da aplicação; e finalmente, uma camada de tolerância a falhas, que objetiva detectar e tratar falhas no ambiente de execução.

A camada de escalonamento dinâmico oferece dois protocolos de redistribuição de tarefas: local e global. A redistribuição local ocorre apenas dentro de um *site*, enquanto que a redistribuição global envolve máquinas de *sites* distintos. Os protocolos de redistribuição permitem a re-alocação de processos da aplicação, que ainda não foram disparados, e o redirecionamento das mensagens endereçadas às tarefas envolvidas no processo de redistribuição.

A principal característica do sistema de gerenciamento proposto é a sua capacidade de executar aplicações MPI pré-existentes, desenvolvidas para *cluster*, em um ambiente dinâmico de forma controlada e eficiente. Os resultados apresentados mostram a rapidez das execuções de aplicações Smart G-App quando comparadas com as suas versões estáticas originais. A sobrecarga da criação dinâmica de processos cai significativamente, à medida que cresce o número de tarefas da aplicação e o custo computacional de cada uma dessas tarefas. O monitoramento da aplicação também não acrescentou nenhum custo significativo à execução de tais aplicações, uma vez que a camada de monitoramento concentra-se apenas na obtenção de informações que são consideradas extremamente necessárias para as políticas de escalonamento dinâmico de tarefas e tolerância a falhas.

Outra vantagem do SGA proposto, observada nos experimentos realizados, é a escala-

bilidade do *middleware* em relação ao número de tarefas da aplicação. Execuções com os modos estáticos do MPI apresentaram limitações consideráveis em relação ao número de processos em execução concorrentemente, uma vez que, no modo estático do MPI, todos os processos de uma aplicação são disparados ao mesmo tempo. Já os experimentos com o SGA não apresentaram tais limitações. Em aplicações Smart G-App, o número de processos em execução simultaneamente é controlado pelo processo gerenciador da máquina local. Esse controle é possível através da utilização de funções de gerenciamento dinâmico de processos presentes na versão LAM da biblioteca MPI.

Uma das necessidades observadas no desenvolvimento do SGA é a implementação de um algoritmo capaz de controlar o fluxo de mensagens trocadas entre os processos gerenciadores. O gerenciamento de mensagens evitará possíveis gargalos provocados pela exaustão dos *buffers* de recebimento utilizados pelo MPI.

A estrutura em camadas de cada processo gerenciador permite que diferentes algoritmos de escalonamento dinâmico sejam desenvolvidos levando em consideração características específicas de cada um dos níveis da hierarquia de gerenciadores. Ou seja, em uma execução de uma aplicação Smart G-App cada processo gerenciador (GM, GS e GG) pode adotar um algoritmo de escalonamento dinâmico diferente. A abordagem hierárquica distribuída, adotada pelo SGA EasyGrid, abre um amplo caminho para o estudo de algoritmos de escalonamento dinâmico de tarefas.

Algoritmos de escalonamento dinâmico de tarefas que levam em consideração a carga do sistema e informações sobre a execução da aplicação já estão sendo desenvolvidos para aplicações do tipo *bag-of-task* (mestre-escravo e *parameter-sweep*). No futuro, algoritmos que levem em consideração a precedência entre as tarefas (GAD), e conseqüentemente o custo de comunicação entre as mesmas, serão desenvolvidos e acrescentados à biblioteca de escalonamento dinâmico do *Framework* EasyGrid.

Até o momento, a camada de tolerância a falhas dos processos gerenciadores é capaz de identificar falha em um processo da aplicação através do uso de sinais da biblioteca MPI (MPIL_Signal()). Após identificada a falha, a camada de gerenciamento de processos é acionada e um novo processo é disparado no mesmo *host* em que o processo estava executando antes de ocorrer a falha. A integração entre as camadas de escalonamento dinâmico de tarefas e tolerância a falhas já está sendo realizada com o objetivo de permitir a re-execução de processos da aplicação atribuídos a uma máquina do ambiente Grid que tenha sofrido algum tipo de falha. Resultados iniciais apresentados em [36] mostram que o custo para identificação de falha em um processo da aplicação, através do uso da função

MPIL_Signal(), e a conseqüente re-criação do processo que falhou podem ser considerados não muito altos.

Algoritmos específicos, que identifiquem e tratem falhas em processos gerenciadores, serão estudados de maneira a identificar o tipo de abordagem que mais se adapta ao tratamento de falhas para cada um dos processos gerenciadores de acordo com suas funcionalidades dentro de cada nível hierárquico. Além da recuperação de processos gerenciadores, novas metodologias estão sendo estudadas para a identificação e recuperação de processos da aplicação. Entre elas, a técnica de *checkpointing/restart* e *log* de mensagens [66] que podem inclusive, permitir a implementação da migração de processos em execução.

Para o desenvolvimento do SGA proposto, foi utilizada a versão LAM da biblioteca MPI. O LAM/MPI, a partir da versão 7.0, passou a oferecer suporte para a execução de aplicações paralelas em Grids Computacionais. Ao contrário do MPICH, o LAM/MPI implementou as funções de gerenciamento dinâmico de processos propostas pelo padrão MPI2. No futuro, será avaliada a eficiência da versão MPICH-G2 (MPICH habilitado para Grid) em relação à versão LAM/MPI. Entretando, é preciso esperar que as funcionalidades referentes ao gerenciamento dinâmico de processos sejam incorporadas a essa versão do MPI.

O desenvolvimento e a utilização de aplicações paralelas de grande escala e a execução dessas aplicações em ambientes Grids cada vez maiores, também estão previstos em trabalhos futuros.

Referências

[1] APST applications. http://grail.sdsc.edu/projects/apst/applications.html.

[2] Cactus. http://www.cactuscode.org/.

[3] Condor-G. http://www.cs.wisc.edu/condor/manual/v6.4/5_3Condor_G.htm.

[4] GridLab. http://www.gridlab.org/.

[5] Hawkeye. http://www.cs.wisc.edu/condor/hawkeye/.

[6] Legion. http://www.cs.virginia.edu/~legion/.

[7] Message passing interface forum. http://www.mpi-forum.org/.

[8] Network Weather Service. http://nws.cs.ucsb.edu/.

[9] OurGrid Project. http://www.ourgrid.org/.

[10] The Globus Alliance. http://www.globus.org/.

[11] Top500 supercomputer sites. http://www.top500.org/list/2004/11/.

[12] Agbaria, A., and Friedman, R. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *Cluster Computing* (July 2003), Springer Science+Business Media B.V., Formerly Kluwer Academic Publishers B.V., pp. 227–236.

[13] Allen, G., Benger, W., Dramlitsch, T., Goodale, T., Hege, H.-C., Lanfermann, G., Merzky, A., Radke, T., and Seidel, E. Cactus grid computing: Review of current development. *Lecture Notes in Computer Science 2150* (August 2001), 817–.

[14] Allen, G., Davis, K., Dolkas, K., Doulamis, N., Goodale, T., Kielmann, T., Merzky, A., Nabrzyski, J., Pukacki, J., Radke, T., Russell, M., Seidel, E., Shalf, J., and Taylor, I. Enabling applications on the grid: A GridLab overview, 2003.

[15] Andrade, N., Cirne, W., Brasileiro, F., and Roisenberg, P. Ourgrid: An approach to easily assemble grids with equitable resource sharing. Springer-Verlag GmbH, pp. 61–86.

[16] Andreozzi, S., Bortoli, N. D., Fantinel, S., Ghiselli, A., Tortone, G., and Vistoli, C. GridICE: A monitoring service for the grid, October 2003.

[17] Asadzadeh, P., Buyya, R., Kei, C. L., Nayar, D., and Venugopal, S. Global grids and software toolkits: A study of four grid middleware technologies. *High Performance Computing: Paradigm and Infrastructure* (2004).

[18] Baker, M., Buyya, R., and Laforenza, D. Grids and grid technologies for wide-area distributed computing. *Software: Practice and Experience 32*, 15 (2002), 1437–1466.

[19] Baker, M. A., and Smith, G. C. GridRM: An extensible resource monitoring system. In *Proceedings of the Fifth IEEE International Cluster Computing Conference* (December 2004), pp. 207–214.

[20] Balaton, Z., Kacsuk, P., and Podhorszki, N. Application monitoring in the grid with GRM and PROVE. In *ICCS '01: Proceedings of the International Conference on Computational Science* (London, United Kingdom, 2001), Springer-Verlag, pp. 253–262.

[21] Balis, B., Bubak, M., Funika, W., Szepieniec, T., Wismller, R., and Radecki, M. Monitoring grid applications with grid-enabled omis monitor. In *Proceedings of the Grid Computing: First European Across Grids Conference* (Santiago de Compostela, Spain, 2004), vol. 2970, Springer-Verlag GmbH, pp. 230–239.

[22] Batchu, R., Neelamegam, J., Dui, Z., Beddhua, M., Skjellum, A., Dandass, Y., and Apten. MPI/FT (TM): Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *First IEEE International Symposium of Cluster Computing and the Grid* (Melbourne, Australia, 2001).

[23] Berman, F., and Wolski, R. The AppLeS project: A status report, May 1997.

[24] Boeres, C., Lima, A., and Rebello, V. Hybrid task scheduling: Integrating static and dynamic heuristics. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing* (Washington, DC, United States, November 2003), IEEE Computer Society, pp. 199–206.

[25] Boeres, C., Nascimento, A., Rebello, V., and Sena, A. Hierarchical self-scheduling for MPI applications executing in computational grids. *3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing* (to appear).

[26] Boeres, C., and Rebello, V. E. F. EasyGrid: Towards a framework for the automatic grid enabling of MPI applications. In *Middleware Workshops* (2003), pp. 256–260.

[27] Bonnassieux, F., Harakaly, R., and Primet, P. MapCenter: An open grid status visualization tool, September 2002.

[28] Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., and Selikhov, A. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. *Proceedings of ACM/IEEE SuperComputing Conference (SC)* (November 2002), 29–47.

[29] Buyya, R., Abramson, D., and Giddy, J. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid, 2000.

[30] Casanova, H., and Dongarra, J. NetSolve: A network server for solving computational science problems. Tech Report CS-96-328, Knoxville, Tennessee, United States, 1996.

[31] Casavant, T. L., and Kuhl, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering 14*, 2 (February 1988), 141–154.

[32] Chapin, S. J., Katramatos, D., Karpovich, J., and Grimshaw, A. S. The legion resource management system. In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Springer Verlag, 1999, pp. 162–178.

[33] Cirne, W., da Silva, D. P., Costa, L., Santos-Neto, E., Brasileiro, F. V., Sauvé, J. P., Silva, F. A. B., Barros, C. O., and Silveira, C. Running bag-of-tasks applications on computational grids: The MyGrid approach. In *Proceedings of the 32nd International Conference on Parallel Processing (ICPP)* (2003), IEEE Computer Society, pp. 407–417.

[34] Cooke, A., Gray, A. J. G., Ma, L., Nutt, W., Magowan, J., Oevers, M., Taylor, P., Byrom, R., Field, L., Hicks, S., Leake, J., Soni, M., Wilson, A., Cordenonsi, R., Cornwall, L., Djaoni, A., Fisher, S., Podhorszki, N., Coghlan, B., Kenny, S., and O'Callaghan, D. R-GMA: An information integration system for grid monitoring. In *Lecture Notes in Computer Science* (October 2003), vol. 2888.

[35] de Camargo, R. Y., Goldchleger, A., Kon, F., and Goldman, A. Checkpointing-based rollback recovery for parallel applications on the integrade grid middleware. In *Proceedings of the 2nd workshop on Middleware for Grid Computing* (Toronto, Ontario, Canada, 2004), ACM Press, pp. 35–40.

[36] de P. Nascimento, A., da C. Sena, A., da Silva, J. A., de C. Vianna, D. Q., Boeres, C., and Rebello, V. E. F. Managing the execution of large scale MPI applications on computational grids. *17th. International Symposium on Computer Architecture and High Performance Computing* (October 2005).

[37] Du, C., Ghosh, S., Shankar, S., and Sun, X.-H. A runtime system for autonomic rescheduling of MPI programs. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP 2004)* (Washington, DC, United States, 2004), IEEE Computer Society, pp. 4–11.

[38] Erwin, D. UNICORE and the project UNICORE plus. In *Presented at the ZKI Working Group Supercomputing Meeting* (May 2000).

[39] Forum, M. P. I. MPI: A Message-Passing Interface Standard. Tech Report UT-CS-94-230, 1994.

[40] Foster, I., Geisler, J., Nickless, W., Smith, W., and Tuecke, S. Software infrastructure for the I-WAY high-performance distributed computing experiment. IEEE Computer Society Press, 1996, pp. 562–571.

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

[42] Foster, I., and Kesselman, C. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, California, United States, 1998.

[43] Foster, I., Kesselman, C., Nick, J., and Tuecke, S. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[44] Foster, I., Kesselman, C., and Tuecke, S. The anatomy of the grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science 2150* (2001), 1–25.

[45] Freire, P. M. S. Monitoramento para aplicações MPI system-aware. Tese de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2003.

[46] Frey, J., Tannenbaum, T., Foster, I., Livny, M., and Tuecke, S. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing 5*, 3 (July 2002), 237–246.

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

[48] Goldchleger, A., Kon, F., Goldman, A., Finger, M., and Bezerra, G. C. InteGrade object-oriented grid middleware leveraging the idle computing power of desktop machines: Research articles. *Concurr. Comput.: Pract. Exper. 16*, 5 (2004), 449–459.

[49] Grimshaw, A. S., and Wulf, W. A. The legion vision of a worldwide virtual computer. *Communications of the ACM 40*, 1 (January 1997), 39–45.

[50] Gropp, W., and Lusk, E. Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications 18*, 3 (2004), 363–372.

[51] Gunter, D., Tierney, B., Crowley, B., Holding, M., and Lee, J. NetLogger: A toolkit for distributed system performance analysis, 2000.

[52] Kennedy, K., Mazina, M., Mellor-Crummey, J., Cooper, K., Torczon, L., Berman, F., Chien, A., Dail, H., Sievert, O., Angulo, D., Foster, I., Gannon, D., Johnsson, L., Kesselman, C., Aydt, R., Reed, D., Dongarra, J., Vadhiyar, S., and Wolski, R. Toward a framework for preparing and executing adaptive grid programs. In *16th Internationa Parallel and Distributed Processing Symposium (IPDPS 2002 (IPPS and SPDP))* (Washington - Brussels - Tokyo, April 2002), IEEE.

[53] Krauter, K., Buyya, R., and Maheswaran, M. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience 32*, 2 (November 2001), 135–164.

[54] Laforenza, D. Grid programming: Some indications where we are headed. *Parallel Computing 28*, 12 (December 2002), 1733–1752.

[55] Louca, S., Neophytou, N., Lachanas, A., and Evripidou, P. MPI-FT: Portable fault tolerance scheme for mpi. *Parallel Processing Letters 10*, 4 (2000), 371–382.

[56] Massie, M. L., Chun, B. N., and Culler, D. E. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing 30*, 5-6 (2004), 817–840.

[57] Mendes, H. A. HLogP: Um modelo de escalonamento para a execução de aplicações MPI em grades computacionais. Tese de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2004.

[58] Mills, D. Network Time Protocol (version 3) specification, implementation and analysis, March 1992.

[59] Nakada, H., Sato, M., and Sekiguchi, S. Design and implementations of Ninf: Towards a global computing infrastructure. *Future Generation Computer Systems 15*, 5 (1999), 649–658.

[60] Rao, S., Alvisi, L., and Vin, H. M. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Proceedings of the Twnty-Ninth Annual International Symposium on Fault-Tolerant Computing* (Washington, DC, United States, 1999), IEEE Computer Society, pp. 48–55.

[61] Ribler, R. L., Vetter, J. S., Simitci, H., and Reed, D. A. Autopilot: Adaptive control of distributed applications. In *Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing* (Washington, District of Columbia, United States, 1998), IEEE Comuter Society, pp. 172–179.

[62] Smith, W. A system for monitoring and management of computational grids. In *Proceedings of the 31st International Conference on Parallel Processing (ICPP)* (2002), pp. 55–62.

[63] Souto, H., da Silveira Filho, O., Moyne, C., and Didierjean, S. Thermal dispersion in porous media: Computations by the random walk method. *Journal of Computational and Applied Mathematics 21*, 2 (2002), 513–544.

[64] Stelling, P., DeMatteis, C., Foster, I. T., Kesselman, C., Lee, C. A., and von Laszewski, G. A fault detection service for wide area distributed computations. *Cluster Computing 2*, 2 (June 1999), 117–128.

[65] Stellner, G. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)* (Honolulu, Hawaii, United States, 1996).

[66] Treaster, M. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *ArXiv Computer Science e-prints* (December 2005), 1–11.

[67] Vadhiyar, S. S., and Dongarra, J. J. Self adaptivity in grid computing. *Concurrency and Computation: Practice and Experience 17*, 2-4 (2005), 235–257.

[68] Wolski, R., Spring, N. T., and Hayes, J. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems 15*, 5 (1999), 757–768.

[69] Ziv, A., and Bruck, J. *Checkpointing in Parallel and Distributed Systems.* Parallel and Distributed Computing Handbook. McGraw-Hill, 1996, chapter 10, pp. 274–302.