Universidade Estadual de Campinas
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

Estudo sobre Processamento Maciçamente Paralelo na Internet

Autor: Eduardo Javier Huerta Yero

Orientador: Prof. Dr. Marco Aurélio Amaral Henriques

Comissão Julgadora

Edmundo Roberto Mauro Madeira - IC/UNICAMP
Edson Toshimi Midorikawa - EPUSP
Islene Calciolari Garcia - IC/UNICAMP
Fernando José Von Zuben - FEEC/UNICAMP
Ivan Luiz Marques Ricarte - FEEC/UNICAMP
Marco Aurélio Amaral Henriques - FEEC/UNICAMP

Tese apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos necessários para a obtenção do título de Doutor em Engenharia Elétrica.

Campinas
Julho, 2003

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

H871e Huerta Yero, Eduardo Javier
Estudo sobre processamento maciçamente paralelo na Internet / Eduardo Javier Huerta Yero. – Campinas, SP: [s.n.], 2003.

Orientador: Marco Aurélio Amaral Henriques.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Processamento paralelo (Computadores) 2. Internet (Redes de computação). 3. Processamento eletrônico de dados - Processamento distribuído. 4. Cliente/servidor (Computação). I. Henriques, Marco Aurélio Amaral. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Este trabalho estuda a possibilidade de aproveitar o poder de processamento agregado dos computadores conectados pela Internet para resolver problemas de grande porte. O trabalho apresenta um estudo do problema tanto do ponto de vista teórico quanto prático. Desde o ponto de vista teórico estudam-se as características das aplicações paralelas que podem tirar proveito de um ambiente computacional com um grande número de computadores heterogêneos fracamente acoplados. Desde o ponto de vista prático estudam-se os problemas fundamentais a serem resolvidos para se construir um computador paralelo virtual com estas características e propõem-se soluções para alguns dos mais importantes como balanceamento de carga e tolerância a falhas. Os resultados obtidos indicam que é possível construir um computador paralelo virtual robusto, escalável e tolerante a falhas e obter bons resultados na execução de aplicações com alta razão computação/comunicação.

Palavras Chave: processamento paralelo, sistemas distribuído, grids, Internet

Abstract

This thesis explores the possibility of using the aggregated processing power of computers connected by the Internet to solve large problems. The issue is studied both from the theoretical and practical point of views. From the theoretical perspective this work studies the characteristics that parallel applications should have to be able to exploit an environment with a large, weakly connected set of computers. From the practical perspective the thesis indicates the fundamental problems to be solved in order to construct a large parallel virtual computer, and proposes solutions to some of the most important of them, such as load balancing and fault tolerance. The results obtained so far indicate that it is possible to construct a robust, scalable and fault tolerant parallel virtual computer and use it to execute applications with high computing/communication ratio.

Keywords: parallel processing, distributed systems, grids, Internet

A Juliana, minha esposa, e aos filhos que ainda não tive com ela

Agradecimentos

Esta história começou muito antes de eu chegar no Brasil, quando meus pais possibilitaram que eu terminasse a graduação com um esforço imenso, difícil de descrever a quem não conhece a realidade cubana. Mesmo na distância o amor deles me inspirou sempre, e sem eles este doutorado simplesmente não teria sido possível.

Dois anos depois de terminar a graduação cheguei no Brasil para fazer Mestrado, morrendo de medo, sem saber português e com uma fome tal que Luisma e Marty, os amigos que me receberam, tinham por costume jogar comida na sala e se esconder no quarto. "Eu acho que não tem perigo enquanto a gente o mantiver bem alimentado", susurravam assomando a cabeça pela porta. Eles eram meus amigos naquele momento, e foram meus irmãos a partir daí.

Minha família em Cuba é sensacional. Sempre pedindo notícias, torcendo e se alegrando com cada pequena vitória. Quando minha irmã Odalys e meu cunhado Roberto vieram morar no Brasil, um pedaço dessa família se firmou aqui para sempre. Pelo apóio e carinho nestes anos eles merecem muito mais do que serem citados nos meus agradecimentos, mas de todas as opções esta era a mais barata.

Marco Aurélio, meu orientador, é um exemplo de paciência e dedicação ao trabalho. Ao seu lado cresci como pesquisador e como pessoa. Espero de todo coração conseguir mostrar isso a todos no futuro, já que até o momento não foi possível.

Os últimos meses da tese, justo quando o trabalho aumenta, foram bastante agitados na minha vida pessoal. Tive a sorte de conhecer e começar a namorar com Juliana, uma mulher cheia de vida cuja chegada acelerou a terminação do trabalho. Ju, hoje minha esposa, nunca reclamou por passar finais de semana ao meu lado no laboratório; mantinha-se sempre sorridente e positiva, me estimulando quando as forças me faltavam. A ela entrego este agradecimento, a dedicatória e o resto da minha vida.

Sumário

Capítulo 7

Tolerância a falhas em um grid

O Capítulo 6 mostrou uma abordagem para o escalonamento de tarefas que é capaz de concluir a execução de uma aplicação mesmo se algumas unidades trabalhadoras falharem. Apesar desta estratégia tratar as falhas que ocorrem com mais freqüência em um grid, ela não é capaz de se recuperar de falhas nos componentes coordenadoras ou no servidor, segundo a arquitetura de grid proposta no Capítulo 5.

No entanto, os coordenadores e o servidor são peças fundamentais no funcionamento do grid. São esses componentes os que armazenam a informação de estado do sistema. Por exemplo, se um coordenador falha, o estado das aplicações sendo executadas (isto é, quais tarefas já terminaram, quais tarefas estão sendo executadas, onde elas estão e quais tarefas estão aguardando para serem alocadas a um trabalhador) é perdido, o que implica que as aplicações devem ser reiniciadas. Este risco é inaceitável para aplicações que podem passar dias ou semanas executando.

Este capítulo mostra a abordagem proposta neste trabalho para o problema de falhas no servidor e coordenadores da plataforma. Como já foi mencionado no Capítulo 5, a abordagem é baseada em uma combinação de técnicas de log/replay e checkpoint/rollback. Para implementar o log/replay a proposta baseia-se no fato de que tanto o servidor quanto os coordenadores são compostos por um conjunto de serviços *piecewise-deterministic* que interagem de forma controlada pela plataforma. Qualquer interação entre dois serviços é armazenada em um *log* de forma tal que, no evento de uma falha, elas podem ser reproduzidas e os serviços conduzidos até o mesmo estado em que se encontravam antes da falha. O mecanismo de checkpoint/rollback auxilia no controle do tamanho do log e, portanto, no controle do tempo necessário para a recuperação, armazenando periodicamente checkpoints de cada serviço para permitir a remoção de entradas no arquivo de log.

Os detalhes desta proposta são apresentados no artigo *"A Fault Tolerance Mechanism for Scalable Grids"*, mostrado a seguir.

Parte I

Estado da Arte

Capítulo 1

Introdução

Muitos esforços têm sido dedicados à tentativa de explorar o poder computacional disponível — e muitas vezes ocioso — na Internet para executar aplicações de grande porte. Por um lado, aplicações com grandes necessidades de processamento nas áreas de economia, engenharia e biologia, entre outras, não podem ser executadas em um tempo razoável por nenhum computador existente. Por outro lado, o enorme poder agregado dos computadores conectados a Internet permanece subutilizado na maior parte do tempo. Não é difícil relacionar estes fatos: será possível construir uma grade (mais conhecida pelo termo em inglês *grid*) de computadores que são cada vez mais poderosos, conectados por redes já instaladas e que aumentam constantemente sua velocidade e confiabilidade, e usá-la para resolver problemas de grande porte? Apesar de que intuitivamente o casamento parece perfeito, um conjunto de problemas de difícil solução se interpõe entre o mundo digital atual e um mundo em que a participação em grids seja tão comum quanto o acesso à Internet.

A própria terminologia nesta área é usada ainda de maneira relativamente informal, como reconhece Ian Foster no seu artigo *"What is the Grid? A Three Point Checklist"* (Grid Today, julho de 2002). Foster, um dos visionários na área de grids, define *grid computing* como *"a resolução de problemas e o compartilhamento coordenado de recursos em uma organização virtual dinâmica e multi-institucional"*. Para ele, um grid deve satisfazer três características básicas, listadas a seguir.

1. Coordenar recursos que não estão sujeitos a um controle centralizado.

2. Usar protocolos e interfaces seguindo padrões bem estabelecidos, de propósito geral e abertos.

3. Oferecer uma qualidade de serviço não trivial.

Segundo esta definição, os chamados *clusters de computadores* — conjunto relativamente pequeno de computadores conectados por redes locais — como o PVM e o Sun Grid Engine — não podem ser classificados como *grids*, pois eles não atendem às características 1 e 2 enunciadas acima. No entanto, o autor classifica sistemas como Condor, Entropia e United Devices — os quais podem gerenciar computadores localizados em redes diferentes de forma decentralizada — como *grids de primeira geração*, por atenderem as características 1 e 3, mas não usar protocolos e interfaces padronizados.

Neste trabalho estudaremos as dificuldades de se construir um *grid* de primeira geração que, por simplicidade, chamaremos *grid* no restante do texto. Consideramos que, no momento em que o trabalho está sendo desenvolvido, ainda não existem padrões suficientemente amadurecidos para as interfaces e os protocolos do *grid* que justifiquem um investimento de tempo maior nesta área. O objetivo fundamental do *grid* proposto é executar aplicações paralelas de grande porte e, como tal, seu desenvolvimento tem maior ênfase em temas como escalabilidade, tolerância a falhas e segurança.

De forma geral as dificuldades para se construir grid podem ser divididas em três grandes grupos, mostrados a seguir.

- A falta de base teórica para entender as características do sistema e, conseqüentemente, das aplicações paralelas apropriadas para os mesmos.
- As dificuldades técnicas envolvidas no projeto, implementação, implantação e gerenciamento de um sistema que possa comportar um grande número de computadores simultaneamente.
- As barreiras psicológicas inerentes ao ser humano e que dificultam a adoção de uma nova visão da Internet. Para entender estes receios é suficiente fazer a seguinte pergunta a nós mesmos: eu colocaria meu computador à disposição de pessoas desconhecidas para resolver problemas delas?

Nas seguintes seções abordaremos estas dificuldades com maiores detalhes.

1.1 Fundamentos teóricos

Para aproveitar efetivamente o poder computacional de um grid é necessário desenvolver um embasamento teórico sólido que permita entender a relação entre a aplicação paralela e o grid. Este conhecimento deve permitir avaliar se uma aplicação em particular é apropriada ou não para o ambiente em questão.

Existem basicamente dois tipos de paralelismo: funcional e de dados. Uma aplicação apresenta paralelismo funcional quando é formada por componentes paralelos com comportamentos diferentes. Cada componente realiza uma função específica e interage com os outros componentes quando necessário. Um exemplo clássico de paralelismo funcional é o *pipeline*, em que os componentes paralelos são arranjados em fila (os dados de saída do componente $i-1$ constituem a entrada do componente i) e cada um executa um conjunto específico de operações sobre os dados.

As aplicações com paralelismo de dados, por outro lado, são formadas por tarefas iguais que processam partes diferentes dos dados de entrada. Este tipo de aplicação é comumente estruturada segundo o modelo Mestre-Escravo. Neste modelo um componente Mestre divide os dados a serem processados em subconjuntos menores que são entregues aos componentes Escravos para serem processados. Ao finalizar o processamento os componentes Escravos entregam um resultado parcial ao Mestre, que os usa para gerar o resultado final. Algumas aplicações (e.g. otimização combinatorial) repetem este processo até atingir um determinado critério de parada. Existem outros modelos possíveis, como o modelo hierárquico em que, entre o Mestre

e os Escravos, interpõem-se várias camadas de Sub-Mestres para gerenciar subconjuntos de Escravos.

Intuitivamente é possível imaginar que as aplicações com paralelismo de dados são as melhores candidatas para uma abordagem maciçamente paralela. Dificilmente encontraremos aplicações com milhares de componentes paralelos, cada um com uma função diferente (paralelismo funcional). No entanto, é possível imaginar aplicações com uma grande quantidade de dados a serem processados usando o mesmo algoritmo (por exemplo criptoanálise, processamento de imagens, montagem de fragmentos de DNA, reconhecimento de padrões). Ainda intuitivamente podemos inferir que se estes dados podem ser divididos em subconjuntos menores e processados de forma independente então a aplicação deve ser uma candidata a uma implementação eficiente em um grid.

Estes conceitos, apresentados aqui informalmente, devem ser estudados a fundo. O efeito da contenção pelo uso de recursos compartilhados como os enlaces de rede, por exemplo, é um fator cuja importância aumenta na medida em que o número de processadores do grid aumenta. Como avaliar se a contenção chegará a impedir uma aplicação de executar em um grid? Quais as características de uma aplicação "apropriada" para um grid? Para responder estas perguntas é necessário desenvolver um modelo consistente e prático para descrever o desempenho da aplicação paralela no grid.

1.2 Barreiras tecnológicas

Projetar, implementar, implantar e gerenciar um grid formado por um grande número de computadores não é simples. Além dos problemas relacionados ao gerenciamento de uma grande quantidade de computadores, a maior probabilidade de falhas tanto da rede quanto dos computadores em si, as ameaças à segurança dos dados (dos computadores participantes e das aplicações) e as variações constantes na disponibilidade de recursos (poder de processamento, largura de banda) devido ao uso irregular dos mesmos introduzem questões de difícil solução prática.

Um grid deve ser escalável. Informalmente entende-se por escalabilidade a ausência de gargalos – ou pontos de congestionamento – à medida em que o número de computadores aumenta. Um dos mandamentos para sistemas distribuídos dita: para um sistema ser escalável devem-se evitar componentes centralizados. Porém, nas condições de um grid é impossível imaginar algoritmos totalmente democráticos (distribuídos), em que cada passo requer a coordenação de todos os participantes. Isto implica que algum componente coordenador deverá existir, mesmo que seja para gerenciar apenas um subconjunto dos computadores do grid. Mas, quantos coordenadores? Como eles colaboram para executar uma aplicação? Qual a carga máxima que um coordenador pode gerenciar sem provocar gargalos? Estas e outras perguntas precisam ser abordadas no projeto da arquitetura do grid.

A tolerância a falhas é outro desafio para o projeto de um grid. A grande quantidade de computadores, a baixa confiabilidade dos mesmos e dos enlaces de rede e o (possivelmente) longo tempo de execução das aplicações paralelas aumentam a probabilidade da ocorrência de falhas durante tal execução. No caso de um grid, uma falha não deve implicar na perda de todos os resultados obtidos pela aplicação, porém é permissível que alguns destes resultados sejam perdidos. As soluções comuns para tolerância a falhas em sistemas distribuídos envolvem

registros de mensagens, replicação de componentes e salvamento periódico do estado do sistema (comumente conhecido pelo termo em inglês *checkpointing*). No entanto, mesmo usando uma destas técnicas – ou uma combinação de algumas delas – ainda é necessário procurar implementações escaláveis e que não comprometam o desempenho do sistema em situações normais. Por exemplo, uma solução que gere constantemente tráfego pela rede pode ser inadmissível.

Outra parte importante em um grid é o escalonador, que é responsável por distribuir os componentes paralelos da aplicação entre os computadores. Um escalonador eficiente deve conseguir aproveitar melhor os recursos heterogêneos que o grid oferece para minimizar o tempo de execução da aplicação. No entanto, os recursos disponíveis no grid variam constantemente, tanto quantitativa quanto qualitativamente. A quantidade de computadores disponíveis pode variar, seja produto de falhas ou do fluxo normal de entrada e saída destes. A qualidade dos recursos, como poder de processamento e largura de banda varia também, devido ao uso dos mesmos para tarefas alheias ao processamento no grid. A tentação de fazer um escalonador totalmente dinâmico, que avalie constantemente os recursos disponíveis, deve ser cuidadosamente avaliada, para evitar que a sobrecarga imposta pelo algoritmo seja muito grande. Surge então a pergunta: onde estaria o ponto de equilíbrio entre a eficiência do algoritmo e a sobrecarga imposta pelo mesmo ao grid?

Como qualquer sistema distribuído, um grid deve ter a capacidade de proteger seus elementos de ações mal-intencionadas de terceiros. Os programadores de aplicações devem ser protegidos contra o acesso não autorizado aos dados ou código da aplicação, visto que a mesma será executada em computadores de terceiros. Por outro lado, os donos dos computadores participantes devem ser protegidos contra aplicações que possam causar danos nos mesmos ou executar ações ilegais no nome destes, entre outros possíveis ataques. Embora as soluções existentes na área de segurança para autenticação e controle de acesso possam ser usadas em grids, ainda é necessário um estudo detalhado para determinar a abordagem mais escalável e menos invasiva ao funcionamento normal do sistema.

Um outro assunto prático, também de grande impacto no estabelecimento dos grids como alternativa de processamento, é o gerenciamento do mesmo. Por gerenciamento entenda-se a instalação e manutenção do código e o monitoramento do estado do sistema. Considerando que pode existir um grande número de computadores no grid, instalar o código pela primeira vez e distribuir novas versões do mesmo passam a ser problemas de grandes dimensões. Da mesma forma, os mecanismos de monitoramento poderão gerar grandes quantidades de informação que precisam ser transmitidas através da rede. Projetar mecanismos viáveis para gerenciar um grid interferindo o mínimo possível no funcionamento do mesmo é de vital importância para seu sucesso no mundo real.

1.3 Fatores humanos

Imaginemos agora que todos os fundamentos teóricos foram estabelecidos e as dificuldades técnicas superadas. Temos, pronto para instalar, um grid que poderá revolucionar a forma de tratar problemas computacionais de grande porte. Pesquisas contra o câncer, detecção de sinais de vida extraterrestre, mapeamento genéticos de espécies, previsões meteorológicas mais exatas, enfim, problemas de grande impacto para a humanidade, podem ser resolvidos por um computador maciçamente paralelo de baixo custo, formado por computadores espalhados pela

Internet. Só falta convencer os donos desses computadores a participarem no grid.

Algumas iniciativas mostram que existem resistências para que as pessoas coloquem seus computadores em um grid. Para o usuário comum um grid é uma possível fonte de problemas, sejam eles relacionados com segurança ou com os recursos econômicos investidos (energia elétrica, pulsos telefônicos), sem retorno palpável. Os potenciais colaboradores se perguntam: o que eu ganho com minha participação?

Existem várias formas de vencer estas barreiras. Recentemente o projeto SETI@Home – que procura indícios de inteligência extraterrestre processando sinais captados por radiotelescópios – conseguiu atrair o interesse de mais de 4 milhões de usuários, segundo estatísticas de junho/2003, sendo que mais de 600 000 deles mantinham-se ativos, i.e. processando dados (www.setiathome.org, junho 2003). Estes usuários doaram tempo de processamento dos seus computadores ao projeto, após terem o trabalho de descarregar um programa através da rede e instalá-lo localmente. Por quê? Esta participação maciça foi conseguida devido a uma publicidade intensa que apelava para um fator subjetivo: a possibilidade do colaborador ser o primeiro ser humano a descobrir indícios científicos da existência de inteligência extraterrestre. A campanha publicitária encarregou-se de inibir possíveis dúvidas referentes à segurança e o projeto foi um sucesso.

O projeto SETI@Home abriu o caminho para a introdução de grids na sociedade. Seu impacto só não tem sido maior pela ausência de resultados visíveis até o momento – é possível imaginar a visibilidade que teriam os grids se SETI tivesse descoberto provas da existência de vida fora da Terra. No entanto, este projeto colocou os grids na vista do público e encorajou iniciativas similares. A United Devices, em parceria com outras empresas como IBM e Accelerys, está promovendo projetos que pesquisam novas drogas contra o câncer, Anthrax e catapora, mediante o uso de um grid formado por usuários da Internet que doam seu tempo de processamento (www.ud.com, junho 2003). Segundo estatísticas da United Devices, em junho/2003 mais de 2 milhões de computadores estavam participando no seu grid.

Existem outras alternativas para estimular os donos dos computadores a participarem de um grid, mesmo quando o problema sendo resolvido não seja de fundamental importância para a humanidade. Estímulos econômicos como dinheiro real ou virtual, descontos para determinadas compras, tickets para sorteios ou assinaturas para sites podem ser usados em troca da disponibilização do computador para o grid por um determinado período de tempo. Em qualquer caso, os administradores do grid terão que convencer os donos dos computadores de que não existem perigos para a integridade do computador e os dados nele contidos se ele for disponibilizado para uso por terceiros. Os argumentos para isto devem ser tanto práticos (medidas de segurança tomadas pelo grid) quanto subjetivos (confiança na empresa ou instituição que gerencia o grid). Pode-se concluir que esta área também requer ainda um grande esforço de pesquisa e desenvolvimento antes de que possa ser considerada amadurecida e eficaz na conquista de novos colaboradores para um grid.

1.4 Objetivos

Neste trabalho abordaremos dois dos três grandes desafios associados à computação paralela em grids: a fundamentação teórica e os problemas técnicos ligados à implementação. Dentro da fundamentação teórica apresentaremos um estudo analítico cujo objetivo fundamental é desen-

volver as ferramentas necessárias para avaliar as características de um grid e, consequentemente, das aplicações paralelas apropriadas para ele. No que diz respeito à implementação de um grid, introduziremos novas soluções para problemas como escalabilidade e tolerância a falhas, fundamentais no sucesso de um sistema como o proposto.

Os aspectos relacionados com os fatores humanos não serão diretamente abordados por não serem de caráter estritamente técnico. No entanto, quando for apropriado, destacaremos providências que podem ser tomadas no sistema para facilitar na adoção de um grid pelo público.

1.5 Organização do trabalho

O trabalho está dividido em quatro partes. A Parte I, além deste capítulo de introdução, contém no Capítulo 2 uma revisão bibliográfica das propostas de grids existentes, classificando-os de acordo com a abordagem usada. Pretendemos com esta primeira parte dar uma visão geral do panorama atual na computação usando grids.

A Parte II está dedicada a mostrar as propostas deste trabalho para modelagem de grids e de suas aplicações paralelas. O Capítulo 3 mostra uma estratégia de predição de desempenho de aplicações paralelas em sistemas heterogêneos. A estratégia proposta é estática, ou seja, usa apenas informações disponíveis em tempo de compilação (estrutura da aplicação, quantidade de operações executadas em cada componente paralelo) para gerar um modelo analítico parametrizado. Este modelo pode ser instanciado para estimar o tempo de execução da aplicação para casos diferentes. Apesar de ser um modelo estático, a solução proposta considera a contenção pelo uso de recursos em tempo de execução, o que normalmente era considerado apenas em abordagens baseadas em simulações. O Capítulo 4 estuda o desempenho de aplicações Mestre-Escravo em grids do ponto de vista analítico. Inicialmente define-se um modelo de aplicação e de grid. Estes modelos são usados para obter expressões analíticas para o desempenho da aplicação, que são usadas para analisar o ganho em velocidade (*speedup*) máximo esperado para a aplicação. Avalia-se também a escalabilidade da aplicação, isto é, a habilidade da aplicação para aproveitar uma grande quantidade de processadores disponíveis. O trabalho estabelece condições necessárias e suficientes para uma aplicação Mestre-Escravo ser escalável, e mostra o impacto da contenção pelo uso da rede no desempenho da aplicação.

A Parte III do trabalho está dedicada a avaliar as dificuldades técnicas para se construir um grid e propor soluções para as principais delas. O Capítulo 5 apresenta JoiN, um sistema desenvolvido como parte deste trabalho para servir de plataforma de testes às idéias nele propostas. JoiN começou a ser desenvolvido em 1997 e desde então passou por constantes aperfeiçoamentos que fizeram dele um sistema flexível, confiável e fácil de usar.

O Capítulo 6 está dedicado ao gerenciamento de aplicações em um grid. Nele é proposto um modelo de aplicação consistente com as características esperadas de uma aplicação maciçamente paralela, e um algoritmo de escalonamento flexível o suficiente para ser usado em um sistema em que a disponibilidade de recursos varia constantemente. São apresentados resultados práticos que avaliam a escolha feita.

O Capítulo 7 aborda mecanismos de tolerância a falhas. Nele estudam-se as soluções existentes e avalia-se a dificuldade de usá-las diretamente em grids, seja por problemas de escalabilidade ou desempenho. O capítulo propõe uma solução de baixa sobrecarga baseada em

uma combinação de replicação, checkpointing e logs. A solução está orientada a proteger de falhas os componentes vitais do sistema (os encarregados de gerenciar o grid) enquanto que as falhas nos componentes que executam as tarefas paralelas são somente detectadas e as tarefas replicadas.

A Parte IV é formada pelo Capítulo 8, que apresenta as conclusões e os trabalhos futuros a serem realizados. Apesar de existir ainda um longo caminho a percorrer, os resultados obtidos nesta pesquisa sugerem que estamos na direção certa e nos estimulam a continuar os esforços nesta área.

Capítulo 2

Revisão bibliográfica

Este capítulo oferece uma visão do espectro de propostas já existentes relacionadas com grids. Apesar do pouco tempo de existência desta área de pesquisa já é possível encontrar na literatura uma ampla variedade de abordagens diferentes para projetar e implementar um grid. Para cada uma delas apresentaremos um resumo de suas características principais, pois uma análise detalhada de cada um está fora do escopo deste trabalho.

Para facilitar a apresentação dos sistemas pesquisados é útil fazer uma classificação de acordo com um determinado critério. Esta classificação pode ser feita, por exemplo, de acordo com a abordagem adotada pelo sistema para resolver problemas como escalabilidade, tolerância a falhas, segurança, entre outros. Para esta exposição dos sistemas considerou-se mais adequado utilizar como critério de classificação a abordagem para o problema da heterogeneidade de recursos de computação. Esta classificação gera um número pequeno de categorias (somente 3) e as diferenças entre as categorias são bem claras e representativas.

A classificação dividirá os sistemas em 3 grandes classes: os independentes de plataforma, os dependentes de plataforma e os que utilizam uma combinação de técnicas dependentes e independentes de plataforma.

2.1 Sistemas dependentes de plataforma

Os sistemas dependentes de plataforma se caracterizam por serem implementados em linguagens de programação como C/C++, cuja portabilidade entre plataformas diferentes não é garantida. As aplicações programadas para estes sistemas normalmente precisam gerar código binário para cada um dos tipos diferentes de plataformas que participam no grid. Em alguns casos, como os de Condor [42] e NOW[2], o sistema exige que os computadores que formam a máquina virtual apresentem a mesma arquitetura. A principal vantagem desses sistemas é que eles conseguem explorar as características particulares de cada tipo de arquitetura para obter implementações com bons desempenhos.

NOW (Network of Workstations) [2] é um sistema formado por várias estações de trabalho UNIX de alto desempenho conectadas por uma rede dedicada de alta velocidade. Os objetivos fundamentais do projeto são obter uma melhor relação custo-benefício para aplicações paralelas que os MPPs (Massively Parallel Processors) e obter um desempenho para as aplicações seqüenciais normais melhor que o de uma estação de trabalho isolada. Para isso NOW usa tecnologias de comunicação avançadas (ATM) e um sistema operacional chamado GLUnix (GLobalUnix)

que manipula os recursos das estações de trabalho como se fossem um único computador paralelo.

PVM (Parallel Virtual Machine) [18] é um sistema que permite simular um computador paralelo usando máquinas conectadas por uma rede. Este computador paralelo é chamado de máquina paralela virtual e sua implementação baseada em troca de mensagens é simples e eficiente. Atualmente existem implementações do PVM em muitas arquiteturas, tanto para estações de trabalho conectadas a uma rede quanto para computadores paralelos reais, tornando as aplicações que usam PVM portáveis para um grande número de ambientes. Embora o PVM seja baseado em protocolos TCP e UDP, a configuração do PVM em redes tipo Internet é difícil, pois o usuário PVM precisa de privilégios em cada um dos computadores que formam a máquina virtual. Por este motivo, e pela falta de mecanismos no sistema para garantir escalabilidade, o seu uso é geralmente restrito a redes locais. PVM serviu de base para o sistema PARA++ [12], que é uma abordagem orientada a objetos para o problema da programação paralela.

MPI (Message Passing Interface) [38] é um sistema com objetivos similares a PVM. Enquanto PVM foi um projeto de pesquisa que evoluiu até virar um padrão de fato, MPI surgiu de um esforço consciente de um grupo de engenheiros para desenvolver um padrão de troca de mensagens com o objetivo de aumentar a portabilidade de aplicações paralelas escritas para MPPs. MPI tem sido implementado em várias arquiteturas paralelas, e existem algumas implementações que trabalham em conjuntos de computadores conectados por uma rede. MPI também inspirou sistemas orientados a objeto para a programação paralela, tais como mpi++ [21], OOMPI [39], PARA++ [12] e JOINT [43].

Virtual Clusters [13] tem como objetivo fundamental construir clusters sem interferir no dia-a-dia dos donos dos computadores. Para isso o sistema executa o software associado ao cluster em um ambiente separado no mesmo computador. Ele instala um sistema operacional independente, usando uma partição separada do disco rígido. Quando o computador permanece ocioso por um determinado periodo de tempo um código específico é ativado para salvar o estado atual do computador e iniciar um novo sistema operacional. Neste ambiente é executado o código que permitirá ao computador participar no cluster. Quando o dono do computador voltar a usar o mesmo, a participação no cluster é interrompida e o estado restaurado.

Condor [42] é um sistema tipo batch, em que o usuário submete as tarefas e o sistema se encarrega de procurar um computador ocioso onde executá-la. Se o computador volta a ser utilizado por seus usuários regulares (isto é, deixa de ser ocioso), Condor automaticamente migra a tarefa a um outro computador que esteja ocioso. Não é necessária uma programação especial para usar Condor, pois a tarefa é executada com a ilusão de que está trabalhando somente na máquina em que foi submetida. Para programar aplicações paralelas usando o Condor é necessário instalar o PVM e uma interface especial entre o Condor e o PVM chamada CARMI [34]. Existe também a possibilidade de interação do Condor com o Globus [15] usando uma interface entre os mesmos conhecida como Condor-G [17].

Legion [20] tem como objetivo agrupar computadores heterogêneos e apresentá-los ao usuário como um computador virtual. Os componentes fundamentais de Legion são objetos que se comunicam através de invocações a métodos. Cada método tem uma assinatura que descreve seus parâmetros e seu valor de retorno. Estas assinaturas podem ser descritas usando CORBA IDL (Interface Definition Language) ou MPL (Mentat Programming Language). Como princípio de projeto, o sistema especifica somente a funcionalidade dos seus componentes, e oferece implementações simples para cada uma delas. O usuário poderá substituir um componente por

outro que se adapte melhor às suas necessidades.

NetSolve/GridSolve [4] é um sistema projetado para resolver problemas científicos em ambientes distribuídos. A resolução de problemas é fortemente baseada no modelo cliente-servidor. Um cliente envia pedidos a um *agente NetSolve*, que se encarrega de procurar na rede os recursos necessários e de coordenar a execução do pedido. NetSolve oferece interfaces de programação para C, Fortran, MATLAB e Java.

MyGrid [11] constrói um grid com os recursos acessíveis para um usuário. O usuário deve fornecer ao sistema meios para transferir arquivos e executar programas remotos nos computadores que ele pretende usar. MyGrid fornece recursos básicos de administração e escalonamento para o grid. Uma extensão deste trabalho chamada OurGrid [3] permite unir o esforço de vários grids na execução de uma aplicação.

Globus [15] tem como principal objetivo construir um grid que forneça acesso constante e confiável a recursos computacionais de alto desempenho. Os serviços oferecidos por Globus são baseados em um *toolkit*, que provê os mecanismos básicos de comunicação, segurança, alocação e manipulação de recursos, entre outros. Globus não especifica um modelo de programação em particular. Sistemas de mais alto nível, como Legion, compiladores de CC++ (Compositional C++) e implementações de MPI podem ser desenvolvidos em um grid usando os serviços básicos oferecidos por Globus. Este sistema é um dos projetos de grids mais bem sucedidos até o momento, tendo realizado experimentos com 330 computadores e 3600 processadores.

Estimulados pelo sucesso de alguns grids experimentais, algumas empresas lançaram-se ao mercado oferecendo soluções baseadas em grids. Por serem comerciais existem poucos detalhes técnicos disponíveis sobre seu funcionamento, o que impede uma avaliação aprofundada. Porém, o fato de todas elas disponibilizarem código diferente para cada sistema operacional deixa claro que as soluções oferecidas são dependentes de plataforma.

Parabon (www.parabon.com, junho 2003) é um grid em operação desde junho do ano 2000. Ele participa em projetos acadêmicos e comerciais. Atualmente envolvido em parcerias com a Celera Genomics, o site disponibiliza meios para os usuários participarem no descobrimento de drogas para a cura do câncer. Uma das características que identificam Parabon é que a empresa prevê uma forma de pagamento aos participantes caso a aplicação paralela sendo executada seja comercial.

United Devices (www.ud.com, junho 2003) oferece diversas formas para construir grids, sejam eles empresariais ou globais. Em operação desde junho do ano 2000, o projeto participa em pesquisas contra o câncer, Anthrax e catapora. Atualmente a empresa participa em parcerias com a IBM, a Intel e a Accelerys para atingir seus objetivos.

Similar aos dois sistemas anteriores, o Entropia (www.entropia.com, junho 2003) participa em um programa para combater a AIDS. Em operação desde agosto do ano 2000, o Entropia tem também parcerias com a IBM. O grid é usado principalmente para aplicações financeiras, farmacêuticas e de bioinformática, entre outras.

2.2 Sistemas independentes de plataforma

Os sistemas independentes de plataforma se caracterizam por serem baseados em tecnologias portáveis, tais como Java [19]. Java tem sido considerada uma das principais promessas para a computação de alto desempenho, apesar de alguns problemas de projeto da linguagem que

dificultam a programação de aplicações científicas [16, 32]. Java permite aos programadores realizar o sonho *write once, run everywhere*. Uma aplicação escrita em Java é compilada em um *bytecode*, que é depois interpretado por uma Máquina Virtual Java. Sob este esquema, basta implementar Máquinas Virtuais Java para cada uma das plataformas existentes e as aplicações Java poderão ser executadas em qualquer arquitetura sem nenhuma mudança. Atualmente, a Máquina Virtual Java está implementada para as principais arquiteturas no mercado. Além disso, existem implementações da Máquina Virtual Java fazendo parte dos navegadores WWW mais conhecidos, criando a possibilidade de se executar código Java em qualquer computador que tenha instalado um navegador. Alguns dos problemas iniciais com a plataforma Java, tais como os problemas de desempenho (derivados do fato de que o *bytecode* é interpretado pela Máquina Virtual), estão sendo eliminados com novas técnicas, como a compilação JIT (*Just In Time*). Estes fatos têm levado a comunidade científica de processamento de alto desempenho a acreditar em Java como uma opção real para a implementação de grids que possam tirar proveito do maior número possível de computadores heterogêneos ligados à Internet.

É necessário ressaltar que Java não é a única opção para sistemas independentes de plataforma. Pacotes de software como Matlab e Mathematica oferecem também portabilidade para plataformas diferentes. No entanto, não se tem notícias de seu uso como plataforma de programação para grids.

Nesta seção serão apresentados alguns dos sistemas existentes que são baseados na tecnologia Java. Para uma melhor compreensão, estes sistemas serão divididos em sistemas baseados em aplicações Java e sistemas baseados em *applets* Java.

2.2.1 Sistemas baseados em aplicações Java

Os sistemas baseados em aplicações Java se diferenciam dos sistemas apresentados na seção anterior somente pelo fato de que usam Java para produzir código independente de plataforma, e assim evitar ter que portar os códigos binários para cada arquitetura existente no sistema.

ParaWeb [8] é um sistema baseado em Java que implementa duas abordagens para a programação de aplicações paralelas: o JPRS (Java Parallel Runtime System) e o JPCL (Java Parallel Class Library). O JPRS estende o ambiente de execução de Java – alterando a Máquina Virtual Java – para permitir a criação remota de linhas de controle (*threads*). A implementação atual do JPRS é baseada em memória compartilhada. O JPCL, no entanto, estende a interface de programação de Java mediante um conjunto de classes, sem alterar a Máquina Virtual Java. A implementação atual do JPCL é baseada em troca de mensagens.

Albatross [25] é um projeto cujo principal objetivo é explorar ambientes de programação formados por vários *clusters* de computadores conectados por redes tipo WAN. O projeto assume que os clusters estão ou podem ser estruturados de forma hierárquica. A abordagem utiliza este fato para diminuir a comunicação entre clusters. As principais idéias do projeto são exploradas na implementação do sistema Manta. Manta [29] é um sistema baseado em modificações na linguagem Java para permitir a programação de aplicações paralelas em ambientes distribuídos. Manta introduz a palavra chave *remote* para identificar as classes cujos métodos podem ser invocados de forma remota. O compilador de Java é modificado para reconhecer essa nova palavra chave e para gerar código nativo diretamente, ao invés de gerar o *bytecode*. Além disso, Manta altera a implementação de RMI (*Remote Method Invocation*) para que seja mais simples e flexível.

ProActive PDC [9] é uma biblioteca Java para a programação paralela e distribuída. O objetivo principal do sistema é o desenvolvimento de um grid que permita programar aplicações que possam ser executadas indistintamente em multiprocessadores com memória compartilhada, em clusters de estações de trabalho, ou em qualquer combinação das anteriores. A biblioteca é totalmente baseada em RMI e nas facilidades de reflexão do JDK (*Java Development Kit*), sem alterar nenhum elemento de ambiente de programação de Java. O sistema permite polimorfismo entre objetos locais e remotos e oferece mecanismos de sincronização de alto nível entre os objetos.

IceT [40] tenta combinar idéias encontradas em computação distribuída de alto desempenho (tipo PVM) com idéias de computação na Internet. Em geral, o modelo proposto por IceT é um conjunto de *ambientes virtuais*, pertencentes cada um a um usuário diferente, cada um com níveis diferentes de segurança e acessibilidade. O sistema estende o mecanismo para carregar classes de forma remota para permitir que o código das aplicações possa ser carregado da mesma forma em que os navegadores WWW carregam os *applets* Java. A interface de programação de aplicações para IceT é similar à do PVM.

JPVM [14] é uma implementação de PVM em Java. Como Java é uma linguagem orientada a objetos pura, a implementação teve que adaptar a interface de programação de PVM a uma interface orientada a objetos, mas de forma que a nova interface ficasse o mais parecida possível com a original.

Ninflet [41] é um sistema para a computação global baseado em Java. Ninflet estende a linguagem Java introduzindo a palavra chave `remote`, para identificar as classes cujos métodos poderão ser invocados de forma remota. O sistema está formado por 3 componentes fundamentais: os Servidores (*Ninflet Servers*), que se colocam à disposição do sistema para executar *ninflets*, os clientes (*Ninflet Clients*), que se conectam ao sistema para solicitar a execução de *ninflets*, e os Coordenadores (*Ninflet Dispatchers*), que coordenam a interação entre clientes e servidores.

JavaParty [33] é um sistema para a computação paralela otimizado para sistemas que interconectam estações de trabalho usando hardware especializado. O sistema introduz a palavra chave `remote` na linguagem Java para identificar os objetos que devem ser distribuídos pela rede. A partir desse momento, o programador não tem que se preocupar mais com a migração destes objetos, e os utiliza sem saber sua localização física atual. Sempre que o sistema decide por alguma razão migrar um objeto, a migração é feita de forma totalmente transparente ao programador. A migração de um objeto é possível somente se nenhum método está sendo executado sobre o objeto e se nenhuma outra migração está tendo lugar no momento. Para atingir seus objetivos JavaParty utiliza um preprocessador que analisa o código usando a palavra chave `remote` e gera código Java convencional, e um compilador que gera o bytecode necessário. JavaParty não faz nenhuma modificação na Máquina Virtual (interpretador) Java.

Parallel Java [22] é um sistema baseado em Charm++ [24] que implementa uma extensão a Java para a programação paralela baseada em uma biblioteca e um ambiente de execução. A solução proposta combina elementos de sistemas orientados a objetos e sistemas baseados em troca de mensagens. Existem objetos que podem ser criados de forma remota. Um objeto pode obter uma referência remota de outro objeto e chamar as *funções de entrada* desse objeto. As funções de entrada são as únicas funções do objeto que podem ser chamadas de forma remota. Elas recebem uma mensagem como único parâmetro e não retornam nada. O sistema é implementado usando Converse [23], uma plataforma de interoperabilidade que permite integrar

módulos escritos em diferentes linguagens.

Unified Computing Environments (UCEs) [26] são conjuntos de *daemons* — chamados UCEMs (*Unified Computing Environment Managers*) — executados no nível de usuário nos computadores participantes. Cada usuário configura seu próprio UCE. Um mesmo computador pode ter UCEMs de diferentes usuários sendo executados. Quando um usuário deseja executar uma aplicação determinada, utiliza um navegador para acessar uma página Web contendo um *applet* que implementa a aplicação. Este acesso é detectado por um *proxy* especial, que substitui a referência ao *applet* na página Web acessada por uma referência a um *plug-in*. O *proxy* então executa a aplicação – implementada no *applet* – no UCE configurado pelo usuário, enquanto o usuário observa o progresso da aplicação com ajuda do *plug-in*. Para isso, os navegadores têm que ser configurados para utilizarem esse *proxy*.

2.2.2 Sistemas baseados em *applets* Java

Diferente das abordagens introduzidas acima, os sistemas apresentados nesta seção exploram a capacidade dos navegadores WWW de executar código Java – em forma de *applets* – para formar grids com um número potencial de participantes muito elevado e sem a necessidade de se instalar e configurar software – além do navegador WWW – em cada um dos computadores. O principal problema desta abordagem é o modelo de segurança extremamente restritivo imposto aos *applets* Java – chamado de *sandbox* – que impede que os *applets* executem um conjunto de operações que são vitais para um grid. Apesar de que algumas dessas restrições já foram levantadas em versões mais recentes do JDK, elas influenciaram decisivamente o projeto de alguns dos sistemas apresentados aqui.

JeT [31] é um sistema baseado na execução de *applets* Java que está limitado à aplicações mestre-escravo em que as tarefas escravas não armazenam informação de estado (ou seja, são *stateless*). Elas são implementadas em *applets* Java que são descarregados nos computadores participantes no sistema. O processo mestre é uma tarefa especial que precisa ser executada como uma aplicação Java no mesmo computador em que está localizado o servidor Web que descarregou os *applets* Java. Para aumentar a eficiência, a comunicação no JeT é baseada em UDP (Universal Datagram Protocol). O ambiente de execução garante a ordenação entre as diferentes mensagens e a entrega sem erros. O JeT implementa também um algoritmo de tolerância a falhas muito simples, pois ele precisa se preocupar somente com a tarefa mestre. As tarefas escravas podem falhar sem afetar a execução da aplicação, pois basta realocar o trabalho que ela estava executando a outra tarefa escrava. Para que esse esquema funcione, é vital que as tarefas escravas não mantenham informação de estado e nem alterem de forma permanente o ambiente de execução (por exemplo, escrevendo em um arquivo).

Javelin [28] é também um sistema baseado em *applets* Java, desenvolvido como o sucessor de SuperWeb [1]. O sistema contém 3 componentes fundamentais: os clientes, os servidores e os *brokers*. O cliente é um processo que necessita recursos de processamento para executar uma determinada aplicação. O servidor é um processo que oferece recursos de computação para serem utilizados por aqueles que os necessitem, e o broker é um processo que coordena a interação de clientes com servidores. Tanto os clientes como os servidores podem ser implementados através de um navegador WWW. O cliente pode utilizar o navegador para carregar (*upload*) código no broker. O servidor pode se conectar ao broker utilizando um navegador e receber dele uma ou várias tarefas para serem executadas. No caso do servidor, o navegador tem

que ter capacidade para executar *applets* Java. Por causa das limitações do sistema proposto, os autores propuseram o Javelin++ [27], que é baseado em RMI ao invés de UDP e troca o uso de *applets* por aplicações Java. Além disso, o sistema implementa uma "rede de brokers" para aumentar a escalabilidade.

Bayaniham [36] é um sistema que combina as vantagens derivadas do uso da linguagem Java – como a independência de plataforma e a possibilidade de executar código dentro de navegadores WWW – com as vantagens de utilizar um modelo de objetos distribuídos padrão, como CORBA [35]. Bayaniham utiliza uma implementação de CORBA chamada HORB [37] que pode ser utilizada em aplicações e *applets* Java. O sistema usa a orientação a objetos de forma extensiva. Para programar uma aplicação para Bayaniham basta escolher os componentes gerais que melhor se adaptem às necessidades da aplicação e estendê-los – mediante o uso da herança – para incluir funcionalidades particulares da aplicação. O artigo citado usa como aplicação protótipo um exemplo pouco convencional e bem interessante: uma aplicação de *Web crawling*, ou seja, uma aplicação que, dado um link inicial de uma página Web, segue recursivamente todos os links que se derivam do inicial. O exemplo mostra que um grid pode ser visto não somente como uma soma de poder de cômputo, mas também como uma soma de largura de banda para resolver alguns problemas.

PopCorn [30] implementa uma máquina paralela virtual baseada também em *applets* Java. A principal diferença de PopCorn em relação às outras propostas está relacionada com o mecanismo utilizado para incentivar os donos dos computadores a participarem do sistema. PopCorn utiliza um mecanismo de pagamento baseado em regras de mercado como mecanismo de estímulo. O modelo de programação proposto pelo sistema é baseado em *computelets*, que são entidades executadas de forma independente. Cada *computelet* contém o código e os dados de que precisa para executar uma tarefa.

KnittingFactory [6] é um sistema baseado em *applets* Java que foi projetado para servir de base a outros sistemas de mais alto nível, e não para ser programado diretamente. KnittingFactory implementa 3 serviços fundamentais: o serviço de diretórios, que permite a usuários usando navegadores WWW encontrar outros membros de uma sessão distribuída, o serviço de classes, que permite a um usuário iniciar um processamento em qualquer computador do sistema, e o serviço de *applets*, que permite a comunicação direta entre *applets*. Assim como outras propostas, KnittingFactory depende de uma aplicação servidora para cada aplicação paralela implementada. No entanto, KnittingFactory implementa um servidor WWW bem simples que lhe permite colocar a aplicação servidora em qualquer computador, e não obrigatoriamente no mesmo computador em que está o servidor WWW principal do site.

2.3 Sistemas mistos

Embora não constituam uma maioria, algumas propostas tentam combinar as vantagens de Java como ambiente independente de plataforma e o bom desempenho obtido pelos sistemas dependentes. Desta forma é possível também aproveitar código já existente e fazer a transição para Java de forma gradual.

JAVADC [10] é um sistema baseado em Java e WWW cujo objetivo principal é a execução de aplicações paralelas tipo SPMD (Single Program, Multiple Data) que usam PVM e MPI. As aplicações paralelas são implementadas usando o PVM, o pPVM ou o MPI. Java, e em particular

os *applets* Java, são utilizados somente para implementar um sistema cliente-servidor com interface gráfica que permite configurar um cluster remoto de estações de trabalho, executar uma aplicação paralela nesse cluster e monitorar seu progresso. Este esquema permite configurar clusters e executar aplicações paralelas nos mesmos desde qualquer parte do mundo.

ATLAS [5] é um sistema baseado em Java e Cilk [7], uma linguagem paralela junto com um ambiente de execução baseado em C e que permite várias linhas de controle (*threads*). ATLAS manipula os recursos do grid de forma transparente ao programador. O sistema permite a utilização de código nativo para aumentar o desempenho das aplicações, ao custo da diminuição da portabilidade. Parte do ambiente de execução também não é baseado em Java, por ser herdado de Cilk. ATLAS apresenta uma arquitetura similar à de Javelin. O sistema é composto de *servidores de computação*, que executam um daemon e se colocam à disposição do *administrador*. O administrador recebe pedidos dos *clientes* para executar código nos servidores de computação. Durante a execução, se um servidor está sem trabalho, ele toma (rouba) trabalhos de outro servidor escolhido aleatoriamente. O esquema de obtenção (roubo) de trabalho é organizado de forma hierárquica, para aumentar a escalabilidade do sistema.

2.4 Conclusões

Este capítulo apresentou alguns dos projetos de grids encontrados na literatura. Os mesmos foram classificados de acordo com a solução dada ao problema da heterogeneidade de recursos. Foram identificadas 3 grandes categorias: os sistemas dependentes de plataforma, os sistemas independentes de plataforma e os sistemas que utilizam uma combinação de ambas as técnicas. Os sistemas independentes de plataforma foram divididos em duas sub-categorias: sistemas baseados em aplicações Java e sistemas baseados em *applets* Java. JoiN, o sistema projetado e implementado neste trabalho, se enquadra na categoria de sistemas independentes de plataforma baseados em aplicações Java. No entanto, JoiN apresenta várias características essenciais que o diferenciam de outras propostas e que possibilitam que ele trabalhe com um grande número de computadores. Estas características serão descritas em detalhes na Parte III do texto.

Referências Bibliográficas

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, June 1997.

[2] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1), Fevereiro 1995.

[3] Nazareno Andrade, Walfredo Cirne, and Francisco Brasileiro. Ourgrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[4] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users guide to netsolve. Innovative Computing Department Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002. Also in the proceedings of Supercomputing'96, Pittsburgh.

[5] J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[6] Arash Baratloo, Mehmet Karaul, Holger Karl, and Zvi M. Kedem. An Infrastructure for Network Computing with Java Applets. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, Standford University, Palo Alto, California, February 1998.

[7] Robert D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth Symposium on Principles and Practice of Parallel Programming*, 1995.

[8] Tim Brecht, Harjinder Sandhu, Meijuan Shan, and Jimmy Talbot. ParaWeb: Towards World-Wide Supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[9] Denis Caromel and Julien Vayssiere. A Java Framework for Seamless Sequential, Multithreaded, and Distributed Programming. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, Standford University, Palo Alto, California, February 1998.

[10] Zhikai Chen, Kurt Maly, Piyush Mehrotra, Praveen K. Vangala, and Mohammad Zubair. Web based Framework for Distributed Computing. In *Proceedings of the ACM 1997 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997.

[11] Walfredo Cirne and Keith Marzullo. Open grid: A user-centric approach for grid computing. In *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing*, September 2001.

[12] Oliver Coulaud and Eric Dillon. PARA++: C++ bindings for Message Passing Libraries. Technical report, Institut National de Recherche en Informatique et en Automatique, 1995. Versão 2.1.

[13] César A. F. de Rose, Fausto Blanco, Nicolas Maillard, Katia Saikoski, Reynaldo Novaes, Olivier Richard, and Bruno Richard. The virtual cluster: a dynamic environment for exploitation of idle network resources. In *14th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2002)*, 2002.

[14] Adam J. Ferrari. JPVM; Network Parallel Computing in Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, Standford University, Palo Alto, California, February 1998.

[15] Ian Foster, Carl Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002.

[16] Geoffrey C. Fox and Wojtek Furmanski. Java for parallel computing and as a general language for scientific and engineering simulation and modelling. *Concurrency: Practice and Experience*, June 1997.

[17] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-G: A computation manager agent for multi-institutional grids. *Journal of Cluster Computing*, 5:237–246, 2002.

[18] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Computer. A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The java language specification. Sun Microsystems, 2002.

[20] Andrew S. Grimshaw and Wm. A. Wulf. Legion - A View From 50,000 Feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, Agosto 1996. IEEE Computer Society Press.

[21] Dennis Kafura and Liya Huang. mpi++: A C++ language binding for MPI. In *Proceedings MPI Developers Conference*, South Bend, Indiana, USA, June 1995. University of Notre Dame.

[22] L. V. Kalé, Milind Bhandarkar, and Terry Wilmarth. Design and Implementation of Parallel Java with Global Object Space. In *Proceedings of Conference on Parallel and Distributed Processing Technology and Applications*, Las Vegas, Nevada, 1997.

[23] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the* 10^{th} *International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.

[24] L. V. Kale and S. Krishnan. Charm++: A Portable Concurrent Object Oriented System based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[25] Thilo Kielmann, Henri E. Bal, Jason Maasen, Rob van Nieuwpoort, Ronald Veldema, Rutger Hofman, Ceriel Jacobs, and Kees Verstoep. The albatross project: Parallel application support for computational grids. In *Proceedings of the* 1^{st} *European Grid Forum Workshop*, pages 341–348, Poznan, Poland, April 2000.

[26] Yoshimasa Masuoka, Toyohiko Kagimasa, Katsuyoshi Kitai, Fumio Noda, Jinghua Min, and Shigekazu Inohara. A Facility Providing Flexible Processor Pools for WWW/Java-based Distributed Applications. In *WWW6 - Sixth International World Wide Web Conference*, Santa-Clara, April 1997.

[27] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins, and Peter Capello. Javelin++: Scalability Issues in Global Computing. In *Proceedings of the ACM 1999 Java Grande Conference*, Palo Alto, California,, June 1999.

[28] Michael O. Neary, Bernd O. Christiansen, Peter Capello, and Klaus E. Schauser. Javelin: Parallel Computing on the Internet. *FGCS Special Issue on Metacomputing*, 1999. Elsevier Science, Amsterdam, Netherlands.

[29] Rob Van Nieuwpoort, Jason Maasen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing using the remote invocation model. *Concurrency: Practice and Experience*, 12(8):643–666, 2000.

[30] Noam Nisan, Shmulik London, Ori Regev, and Noam Camiel. Globally Distributedcomputation over the internet - The POPCORN project. In *WWW6 - Sixth International World Wide Web Conference*, Santa-Clara, April 1997.

[31] Hernâni Pedroso, Luis M. Silva, and João Gabriel Silva. Web-based Metacomputing with JET. *Concurrency Practice and Experience*, 9(11):1169–1173, Nov 1997.

[32] Michael Philippsen. Is Java ready for computational science? In *Proceedings of the* 2^{nd} *European Parallel and Distributed Systems Conference*, pages 299–304, Vienna, Austria, July 1998.

[33] Michael Philippsen and Matthias Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[34] Jim Pruyne and Miron Livny. Interfacing Condor and PVM to Harness the Cycles of Workstations Clusters. *Journal of Future Generation Computer Systems*, (12):67–85, 1996.

[35] Arno Puder and Kay Romer. *CORBA: Theory and Practice*. Elsevier, 2001.

[36] Luis F. G. Sarmenta, Satoshi Hirano, and Stephen A. Ward. Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Standford University, Palo Alto, California, February 1998.

[37] Hirano Satoshi. Distributed Execution of Java Programs. In *WorldWide Computing and Its Applications*. Springer Lecture Notes in Computer Science 1274, 1997.

[38] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[39] Jeffrey M. Sqyres, Brian C. McCandless, and Andrew Lumsdaine. Object Oriented MPI. A Class Library for the Message Passing Interface. In *Proceedings of the 1996 Parallel Object-Oriented Methods and Applications Conference*, Santa Fe, New Mexico, February 1996.

[40] Vaidy S. Sunderam. Metacomputing with the harness and icet systems. In *International Conference on Computational Science*, 2001.

[41] Hiromitsu Takagi, Satoshi Matsuoka, Hidemoto Nakada, Satoshi Sekiguchi, Mitsuhisa Satoh, and Umpei Nagashima. Ninflet: A Migratable Parallel Objects Framework using Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford University, Palo Alto, California, USA, February 1998.

[42] Douglas Thain, Todd Tanenbaum, and Miron Livny. Condor and the Grid. In Fran Berman, Anthony J. G. Hey, and Geoffrey Fox, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[43] Eduardo Javier Huerta Yero, Marco A. Amaral Henriques, Javier R. García, and Alina C. Leyva. JOINT: An Object Oriented Message Passing Interface for Parallel Programming in Java. In *Java for High Performance Computing Workshop, High Performance Computing Networks 2001*, Amsterdã, Holanda, June 2001.

Parte II

Fundamentos Teóricos

Capítulo 3

Predição de tempo de execução de aplicações

O objetivo deste capítulo é apresentar uma proposta para estimar o tempo de execução de uma aplicação paralela em um grid. Um modelo que consiga predizer o tempo de execução de uma aplicação pode ser útil para o escalonador do sistema e até para os usuários do mesmo, que podem determinar o tamanho da instância do problema que pode ser resolvido em um tempo razoável.

Neste capítulo é proposta uma estratégia para modelar a execução de uma aplicação paralela em um sistema distribuído heterogêneo (categoria à qual pertencem os grids) baseada em uma combinação de duas abordagens existentes. A primeira é PAMELA, uma linguagem para a modelagem de sistemas paralelos que tem a capacidade de considerar características dinâmicas do sistema, como a contenção pelo uso de recursos, mesmo fazendo uma análise estática do mesmo (isto é, sem fazer simulações). Esta abordagem é estendida por uma segunda (intervalos) para considerar a contenção induzida por agentes externos à aplicação paralela, como outros processos executando em um computador do sistema. A extensão faz uso de intervalos de valores para representar o desempenho dos recursos que formam o sistema, no lugar dos valores escalares usados por PAMELA. Assim, ao invés do desempenho de um processador ser representado por um escalar (por exemplo, x Gigaflops), ele é representado por um intervalo de possíveis valores ($[x_{min}, x_{max}]$ Gigaflops). É mostrado o impacto que a inclusão dos intervalos tem na qualidade das predições feitas por PAMELA.

A abordagem é explicada com detalhes no artigo *"Contention-Sensitive Static Performance Prediction for Parallel Distributed Applications"*.

Contention-Sensitive Static Performance Prediction for Parallel Distributed Applications*

Eduardo J. Huerta Yero† and Marco A. Amaral Henriques
DCA - FEEC - UNICAMP
[huerta,marco]@dca.fee.unicamp.br

Abstract

Performance prediction for parallel applications running in heterogeneous clusters is difficult to accomplish due to the unpredictable resource contention patterns that can be found in such environments. Typically, components of a parallel application will contend for the use of resources among themselves and with entities external to the application, such as other processes running in the computers of the cluster. The performance modeling approach should be able to represent these sources of contention and to produce an estimate of the execution time, preferably in polynomial time. This paper presents a polynomial time static performance prediction approach in which the prediction takes the form of an interval of values instead of a single value. The extra information given by an interval of values represents the variability of the underlying environment more accurately, as indicated by the practical examples presented.

1 Introduction

Performance prediction plays an important role in parallel systems. In order to better exploit the computational resources available, parallel systems designers often use performance prediction techniques to assist them during the design phase of the system. In runtime, the parallel system can use these techniques to predict the execution time of the application under different schedulings, and thus help determine the best scheduling for the application.

One of the main problems faced by performance prediction techniques is how to model and analyze the contention on the use of resources. If the parallel application is executed in a distributed environment, the parallel components will have to contend for the resources with entities external to the parallel application, such as other processes running on the computers that form the distributed system. This type of contention will be referred to as **external contention**.

External contention is very hard to model since it would imply that every possible entity external to the application would have to be modeled as well. Instead, typical approaches

*Work partially supported by grant 98/04305-9 of the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

†On release from University of Havana, Cuba

use intervals of possible values to represent the availability of resources in the system [2, 5]. Parallel applications are then modeled by a paradigm, such as Structural Models [8, 9, 10], using intervals instead of point values to represent resource availability.

Aside from external contention, there are other sources of resource contention in a distributed environment. The parallel application itself induces contention on the resources, since a common parallel application is formed by several parallel components, which will typically compete for resources during their lifetimes. This is particularly the case when several parallel components of the same parallel application are executed on the same computer of a distributed system. This type of contention will be referred to as **internal contention**, to distinguish it from the contention induced by external entities.

Internal contention is easier to model, since it only involves the parallel components forming the application. However, due to the dynamic nature of the execution, analyzing internal contention involves considering all possible execution traces, which can imply in exponential complexity, as it is the case with Stochastic Petri Nets [6] and process algebras [12]. Even polynomial solutions, such as those derived from combinations of directed acyclic task graphs and queueing networks [13, 14] are still costly for large problem sizes, mostly because they are based on numerical analysis.

Static symbolic techniques provide a much simpler alternative. These techniques rely only upon the static information available in the model, such as the precedence relations among the parallel components, to produce a symbolic model which can later be evaluated for different problem and system settings. The symbolic model is able to capture regularities in the parallel programs and machines and exploit them through symbolic simplifications. This may reduce the evaluation cost in many orders of magnitude. PAMELA [3, 4] is a symbolic approach for performance modeling and analysis. It includes a form of contention analysis while still conserving the polynomial time characteristics of the static solutions. PAMELA has been successful in modeling concurrent systems due to the fact that it is based in an imperative process-oriented language, which eases the task of automating the generation of the performance model from the original program. However, it is still difficult for PAMELA to account for external contention.

The goal of this paper is to present a static performance analysis technique that accounts for both external and internal contention. The proposed solution builds upon the approach taken by PAMELA to represent internal contention, and extend it to use intervals of values to account for external contention. The intervals substitute point values as representations of resource performance (links, processors) in the distributed cluster. The paper demonstrates that this approach conserves the prediction accuracy exhibited by PAMELA for a parallel application running in a heterogeneous cluster, while producing extra information which can be useful to deal with the oscillations in resource availability in such environments.

It is worth noticing that the proposal presented here does not intend to be a new method for static performance prediction, but an extension of an existent method, in this case PAMELA. Being so, the paper does not focus in proving the accuracy of the method itself, which has been extensively reported [3, 4]. The results presented here focus on the additional information brought by using intervals in this method and on interval-related issues, such as the algorithm for constructing intervals from a set of point values. The paper argues that the combination of the ability of PAMELA to predict execution time with the extra information gathered by using intervals increase the quality of the information produced by the original PAMELA.

The remaining of the paper is organized as follows. Section 2 introduces modeling in PAMELA, as well as the techniques it uses to analyze a model. Section 3 introduces the basic definitions and arithmetic operations over intervals, as well as a new approach to interval construction proposed in this paper. Section 4 shows how to combine PAMELA with the use of intervals to obtain more descriptive predictions. Finally, Section 5 presents the conclusions of the paper.

2 PAMELA

2.1 Definitions

As stated in the introduction, the solution presented here will build upon the ability of PAMELA to perform static contention analysis and extend it to work with intervals. PAMELA is an imperative, process-oriented language with a relatively small set of instructions. The paper will present only the subset of instructions necessary to understand the approach. A more detailed description of the language can be found in [3] and [4].

delay(τ): Increments the virtual time by a value τ. If a model L is given by $L = delay(\tau)$, then the time T associated with L is $T = \tau$.

; sequential operator: Specifies a strict sequence between two processes or submodels. For example, if $L = delay(\tau_1); delay(\tau_2)$ then $T = \tau_1 + \tau_2$.

|| parallel operator: Specifies two processes running in parallel. If $L = delay(\tau_1)||delay(\tau_2)$, then $T = \max(\tau_1, \tau_2)$.

seq(i=a,b) L_i: The sequential reduction operator. By definition, $seq(i = a, b)L_i = L_a; ...; L_b$.

par(i=a,b) L_i: The parallel reduction operator. By definition, $par(i = a, b)L_i = L_a||...||L_b$.

use(U, τ): Indicates that resource U will be used for a time τ. This instruction is employed to model the contention on the use of resources. There may be more than one instance of resource U, so $\#U$ is defined to be the cardinality of U.

As an example, the next section presents the PAMELA model of a program which multiplies two matrixes.

2.2 Example: The PAMELA model for the matrix multiplication problem

The matrix multiplication problem is commonly used as an example in parallel environments, due to the importance of matrix manipulation in scientific computing. In the approach presented here, two $N \times N$ matrixes, M_1 and M_2, will be multiplied in parallel. The algorithm uses one master task and S slave tasks. The master task is responsible for subdividing matrix M_1 in S submatrixes $(N/S) \times N$ and sending them to the S slaves along with M_2 (for simplicity, let us assume that $N \bmod S = 0$). Each slave then multiplies the $(N/S) \times N$ submatrix by M_2

and returns the result to the master. Finally, the master task merges the partial results into the final result.

The objective of PAMELA is to develop a model capable of reflecting the behavior of a parallel application in a distributed heterogeneous environment. Such an environment is formed by a number of non-dedicated heterogeneous computers connected by a network. Therefore, the execution times of the basic low-level operations may vary from machine to machine. The model presented here reflects this fact by indexing the basic operations with the machine number i, $0 \leq i \leq P$, where $P+1$ is the number of computers (P worker computers and one coordinator). The model is defined as follows.

```
L = Split;Process;Gather
Split = seq(j=1, N*N) {
            use(comp_0, a * t_op0) //split matrix M_1
        }
        seq(i=1,P){
            use(link, (S/P)(N * N/S + N * N) * t_send0i) // send data to slaves
        }
Process = par(i=1,P){
              par(j=1,S/P){
                  seq(k=1, N*N*N/S) {
                      use(comp_i, b * t_opi); // submatrix × matrix
                  }
                  use(link,(N * N/S) * t_sendi0); // send data back to Master
              }
          }
Gather = seq(j=1, N*N) {
             use(comp_0,c * t_op0) // generate final matrix
         }
```

To avoid presenting the algorithm in excessive detail we have assumed that a, b and c are the number of basic operations (array access, assignment, sum, multiplication, integer division, integer rest) executed on each iteration to split, multiply and gather matrix elements. Parameter t_{op_i} is a scalar value representing the time spent on executing an operation in computer i. For the sake of simplicity the model considers these operations to be equivalent in terms of execution time, although in practice this is not the case. A more detailed model would define t_{add_i}, t_{access_i} and so on as individual operations at the cost of more complexity. The parameter $t_{send_{i0}}$ represents the time required to send a matrix element from computer i to computer 0. Again, a more detailed model for $t_{send_{i0}}$ is possible, considering for instance network topology and protocol overhead, at the cost of increased complexity.

This model makes two more assumptions. First, the Master task, which is formed by subtasks *Split* and *Gather*, is always executed in computer 0. Second, the distributed parallel machine uses a naive scheduling algorithm whose main goal is to maintain the number of tasks running in each computer balanced (each worker executes S/P slaves).

2.3 Performance Analysis

It is not a purpose of this paper to present the PAMELA performance analysis algorithm used for the general case[3]. Instead, we will present the basic ideas of the process by analyzing the matrix multiplication example.

Performance analysis in PAMELA is made by establishing a tight lower bound T^l to the execution time T. The lower bound was chosen as an estimate because statistical studies indicate that, with random resource usage, the mean execution time is closer to the lower bound than to the upper bound [3]. Besides that, the lower bound is easier to calculate than the upper bound.

To calculate T^l two different lower bounds are first estimated. Let L be the model being analyzed. The two lower bounds $\varphi(L)$ and $\omega(L)$ are characterized below.

$\varphi(L)$**:** Execution time of model L disregarding resource contention. This analysis, also known as "critical path analysis", assumes that every resource involved in model L has cardinality $+\infty$, and focuses only in precedence relations among the processes involved in the model. As a consequence, every use(U, τ) operation in model L becomes simply a delay(τ). This is the type of analysis made in traditional static approaches, such as Structural Models [8, 9, 10].

$\omega(L)$**:** Limit to the execution time established by resource contention. The analysis of this limit involves only *use* constructs, and focuses on determining which resource will take the longest to attend all the requests made to it in model L.

Basically, $\varphi(L)$ establishes a lower bound on the execution time in the ideal world, where no resource contention exists. On the other hand, $\omega(L)$ estimates the limit imposed by the slowest contended resource in the model. Thus, the lower bound T^l for a model L can be given by the equation

$$T^l \ = \ \max(\varphi(L), \omega(L)). \tag{1}$$

Let us first analyze $\varphi(L)$ for the matrix multiplication. As stated above, $\varphi(L)$ does not consider resource contention, so all $use(r, \tau)$ operations become $delay(\tau)$.

```
L = Split;Process;Gather
Split = seq(j=1, N*N) {
            delay(a * t_op0) //split matrix M1
        }
        seq(i=1,P){
            delay((S/P)(N * N/S + N * N) * t_send0i) // send data to slaves
        }
Process = par(i=1,P){
              par(j=1,S/P){
                  seq(k=1, N*N*N/S) {
                      delay(b * t_opi); // submatrix × matrix
                  }
                  delay((N * N/S) * t_sendi0); // send data back to Master
```

```
            }
        }
Gather = seq(j=1, N*N) {
            delay(c * t_op0) // generate final matrix
        }
```

The following three equations can be used in the reduction process to analyze the sequential and parallel operators.

$$\varphi(delay(\tau)) = \tau \tag{2}$$

$$if\ L = seq(i = a, b)L_i = L_a; ...; L_b\ then\ \varphi(L) = \sum_{i=a}^{b} \varphi(L_i) \tag{3}$$

$$if\ L = par(i = a, b)L_i = L_a||...||L_b\ then\ \varphi(L) = \max_{a \le i \le b} \varphi(L_i) \tag{4}$$

Observe that Eq. 4 implicitly states that there is no contention in the execution of parallel models, since the time required to execute the model is equal to the maximum value of the time required to execute a submodel. This approach will be later contrasted to that of $\omega(L)$.

Using these equations it is possible to reduce the model to an analytic expression.

$$L = Split; Process; Gather \tag{5}$$

$$\varphi(L) = \varphi(Split) + \varphi(Process) + \varphi(Gather) \tag{6}$$

$$\varphi(Split) = a * N * N * t_{op_0} + \sum_{i=1,P} (\frac{S}{P}(N * N/S + N * N) * t_{send_{0i}} \tag{7}$$

$$\varphi(Process) = \max_{i=1,P}(\max_{j=1,S/P}(b * N * N * (N/S) * t_{op_i}) + N * (N/S) * t_{send_{i0}}) \tag{8}$$

$$\varphi(Gather) = c * N * N * t_{op_0} \tag{9}$$

Resuming, $\varphi(L)$ produces an analytical estimative of the execution time considering only precedence relationships among tasks and disregarding resource contention.

To analyze resource contention PAMELA focuses on the workload imposed to each resource by the model. In the matrix multiplication example one can identify three resources: computer 0, where the Master tasks executes, computer $i, 1 \le i \le P$, where Slave tasks execute and the network link, which carries information from Master to Slaves and vice-versa. Considering $\omega_{comp_0}(L), \omega_{comp_i}(L)$ and $\omega_{link}(L)$ as the time imposed by the workload in the model to each of these resources, we have

$$\omega(L) = \max(\omega_{comp_0}(L), \max_{i=1,P}(\omega_{comp_i}(L)), \omega_{Link}(L)). \tag{10}$$

To calculate the elements of Eq. 10 the process should focus on each resource individually. For example, $comp_0$ is responsible for executing a part of *Split* and *Gather*. Therefore, $\omega^{comp_0}(L)$ can be calculated as follows.

$$\omega_{comp_0}(L) = a * N * N * t_{op_0} + c * N * N * t_{op_0} = (a + c) * N * N * t_{op_0} \quad (11)$$

Similarly

$$\omega_{comp_i}(L) = b * N * N * N/S * t_{op_i} \quad (12)$$

and

$$\omega_{link}(L) = \sum_{i=1}^{P}(\frac{S}{P}(N * N/S + N * N) * t_{send_{0i}}) + \sum_{i=1}^{P}\frac{S}{P}(N * N/S)t_{send_{i0}} \quad (13)$$

Considering $t_{send_{i0}} = t_{send_{0i}}, \forall i$ we can reduce it to

$$\omega^{Link}(L) = \sum_{i=1}^{P}(\frac{N * N}{P}(S + 2)t_{send_{i0}}) \quad (14)$$

So, for the matrix multiplication example the prediction made by PAMELA is given by Eq. 1, where $\varphi(L)$ and $\omega(L)$ are given by Eq. 6 and Eq. 10 respectively.

2.4 Performance prediction with PAMELA

This section presents some practical results obtained to validate the ideas proposed. The matrix multiplication problem was implemented and executed in JoiN [1], a distributed parallel environment based in Java. The tests were conducted using 8 machines with the following configurations: a Sun SparcStation4 with a 110 MHz CPU and 64 Mb RAM, a Sun Ultra10, with an UltraSPARC-II CPU, 360 MHz clock and 256 Mb RAM, a PC with a 800MHz Pentium III CPU and 384 Mb RAM, a Sun SparcStation 5, 70 MHz CPU and 32 Mb RAM, a Sun SparcStation5 with a 110 MHz CPU and 32 Mb RAM, a Sun Ultra1, with a UltraSparc CPU, 143 MHz and 64 Mb RAM, a Sun SparcStation4, 110 MHz CPU and 160 Mb RAM, and a Sun Ultra60, 360 MHz UltraSPARC-II CPU and 512 Mb RAM. All Sun stations were running under SunOs release 5.7, and the PC was using Linux 1.2. The computers were connected by a local area network based on Ethernet (10 Mbps) and Fast Ethernet (100 Mbps). The environment was explicitly constructed to be heterogeneous to make the task of obtaining successful predictions difficult. The values used for a, b and c were 5,4 and 2 respectively, representing the number of operations made in each iteration by the algorithm.

We developed a specialized application in JoiN to gather the information necessary to make the predictions. The application was executed for a relatively long period of time (typically around 3 hours) at different times of the day and days of the week. The data it produced was used to calculate mean values for t_{op_i} and $t_{send_{i0}}$ from the individual measures in each computer and then used in Eq. 1, Eq. 6 and Eq. 10 to make the performance predictions. Finally, the real matrix multiplication application was executed and the results compared with the predictions made.

During the tests, over 40 matrix multiplications were executed with different number of slave tasks, which were evenly distributed among the computers. Fig. 1 shows a comparison between the PAMELA prediction and the execution times.

Observe that, although the predictions made by PAMELA are supposed to be lower bounds to the execution times, some actual execution times are smaller than the predicted values. The

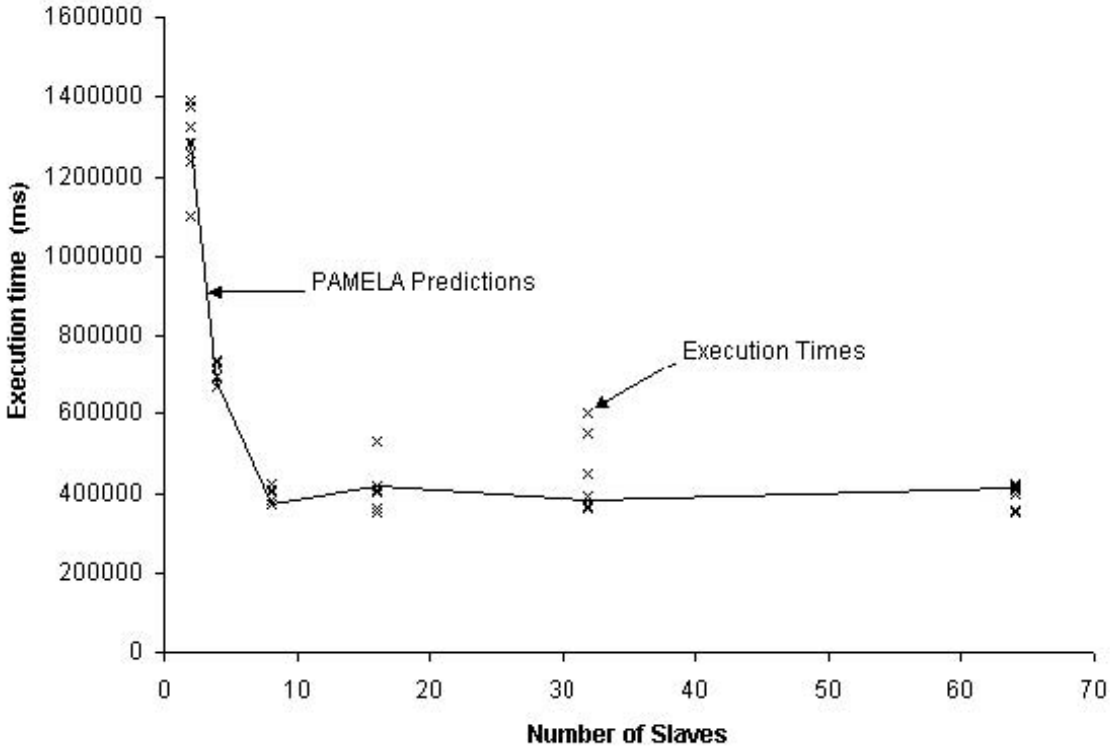

Figure 1: Predictions vs. real Values for PAMELA.

cause of this "anomaly" is that the actual values for t_{op_i} and $t_{send_{i0}}$ during the execution of some instances were smaller than the mean values, thus rendering a smaller execution time. The fact that we chose the matrix multiplication – a regular, well-behaved parallel application – as the example to show that the oscillations in execution time are caused by the oscillations in resource availability.

The rest of this paper presents a proposal to use intervals instead of point values to represent resource availability. As a result predictions will take the form of intervals of values and bring more information to the end user about the predicted behavior of the application.

3 The use of intervals

Intervals are typically constructed from a set of point values representing different measures of a resource over time. For example, one can collect a set of point values derived from the execution of some benchmark in a computer, and then construct an interval that represents the performance behavior of that computer. There are two desired characteristics for such an interval: (a) The interval has to capture the core behavior of the resource, and (b) the interval should be as small as possible.

The probabilistic distribution of values inside the interval also plays an important role in describing the behavior of a resource. Between the lower and upper bounds of the interval the values can distribute according to a known distribution function (e.g. Uniform, Normal, Poison). However, due to the dynamic and unpredictable usage patterns of the resources involved in a distributed heterogeneous environment, there is not – to the best of our knowledge – a well-established study associating the performance characteristics of processors or networks to a specific distribution function.

There are two workarounds for this problem. It is possible to use histograms inside the intervals [11] or simply assume that values distribute uniformly inside the interval. The first approach uses the point values derived from the benchmarks to construct an histogram representing the probability of a value falling in a specific portion of the interval. This approach produces a final estimative in the form of an interval with an histogram indicating the probability of the execution time falling in each portion of the interval. Although it produces a slightly more informative output, the interval arithmetic associated with this approach is more complex and could hinder a clear presentation of the message stated by this paper. Being so, the rest of this paper we will assume that values inside the interval distribute uniformly, and thus define an interval based only in its lower and upper bounds.

Definition

An interval X is defined as the tuple

$$X = [\underline{x}, \overline{x}]; \;\; where \; \{x \in X | \underline{x} \leq x \leq \overline{x}\} \tag{15}$$

where $\underline{x}$ is known as the *lower bound* and $\overline{x}$ is the *upper bound* of the interval.

It is also necessary to define some basic arithmetic operators over intervals. Let X and Y be intervals and p be a scalar value; then it is possible to define the operations below.

$X + Y$	$[\underline{x} + \underline{y}, \overline{x} + \overline{y}]$
$p + X$	$[p + \underline{x}, p + \overline{x}]$
$p * X$	$[p * \underline{x}, p * \overline{x}]$
$\max(X, Y)$	$[\max(\underline{x}, \underline{y}), \max(\overline{x}, \overline{y})]$

3.1 Interval construction

Naive solutions to interval construction, which take the minimum and maximum values of the set of point values (or the second minimum and second maximum) succeed at capturing the core behavior of the resource, but often produce intervals larger than necessary. Other approaches are based on statistical measures such as mean and standard deviation, typically constructing an interval of the form $[mean - K * stddev, mean + K * stddev]$ for some value of K. These approaches work well for normally distributed measures, since there is a well-known relation between the value of K and the amount of values inside the interval (e.g. for $K = 2$ it is guaranteed that 95.45% of the values are inside the interval). However, if the distribution is not normal, this approach fails both to capture the core behavior of the resource and to produce small intervals.

A simple and efficient algorithm was developed to construct intervals with the desired characteristics. This algorithm is able to capture the behavior of the resource while maintaining the interval size as small as possible. The problem was stated in the following terms: Let $S = \{x_1, x_2, ..., x_N\}$ be an ordered sequence of point values produced by the individual tests conducted on the resource. The problem is to find the smallest interval X' which contains at least $C\%$ of the point values.

To simplify the notation, let us define $X_{i,j} = [x_i, x_j]$. This notation allows us, for example, to refer to the interval formed by the minimum and the maximum of S as $X_{1,N}$. The idea of the algorithm is to discard D points from S, where

$$D = \left\lfloor \frac{(100 - C)N}{100} \right\rfloor \tag{16}$$

In other words, if D is less or equal to the $(100 - C)\%$ of N, it is guaranteed that discarding D points from S produces a set with at least $C\%$ of the points in S. Since points can only be discarded from the beginning or the end of S, the goal is to find an interval of the form

$$X_{i,j};\ \ 1 \leq i \leq D + 1 \ and \ j = N - D + i - 1 \tag{17}$$

An interval $X_{i,j}$ discards $i - 1$ points from the beginning of S and $N - j$ points from the end. With j defined as above, the sum $(i - 1) + (N - j)$ is always equal to D.

To finalize, let $A(X_{i,j}) = x_j - x_i$ be the amplitude of the interval $X_{i,j}$. The algorithm has to find

$$X' = \min_{A(X_{i,j})} (X_{i,j}); \tag{18}$$

setting i to the $D + 1$ different values in the interval $[1, D + 1]$ and calculating j from Eq.18.

Approach	Size	Percentage of points
minimum size	1	95%
$[mean - 2 * stddev, mean + 2 * stddev]$	1.7	96%
$[min, max]$	1.8	100%

Table 1: Size vs percentage of points included by each approach to construct intervals.

Setting the value of C adequately (for example, to 95%) guarantees that the core behavior of the resource is captured, while producing an interval with the smallest amplitude possible. Besides, this interval is cheaper to calculate than those based in statistical measures, since it involves an iteration over D values of S, instead of iterating over the whole set to calculate the mean and standard deviation. Figure 2 shows a comparison between this approach, the $[min, max]$ approach and the interval formed by $[mean - 2 * stddev, mean + 2 * stddev]$ with data from real CPU speed tests.

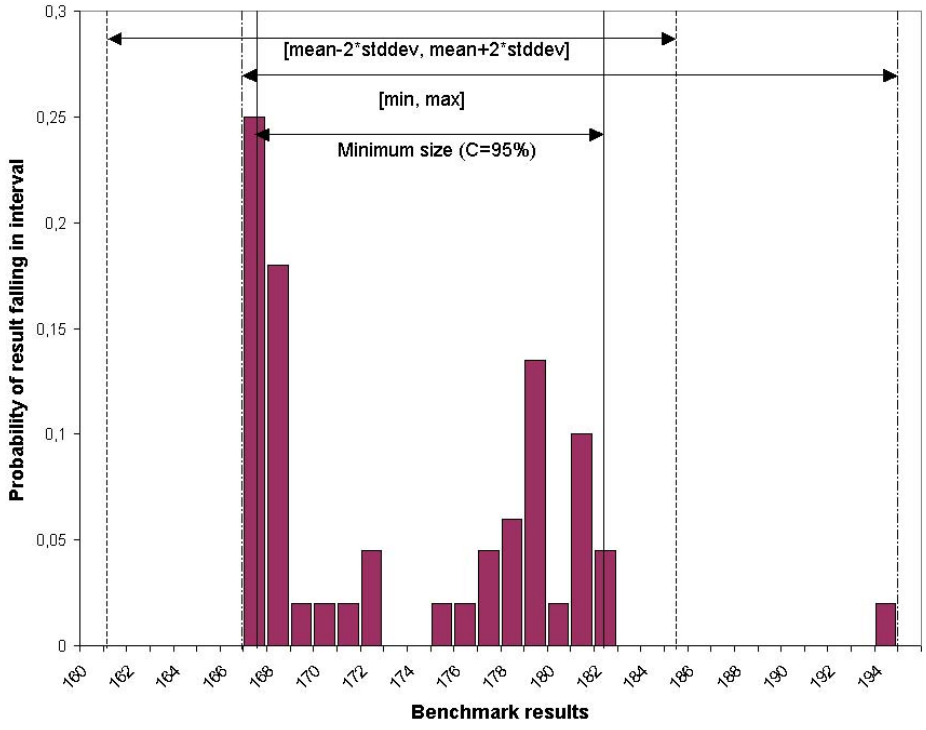

Figure 2: Comparison among intervals.

Table 1 also compares these approaches based on interval size and percentage of point values included in the interval. The size of the interval in each approach is normalized with respect to the size obtained by the algorithm presented above, which is referred to as *minimum size* approach.

4 Intervals and PAMELA

Analyzing Eq. 1, Eq. 6 and Eq. 10 it is possible to observe that the values representing resource behavior (t_{op_i} and $t_{send_{i0}}$) are involved only in additions, multiplications by a scalar and max operations. These values can be obtained from the mean of a set of point values produced by benchmarks executed in the system.

Since addition, multiplication by a scalar and max operations were also defined for intervals in Section 3 it is possible to substitute t_{op} and t_{send} in Eq. 1, Eq. 6 and Eq. 10 by an interval of values constructed from the same set of benchmarks used to calculate the mean. This approach is outlined below.

- Calculate intervals $\bar{t}_{op_i}$ and $\bar{t}_{send_{i0}}$ for $0 \leq i \leq P$ using the algorithm proposed in Section 3.1. These intervals are derived from the same set of benchmarks used to obtain results in Section 2.4.
- Substitute t_{op_i} and $t_{send_{i0}}$ by $\bar{t}_{op_i}$ and $\bar{t}_{send_{i0}}$ in Eq. 1, Eq. 6 and Eq. 10.
- Calculate the predictions, which now take the form of an interval of values.

Fig. 3 shows the predictions made by this new approach using the same environment presented in section 2.4. Observe that predictions now give a more accurate view of the expected execution times as compared to the point value prediction shown in Fig.1.

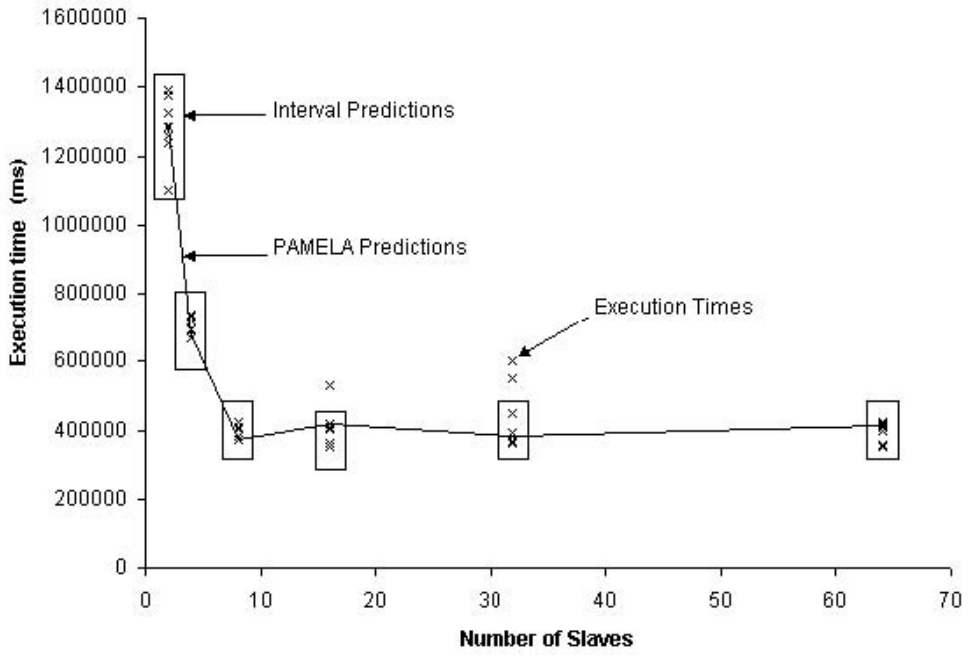

Figure 3: Interval predictions using PAMELA.

Over 90% of the actual execution times fell inside the predicted intervals. This results are not absolute, since they depend on many factors such as the quality of the benchmarks taken,

the value of C used for the construction of the intervals and the resource usage patterns during the actual execution of the applications. However, since the results presented here were obtained by executing the benchmarks and the application on a heavily used heterogeneous network, in different times of the day and different days in the week, we feel that they demonstrate the benefits of adopting this technique.

The percentage of success is not the only significant measure for a performance prediction approach based in intervals. It is also important to see how the real values are distributed inside the interval. If, for example, all real execution times were near the upper bound of the interval then augmenting the lower bound would not affect the percentage of success, although it would improve the quality of the prediction. To this end, the intervals were divided in 3 parts, and the percentage of real execution times that fell inside each of the subintervals was calculated, as well as outside the whole interval. Thus, the whole space was divided in 4 parts: Low, Center, High and Outside. The table below shows the percentage of the real execution times that fell in each of the 4 parts of the predicted interval.

Low	Center	High	Outside
21.5%	40.5%	31%	7%

It is possible to observe that the values of the predictions are well distributed inside the intervals, so that if the intervals were made smaller, the number of real execution times falling outside would increase considerably.

5 Conclusions and future work

This paper presented a static performance analysis technique based on intervals for distributed heterogeneous parallel environments. This technique effectively reflects the contention on the use of resources induced both by the tasks of the parallel application (internal contention) and by entities unrelated to the application (external contention). The effect of internal contention is estimated by using the PAMELA approach, by which the influence of resource usage in the execution time can be modeled and predicted statically. External contention is modeled by using intervals instead of point values for the measurements of the basic operations.

The resulting approach was used to predict the behavior of a parallel matrix multiplication application. The results obtained indicated that PAMELA combined with intervals was able to successfully predict the execution time in over 90% of the tests. Furthermore, the execution times were well distributed inside the prediction interval, suggesting that it could not be reduced without the risk of losing accuracy.

Future work on this subject includes the automation of the performance prediction process. This will allow to make performance predictions both to benefit end users who want to estimate the execution time of their applications and to help the scheduler of a parallel computing system to improve resource usage.

Acknowledgments

We would like to thank the creator of PAMELA, Prof. Arjan J. C. Van Gemund, from Information Technology and Systems Department of Delft University of Technology, The Netherlands, for his insightful commentaries which helped us to fully understand the PAMELA approach.

References

[1] M. A. Amaral Henriques. A proposal for Java based massively parallel processing on the web. In *Proceedings of The First Annual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing*, pages 59–66, Rhodes, Greece, June 1999.

[2] M. Braun and G. Kotsis. Interval based workload characterization for distributed systems. In *Proceedings of the International Conference on Modelling Techniques and Tools*, Lecture Notes in Computer Science. Springer, 1997.

[3] A. J. C. V. Gemund. *Performance Modeling of Parallel Systems.* Coronet Books, July 1996.

[4] A. J. C. V. Gemund. Symbolic Performance Modeling of Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, Feb. 2003.

[5] J. Luethi, S. Majumdar, G. Kotsis, and G. Haring. Performance bounds for distributed systems with workload variabilities and uncertainties. *Parallel Computing*, 22(13), Mar. 1997. (Special Issue: Distributed and Parallel Systems: Environments and Tools).

[6] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* Wiley Series in Parallel Computing. John Wiley and Sons, 1995.

[7] R. Nelson. *Probability, Stochastic Processes, and Queuing Theory: The Mathematics of Computer Performance Modelling.* Springer Verlag, 1995.

[8] J. M. Schopf. Structural prediction models for high-performance distributed applications. In *Proceedings of Cluster Computing Conference*, 1997.

[9] J. M. Schopf and F. Berman. Performance prediction using intervals. Technical Report CS97-541, University of California, San Diego, May 1997.

[10] J. M. Schopf and F. Berman. Using stochastic intervals to predict application behavior on contended resources. In *Proceedings of International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN).* IEEE Computer Society Press, 1999.

[11] J. M. Schopf. A Practical Methodology for Defining Histograms for Predictions and Scheduling. In *Parallel Computing: Fundamentals & Applications, Proceedings of the International Conference ParCo'99.* Imperial College Press, 2000, pp-664-771.

[12] N. Götz, U. Herzog and M. Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In *Proc. SIGMETRICS'93.* LNCS 729, Springer, 1993.

[13] H. Jonkers, A. J.C. van Gemund and G. L. Reijins. A probabilistic approach to parallel system performance modelling. In *Proc. 1986 Int. Conf. Par. Proc.*. IEEE, Aug. 1986, pp. 412-421

[14] V. W. Mak and S. F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, vol.1, July 1990, pp. 257-270.

3.2 Sumário do capítulo

A proposta apresentada neste capítulo mostrou uma técnica de predição de tempo de execução eficiente e com uma precisão adequada para as necessidades de um grid. Por ser baseada em PAMELA, uma linguagem imperativa, esta proposta tem a vantagem adicional de que é possível gerar um modelo automaticamente a partir do código fonte da aplicação com relativa facilidade. Este modelo poderia ser usado pelo escalonador para melhorar o desempenho da aplicação. O Capítulo 6 mostra como o sistema implementado prevê esta possibilidade incluindo no arquivo que descreve de uma aplicação paralela um modelo PAMELA de cada tarefa da aplicação. Finalmente, por ser parametrizado o modelo pode ser instanciado para valores diferentes de tamanho de problema e de quantidade de processadores, entre outros, e usado para prever o tempo de execução sob circunstâncias diferentes.

Capítulo 4

Análise de aplicações Mestre-Escravo em Grids

Este capítulo apresenta uma análise teórica sobre a viabilidade de executar aplicações paralelas de grande porte em um grid com uma grande quantidade de processadores. Enquanto o Capítulo 3 foca na modelagem da aplicação com o objetivo de obter uma expressão analítica para o desempenho da mesma, este capítulo parte de uma expressão analítica obtida para aplicações Mestre-Escravo e a usa para estudar suas características de *speedup* e escalabilidade.

O artigo *"Speedup and Scalability Analysis of Master-Slave Applications on Grid Environments"* mostra esta análise. A seqüência de idéias contidas no artigo é apresentada a seguir.

- Define modelos para o grid, a aplicação Mestre-Escravo e a execução da aplicação no grid.
- Usa o modelo para obter uma expressão analítica parametrizada para o tempo de execução da aplicação.
- Usa a expressão analítica obtida para derivar uma nova expressão analítica para o *speedup* da aplicação.
- Analisa os limites teóricos para o *speedup*.
- Define escalabilidade de uma aplicação em um grid baseado no conceito de isoeficiência, isto é, a capacidade de manter a eficiência constante na medida em que o tamanho do grid aumenta.
- Demonstra que uma aplicação é isoeficiente em um grid (e, portanto, escalável) se e somente se

$$\lim_{P\to\infty} \lim_{N\to\infty} E(N,P) > 0,$$

  onde $E(N,P)$ é a eficiência para um problema de tamanho N executado em um grid com P processadores.
- Analisa o efeito da contenção pelo uso da rede nestes resultados e o impacto que pode ter o projeto do grid na contenção obtida.

- Exemplifica os resultados obtidos com uma análise detalhada de uma aplicação de multiplicação de matrizes.

Speedup and Scalability Analysis of Master-Slave Applications on Grid Environments*

Eduardo J. Huerta Yero† and Marco A. Amaral Henriques
DCA - FEEC - UNICAMP
[huerta,marco]@dca.fee.unicamp.br

Abstract

Although grid environments have an enormous potential processing power, real applications that take advantage of this power remain an elusive goal. This is due in part to the lack of understanding about the characteristics of the applications best suited for these environments. This paper focuses on Master/Slave applications for grids. It defines application, cluster and execution models to derive an analytic expression for the execution time. It defines speedup and derives speedup bounds based on the inherent parallelism of the application and the aggregated computing power of the cluster. The paper derives an analytical expression for efficiency and uses it to define scalability of the algorithm-cluster combination based on the isoefficiency metric. Furthermore, the paper establishes necessary and sufficient conditions for an algorithm-cluster combination to be scalable which are easy to verify and use in practice. Finally, it covers the impact of network contention as the number of processors grow.

1 Introduction

Recently grid environments formed by heterogeneous computers connected by generic networks such as Internet have been successfully used to solve large parallel programs [9]. These platforms provide a cost-effective, easy-upgrading computing power which is unmatched by any real parallel computer.

Data parallel applications, particularly Master-Slave applications, are well-suited for this kind of environment. The input data of a data parallel application can be divided to take advantage of a large number of available processors. Master-Slave applications have been the most common choice so far, due mainly to the simplicity of their implementations, tolerance to Slave failures and simple communication topology. The SETI@home project [2], aimed at discovering extraterrestrial intelligence, is an example of a large Master-Slave application running on a grid environment. As grid computing evolves, more effort is being devoted to understand the performance characteristics of parallel applications running in these platforms.

*Work partially supported by grant 98/04305-9 of the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

†On leave from University of Havana, Cuba

Performance of a parallel application is usually evaluated according to three metrics: execution time, speedup and scalability. To predict execution time researchers develop models that abstract the execution behavior of the application [8, 10, 18, 20, 27, 28]. These models can be used to improve execution time through well-informed scheduling decisions or even through structural changes in the application implementation.

Speedup is a dimensionless metric that measures the performance gain obtained by the parallel implementation of an application over its sequential counterpart. Although different definitions exist, speedup is basically the ratio of sequential and parallel execution times for the same problem. Speedup is usually considered to be greater than one (otherwise it is a slowdown) and its upper bound for a fixed size problem was established by Amdahl [1] to be $1/s$, where s is the sequential portion of the algorithm.

Scalability measures the ability of an application to grow in size to take advantage of a larger set of resources. The fact that it is possible to increase the application size to exploit available resources and thus avoid the strict limits imposed by Amdahl's law was first noted by Gustafson [11]. Both Amdahl's and Gustafson's results have had a tremendous impact on parallel computing. A topic named "Amdahl's law and scalability" was recently voted by a panel of experts as the most influential parallel and distributed processing concept of the last millennium [21].

The goal of this paper is to study speedup and scalability issues for Master-Slave applications executing on grid environments. We define cluster, application and execution models consistent with those commonly found in grids. We derive an analytic expression for the execution time and use it to study the speedup bounds for an application. We define scalability using the isoefficiency metric (i.e. the capacity of increasing the application size when system size grows in order to keep efficiency constant) and derive necessary and sufficient conditions for a Master-Slave application to be scalable. Finally we study the impact of network contention on the execution time of the application and modify the scalability conditions to account for it.

The rest of the paper is structured as follows. Section 2 presents the cluster, application and execution models used throughout the paper. Section 3 derives a speedup formula and its upper bounds, both by studying the application and cluster characteristics. Section 4 defines scalability for a Master-Slave application, and studies the necessary and sufficient conditions for such an application to be scalable. Section 5 introduces a simple implementation of the matrix multiplication problem to serve as an example of the results obtained. Section 6 explores the possible impact of resource contention over the performance of a parallel application, and how it affects the results presented. Finally, Section 7 presents the conclusions of this work.

2 Models

In order to establish a solid foundation to build the results presented here, this paper defines models for a heterogeneous cluster, a Master-Slave application and the execution procedure itself. Although some abstractions are made, these models are intended to be as realistic as possible.

2.1 Cluster model

A cluster is usually depicted as a general graph where the nodes represent processors and the edges represent bidirectional communication links [6, 8, 20, 28]. Additional information can be added to nodes and edges to represent computer power and link speed respectively.

The model represented here is a particular case of such a general graph. It is formed by the elements below.

P**:** The number of computers forming the cluster.

$C = \{c_1, c_2, ..., c_P\}$**:** The set of computers forming the cluster. Associated with each computer there is an attribute $t_{op_i}, 1 \leq i \leq P$, indicating the time spent by computer i to execute an operation. t_{op_i} is given in $secs/op$.

$L = \{l_{1,1}, l_{2,1}, ..., l_{P,1}\}$**:** A set of bidirectional links between computer 1 and computer i, $1 \leq i \leq P$. Associated with each link there is an attribute t_{c_i} indicating the time spent to send a data unit between computers 1 and i. The value of t_{c_i} is expressed in $sec/unit$, where $unit$ may be bytes, words or any other data unit. Fig. 1 depicts the cluster model. By definition, t_{c_1} is 0.

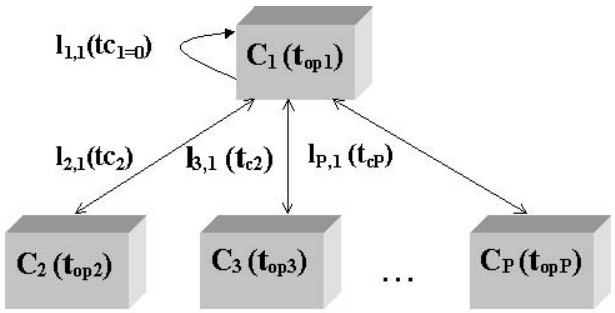

Figure 1: The cluster model

The attributes t_{op} and t_c can be determined through benchmarks for each computer in the cluster. The model assumes that these values are consistent for the application being executed in the sense that if $t_{op_i} = k * t_{op_j}, 1 \leq i, j \leq P, i \neq j, k > 0$, then computer c_j will be able to execute Slave tasks k times faster than c_i.

2.2 Application model

Parallel applications are usually based on tasks graphs [6, 28], with the nodes representing the tasks and the links representing data flow. Shao et al. [20] present a task graph model specific to Master-Slave applications, which is similar in some points with the model presented below.

N : A parameter that denotes the size of the problem. For example, in a multiplication of two square matrices N can be the number of rows in each matrix.

Master task: The Master task has the function of distributing and collecting data from the Slaves, as well as executing whatever serial code is necessary to solve the problem at hand. For simplicity the model assumes that the Master is not involved in the actual processing, although the results presented here can be easily extended to include this case. The Master is composed by:

$Seq_a(N)$**:** An initial sequential code probably used to generate or read some data, or to preprocess the data before the parallel algorithm starts;

$Split(N)$**:** The code used to divide the data to be processed among the Slaves;

$Gather(N)$**:** The code used to collect the results from the Slaves;

$Seq_b(N)$**:** A final sequential code, probably used to generate the final result and/or output it to its final destination (e.g. file, display).

Slave tasks: The Slave tasks have the function of actually solving the problem. They are defined by:

$ST(N)$**:** Number of Slave tasks;

$In(N, ST(N))$**:** Size of the input data for each Slave;

$Out(N, ST(N))$**:** Size of the output produced by each Slave;

$W(N, ST(N))$**:** The amount of work performed by the Slave.

The items $Seq_a(N)$, $Seq_b(N)$, $Split(N)$, $Gather(N)$ and $W(N, ST(N))$ are expressed in number of operations, while $In(N, ST(N))$ and $Out(N, ST(N))$ are expressed in number of data units.

Observe that the model defines $W(N, ST(N))$, $In(N, ST(N))$ and $Out(N, ST(N))$ – which are the elements in the model forming the parallelizable part of the application – as functions of the size of the problem N and the number of Slave tasks $ST(N)$, while not depending on the number of processors P. Although this may seem strange at first, this model gives a more realistic description of the approach taken by grids to parallel processing.

In common approaches to parallel processing the amount of work carried out by a Slave task depends on N and P. Given a fixed number of processors P and a fixed number of tasks, as N increases the size of each Slave task increases as well. This model poses some problems for grid systems, since they are mainly used to solve parallel problems for large values of N:

- If the amount of work assigned to a Slave task is too large, it may not be able to execute in any computer of the cluster due to memory restrictions. Although this problem can be solved by allowing the application to have access to the hard disks, this solution raises security issues with potentially serious consequences.

- Even if the Slave task is able to execute, it may take too long to produce results due to the amount of work assigned to it. Since individual computers in grid systems tend to be available for unpredictable (and normally small) periods of time, this would mean that either the parallel application has to be equiped with some sort of checkpointing ability or it will not be able to complete execution at all.

The solution adopted to implement a large Master-Slave application that is able to take advantage of a heterogeneous, dinamic environment such as a grid is to make parallel tasks relatively small. If the size of the problem N increases, then it is the number of tasks that should increase, and not the size of each task. This reasoning leads to the following assumptions, which will be used throughout the paper.

H1 : $In(N, ST(N))$, $Out(N, ST(N))$ and $W(N, ST(N))$ are equal for all Slave tasks, i.e. the work can be perfectly divided among the Slaves.

H2 : $\lim_{N\to\infty} In(N, ST(N)) = IN < \infty$, i.e. the size of the input data must not grow indefinitely with N.

H3 : $\lim_{N\to\infty} Out(N, ST(N)) = OUT < \infty$. Likewise, the size of the output data must be bounded for all values of N.

H4 : $\lim_{N\to\infty} W(N, ST(N)) = PAR < \infty$. The amount of work performed by each Slave must also be bounded for all values of N.

H5 : $\lim_{N\to\infty} ST(N)W(N, ST(N)) = \infty$. The amount of work forming the parallelizable part of the problem goes to ∞ with N.

H6 : $ST(N) \geq P$, $\forall N, P$. The number of Slave tasks must be greater than or equal to P for any P and any problem size, otherwise there would be unused processors.

H7 : $Seq_a(N)$, $Seq_b(N)$, $Split(N)$, $Gather(N)$, $In(N, ST(N))$, $Out(N, ST(N))$ and $W(N, ST(N))$ are continuous functions of N and $ST(N)$ for $N > 0$.

H8 : $t_{op_i}, 1 \leq i \leq P$ does not depend on N, i.e. an increase in problem size does not induce an increase in the time to execute an operation.

From **H4** and **H5** it is possible to deduce that

$$\lim_{N\to\infty} ST(N) = \infty. \tag{1}$$

H4 states that $W(N, ST(N))$, the amount of work executed by each Slave, is bounded, while **H5** says that $ST(N)W(N, ST(N))$, the total amount of parallelizable work in the application, tends to ∞ with N. The only way both conditions can hold simultaneously is that Eq. (1) also holds.

2.3 Execution model

With the application and cluster models defined it is possible to model the execution of the application in the cluster.

It is assumed that the Master executes in computer c_1 and the Slaves execute in $c_i, 1 \leq i \leq P$. Since there are $ST(N) \geq P$ Slaves, they are divided in P groups, each with Q_i, $1 \leq i \leq P$ Slaves, so

$$ST(N) = \sum_{i=1}^{P} Q_i. \tag{2}$$

This model is depicted in Fig. 2. Data from Master to Slaves is distributed concurrently, i.e. the time spent to send input data to all Slaves is equal to the time spent by the slowest communication link. We will disregard the effects of communication contention for now, as this topic is explored in Section 6.

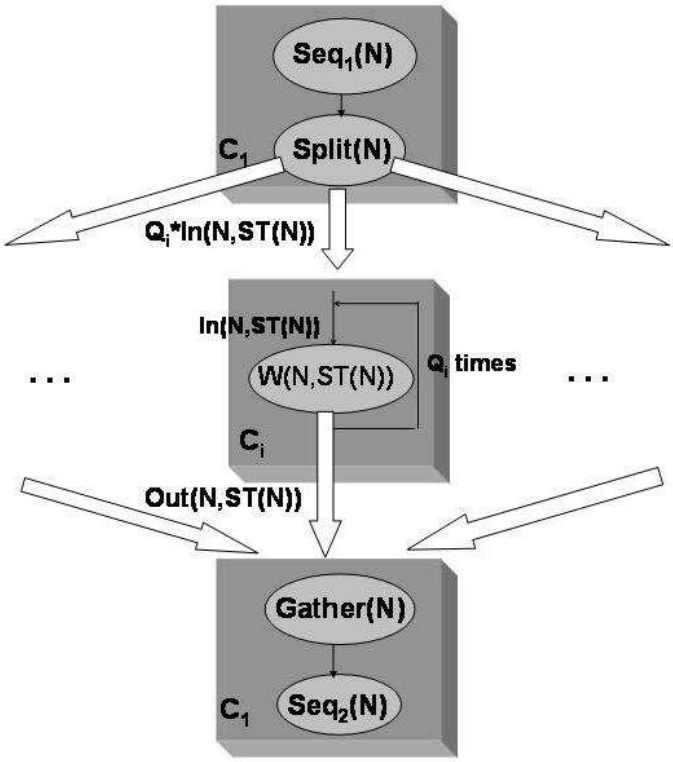

Figure 2: The execution model

Every computer c_i executes its Q_i assigned tasks sequentially. At the end of each task, results of size $Out(N, ST(N))$ are sent back to the Master. Again, a maximum of P processors may send their outputs to the Master simultaneously, but the contention induced by this fact is ignored at this stage.

2.4 Model analysis

This section derives an analytic expression for the execution time associated with the models presented so far. Consider $Q^{'} = \{Q_1, Q_2, ..., Q_P;\ Q_i \in \Re\}$ a task distribution among processors. Lets examine the time T_i in a single path from and back to the Master, considering only computer c_i.

$$\begin{aligned} T_i &= Seq_a(N)t_{op_1} + Split(N)t_{op_1} + Q_i[In(N, ST(N))t_{c_i} + W(N, ST(N))t_{op_i} \\ &\quad + Out(N, ST(N))t_{c_i}] + Gather(N)t_{op_1} + Seq_b(N)t_{op_1} \end{aligned} \tag{3}$$

Since Eq. (3) holds for every $1 \leq i \leq P$, the time for the application to execute is given by

$$T_{Q'} = \max_{i=1,P}(T_i). \tag{4}$$

Extracting the terms which are not related to i from the max we have

$$\begin{aligned} T_{Q'} &= (Seq_a(N) + Seq_b(N) + Split(N) + Gather(N))t_{op_1} \\ &\quad + \max_{i=1,P}\{Q_i[W(N, ST(N))t_{op_i} + (In(N, ST(N)) + Out(N, ST(N)))t_{c_i}]\}. \end{aligned} \tag{5}$$

Now defining

$$\begin{aligned} Seq(N) &= (Seq_a(N) + Seq_b(N))t_{op_1} && (6)\\ Ov(N) &= (Split(N) + Gather(N))t_{op_1} && (7)\\ Comm_i(N, ST(N)) &= (In(N, ST(N)) + && (8)\\ & \quad Out(N, ST(N)))t_{c_i} && (9)\\ Par_i(N, ST(N)) &= W(N, ST(N))t_{op_i} && (10) \end{aligned}$$

we have

$$T_{Q'} = Seq(N) + Ov(N) + \max_{i=1,P}\{Q_i[Par_i(N, ST(N)) + Comm_i(N, ST(N))]\}. \tag{11}$$

The parallel execution time $T_{Q'}$ depends on the scheduling $Q^{'}$ used. This paper considers the time T_{min} derived from the optimum scheduling decision, i.e. T_{min} is the minimum execution time obtainable from the algorithm in cluster C. If Q is the set of all possible scheduling decisions $Q^{'}$ for $ST(N)$ Slave tasks, then T_{min} can be stated as

$$T_{min} = \min_{Q' \in Q}(T_{Q'}). \tag{12}$$

Now consider the time T^- defined as

$$T^- = Seq^-(N) + Ov^-(N) + \frac{ST(N)}{P}[Par^-(N, ST(N)) + Comm^-(N, ST(N))]. \tag{13}$$

where

$$\begin{aligned} Seq^-(N) &= [Seq_a(N) + Seq_b(N)]t^-_{op} && (14)\\ Ov^-(N) &= [Split(N) + Gather(N)]t^-_{op} && (15)\\ Comm^-(N, ST(N)) &= [In(N, ST(N)) + Out(N, ST(N))]t^-_c && (16)\\ Par^-(N, ST(N)) &= W(N, ST(N))t^-_{op} && (17) \end{aligned}$$

and

$$t_{op}^{-} = \min_{i=1,P}(t_{op_i}) \tag{18}$$

$$t_{c}^{-} = \min_{i=1,P}(t_{c_i}). \tag{19}$$

For a homogeneous cluster, i.e. all computers and communication links are equal, T^- is the execution time derived from the optimum scheduling ($T^- = T_{min}$). For the heterogeneous cluster case, T^- can be regarded as the time it would take the application to execute in C^-, a virtual homogeneous cluster formed by P replicas of the fastest computer and communication link in the original cluster. The following proposition formally describes the relationship between T_{min} and T^- for the heterogeneous case.

Lemma 1 *There exists $K > 1$ such that $\forall\ N > 0$ and $\forall\ P > 1$ Eq. (20) holds.*

$$T^- < T_{min} < KT^- \tag{20}$$

Proof. Let us start by defining T^+, which is the opposite of T^-, i.e. the execution time of the Master-Slave application when executed in C^+, a virtual homogeneous cluster formed by replicas of the worst computer and communication link in the heterogeneous cluster.

$$T^+ = Seq^+(N) + Ov^+(N) + \frac{ST(N)}{P}(Par^+(N, ST(N)) + Comm^+(N, ST(N))),$$

where

$$\begin{aligned} Seq^+(N) &= [Seq_a(N) + Seq_b(N)]t_{op}^+ && (21)\\ Ov^+(N) &= [Split(N) + Gather(N)]t_{op}^+ && (22)\\ Comm^+(N, ST(N)) &= [In(N, ST(N)) + Out(N, ST(N))]t_c^+ && (23)\\ Par^+(N, ST(N)) &= W(N, ST(N))t_{op}^+ && (24) \end{aligned}$$

and

$$t_{op}^{+} = \max_{i=1,P}(t_{op_i}) \tag{25}$$

$$t_{c}^{+} = \max_{i=1,P}(t_{c_i}). \tag{26}$$

The best execution time in the heterogeneous cluster satisfies

$$T^- < T_{min} < T^+. \tag{27}$$

The relation $T^- < T_{min}$ is guaranteed by the fact that the heterogeneous cluster C must have a computer and/or communication link whose performance is worse than its similar in C^-, otherwise they would be identical. Similarly, $T_{min} < T^+$ holds because C must have a computer and/or communication link with better performance than C^+.

The difference between T^- and T^+ is actually a constant, since it is possible to find k_1 and k_2 such that

$$t_{op}^+ = k_1 t_{op}^- \quad (28)$$
$$t_c^+ = k_2 t_c^-, \quad (29)$$

where $k_1, k_2 > 1$.
Hence, for $K = \max(k_1, k_2)$ the relationship

$$T^+ < KT^- \quad (30)$$

holds. Substituting for T^+ in Eq. (27) we have

$$T^- < T_{min} < T^+ < KT^- \quad (31)$$

which shows that Eq. (20) holds. $\diamondsuit$

This section modeled the execution of a Master/Slave application on a grid. As a result we obtained an analytical expression for the execution time of the application (Eq. (12)) and the relationship among T^-, T_{min} and T^+ given by Eq. 27. These results will be used in the next section to analyze the speedup and efficiency characteristics of the application.

3 Speedup

3.1 Definitions and previous work

Although it has been around for a few decades, the concept of speedup has yet to find a widely accepted definition. In traditional parallel systems it is usually defined as T_{seq}/T_{par}, where T_{seq} is the time spent by the best sequential algorithm for the problem executing on a single processor and T_{par} is the time spent by a parallel algorithm on a parallel machine with P processors [5, 7]. However, this simple definition has been focus of constant improvements.

Sun and Gustafson [23] proposed a generalized speedup formula which is the ratio of parallel to sequential execution speed. Sun and Ni [24] further generalized Gustafson's result to fixed-time relative speedup, which adjusts grid parameters such as number of steps and expected accuracy so that the parallel execution time of the adjusted instance remains equal to the sequential execution time of the unadjusted instance. Moreover, Sun and Ni proposed another speedup model called memory-bounded speedup, which adjusts problem parameters so that the parallel algorithm uses as much memory per processor as the sequential unadjusted algorithm. Wu and Li [27] extend the notion of speedup to be the ratio of execution time in smaller to larger clusters. Donaldson et. al. [6] propose a generalization of traditional speedup for heterogeneous networks. A thorough study of speedup models, together with their advantages and disadvantages, is presented by Sahni and Tahnvantri [19].

Observe that speedup is normally defined as the execution time of the *best* sequential algorithm over the parallel algorithm (also known as absolute speedup [19]), therefore implying that the sequential and parallel algorithms might be different. A different approach, known as relative speedup, considers the parallel and sequential algorithms to be the same [19]. While

absolute speedup calculates the performance gain for a particular problem using any algorithm, relative speedup focuses on the performance gain for a specific algorithm that solves the problem. This work uses the relative speedup definition, which was also used by Amdahl in [1], and later by Gustafson in [11].

Grids introduce the additional complexity of choosing the processor on which to measure T_{seq}. Several approaches have been reported on this issue, such as picking a generic Reference Processor [4], defining a virtual processor whose performance characteristics correspond to the average characteristics of computers in the grid [18] or defining the speedup relative to the fastest machine in the grid [6, 27]. The work presented here uses the latter approach, since it answers common question in the minds of parallel application users: given a cluster C, how much faster will the parallel program execute relative to the best sequential time possible?

Sequential execution time T_{seq} is defined as follows.

$$T_{seq} = Seq^{-}(N) + ST(N)Par^{-}(N, ST(N)) \tag{32}$$

That is, the algorithm consists of the sequential execution of all the Slave tasks in the fastest computer. Since data distribution is not involved, the terms $Ov(N)$ and $Comm(N, ST(N))$ are not present. Observe that Eq. (32) forces the sequential and parallel algorithms to be the same. As it was noted previously in this section, this is the case in relative speedup.

Now we can formally define relative speedup as

$$S(N, P) = \frac{T_{seq}}{T_{min}}, \tag{33}$$

with T_{min} defined by Eq. 12. Similarly, we can define

$$S^{+}(N, P) = \frac{T_{seq}}{T^{-}}. \tag{34}$$

Lemma 2 *There exists* $0 < \alpha < 1$ *such that*

$$\alpha S^{+}(N, P) < S(N, P) < S^{+}(N, P) \tag{35}$$

Proof.

Inverting all the terms in Eq. (20) and multiplying by T_{seq} we have

$$\frac{1}{K} S^{+}(N, P) < S(N, P) < S^{+}(N, P). \tag{36}$$

Making $\alpha = 1/K$ the demonstration is complete. $\diamondsuit$

Before entering the discussion about speedup bounds, let us make two assumptions about $S(N, P)$.

H9 : $S(N, P)$, and thus $S^{+}(N, P)$, are continuous and strictly increasing functions of N and P. Since we are disregarding the effects of contention, when the parallel program and cluster size increase, the parallel execution time, consequently, decreases.

H10 : $\min_{\forall N,P} S(N, P) = 1$. If the problem size is so small that $T_{min} > T_{seq}$, then the sequential approach is used instead of the parallel one. The same can be assumed about $S^{+}(N, P)$, i.e. $\min_{\forall N,P} S^{+}(N, P) = 1$.

Hypothesis **H10** limits the analysis to the cases of interest, i.e. when the speedup is greater than or equal to 1. There is no reason to go through the difficulties of running a parallel program in a grid if it will take longer to execute than a sequential version of the same program.

3.2 Speedup bounds

There are two elements that limit the speedup obtained from the parallel execution of an application: The structure of the application and the computing power of the cluster where the application will be executed. This section examines both in detail.

3.2.1 Application structure

A parallel application can be subdivided into its sequential and parallel parts. Amdahl's law [1] states that, if s is the sequential portion of a problem, then the speedup of any parallel implementation is bounded by $1/s$. This result was the first to establish a clear relationship between the application structure and the speedup obtained from its parallel implementation.

Since it is stated for general problems, Amdahl's law does not take into account the overhead derived from managing a parallel program. With a definition of speedup as that in Eq. 33 it is possible to propose a tighter bound.

Lemma 3 *For a problem of fixed-size N and for all values of P, program speedup satisfies*

$$S(N,P) < \frac{1}{s^- + o^-} \tag{37}$$

where

$$s^- = \frac{Seq^-(N)}{T_{seq}} \tag{38}$$

$$o^- = \frac{Ov^-(N)}{T_{seq}} \tag{39}$$

Proof.

According to assumption **H9** $S^+(N,P)$ is a strictly increasing function of P for a fixed value of N. This means that, $\forall P > 1$, it holds

$$S^+(N,P) < \lim_{P\to\infty} S^+(N,P) \tag{40}$$

To calculate $\lim_{P\to\infty} S^+(N,P)$ we must first observe that T_{seq} does not depend on P, so

$$\lim_{P\to\infty} S^+(N,P) = \frac{T_{seq}}{\lim_{P\to\infty} T^-} \tag{41}$$

Looking at (Eq. 13) it is possible to determine that

$$\lim_{P\to\infty} T^- = Seq(N) + Ov(N). \tag{42}$$

Substituting Eq. (42) in Eq. (41) and dividing the numerator and denominator by T_{seq} we have

$$\lim_{P\to\infty} S^+(N,P) = \frac{1}{s^- + o^-}. \tag{43}$$

Since Eq. (40) and Eq. (43) guarantee that

$$S^+(N,P) < \frac{1}{s^- + o^-} \tag{44}$$

and Lemma 2 guarantees that $S(N,P) < S^+(N,P)$ we can conclude that

$$S(N,P) < \frac{1}{s^- + o^-}. \tag{45}$$

$\diamondsuit$

3.2.2 Computing power

The other limiting factor for application speedup is the computing power of the cluster being used. It is usually considered that speedup cannot be greater than P, the number of processors used, since the parallel implementation incurs in additional overhead due to communication. Thus,

$$S(N,P) \leq P, \quad \forall P \tag{46}$$

It is worth noticing that the models presented in this paper do not allow superlinear speedups ($(S(N,P) > P)$) to occur. Such speedups occur mainly because of two reasons: (a) since there is more available RAM memory when the parallel algorithm is used, the time t_{op_i} to execute an operation may decrease; and (b) the parallel version executes less work, using information derived early in the processing to avoid exploring certain branches of the solution space. The grid model presented here considers t_{op_i} fixed (assumption **H8**), independent of the problem size N. Moreover, Eq. 32 states that the amount of work carried out by the sequential version of the program (T_{seq}) is the sum of the sequential portion of the program ($Seq^-(N)$) plus the work carried out by the Slaves ($ST(N)Par(N, ST(N))$). This definition eliminates the possibility of executing fewer operations in the parallel version.

The speedup limit P can be considered tight enough for traditional parallel machines and homogeneous clusters, since the processing power available is P times greater than the power of any single processor. However, for a heterogeneous cluster this may not be the case. Colombet and Desbat [4] propose a speedup bound where the sequential time is measured in a generic reference processor. Since the performance characteristics of this processor are not specified, under ideal conditions their speedup can still be P. However, when the sequential time is measured on the fastest processor of a heterogeneous system, the speedup must be less than P, as we formally demonstrated below.

Definition (Processing Power): The processing power Pw_i of computer $i, 0 \leq i \leq P$ is defined as the inverse of the time per operation on this computer.

$$Pw_i = \frac{1}{t_{op_i}} \tag{47}$$

As a particular case of the above definition, we can define

$$Pw^{+} = \frac{1}{t_{op}^{-}} \tag{48}$$

Definition (Cluster Processing Power): The Cluster Processing Power Pw of a cluster with P processors is defined as

$$Pw = \sum_{i=1}^{P} Pw_i \tag{49}$$

Lemma 4 *For a heterogeneous cluster with P processors*

$$S(N,P) < P \tag{50}$$

holds.

Proof. Let us consider the ideal case where $Seq(N) = Ov(N) = Comm(N, ST(N)) = 0$ and where the optimum scheduling decision has been taken. Eq. 33 becomes

$$\begin{aligned} S(N,P) &= \frac{ST(N)Par^{-}(N,ST(N))}{\max_{i=1,P}(Q_i Par_i(N,ST(N)))} \\ &= \frac{ST(N)W(N,ST(N))t_{op}^{-}}{\max_{i=1,P}(Q_i W(N,ST(N))t_{op_i}))} \\ &= \frac{ST(N)}{Pw^{+}\max_{i=1,P}(\frac{Q_i}{Pw_i})} \end{aligned} \tag{51}$$

The term Q_i corresponds to a scheduling decision. The optimum scheduling occurs when all tasks finish at exactly the same time, that is,

$$\frac{Q_i}{Pw_i} = \frac{Q_j}{Pw_j} = L_R \quad \forall i,j, \;\; 1 \leq i,j \leq P. \tag{52}$$

Under these conditions it is possible to state that

$$\begin{aligned} \frac{ST(N)}{Pw} &= \frac{\sum_{i=1}^{P} Q_i}{\sum_{j=1}^{P} Pw_j} \\ &= \frac{\sum_{i=1}^{P} \frac{Q_i Pw_i}{Pw_i}}{\sum_{j=1}^{P} Pw_j} = \frac{\sum_{i=1}^{P} L_R Pw_i}{\sum_{j=1}^{P} Pw_j} \\ &= \frac{L_R \sum_{i=1}^{P} Pw_i}{\sum_{j=1}^{P} Pw_j} = L_R. \end{aligned} \tag{53}$$

Using this result in Eq. 51

$$S(N,P) = \frac{ST(N)}{Pw^{+}\max_{i=1,P}(\frac{ST(N)}{Pw})} = \frac{Pw}{Pw^{+}} \tag{54}$$

Since the cluster is heterogeneous, there must exist $1 \leq i \leq P$ such that

$$Pw_i < Pw^+ \tag{55}$$

then

$$\begin{aligned} Pw &= \sum_{i=1}^{P} Pw_i < P * Pw^+ \\ \frac{Pw}{Pw^+} &< P \\ S(N,P) &< P. \end{aligned}$$

$\diamondsuit$

This result implies that even in ideal conditions it is impossible to obtain a speedup of P for a heterogeneous system. Fig. 3 shows the shape of the ideal speedup curve for an imaginary heterogeneous system on which machine powers are randomly selected between 70% and 100% of Pw^+.

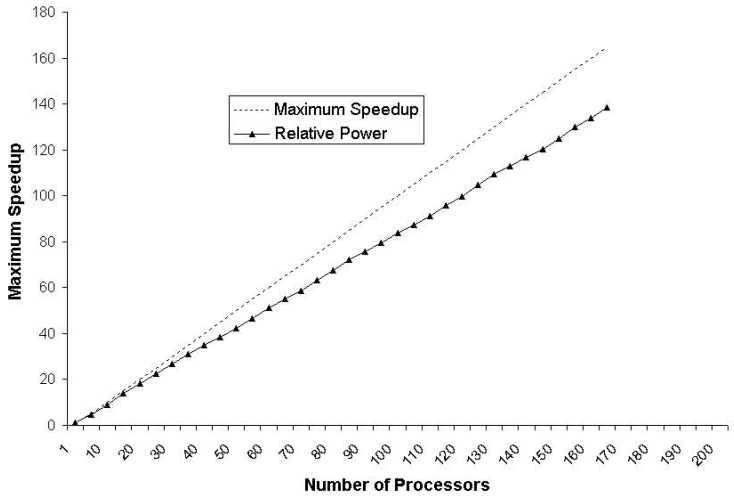

Figure 3: Maximum speedup for a heterogeneous system.

The result of Theorem 3 and Lemma 4 can be combined to determine the actual bounds of speedup for a parallel application. As an example, Fig. 4 presents an analytical comparison of the curves generated by the traditional bounds (set by the Amdahl law and the number of processors P) with the ones determined by Theorem 3 and Lemma 4 for a matrix multiplication algorithm ($N = 100$). Note that the actual application speedups must be located below both curves. Fig. 4 shows that the limits imposed by the results presented in this section are tighter than the ones proposed by Amdahl's law and the number of processors P.

4 Scalability

In 1988 John Gustafson presented results obtained in the Sandia National Laboratories that put a new perspective on the way parallel processing was being approached [11]. In short,

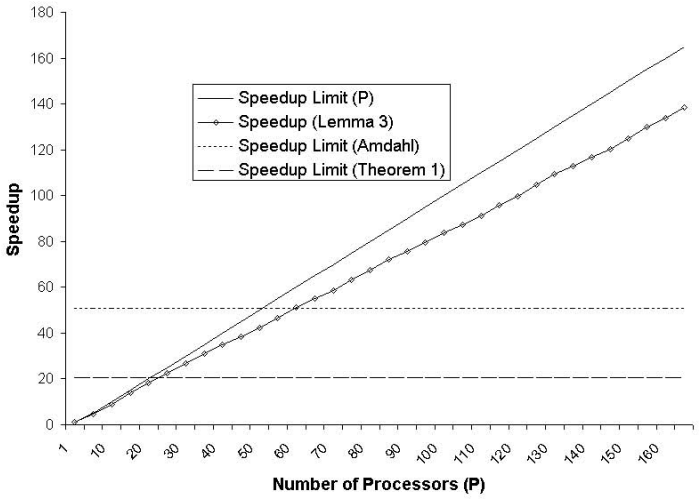

Figure 4: Speedup bounds for a parallel matrix multiplication.

experiments at Sandia proved that, although Amdahl's result was valid for fixed-size problems, it was possible to increase the size of the problem with the number of processors and to obtain speedups larger than those originally predicted by Amdahl's law. The ability of a problem to increase its size to take advantage of a larger number of processors is called *scalability*, and it has been the focus of a large research effort in the last decade [13, 14, 17, 22, 25, 27].

More formally, a parallel program is said to be scalable when its performance can be made linear to the system size by properly adjusting the problem size [17, 27]. Different metrics have been developed to better understand this property. In [17] Müller-Wichards and Rönsch study the behavior of an overhead function which would be responsible for the lack of scalability as system and problem size increase. Sun and Rover [25] proposed the isospeed metric, which determines the ability of the algorithm/grid combination to keep the speed (i.e Work/Time) constant. Wu and Li [27] define scalability based on a fixed computation/communication ratio. Jogalekar and Woodside [13] present a scalability metric for general purpose distributed systems which is based on the throughput of the system. Sun [22] studies the relationship between scalability and execution time through a technique called crossing point analysis.

The work presented in this paper is based on the isoefficiency metric, first proposed by Kumar and Rao in [14]. According to this metric, an algorithm/grid combination is said to be scalable if problem size can be adjusted to maintain efficiency constant as the system grows. This metric provides a simple yet effective way to analyze scalability, especially as problem and system size approach infinity. In this case $S^+(N,P)$ is more useful, since it does not depend on the scheduling parameter Q_i and it has the same behavior as $S(N,P)$ while N and P approach infinity (Lemma 2).The formal definitions are presented below.

Definition (Efficiency): For a parallel problem with size N and a system with P processors, efficiency $E(N,P)$ is defined as

$$E(N,P) = \frac{S(N,P)}{P} \tag{56}$$

Definition (Isofficiency function): The isoefficiency function $I(P, e) = N_e$ denotes the amount of work (N_e) necessary to make a system with P processors perform with an efficiency $e > o$.

Definition (e-Isoefficiency): A problem is said to be *e-Isoefficient* in a system with P processors if $\exists N_e$ and $P^{'} > 0$ such that for every$P > P^{'}$

$$E(N_e, P) = e \tag{57}$$

Isoefficiency defines a relationship I between N and P such that $I(P, e) = N$ is the amount of work necessary to make the efficiency e when P processors are used. The function I is called the isoefficiency function.

Definition (Scalability): An algorithm-grid combination is scalable if and only if the isoefficiency property holds.

If $E(N, P)$ is a constant (as in the ideal case, where $E(N, P) = 1$) then the algorithm-grid combination is isoefficient and thus scalable. However, efficiency values which are independent of N and P are rarely found in practice. In the following text we consider the general case where $E(N, P)$ depends on both N and P. Since by **H9** $S(N, P)$ is a strictly increasing function of N, then, by definition, $E(N, P)$ is also strictly increasing with N for a fixed value of P. Inversely, for a fixed value of N, when P increases $E(N, P)$ decreases, since, according to Theorem 3, the value of $S(N, P)$ will saturate in its limit at $1/(s^- + o^-)$.

Now consider the term

$$E(\infty, P) = \lim_{N \to \infty} E(N, P). \tag{58}$$

Since $E(N, P)$ is an increasing function of N, $E(\infty, P)$ represents the maximum attainable efficiency for a specific value of P. It is worth analyzing how this value behaves as P increases. The following theorem establishes a relationship between $E(\infty, P)$ and isoefficiency.

Theorem 1 *Consider an algorithm-grid combination with efficiency $E(N, P)$. Then for $0 < e < 1$*

$$\lim_{P \to \infty} E(\infty, P) = e \tag{59}$$

holds if and only if the combination is e-isoefficient but not $e^{'}$-isoefficient for $e^{'} > e$.

Proof. $\Rightarrow$:

if $\lim_{P \to \infty} E(\infty, P) = e$ then e-isoefficiency holds and $e^{'}$-isoefficiency does not.

Since $\lim_{P \to \infty} E(\infty, P) = e$ and $E(N, P)$ decreases with P, then

$$E(\infty, P) > e \ \ \forall P. \tag{60}$$

Also, by hypothesis **H10** and the definition of Efficiency

$$\min_{\forall N, P} E(N, P) = \frac{1}{P}. \tag{61}$$

For a $P > \frac{1}{e}$ it follows

$$\min_{\forall N, P} E(N, P) < e < E(\infty, P). \tag{62}$$

Since by hypothesis **H9** $E(N,P)$ is continuous, there exists $N^{'}$ such that

$$E(N^{'},P) = e, \tag{63}$$

which proves e-isoefficiency. To prove that $e^{'}$-isoefficiency is not satisfied it is enough to see that, for a sufficiently large value of P

$$E(\infty,P) < e^{'}, \tag{64}$$

holds since $\lim_{P\to\infty} E(\infty,P) = e$ and $e < e^{'}$. For these values of P it is impossible to find $N^{'}$ such that

$$E(N^{'},P) = e^{'}, \tag{65}$$

therefore $e^{'}$-isoefficiency does not hold.

$\Leftarrow$:

If e-isoefficiency holds and $e^{'}$-isoefficiency is not satisfied for $e^{'} > e$, then

$$\lim_{P\to\infty} E(\infty,P) = e. \tag{66}$$

According to its definition, e-isoefficiency means that $\forall P > P^{'}$ there exists $N_e = I(P,e)$ such that

$$E(N_e,P) = e. \tag{67}$$

Since $E(N,P)$ is increasing with N then $E(\infty,P) > e$. This remains true as P approaches infinity, so

$$\lim_{P\to\infty} E(\infty,P) \geq e. \tag{68}$$

However, $\lim_{P\to\infty} E(\infty,P) = e^{'} > e$, would mean, by the first part of this demonstration, that $e^{'}$-isoefficiency is possible, which contradicts the hypothesis. Therefore

$$\lim_{P\to\infty} E(\infty,P) = e, \tag{69}$$

which is what we intended to prove. $\diamondsuit$

The above theorem establishes that if the limit of efficiency as N and P go to infinity is $e > 0$ then e-isoefficiency and thus scalability can be guaranteed. Now it is only necessary to determine under which conditions this limit is greater than zero. The following theorem specifies such conditions.

Theorem 2 *For a Master-Slave application with size N executing in a cluster with P processors*

$$\lim_{P\to\infty} E(\infty,P) = e > 0 \tag{70}$$

holds if and only if

$$\lim_{N\to\infty} \frac{Seq^{-}(N)}{ST(N)} = 0 \tag{71}$$

$$\lim_{N\to\infty} \frac{Ov^{-}(N)}{ST(N)} = 0 \tag{72}$$

$$\tag{73}$$

Proof.

Let us first calculate $E(\infty, P)$, defined by Eq. (58).

Consider

$$\lim_{N\to\infty} \frac{Seq(N)}{ST(N)} = \alpha \tag{74}$$

$$\lim_{N\to\infty} \frac{Ov(N)}{ST(N)} = \beta \tag{75}$$

$$\tag{76}$$

Using the definition of $E(N, P)$, Eq. (58) can be rewritten as

$$E(\infty, P) = \lim_{N\to\infty} \frac{T_{seq}}{PT_{min}}. \tag{77}$$

Since Eq. 20 stated that the difference between T_{min} and T^{-} is bounded by a constant K and this theorem deals with behavior near ∞, we can reformulate the equation above as

$$E(\infty, P) = \lim_{N\to\infty} \frac{T_{seq}}{PT^{-}}. \tag{78}$$

Dividing both numerator and denominator by $ST(N)$ and calculating the limit we have

$$E(\infty, P) = \frac{\alpha + t_{op}^{-} PAR}{P(\alpha + \beta) + t_{op}^{-} PAR + t_{c}^{-}(IN + OUT)} \tag{79}$$

with the terms IN, OUT and PAR derived from hypothesis **H2**, **H3** e **H4** respectively. To solve $\lim_{P\to\infty} E(\infty, P)$ we have to consider all possibilities for α and β.

$$\lim_{P\to\infty} E(\infty, P) = \begin{cases} 0 & if\ \alpha \neq 0\ or \\ & \beta \neq 0 \\ \frac{t_{op}^{-} PAR}{t_{op}^{-} PAR + t_{c}^{-}(IN+OUT)} & if\ \alpha = 0\ and \\ & \beta = 0 \end{cases} \tag{80}$$

Eq. 80 proves that $\lim_{P\to\infty} E(\infty, P) > 0$ if and only if the two conditions stated in the theorem are true. $\diamondsuit$

Since IN, OUT, PAR, t_{op}^{-} and t_{c}^{-} are constants, it is possible to find r such that

$$r = \frac{t_{op}^{-} PAR}{t_{c}^{-}(IN + OUT)}. \tag{81}$$

The constant r represents the ratio between processing and communication time for a Slave task running in the fastest processor.

Substituting in Eq. 80 we have

$$\lim_{P\to\infty} E(\infty, P) = \frac{r}{r+1}. \tag{82}$$

Eq. 82 confirms that there exists a direct relationship between the computing/communication ratio r and the expected efficiency. The larger the value of r, the closer $\lim_{P\to\infty} E(\infty, P)$ gets to the ideal value of 1. The following corollary summarizes the results presented so far into a usable mechanism to analyze scalability.

Corollary 1 *An algorithm-cluster combination is scalable if and only if the two conditions stated by Eq. 71 and Eq. 72 in Theorem 2 are true.*

Proof. An algorithm-cluster combination is scalable by definition if the isoefficiency property holds. Theorem 1 established the equivalence between $\lim_{P\to\infty} E(\infty, P) = e > 0$ and e-isoefficiency, while Theorem 2 demonstrated that $\lim_{P\to\infty} E(\infty, P) = e > 0$ is equivalent to the two conditions stated by Eq. 71 and Eq. 72 in Theorem 2; therefore these conditions guarantee scalability. $\diamondsuit$

The corollary above gives a practical method to determine whether an algorithm is scalable while executing in the given grid by either checking the limit of the efficiency expression as N and P approach infinity or by checking the two conditions stated in Theorem 2. The following section shows an example on how to use this method.

5 Example

This section presents a naive parallel implementation of the well-known matrix multiplication problem. It is not the purpose of this algorithm to be particularly efficient, but to be simple enough to ease the analysis. For efficient parallel matrix multiplication algorithms refer to [3, 16].

The algorithm presented here multiplies two $N \times N$ matrices A and B. Each line and column of A and B is divided in N/D blocks of size D. As a result, each matrix is divided in N^2/D^2 submatrices. From the multiplication algorithm it follows that each submatrix in A has to be multiplied by N/D submatrices in B. For example, if $N = 9$ and $D = 3$, then $A_{1,1}$ (a 3×3 submatrix) has to be multiplied by $B_{1,1}$, $B_{1,2}$ and $B_{1,3}$.

The algorithm assigns each submatrix multiplication to a different slave task. Since D is a constant, the amount of work performed by each task (D^3) and the size of the data that has to be sent (D^2) are independent of the problem size N. Observe that this approach satisfies hypothesis **H1-H8**.

With this information it is possible to define each of the model components. For the sake of simplicity each component will be stated only regarding its dependency with N and D, discarding any constant value. For example, a matrix multiplication will require D^3 operations, instead of $aD^3 + b$.

$Seq^-(N) = N^2 t_{op}^-$: Required to obtain the original matrices and output the final result.

$Ov^-(N) = N^2 t_{op}^-$: Needed to divide the original matrices in $D \times D$ submatrices and assemble the final result.

$Par^-(N, ST(N)) = D^3 t_{op}^-$: Used to multiply two $D \times D$ submatrices.

$Comm^-(N, ST(N)) = D^2 t_c^-$: Amount of time to send and receive data from each Slave task.

Using these definitions we have from Eq. (32) and Eq. (13),

$$T_{seq} = (N^2 + N^3) t_{op}^- \tag{83}$$

$$T^- = 2N^2 t_{op}^- + \frac{N^3}{D^3 P}(D^3 t_{op}^- + D^2 t_c^-) \tag{84}$$

Now consider the speedup formula from Eq. 34. Substituting T_{seq} and T^- we have

$$S^+(N, P) = \frac{(1+N) t_{op}^-}{2t_{op}^- + \frac{N}{D^3 P}(D^3 t_{op}^- + D^2 t_c^-)}. \tag{85}$$

Now let us calculate $\lim_{P\to\infty} S^+(N, P)$, the maximum attainable speedup for specific matrix dimensions.

$$\lim_{P\to\infty} S^+(N, P) = \frac{1+N}{2}. \tag{86}$$

To analyze scalability let us first consider the conditions stated by Theorem 2.

$$\lim_{N\to\infty} \frac{Seq^-(N)}{ST(N)} = \lim_{N\to\infty} \frac{N^2 D^3 t_{op}^-}{N^3} = 0 \tag{87}$$

$$\lim_{N\to\infty} \frac{Ov^-(N)}{ST(N)} = \lim_{N\to\infty} \frac{N^2 D^3 t_{op}^-}{N^3} = 0 \tag{88}$$

From the results in Corollary 1 these two conditions imply that the algorithm-cluster combination is isoefficient and thus scalable. To further study the problem, lets consider $E(\infty, P)$, the maximum attainable efficiency for P processors. Having $S^+(N, P)$ defined by Eq. 85 we have

$$E(\infty, P) = \lim_{N\to\infty} \frac{(1+N) t_{op}^-}{2Pt_{op}^- + \frac{N}{D^3}(D^3 t_{op}^- + D^2 t_c^-)}. \tag{89}$$

$$E(\infty, P) = \frac{D^3 t_{op}^-}{D^3 t_{op}^- + D^2 t_c^-} \tag{90}$$

Eq. 90 shows that $E(\infty, P)$ is a constant, so

$$\lim_{P\to\infty} E(\infty, P) = \frac{D^3 t_{op}^-}{D^3 t_{op}^- + D^2 t_c^-} = e \tag{91}$$

and the isoefficiency condition holds.
Finally lets derive $N_e = I(P, e)$, the isoefficiency function for a specific value of e.

$$E(N_e, P) = \frac{(1+N)t_{op}^-}{2Pt_{op}^- + \frac{N}{D^3}(D^3 t_{op}^- + D^2 t_c^-)} = e \tag{92}$$

Solving for N_e we have

$$N = \frac{(2Pe-1)Dt_{op}^-}{(1-e)Dt_{op}^- - et_c^-}. \tag{93}$$

Now consider the relationship q of communication to execution time.

$$q = \frac{t_c^-}{t_{op}^-} \tag{94}$$

Substituting Eq. 94 in Eq. 93 we have

$$I(P, e) = N_e = \frac{(2Pe-1)D}{(1-e)D - qe}. \tag{95}$$

For the denominator to be greater than zero, it has to hold that

$$e < \frac{D}{D+q}. \tag{96}$$

Eq. 96 establishes a bound to the maximum possible efficiency according to the values of D and q. For large values of D and small values of q (large amount of work for each task and fast communications) the value of e can be close to 1.

6 Resource contention

By definition the values of t_{op_i} and t_{c_i} are measured for every computer in the cluster individually, i.e. they do not express the effects of the execution of the Master/Slave application. According to the definition of the model, the Slaves assigned to a computer are executed sequentially, hence no competition for the processors occurs. Hence the value of t_{op_i} can be considered accurate throughout the execution of the application.

Communication is a different case. During the distribution of data and collection of results, all P computers may be sharing some communication links, thus inducing a slowdown that will affect application performance. The issue to be studied is how this contention reflects on the conditions necessary for an application to be scalable.

Tanenbaum presents in [26] a thorough study of the effect of congestion in network performance. Problems in the medium access sublayer, network layer and transport layer may rapidly decrease performance to the point of rendering the network unusable.

Let us assume that a contention function $C(P)$ exists which slows down communications as P increases. Then Eq. 79 can be rewritten as

$$E(\infty, P) = \frac{\alpha + t_{op}^- PAR}{P(\alpha+\beta) + t_{op}^- PAR + t_c^- C(P)(IN+OUT)} \tag{97}$$

If we assume that

$$\lim_{P\to\infty} C(P) = \infty, \tag{98}$$

that is, the contention grows indefinitely with P, then

$$\lim_{P\to\infty} E(\infty, P) = 0 \tag{99}$$

and isoefficiency conditions, and thus scalability, are not satisfied.

Grid systems are usually designed to keep $C(P)$ from growing indefinitely with P. This bound to contention can be achieved through a decentralized management approach. Computers participating in the grid can be divided in groups. Each group has a coordinator which supervises all computers in the group. Groups are connected among themselves through some logical topology. Fig. 5 presents such a grid.

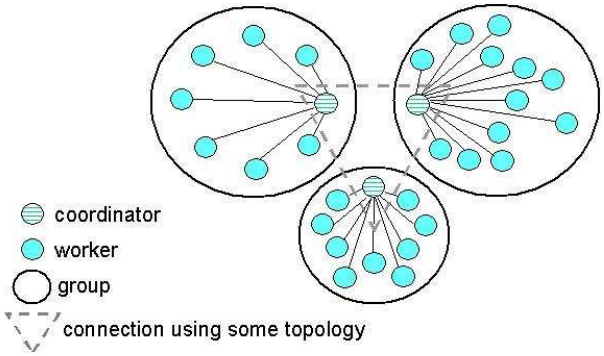

Figure 5: Scalable grid model.

In this approach each group has a limit to the number of computers belonging to it. The system accommodates more computers by creating new groups. Hierarquical ([15]) and hypercubic ([12]) topologies have been proposed to interconnect the groups.

Having this in mind it is possible to consider $C(P)$ as a bounded function of P, that is,

$$\lim_{P\to\infty} C(P) = c < \infty. \tag{100}$$

Being so, the limit in Eq. 80 changes to

$$\lim_{P\to\infty} E(\infty, P) = \frac{t_{op}^{-} PAR}{t_{op}^{-} PAR + ct_c^{-}(IN + OUT)}, \tag{101}$$

which is greater than zero and therefore allows the isoefficiency conditions to be satisfied.

7 Conclusions

The use of grid environments to execute large parallel applications is becoming a common practice. The enormous potential power of grids, together with their easy-upgrading characteristics

and low cost/performance ratio has led researchers to attempt to find effective approaches to exploit these environments.

Despite the promises, applications and environments that extract a significant portion of this power remain an elusive goal. This is in part due to the lack of understanding of the performance characteristics of the applications which are best suited for them.

This paper focuses on Master/Slave applications for Grid environments. It defines application, cluster and execution models to derive an analytic expression for the execution time. This expression is then used to define speedup and efficiency for the application. The paper presents speedup bounds derived from Amdahl's law and the aggregated computing power of the cluster. Furthermore, the paper establishes necessary and sufficient conditions for an algorithm/cluster combination to be scalable based on the isoefficiency metric. Finally it explores the impact of network contention on the results presented. The final conclusion drawn from the results presented is that it is possible to execute large Master/Slave applications on grids as long as the applications are scalable and the grids keep the communication contention bounded as the number of processors grow.

References

[1] G. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference*, pages 483–485, 1967.

[2] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[3] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Matrix multiplications on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), October 2001.

[4] L. Colombet and L. Desbat. Speedup and efficiency of large-size applications on heterogeneous networks. *Theoretical Computer Science*, 196:31–44, 1998.

[5] Paolo Cremonesi, Emilia Rosti, Giuseppe Serazzi, and Evgenia Smirni. Performance evaluation of parallel systems. *Parallel Computing*, 25:1677–1698, 1999.

[6] Val Donaldson, Francine Berman, and Ramamohan Paturi. Program speedup in a heterogeneous computing network. *Journal of Parallel and Distributed Computing*, 21:316–322, 1994.

[7] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3), March 1989.

[8] Silvia M. Figueira and Francince Berman. A slowdown model for applications executing on time-shared clusters of workstations. *IEEE Transactions on Parallel and Distributed Systems*, 12(6), June 2001.

[9] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, first edition, January 1999.

[10] Linguo Gong, Xian-He Sun, and Edward F. Watson. Performance modeling and prediction of nondedicated network computing. *IEEE Transactions on Computers*, 51(9), September 2002.

[11] John Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, May 1988.

[12] Eduardo Javier Huerta Yero and Marco Aurlio Amaral Henriques. A Method to Solve the Scalability Problem in Massively Parallel Processing on the Internet. In *7th Euromicro Workshop on Parallel and Distributed Processing*, University of Madeira, Funchal, Portugal, 3 a 5 de Fevereiro 1999. IEEE Computer Press.

[13] Prasad Jogalekar and Murray Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6), June 2000.

[14] V. Kumar and V. N. Rao. Parallel depth-first ii. analysis. *International Journal of Parallel Programming*, 16, 1987.

[15] Mike Lewis and Andrew Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, aug 1996. IEEE Computer Society Press.

[16] Keqin Li. Scalable parallel matrix multiplication on distributed memory parallel computers. *Journal of Parallel and Distributed Computing*, 61:1709–1731, 2001.

[17] Dieter Müller-Wichards and Wolfgang Rönsch. Scalability of algorithms: An analytic approach. *Parallel Computing*, 21:937–952, 1995.

[18] D. N. Ramos-Hernández and M. O. Tokhi. Performance evaluation of heterogeneous systems. *Microprocessors and Microsystems*, 25:203–212, 2001.

[19] Sartaj Sahni and Venkat Thanvantri. Performance metrics: Keeping the focus in runtime. *IEEE Parallel and Distributed Technology*, pages 43–56, Spring 1996.

[20] Gary Shao, Francine Berman, and Rich Wolski. Master/slave computing on the grid. In *9th Heterogeneous Computing Workshop*, May 2000.

[21] Larry Snyder and Thomas Sterling. Panel: "What are the top ten most influential parallel and distributed processing concepts of the last millenium?". *Journal of Parallel and Distributed Computing*, 61, 2001. Panel held at IPDPS 2000.

[22] Xian-He Sun. Scalability versus execution time in scalable systems. *Journal of Parallel and Distributed Computing*, 62:173–192, 2002.

[23] Xian-He Sun and John Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17, December 1991.

[24] Xian-He Sun and Lionel Ni. Scalable problems and memory bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, September 1993.

[25] Xian-He Sun and Diane T. Rover. Scalability of parallel algorithm-machine combinations. *IEEE Transactions on Parallel and Distributed Systems*, 5(6), June 1994.

[26] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, third edition, 1996.

[27] Xingfu Wu and Wei Li. Performance models for scalable cluster computing. *Journal of Systems Architecture*, 44:189–205, 1998.

[28] Yong Yan, Xiaodong Zhang, and Yongsheng Song. An effective and practical performace prediction model for parallel computing on nondedicated heterogeneous now. *Journal of Parallel and Distributed Computing*, 38(1):63–80, 1996.

4.2 Sumário do capítulo

Este capítulo apresentou uma análise das características de desempenho de uma aplicação Mestre-Escravo executada em um grid. Os resultados demonstram que se uma aplicação satisfaz determinadas condições então ela é escalável. Como as condições impostas não são excessivamente restritivas – são, de fato, esperadas para uma aplicação Mestre-Escravo – estes resultados reafirmam a noção intuitiva de que é possível explorar grids com um grande número de processadores para executar aplicações Mestre-Escravo de grande porte.

Com este capítulo concluímos a fundamentação teórica desta tese. Acreditamos ter provado que vale a pena investir esforços na construção de grids escaláveis e eficientes, uma vez que, se eles existirem, haverá aplicações capazes de explorá-los. Os Capítulos 5, 6 e 7 da Parte III estão dedicados a apresentar aspectos técnicos da construção de um grid.

Parte III

Construção de um Grid

Capítulo 5

Projeto e implementação de um grid

Para provar a validade de uma idéia geralmente não é suficiente uma demonstração teórica. No nosso caso particular isso quer dizer que apesar dos resultados apresentados no Capítulo 4 indicarem claramente a validade de uma abordagem baseada em grids para determinados tipos de aplicações paralelas, ainda é necessário provar esta idéia na prática.

Este capítulo apresenta JoiN, um sistema desenvolvido como parte deste trabalho para testar as idéias nele propostas. Este sistema começou a ser desenvolvido no início de 1997, e desde então tem sido aprimorado constantemente. JoiN oferece uma plataforma totalmente baseada em Java para a execução de aplicações maciçamente paralelas de forma escalável e tolerante a falhas. Para atingir estes objetivos o sistema conta com um gerenciamento descentralizado dos computadores participantes (que são divididos em grupos independentes) e mecanismos de tolerância a falhas capazes de minimizar o impacto causado por problemas em qualquer computador do sistema. JoiN usa uma estratégia de escalonamento semi-estática, que lhe permite se adaptar a mudanças nas condições do grid sem a sobrecarga imposta pelos escalonadores dinâmicos. Atualmente na versão 1.3, JoiN fornece uma plataforma robusta e extensível em que podem ser testadas diversas propostas para grids.

O artigo JoiN*: The Implementation of a Java-based Massively Parallel Grid*, apresentado a seguir, mostra uma visão detalhada do projeto e implementação de JoiN, com destaque para as soluções propostas na área de escalabilidade, escalonamento, tolerância a falhas e segurança. O artigo esclarece também o estado atual do sistema e as direções em que devem ser conduzidos os trabalhos futuros.

JoiN: The Implementation of a Java-based Massively Parallel Grid *

Eduardo J. Huerta Yero†, Fabiano de O. Lucchese,
Francisco S. Sambatti‡, Miriam von Zuben and Marco A. A. Henriques
DCA - FEEC - UNICAMP
[huerta,flucches,sambatti,marco]@dca.fee.unicamp.br

Abstract

This paper presents JoiN, Java-based platform to construct massively parallel grids. JoiN provides a flexible, robust and efficient platform to develop large parallel applications. The system is designed to be scalable by allowing computers in the grid to be separated in independent sets (called *groups*) which are managed independently and collaborate using a logical interconnection topology. JoiN provides advanced fault tolerance capabilities that allow it to withstand failures both in computers executing parallel tasks and in computers managing the groups. The parallel applications executing in the system are formally specified using a rigorously defined application model. JoiN uses a dynamic, flexible scheduling algorithm that adapts to changes in resource availability and replicates parallel tasks for fault tolerance. The platform provides an authentication/access control mechanism based in roles which is embedded in the inner parts of the system. The software architecture is based on the concept of *services*, which are independent pieces of software that can be combined in several ways, providing the flexibility needed to adapt to particular environments. JoiN has been successfully used to implement and execute several parallel applications, such as DNA sequencing, Monte Carlo simulations and a version of the Traveling Salesman Problem.

1 Introduction

Large parallel applications remain a challenge to the parallel processing community. On one side, parallel applications with enormous needs for computing power remain unsolved, while on the other side there is enough potential computing power to solve them. This is particularly true when one considers the aggregated computing power supplied by personal computers and other processing devices connected by high speed networks.

Grid computing is an attempt to fill the gap between these two worlds. The goal of grid systems is to harness the power of geographically distributed, heterogeneous processing devices

*Work partially supported by grant 98/04305-9 of the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

†On leave from University of Havana, Cuba
‡Western Paraná State University

as a single, virtual parallel machine and to use this power to solve large problems. Despite the intuitive notion that such a match is possible, problems from both technological and human origins have prevented so far the pervasive use of grids. As a consequence, the dream of grid researchers – a world where grid services are integrated at the operating system level and where being part of a grid is as common as accessing the Internet – still lies far ahead.

The heterogeneity of resources is perhaps the most challenging problem faced by grid systems. Parallel computers – sometimes aggregated to form grids – usually have specific libraries particularly designed and tuned for them. This fact makes the task of programming applications that run across several of these parallel machines particularly difficult. Desktop computers and other simple processing devices present differences in the processors and operating systems used. While some of these problems have been alleviated by the Java Technology – which allows to write one program and execute it across different architectures – some of them have still to be addressed in a more comprehensive manner, such as the interaction with legacy code and the standardization of the scientific libraries needed by some of the parallel applications.

The scheduling of parallel tasks in a large set of heterogeneous resources is also a complex problem. Heterogeneity forces the scheduling mechanism to be aware of different processor and link speeds, processor types (vector, scalar) and memory restrictions among others. Furthermore, resource availability (number of computers, processor and link speed, amount of free memory, etc) in grids tend to change over time in an unpredictable form. As a consequence, a mechanism to configure a grid to make it optimal for a specific application remains an open problem for which only heuristic solutions exist.

Application models are closely related to scheduling. An application model describes the components forming the application and how they interact during the execution phase. Typically structured as task graphs, application models can describe data flow among components, operations performed by each component, execution restrictions (e.g. amount of memory needed, libraries that must be present in the target machine) and any other characteristic of the application that may help the scheduler to find the optimal match.

The ability of a grid to handle a large number of computers is also vital if one aims at harnessing the power scattered in the Internet. A scalable system architecture is necessary. This architecture has to be capable of managing thousands or even millions of processing devices while keeping overhead and contention effects controled. While these mechanisms exist for other somewhat similar problems (e.g. the Internet Domain Name Service), the immaturity and lack of standards of grid systems have prevented the acceptance of a common solution in this area. If this problem has gone quite unnoticed so far it is only because worldwide massively parallel grids are still a dream.

A system formed by a large number of computers will have to withstand relatively frequent failures in some of its components. This problem is particularly important in grids since applications have a long execution time and it is not acceptable to loose the results every time a system component fails. Replication and checkpointing techniques – the most popular approaches for fault tolerance in distributed systems – must be adapted to work in large, loosely connected grid environments.

Security is also a great concern when designing a grid system. The grid has to be protected against several threats, coming both from the grid participants and the application programmer side. Grid participants may attempt unauthorized access to sensitive data, either at the data location or while the data is in transit. If the application manages sensitive data (e.g. stock

market predictions) the malicious participant may try to obtain this information from the grid components executing in his/her machine or any other machine from the network. Conversely, application programmers may try to take advantage of the fact that his/her application components will be executed in remote machines to perform all kinds of security attacks against the local computer or any other computer connected by the network.

Although of a more practical nature, the administration of a grid system also poses some interesting problems. Installation and maintainance of the grid code as the specifications evolve is a challenging task, specially considering the potential number of computers involved. Grid systems must provide a mechanism that demands from the grid participant the minimum of technical knowledge possible to install and maintain the code.

This paper introduces JoiN, a Java-based massively parallel grid designed to handle heterogeneous resources in a scalable, secure, fault-tolerant manner. The JoiN project started in 1997 [1] and has evolved over time to a stable and efficient code. It features decentralized management to allow the system to grow, a formally defined application model which is used by a dynamic scheduler to take advantage of the processing power available, a lightweight fault tolerance mechanism based on both replication and checkpointing, a security infrastructure based on roles and remote administration to allow for the automatic update of application and system code.

The rest of the paper is organized as follows. Section 2 presents the architecture of the system, along with a detailed explanation of each of its components. Section 3 introduces the application model. Section 4 focuses on the scheduling strategy used in the system. Section 5 introduces the solution proposed to tolerate failures in key components of the architecture. Section 6 presents the authentication and access control mechanisms used in JoiN. Section 7 presents the implementation status of JoiN. Section 8 cites some work related to JoiN, pointing out similarities and differences. Finally, Section 9 presents the conclusions and future work.

2 Architecture of JoiN

The JoiN architecture is shown in Fig. 1. It consists on a *server*, some *administration modules* and several *groups* interconnected using a specific *topology*. Each group is formed by one *coordinator* and several *workers*.

The function of each component is detailed below.

Server: It acts as the first point of contact to enter the system and manages high level operations, such as installing and submitting applications and keeping general information about the system. It does not participate actively in the execution of an application, so in the case of temporal failure it can be recovered without affecting the performance of the applications being executed.

Groups: Each group is formed by exactly one coordinator and several workers. The workers are responsible for actually executing the applications, so there is potentially a large number of them. The coordinator manages the activities of the workers in its group, and it is responsible for communicating with entities (server, other coordinators) outside the group. Each worker communicates only with its coordinator. Communication among workers is forbidden.

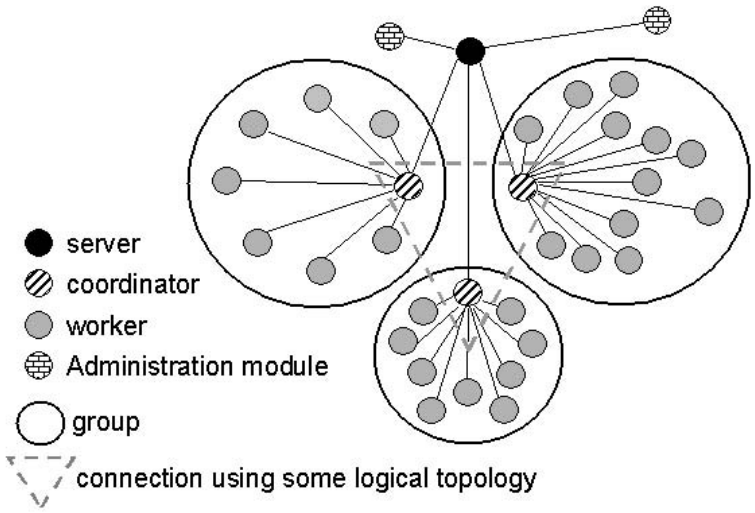

Figure 1: The JoiN architecture.

Interconnection topology: Since there are many groups there has to be a strategy to logically interconnect them. Fully connected graphs, hierarquical graphs [8] and binary hypercubes [9] have already been proposed.

Administration module: Also known as JACK (JoiN Administration and Configuration Kit), its function is to provide a graphical interface on which to perform the basic activities of the system and monitor its status. It is only connected to the server.

The strong point of this architecture is its scalability. As the number of computers in the grid grow, the system can either allocate the newly arriving computers in existing groups or create new groups to accommodate them. If the interconnection topology used among groups is scalable (as the binary hypercube is) then the system can manage a large number of computers without creating any bottleneck.

2.1 Entering the grid

A computer participates in the grid by contacting the server through a well-known address and port. The server decides on which group to allocate the computer and answers back with the address of the coordinator of the group. The computer then contacts the coordinator and from that point on it is treated as a worker of the group. This process is shown in Fig. 2.

In the case of the coordinator the process is similar. The potential coordinator contacts the server, which determines if the system needs a new coordinator. If that is the case, the server starts a new group with that computer as the coordinator. If not, the computer is treated as a worker and the process described in Fig. 2 is carried out. The decision to start a new group can be based on dynamic load information of the current groups or in a previously established configuration (e.g. number of coordinators fixed beforehand).

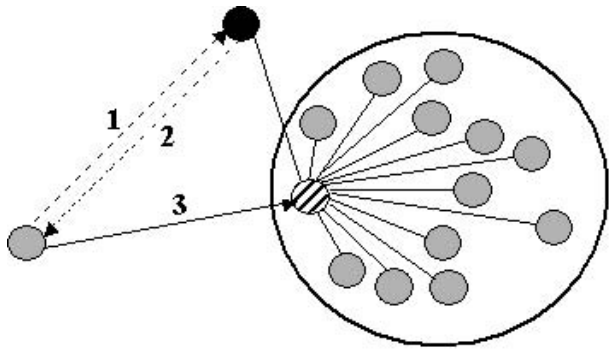

Figure 2: A computer entering JoiN.(1) The computer contacts the server, informing it is available. (2) The server alocates the computer to an existent group. (3) The computer contacts the coordinator of that group.

2.2 Version control

Managing a large grid is a challenging task, specially when considering the efforts that must be employed in mantaining the computational infrastructure up to date. When a new version of the platform is deployed, there is a large number of installed versions that must be updated. JoiN provides an automatic form of version control embedded in the initialization code of the system.

When a computer enters the grid the system performs a check to see if that computer has the latest version of the platform's software. If not, the necessary files are sent to the computer, which updates the files in disk and starts a new attempt to enter the system. Since now the files are updated, this time the request will succeed. This process is shown in Fig. 3.

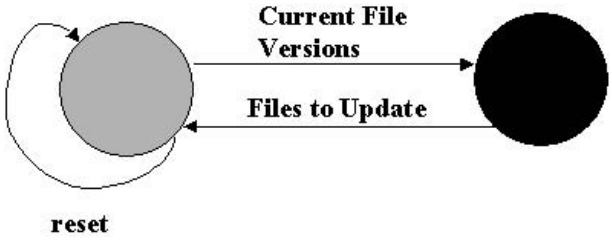

Figure 3: A computer updating system software upon entering JoiN.

To perform a version update in a running grid JoiN provides a reset mechanism using JACK. First, the system code is updated in the server. Then a reset operation is started which forces all grid members to stop executing and reenter the system as if it was the first time. Fig. 4 shows the progress of a reset call.

Since all components will reenter the system the update process described in Fig. 3 is carried out with each of them, thus disseminating the new version to all of the system components.

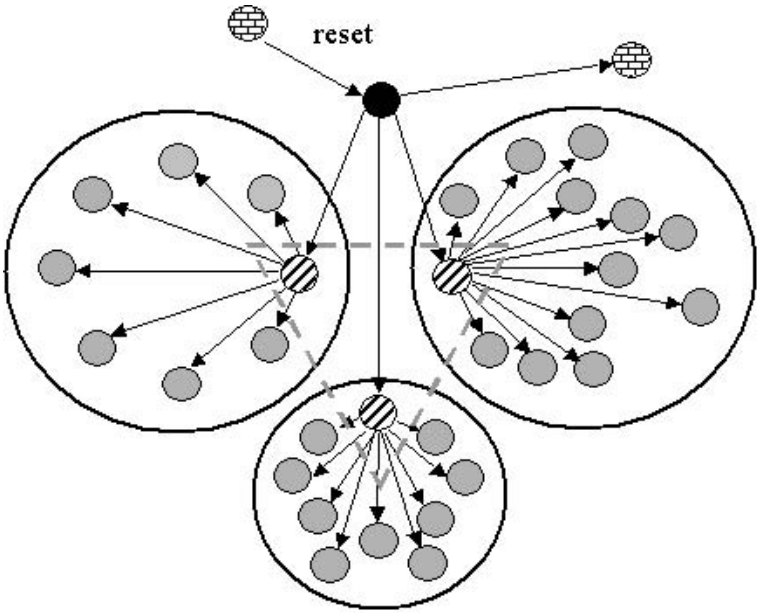

Figure 4: A reset in JoiN. All components are forced to reenter the system.

2.3 Component architecture

Each of the components in JoiN –server, coordinators, workers and JACKs – share the same architecture, regardless of its function in the platform. The architecture of all components is based on the concept of *services*. A service is a software module with a specific function, such as scheduling or fault tolerance. Each component has a *Service Manager*, which is responsible for monitoring the interactions among services. Fig. 5 ilustrates this concept.

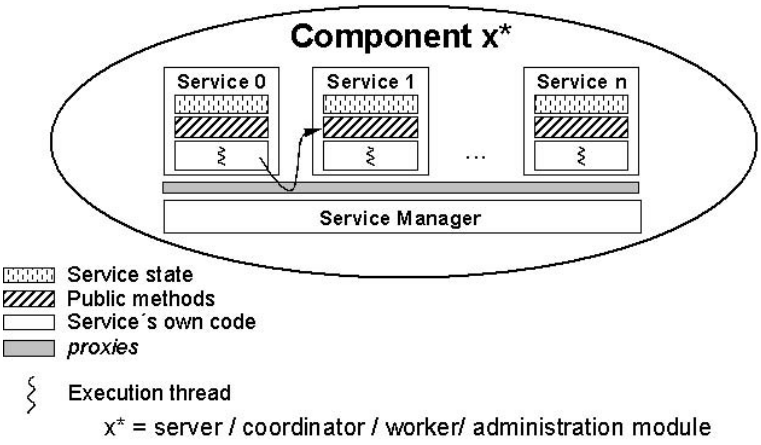

Figure 5: Architecture of a component in JoiN.

Every service can have a representative in each of the components of the platform. Therefore, a service is divided in a *JACK module*, a *server module*, a *coordinator module* and a *worker*

module. A specific component will execute the corresponding modules of all services. For example, the server will execute the server modules of all installed services.

Interactions among different services inside a component are monitored by the service manager through a layer of proxies. This allows for strict security measures to be taken at each step by possibly denying permissions for a service to invoke a method on other service. Interactions among different services outside the frontier of a module are forbidden, but a service can communicate with its peers in other modules.

Several services have already been implemented and are currently part of the platform. Some of them are presented below.

TIDManager: A TID is a Task IDentifier, a unique integer number whose function is to identify an execution thread inside the component.

SecurityManager: Implements the authentication and access control policy inside the system. Every time a service performs a request to any other service inside the same component, the SecurityManager is activated to check if it is a valid interaction. Currently its implementation is based in roles. In this approach users are divided into categories (e.g. administrator, programmer, collaborator) and access control rights are assigned to categories instead of users. This approach allows for a scalable administration as the number of users grow.

ApplicationManager: Responsible for installing, updating, uninstalling, executing and monitoring applications in the system. It has modules in all four components, since all of them are somehow responsible for a part of application execution.

Communicator: This service provides with basic message-passing facilities to allow other services to communicate. It also detects and announces when a component has failed.

TopologyManager: Groups are connected through a logical topology. This service hides the particular details of the topology used and provides a form to communicate among coordinators of different groups.

Scheduler: Divides the work to be done first among groups and then among the workers in each group.

Service-based components provide with a flexible, modular and controllable environment. Each service executes in a sandbox with the points of contact to the outside world being monitored by the service manager. A service can be substituted with a totally different implementation without disturbing any of the other services. New services can be added and existent services removed with minimum impact on the platform. Due to its flexibility, this paradigm has been an enormous help during the development phase of the system, when specifications change constantly. The service paradigm allows strict security measures to be taken at each service interaction without explicitly programming the services to do so.

3 Applications

Applications are installed, submitted and monitored through the JACKs. To install an application it is necessary to inform the set of Java classes containing the code and a Parallel Application Specification (PAS) file describing the structure of the application (precedence relationships and data flow). Applications are installed in the server. Fig. 6 illustrates this process.

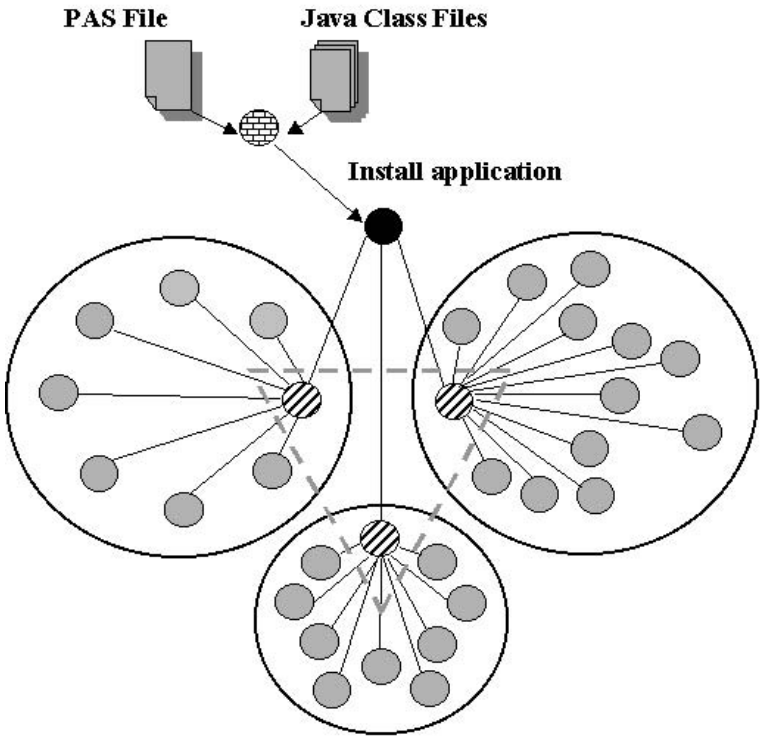

Figure 6: Installing an application in JoiN.

When an application is submitted the request is sent to the server, who sends the request together with the application code to the groups. There are two forms for a server to begin the execution of the application. In the approach described in Fig. 7 the server divides the application among the groups – considering the aggregated power of each group and their current workloads – and directly contacts every coordinator to send the corresponding information. Each coordinator then divides its workload among the workers using a dynamic scheduling algorithm and sends to each worker the corresponding tasks. When the application finishes the generated information makes its way back through the same path: workers to coordinators to server to JACKs.

Although simple, this approach may impose a significative load on the server, specially when considering that it must be kept up to date concerning the load status of all the groups. If the scheduling service decides not to overload the server with the work of dividing the application and monitoring the groups it may take the approach proposed in Fig. 8. In this approach the server simply handles the submit request to one of the groups. The coordinator of this

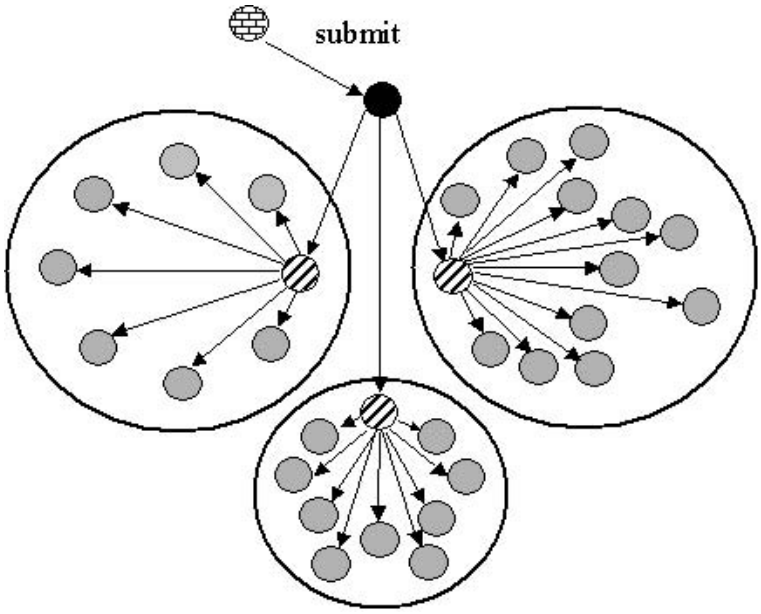

Figure 7: Submitting an application in JoiN. The server divides the work among the groups.

group may divide the workload and share it with other groups using the virtual interconnection topology among them.

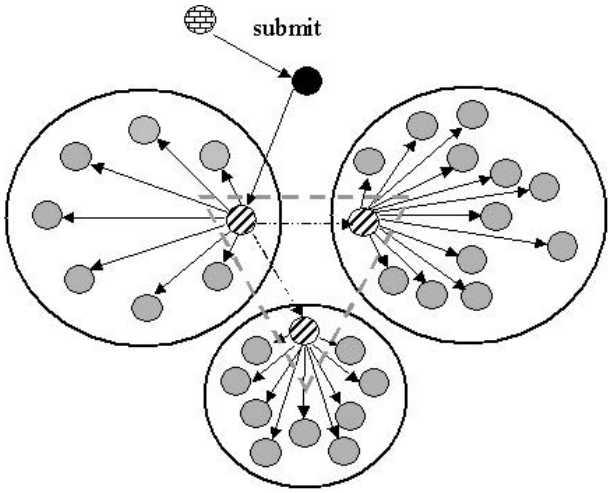

Figure 8: Submitting an application in JoiN. The server handles the application to a coordinator, which uses the interconnection topology to share it with other groups.

3.1 Application model

In order to have better control over the application being executed it is desirable for a grid to know beforehand the structure of the application. Information such as the number of tasks and

the precedence relationships among them can be used by an inteligent scheduler to exploit the computational resources available.

Applications in JoiN can be represented using task graphs. Every node in the graph represents a task in the application while links represent the data flow among the tasks. Every node has an associated *cardinality* that represents the number of instances of the task in the application. Fig. 9 presents an example. The notation T_i^N indicates that task T_i has cardinality N.

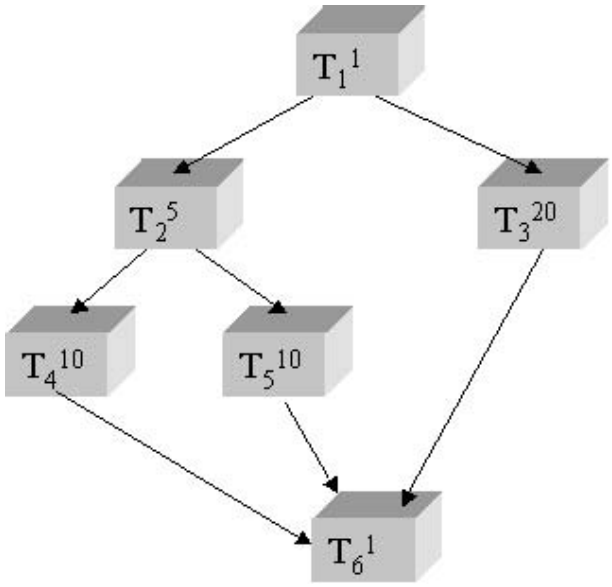

Figure 9: The JoiN application model.

JoiN handles data flow among tasks automatically. A task receives an input and produces an output, unaware of how it will be used when composing the final application. To do so, JoiN demands that a multiplicative relationship exists between tasks in adjacent levels. Observe in Fig. 9 that the output of the 5 instances of task T_2 will be fed to 20 tasks in the subsequent level. JoiN will automatically split the output of the 5 instances of T_2 in 20 parts, liberating the programmer from this tedious job. This approach also allows to implement standalone tasks that can be reused in different applications.

To specify the task graph for a parallel application JoiN uses a PAS (Parallel Application Specification) file. Although a detailed description of the sintax of this file is beyond the scope of this paper, the following text will introduce the main concepts. Fig. 10 shows the PAS file from an actual example of an application that assembles DNA fragments into a single, larger fragment.

A PAS file has three sections: assignment, data link and model. The assignment section declares that the parallel application is formed by three files (which can be found in the `applications/example` directory inside the basic JoiN directory according to the declaration in `path`): `divide.class`, `compare.class` and `assemble.class`. The files are associated with identifiers `T1`, `T2` and `T3` respectively.

```
// Assigment section
path = ‘‘applications/example’’;
T1=divide.class
T2=compare.class;
T3=assemble.class;


// Data link section


T1 = L1(1) << "seqs.txt";
T2 = L2(100) << T1;
T3 = L3(1) << T2 >> "results.txt";


// Model section


L1= delay(10000);
L2= delay(8000000);
L3= delay(10000);
```

Figure 10: An example of a PAS file.

The Data Link section specifies that task `T1` – which is represented by model `L1` and has cardinality `1` – reads its input from file `seqs.txt`. Task `T2` – which is represented by model `L2` and has cardinality 100 – reads its input from `T1`. Finally, task `T3`, who has cardinality 1, reads its input from `T2` and writes its output to file `results.txt`.

Finally, the model section uses the PAMELA language [7] to provide a parameterized model of each task. This information may be used by the scheduler to predict the execution time for a set of candidate allocations. In this example the model shown is extremely simple, using only the `delay` instruction to specify the time it will take each task to execute. More complex models may allow parameters to be specified and instantiatied later according to grid characteristics, providing the scheduler with important performance information.

4 Scheduling

The scheduling algorithm plays an important role in the efficiency of the grid. Being able to understand the differences among the available resources and to adapt the application scheduling after failures or sudden changes in performance is vital for a grid scheduler.

In order to gather the information needed for scheduling JoiN uses a benchmark based on those developed by the Java Grande Forum, a research group devoted to use Java for scientific applications (www.epcc.ed.ac.uk/javagrande, June 2003). This benchmark is executed in the

workers at the moment they enter the system to assess, as accurately as possible, its computing power. This information is sent to the Coordinator of the group, where the scheduling decisions are taken.

The scheduling algorithm used in JoiN is an extension of the Generational Scheduling (GS) proposed by Carter et al. in [4]. Generational Scheduling reduces the scheduling problem by considering only the tasks ready for execution. This reduction creates a subproblem where no precedence relations exist. The subproblem can be solved with any simple scheduling algorithm, such as *trivial*, which attempts to even the number of tasks in each worker, or *best-fit*, which tries to balance the workload among the workers.

Generational Scheduling performs a static scheduling at each stage. Every task ready for execution in a stage is allocated at the beginning of the stage. This fact prevents the algorithm from adapting to changes in worker performance or even failures. Furthermore, once a task begins executing it is no longer considered for scheduling, so if the task stops the application may never finish.

The extension proposed and implemented in JoiN, called *Generational Scheduling with Task Replication* (GSTR), proposes two major modifications to GS. First, at each stage the scheduler considers the tasks ready for execution *and the tasks currently executing*. As a consequence a task may be scheduled to a worker *while it is still executing in other worker*, effectively replicating it. This simple extension provides a powerful fault tolerance mechanism, since a task that failed while executing will eventually be re-scheduled. It also provides a way to adapt to changes in performance. If a task takes too long to finish, it will be re-scheduled to another worker. Finally, GSTR allows the implementation of a mechanism to detect Byzantine failures – on which the tasks keep executing but provide incorrect results – or even malicious tasks knowingly generating erroneous data, simply by replicating the tasks on different workers and comparing the results.

The second modification introduced by GSTR also aims at adapting the scheduling decisions to changes in worker performance. Instead of scheduling all tasks at the beginning of each stage as GS does, GSTR schedules only *part of the tasks*. The rest of the tasks is scheduled on demand, i.e. once a worker finishes executing the tasks statically allocated to it, it begins asking the Coordinator for more tasks. As a consequence, even if the data provided by the benchmarks is no longer valid, this dynamic section of the scheduling will be able to adapt to the new conditions. Several experiments have shown that GSTR behaves as expected, providing both fault tolerance and load balance to the system.

5 Fault Tolerance

Fault tolerance for worker computers is covered by the replication of parallel tasks presented in the previous section. This section presents the fault tolerance mechanisms for the coordinators and the server, which are also needed since these components store important information concerning the status of the system.

JoiN implements a fault tolerance mechanism based on log/replay and checkpoint/rollback techniques for coordinators and servers. This mechanism uses the fact that the platform is formed by a collection of independent services interacting through a layer of proxies. Assumming that each service is deterministic (i.e. given the same initial state it will always reach the same

end state) then the state of a service is affected by: (a) its own (deterministic) set of instructions and (b) its interactions with other services.

JoiN uses the proxy layer in every interaction between services to log enough information to be able to replay it later if the component fails. Should a failure occur, each service is restarted and the interactions among them are replayed. By replay we mean that if a service attempts to make a call that was already made before the failure the proxy layer will use the information stored in the log to return to the caller the same result it obtained the first time. Since each service is deterministic this technique guarantees that every service will reach the same state it had before the failure.

Considering that a grid system like JoiN will be running for long periods of time this simple log/replay approach is not enough, since the size of the log may become too large and the replay phase – if an error occurs – may take too long to execute. JoiN introduces a checkpoint/rollback strategy to solve this problem. Every service checkpoints its status periodically, thus permitting some log entries to be discarded. If a failure occurs each service in the platform is restarted from its last checkpoint and only the interactions that took place between the last checkpoint and the failure need to be replayed. With incremental checkpoints and efficient log management this technique has proven to introduce little overhead to the system performance under normal conditions. According to tests conducted, the logging technique introduced a mean overhead of only $2,48\%$ to the normal functioning of the system.

6 Security

JoiN uses an authentication and access control mechanism based on roles. The system defines some static roles (administrator, service programmers, application programmer, computer owner and guests) and grants some permissions to each role according to the security policy desired. Each user in the system is associated with a role, thereby inheriting the permissions granted to it. User authentication is performed through the usual login/password mechanism, consulting a user database stored in the server.

Once a user authenticates he/she will be associated with his/her permissions, according to the role specified. These permissions will be checked at every service interaction by the Security Manager, a special service activated by the proxy layer to determine if an interaction between services is legal (Fig.11(a)). If the interaction violates the security policy of the system an exception will be raised and the interaction aborted (Fig. 11(b)). As it was the case with fault tolerance, the service-based architecture with a proxy layer allowed a powerful, non-intrusive security mechanism to be implemented.

The choice of role-based access control – instead of the usual discretionary access control where permissions are granted directly to users – was motivated by the fact that grid users can be grouped in large classes according to the nature of their interactions with the grid. Granting permissions to each class of users (roles) facilitates administration, specially considering that the potential number of users may be large.

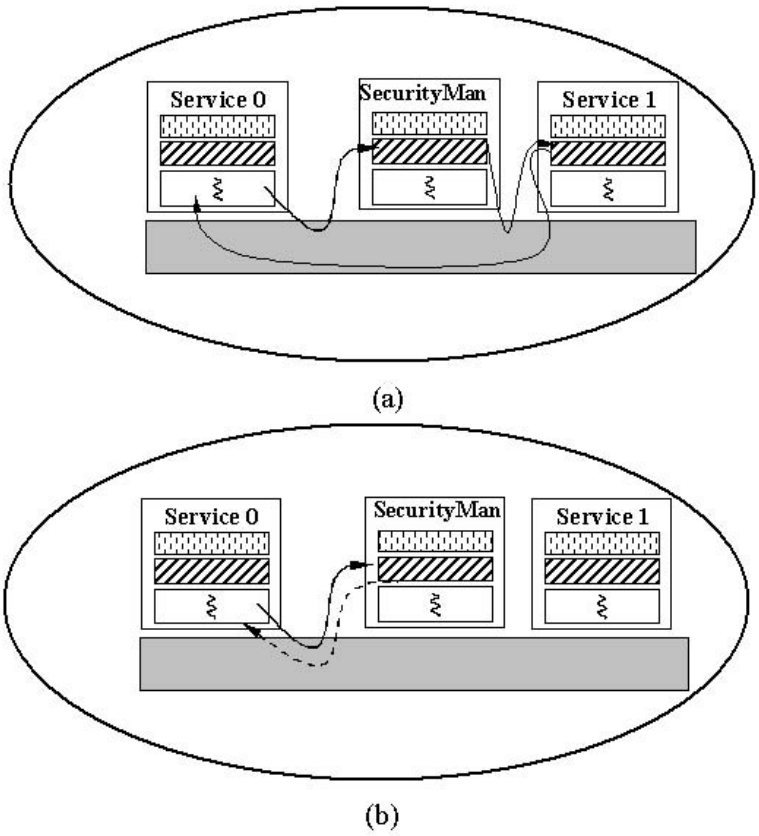

Figure 11: The JoiN access control mechanism. (a) The SecurityManager allows an interaction between services. (b) The SecurityManager denies the interaction.

7 Implementation status

The implementation of the first version of JoiN began early in 1997. Currently in version 1.3, JoiN has proven to be an efficient and robust platform on which to execute parallel applications. Services for scheduling, fault tolerance, security, file input/output and platform administration are already implemented. Other services, such as an advanced performance predictor and the topology service (which handles the interactions among different groups) are already under study.

During these years of development several applications have been implemented in JoiN. Some of them are listed below.

- A DNA sequencing application, which processes a set of DNA fragments in order to build the original sequence.
- An application searching for large prime numbers.
- A factoring application for large integer numbers.
- Monte Carlo simulations to calculate the value of Pi.
- A genetic algorithm for the Traveling Salesman Problem.

- Optimization of the distribution of goods among several sale points.
- A general algorithm to calculate optimal task allocation to processors
- An application to test the scalability of web servers.
- A pattern matching application to recognize paternity based on several DNA samples.
- Analisis of philogenetic trees.

JoiN is currently installed in a set of computers at the Faculty of Electrical and Computing Engineering of the State University of Campinas, Brazil. Efforts are currently under way to expand it to other laboratories inside and outside the University.

8 Related Work

Several solutions have been proposed so far to build massively parallel grids for academic and commercial use. SETI@home [2] is perhaps the most popular grid nowadays. Downloaded by over 4 million users – with over 600 000 of them actively participating in june 2002 – this project showed for the first time the enormous aggregated computing power obtainable from the computers connected to Internet. SETI@home is oriented to execute a single application that searches for signs of extraterrestial inteligence by processing signals generated by radiotelescopes.

Globus [5] is a widely known academic grid. Globus proposes a set of basic grid services – the Globus toolkit – upon which to build more sophisticated systems. The Globus toolkit provides basic services for communication, resource allocation and security. High-level systems such as Legion and Condor use these basic services as a starting point to build a grid.

Condor [12] was one of the first successful attempts to build a geographically distributed grid. Condor is basically a batch system. The user submits his/her applications and Condor takes care of finding an idle processor in the grid to allocate it. If the processor is needed, Condor migrates the application to another idle processor. A subsystem named Condor-G [6] allows Condor to interact with the grid services provided by Globus.

Legion [8] gathers a set of heterogeneous computing resources and presents it to the user as a single virtual computer. The basic components in the system are objects, which communicate through method invocations. Legion promotes flexibility in the system by allowing different implementations of key components.

NetSolve/GridSolve [3] is a system designed to solve scientific problems in distributed environments. Clients send a request to a *NetSolve Agent*, whose job is to find the necessary resources to fulfill the request and coordinate its execution. NetSolve offers programming interfaces for C, Fortran, Matlab and Java.

Java-based systems, such as Manta [13], IceT [11] and Javelin++ [10] leverage the platform independency and standard API of Java to provide simple, unique platforms that can be executed on most of the existing architectures. These solutions differ in the application models they execute and in the attention they pay to key issues such as scalability and fault tolerance.

9 Conclusions and future work

This paper presented JoiN, a scalable, extensible Java-based grid developed in the State University of Campinas. JoiN divides the computers participating in the grid into groups, each having a coordinator and several workers. Groups are connected among themselves through a virtual topology. This architecture guarantees that a large number of computers can be accommodated, since – considering that the virtual topology is chosen wisely – it is possible to add more groups if the existing groups are overloaded.

JoiN features an architecture based on the concept of services. A service is an independent software component that executes across the system to provide a specific functionality. Service collaborate through a layer of proxies, which is responsible for enforcing security policies and registering all service interactions as a part of the fault tolerance solution implemented.

Since its inception in 1997 several applications have been implemented in JoiN. Successful tests conducted so far have provided enough incentive to estimulate further efforts to extend JoiN to other laboratories in the campus and beyond.

Current research is oriented to define, among the possible virtual topologies, which one should be used by the platform. Current candidates are a hierarchy of groups and a binary hypercube. Also, an accurate performance prediction strategy is being developed for the grid. Finally, efforts will be dedicated to propose mechanisms to form groups that maximize the efficiency in the execution of parallel applications.

References

[1] Marco Aurélio Amaral Henriques. A proposal for java based massively parallel processing on the web. In *Proceedings of The First Annual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing*, pages 59–66, Rhodes, Greece, June 1999.

[2] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[3] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users guide to netsolve. Innovative Computing Department Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002. Also in the Proceedings of Supercomputing'96, Pittsburgh.

[4] Brent R. Carter, Daniel W. Watson, Freund Richard, Elaine Keith, Francesco Mirabile, and Howard Jay Siegel. Generational scheduling for dynamic task management in heterogeneous computing systems. *Journal of Information Sciences*, 106:219–236, 1998.

[5] Ian Foster, Carl Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002.

[6] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-g: A computation manager agent for multi-institutional grids. *Journal of Cluster Computing*, 5:237–246, 2002.

[7] Arie Jan Cornelis Van Gemund. *Performance Modeling of Parallel Systems.* Coronet Books, July 1996.

[8] Andrew S. Grimshaw and Wm. A. Wulf. Legion - A View From 50,000 Feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, Agosto 1996. IEEE Computer Society Press.

[9] Eduardo Javier Huerta Yero and Marco Aurélio Amaral Henriques. A Method to Solve the Scalability Problem in Massively Parallel Processing on the Internet. In *7th Euromicro Workshop on Parallel and Distributed Processing*, University of Madeira, Funchal, Portugal, 3 a 5 de Fevereiro 1999. IEEE Computer Press.

[10] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins, and Peter Capello. Javelin++: Scalability Issues in Global Computing. In *Proceedings of the ACM 1999 Java Grande Conference*, Palo Alto, California,, June 1999.

[11] Vaidy S. Sunderam. Metacomputing with the harness and icet systems. In *International Conference on Computational Science*, 2001.

[12] Douglas Thain, Todd Tanenbaum, and Miron Livny. Condor and the Grid. In Fran Berman, Anthony J. G. Hey, and Geoffrey Fox, editors, *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley, 2003.

[13] Rob van Nieuwpoort, Jason Maasen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing using the remote invocation model. *Concurrency: Practice and Experience*, 12(8):643–666, 2000.

5.2 Sumário do capítulo

JoiN é um sistema que permite construir um grid de computadores para a execução de aplicações maciçamente paralelas. A arquitetura do sistema está formada por um servidor e vários grupos de computadores, cada um deles com um coordenador e vários trabalhadores. Existe também a possibilidade de gerenciar o sistema através da interface gráfica implementada em um módulo ligado ao servidor.

Devido à sua arquitetura JoiN pode conter um grande número de computadores sem que ocorram gargalos significativos. Um escalonador semi-estático garante que os trabalhadores são explorados satisfatoriamente em benefício das aplicações sendo executadas, enquanto mecanismos de tolerância a falhas garantem que as aplicações terminem mesmo na presença de falhas em alguns componentes.

Esta é uma plataforma em expansão e constantes esforços estão sendo investidos no seu aprimoramento e implantação estável na maior quantidade possível de computadores.

Nos capítulos seguintes veremos com mais detalhes as soluções propostas neste trabalho para os problemas específicos de escalonamento e tolerância a falhas.

Capítulo 6

Escalonamento em um grid

O Capítulo 5 introduziu a estratégia de escalonamento proposta. O algoritmo utilizado, chamado *Generational Scheduling with Task Replication (GSTR)*, foi desenvolvido como uma extensão ao *Generational Scheduling (GS)*, uma estratégia de escalonamento conhecida e com bons resultados. A principal modificação introduzida é a de permitir a replicação de tarefas como forma de se adaptar a falhas e mudanças significativas de desempenho nos computadores participantes no grid. Este capítulo apresenta uma visão mais detalhada desta solução junto com resultados de vários testes realizados na prática.

Um ponto a ser destacado da proposta de escalonamento apresentada é que, embora tenha sido implementada e testada em JoiN, ela é válida em geral para grids que seguem a arquitetura proposta neste trabalho. Ainda, o algoritmo de escalonamento exige apenas que as relações de precedência entre as tarefas sejam conhecidas de antemão; portanto vários modelos de aplicação paralela – e não apenas o modelo utilizado em JoiN– podem usufruir das vantagens do algoritmo proposto. Por esta razão este capítulo fará referências ao JoiN apenas na apresentação dos resultados práticos.

O artigo *"An Adaptive and Fault Tolerant Scheduler for Grids"*, apresentado a seguir, explica a estratégia utilizada junto com os resultados práticos obtidos e uma análise detalhada dos mesmos.

An Adaptive and Fault Tolerant Scheduler for Grids*

Eduardo J. Huerta Yero†, Fabiano de O. Lucchese,
Francisco S. Sambatti‡ and Marco A. A. Henriques
DCA - FEEC - UNICAMP
[huerta,flucches,sambatti,marco]@dca.fee.unicamp.br

Abstract

The recent development of telecommunication infra-structures such as the world-wide data networks, interconnecting millions of computers spread all over the world, has made possible the use of large computational resources at a rather low cost. Within this new reality research activities and projects related to *grid computing* have emerged and established as a solid trend in distributed parallel computing. However, as in every new domain of research there are many unsolved questions, in particular those related to the management of the processing load into the system. In this work the problem of balancing processing loads on a grid is solved by the introduction of the Generational Scheduling with Task Replication (GSTR) algorithm. A comprehensive set of tests is carried out in order to validate the proposed solution. The JoiN platform is used as a testbed for the evaluation of the performance of the solution when applied to real computational environments.

1 Introduction

As grid computing becomes a commonplace in parallel processing, better models are needed to understand and exploit these large, weakly connected, heterogeneous environments. Executing a parallel application while the number of available resources – and even the quality of the resources – change over time requires a sensible and adaptive load balancing scheme to be used in the grid.

The component responsible for load balancing in a parallel system is called a *scheduler*. Although there are many known scheduling strategies, in general they can be classified in three groups: *static*, *dynamic* and *semi-static* (or *semi-dynamic*) [2] [3] [4]. Static schedulers make the allocation decisions at the beginning of application execution and do not interfere again through the lifetime of the application. Dynamic schedulers allocate tasks as processors become idle, so it keeps working during the execution of the application. Static schedulers tend to need information about the system, such as relative power of the processors and link speed, but they

*Work partially supported by grant 98/04305-9 of the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

†On leave from University of Havana, Cuba

‡Western Paraná State University

do not generate further overhead once the allocations are made. At the other hand, dynamic schedulers can be implemented without knowledge about the system, but requests for work made by idle processors are interleaved with application execution. Semi-static schedulers fall somewhere in the middle in an attempt to leverage the strong points of each approach.

The goal of this paper is to present a semi-static, adaptive, fault tolerant scheduling algorithm for grids. The proposed algorithm, called *Generational Scheduling with Task Replication*, or GSTR, is an extension of the *Generational Scheduling* (GS) approach. Generational Scheduling is a simple, low-complexity algorithm first proposed by Carter et. al. in [4]. The GSTR algorithm makes key modifications to GS in order to enhance its results in large, unstable and heterogeneous environments.

The rest of this paper is organized as follows. Section 2 presents the applications and grid models used by GSTR. Section 3 presents the algorithm, focusing on the extensions proposed to GS. Section 4 shows the comparative results of tests conducted in a set of heterogeneous computers. Finally, Section 5 presents the conclusions and future work.

2 Models

This section defines the application and grid models on which the proposed algorithm operates. This models are designed to be as general as possible to facilitate the inclusion of the GSTR algorithm in a variety of environments.

The only requirement needed for the parallel application is that the precedence relationships among the parallel tasks must be known before the execution starts. Being so, the parallel application is modeled by a general task graph like the one shown in Fig. 1. Nodes in the graph represent a task while links going from task T_i to T_j indicate that T_j can begin executing only after T_i terminates.

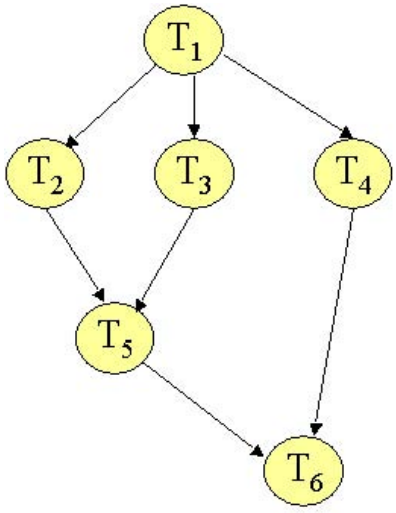

Figure 1: An example of a task graph.

The grid is modeled as a set of *groups*, each composed by a *coordinator* and several *workers* (Fig. 2). The coordinator manages the execution of parallel applications within the group by

allocating parts of the application to the workers and overseeing the execution. Groups can be connected using a *virtual topology* which allows them to collaborate. Only coordinators can communicate with coordinators in other groups.

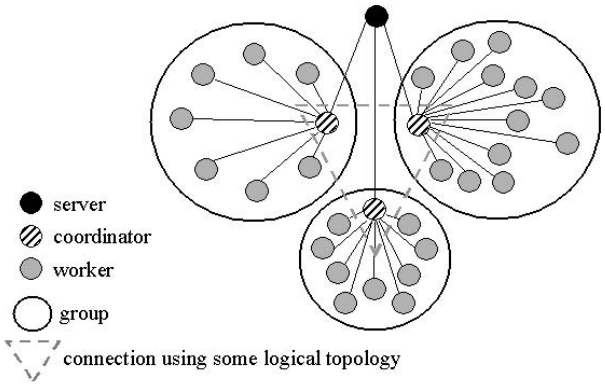

Figure 2: The grid model.

Finally, let us assume that the coordinators are aware of the relative computing power of each of its workers. This measure can be obtained by executing a set of benchmarks on each of the workers, either at startup or periodically. This information will be used by the proposed scheduling algorithm to determine the best way to distribute parallel tasks among the workers.

We believe that the application and grid model used here are general enough to be adapted to many existing grids, such as Condor [6], Globus [7] and Legion [8].

3 Generational Scheduling with Task Replication

To understand the algorithm proposed here let us first introduce Generational Scheduling, the algorithm upon which this proposal is based.

Generational Scheduling reduces the scheduling problem by considering only the tasks ready for execution. This reduction creates a subproblem where no precedence relations exist. The subproblem can be solved with any simple scheduling algorithm, such as *trivial*, which attempts to even the number of tasks in each worker, or *best-fit* which tries to balance the workload among the workers. In GS, all tasks at this stage are statically allocated so, if the benchmarks results are inaccurate or the performance of a worker changes, the algorithm is uncapable of adjusting itself to the new conditions. Furthermore, when a task is allocated it stops being a candidate for allocation, which means that if the task stops, the application may never finish.

The extension presented in this paper, called *Generational Scheduling with Task Replication* (GSTR), proposes two major modifications to GS. First, at each stage the scheduler considers the tasks ready for execution *and the tasks currently executing*. As a consequence a task may be scheduled to a worker (with available processing power) *while it is still executing in other worker*, effectively replicating it. This simple extension provides a powerful fault tolerance mechanism, since a task that failed while executing will somehow be re-scheduled. It also provides a way to adapt to changes in performance. If a task takes too long to finish, it will be re-scheduled to another worker. Finally, GSTR allows the implementation of a mechanism to

detect Byzantine failures – on which the tasks keep executing but provide incorrect results – or even malicious tasks knowingly generating incorrect data – simply by replicating the tasks on different workers and comparing the results.

The second modification introduced by GSTR also aims at adapting the scheduling decisions to changes in worker performance. Instead of scheduling all tasks at the beginning of each stage as GS does, GSTR schedules only a *part of the tasks*. The rest of the tasks is scheduled on demand, i.e. once a worker finishes executing the tasks statically allocated to it, it begins asking the Coordinator for more tasks. As a consequence, if the data provided by the benchmarks is no longer valid, this dynamic section of the scheduling will be able to adapt to the new conditions.

Formally, the GSTR algorithm can be described in four steps.

1. The scheduling problem is formalized, producing a task graph similar to the one in Fig. 1, where tasks and precedence relationships are clearly defined.

2. Tasks that are not ready for execution due to unsatisfied precedence restrictions are temporarily ignored, effectively reducing the task graph to a smaller set where no precedence relationships exist.

3. *Part of the tasks in this set is statically scheduled, while the rest will be allocated dynamically as tasks finish executing.* Any scheduling algorithm (trivial, best-fit) can be used to make the static allocation. At this stage the scheduler considers the tasks waiting to execute *and those who are currently executing.*

4. Re-scheduling events, such as tasks finishing their executions, make the algorithm go back to stage 2 to reconstruct the set of ready tasks, since other precedence relationships may now be satisfied.

The modifications introduced by GSTR ensure that, if a task fails or takes too long to finish due to a sudden change in processor load, it will be replicated in another worker and the application will keep progressing. Also, the fact that only part of the tasks are statically allocated at each stage protects the algorithm against inaccurate data provided by the benchmarks at each worker. Replicated tasks have a lower priority than original ones, in order to avoid that unnecessary task replications affect the overall performance while executing in normal conditions.

Observe that this algorithm can be used both inside a group or among several groups. Within a single group the coordinator executes the algorithm and allocates tasks to the workers as described above. With a simple modification this same scheme may be used with several groups. Fig. 3 ilustrates a possibility where a central component (represented as a black circle) coordinates the execution of applications among several logically interconnected groups. Each group has a coordinator (represented in stripes) and several workers, which are the ones that actually execute tasks. The central component executes the algorithm and allocates the tasks to the coordinator of each group according to the group computing power, which can be obtained by adding the power of all the workers in the group. Each coordinator executes the allocated tasks and inform the central component of re-scheduling events. The meaning of re-scheduling event is maintained in this case, i.e. the conclusion of just one task is sufficient to trigger a new task distribution.

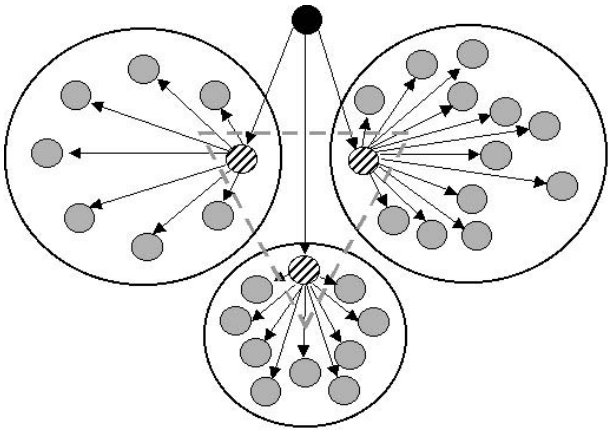

Figure 3: GSTR with several groups and a central component.

A modification of this proposal that does not require a central component is shown in Fig. 4. A group may become part of another group by making its coordinator act as a fake worker in front of the other coordinator. The processing power reported by the fake worker would be the sum of all processing powers of the workers in the coordinator's group. As a consequence the coordinator will receive a workload proportional to the power of its workers, which it will distribute among them. Whenever a task finishes the coordinator acting as a fake worker will inform the actual coordinator, so new tasks can be allocated.

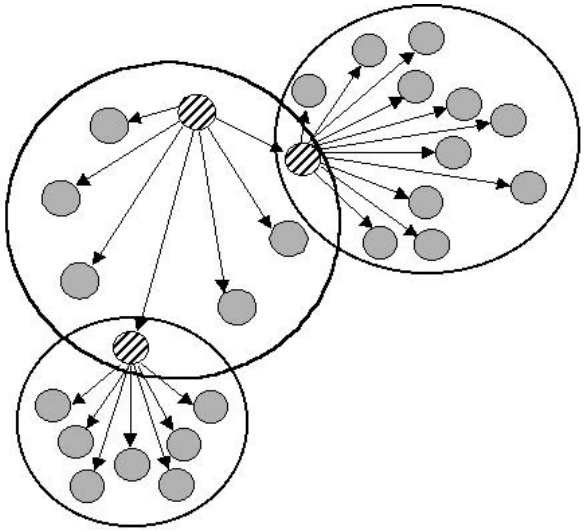

Figure 4: GSTR with several groups and no central component.

4 Test results

This section presents results obtained from tests conducted on the JoiN platform [1]. JoiN is a Java-based grid system with grid and application models like the ones described in Section 2.

Name	EP	RP	Error (%)
Itapua	1.0	1.0	-
Rocas	0.82	0.81	1.23
Dunas	0.54	0.81	-33.3
Jureia	0.82	0.80	2.50
Gorda	0.67	0.80	-16.2
Botafogo	0.53	0.45	17.7
Copacabana	0.44	0.38	15.7
Buzios	0.25	0.38	-34.2
Grumari	0.44	0.38	15.7
Parati	0.37	0.32	15.6
Peruibe	0.21	0.14	50.0
Aracati	0.21	0.14	50.0
Brava	0.21	0.14	50.0
Caragua	0.21	0.13	61.5

Table 1: Performance results for the 14 computers (EP: Estimated Performance, RP: Real Performance).

The goal of these tests was to compare the proposed algorithm with some of the alternatives available for grids.

To test the scheduling algorithms we executed an application that searches for prime numbers inside a specified interval using the Erathostenes Number Sieve algorithm [5]. The application is composed by an initial task that divides the search space into 50 smaller intervals, 50 tasks to process each subinterval looking for prime numbers and a final task that collects the results. Three different interval sizes were tested: 0 to 10^9, 0 to 3×10^9 and 0 to 5×10^9. From now on, these application instances will be referred to as instance 1, 3 and 5, respectively.

The tests were conducted in 14 computers of the Laboratory of Computer Engineering and IndustrialAutomation, Faculty of Electrical and Computer Engineering, State University of Campinas. A set of benchmarks derived from those developed by the Java Grande Forum (www.epc.ed.ac.uk/javagrande, june 2003) was used in order to estimate the processing power of each of these computers, and the results were normalized against the fastest machine. The relative powers obtained are shown in Table 1, ordered from the fastest to the slowest computers.

This table shows three columns: Estimated Performance (EP), Real Performance (RP) and Error. Estimated Performance numbers were derived from the Java Grande Benchmarks, whereas Real Performance figures were derived from executing a sequential version of the parallel application in each computer. The difference between those results is shown in the Error column. This column is an indication of the estimative errors induced by the benchmarks for this particular application. As it can be observed, the error is larger for the less powerful computers. Although it is not common to run a sequential version of the applications to get more precise benchmarks, this technique was used here to produce a deeper analysis of the proposed algorithm.

4.1 Preliminary analysis

Since both the real and estimated performances are known for every computer it is possible to calculate – before the execution – the ideal and actual scheduling produced by these figures. The ideal scheduling, calculated using the RP column in Table 1, is shown in Table 2, with the number of computers varying from 1 to 14. Observe that computers were included in the same order as in Table 1, i.e., from fastest to slowest, based on RP values.

Configuration	**Tasks / Processor**													
1	50	-	-	-	-	-	-	-	-	-	-	-	-	-
2	28	22	-	-	-	-	-	-	-	-	-	-	-	-
3	19	16	15	-	-	-	-	-	-	-	-	-	-	-
4	15	12	12	11	-	-	-	-	-	-	-	-	-	-
5	12	10	10	9	9	-	-	-	-	-	-	-	-	-
6	11	9	9	8	8	5	-	-	-	-	-	-	-	-
7	10	8	8	8	8	4	4	-	-	-	-	-	-	-
8	10	8	8	7	7	4	3	3	-	-	-	-	-	-
9	9	7	7	7	7	4	3	3	3	-	-	-	-	-
10	8	7	7	7	7	3	3	3	3	2	-	-	-	-
11	8	7	7	7	6	3	3	3	3	2	1	-	-	-
12	8	7	7	6	6	3	3	3	3	2	1	1	-	-
13	8	7	6	6	6	3	3	3	3	2	1	1	1	-
14	8	6	6	6	6	3	3	3	3	2	1	1	1	1

Table 2: Ideal scheduling calculated using Real Performance (RP) factors.

The actual scheduling used by the platform, derived from the EP numbers obtained from the benchmarks, is shown in Table 3. In both cases there were 50 tasks scheduled.

Configuration	**Tasks / Processor**													
1	50	-	-	-	-	-	-	-	-	-	-	-	-	-
2	27	23	-	-	-	-	-	-	-	-	-	-	-	-
3	21	18	11	-	-	-	-	-	-	-	-	-	-	-
4	16	13	8	13	-	-	-	-	-	-	-	-	-	-
5	13	11	17	11	8	-	-	-	-	-	-	-	-	-
6	12	9	6	9	7	6	-	-	-	-	-	-	-	-
7	11	9	5	9	7	5	4	-	-	-	-	-	-	-
8	10	9	5	8	6	5	4	2	-	-	-	-	-	-
9	9	8	5	7	6	5	4	2	4	-	-	-	-	-
10	9	7	4	7	6	4	4	2	4	3	-	-	-	-
11	9	7	4	7	6	4	4	2	3	3	1	-	-	-
12	9	7	4	7	6	4	2	2	3	3	1	1	-	-
13	8	7	4	7	6	4	3	2	3	3	1	1	1	-
14	8	7	4	7	5	4	3	2	3	3	1	1	1	1

Table 3: Actual scheduling used by the platform, calculated using Estimated Performance (EP) factors.

A comparison of Table 2 and Table 3 illustrates the impact of the estimative errors induced by the benchmarks. Fig. 5 shows the calculated execution time for both the ideal and actual scheduling. As expected the difference between them grows as the slower computers – who have larger estimative errors – are introduced.

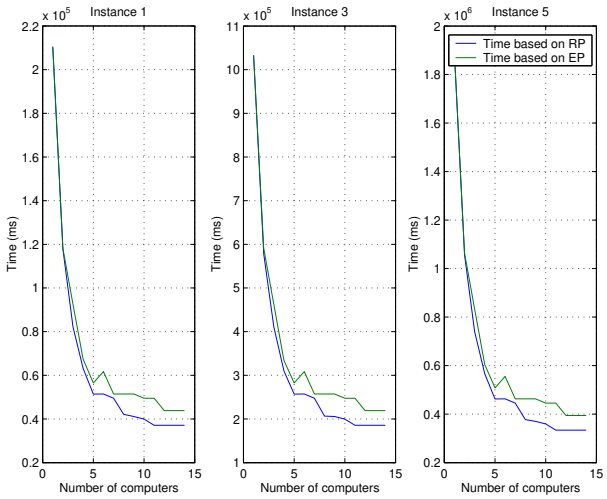

Figure 5: Calculated execution times for the ideal and actual schedulings.

Fig. 6 ilustrates the impact of these errors on the calculated speedup. It shows 3 curves: the ideal speedup curve, obtained by adding the processing powers of the computers executing the application, the speedup obtained by the ideal scheduling (based on RP) and the speedup obtained from the actual scheduling, derived from the benchmarks (EP). In all cases the sequential execution time used to calculate the speedup was measured in the fastest machine.

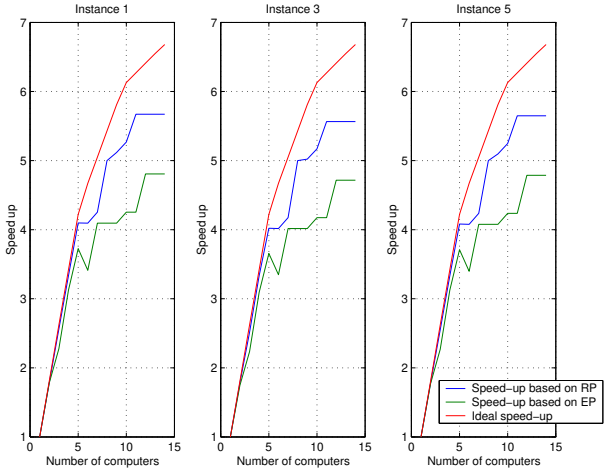

Figure 6: Calculated speedups for the ideal and actual schedulings.

As expected, even the ideal scheduling does not match the ideal speedup curve, due to overhead generated by synchronization and because it is impossible to obtain an exact load balance when allocating tasks, which are an unbreakable set of operations. For example, the *best* scheduling when using `itapua` and `rocas` is 27.6 and 22.4 tasks respectively, but these numbers have to be rounded up with the consequent loss in performance. The speedup of the actual scheduling used is, as expected, consistently lower than the one produced by the ideal scheduling, with the difference growing as the slower computers are introduced.

Performance Fators	Instance	Mean Efficiency
Real	1	91.15 %
	3	90.12 %
	5	90.95 %
Estimated	1	80.13 %
	3	79.02 %
	5	79.90 %

Table 4: Mean efficiency values for the ideal and actual schedulings.

Finally, Table 4 shows the mean efficiency values obtained for both cases, where each value was obtained by dividing the actual speedup by the ideal one.

4.2 Experiments

The experiments conducted used six different scheduling strategies.

trivial: Statically distributes 100% of the tasks by allocating the same number of tasks to each computer.

best-fit: Statically distributes 100% of the tasks according to the processing power of each computer, trying to even time wasted by each of them to execute the tasks.

GS: Allocating statically 50% of the tasks and

- using trivial scheduling at each stage or
- using best-fit scheduling at each stage.

GSTR: Allocating statically 50% of the tasks and

- using trivial or
- using best-fit scheduling at each stage.

For the GS and GSTR algorithms half the tasks were allocated statically, while the other half was left to be scheduled in the dynamic phase of the algorithm.

Fig. 7, Fig. 8 and Fig. 9 present the execution times obtained for the three instances of the problem for each of the six combinations. The ideal time curve represents the time that would be obtained if the best scheduling decision was made, i.e. using RP factors. It can be observed that the size of the problem does not have much impact on the shape of the curves, which are similar for all cases.

Observe that when the trivial algorithm is used (graphic at the left on each figure) there is a significant decrease in performance, since the relative power of each computer is not considered. However, the GS and GSTR schedulers overcome this problem with the dynamic portion of the algorithm, with GSTR having the best results.

Dynamic scheduling and task replication are useful to deal with failures, sudden changes in performance or poorly estimated processing power for the workers. Since one can consider

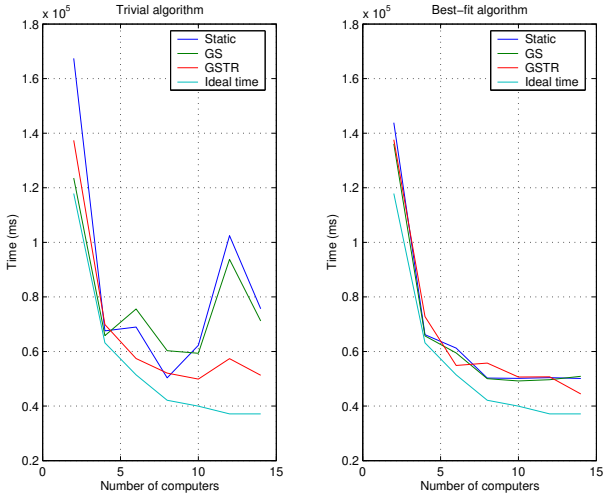

Figure 7: Execution time for Instance 1 of the problem.

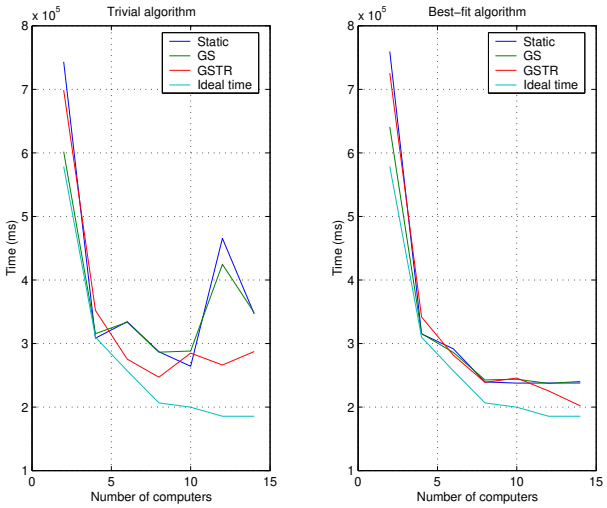

Figure 8: Execution time for Instance 3 of the problem.

the trivial algorithm to be a particular case of best-fit with errors in estimation, it is logical to obtain results where GSTR behaves better.

When best-fit is used, the advantages of GSTR are not as clear. The question is: is GSTR much worse in those cases? The graphics at the right on Fig. 7, Fig. 8 and Fig. 9 show GSTR with results similar to those obtained from GS and the purely static approaches, indicating that the overhead generated by GSTR's dynamic characteristics do not have a significative impact on performance. Fig. 10 shows the speedup values obtained for the largest instance of the problem. These values were obtained by dividing the sequential execution time in the fastest machine by the parallel execution times. Similarly, Fig. 11 presents the efficiency values for this instance, obtained from the actual speedup divided by the ideal speedup.

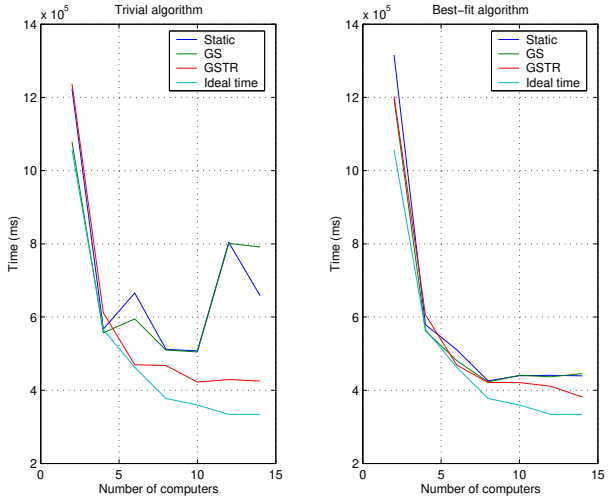

Figure 9: Execution time for Instance 5 of the problem.

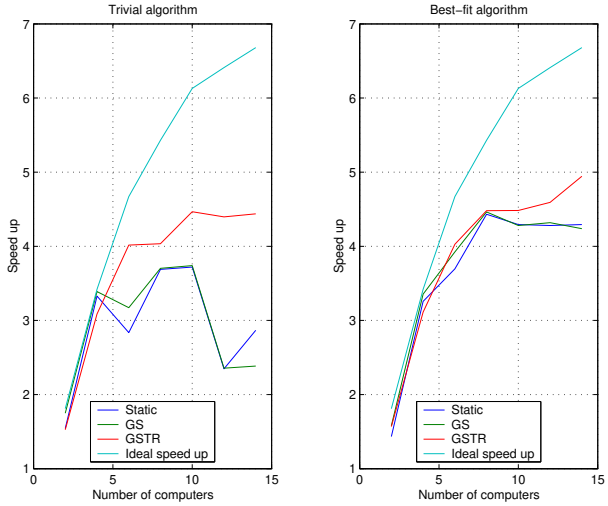

Figure 10: Speedup for Instance 5 of the problem.

Finally, Table 5 presents the mean efficiency values for each technique. These values show GSTR with consistently better results, a measure of how well the algorithm exploit existent resources.

In general the tests indicated that using GSTR with best-fit leads to better results than the other options evaluated. Besides producing lower execution times and behaving more consistently, this algorithm has the unique advantage of being able to adapt to failures and changes in worker performance.

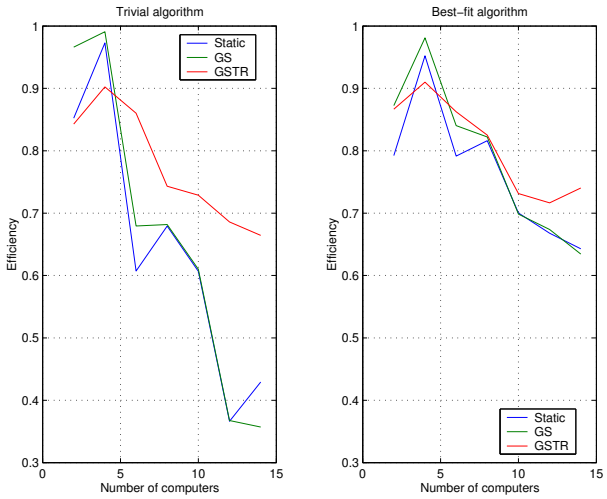

Figure 11: Efficiency for Instance 5 of the problem.

Algorithm	Instance	Scheduler		
		Static	GS	GSTR
Trivial	1	61,70 %	64,12 %	73,33 %
	3	64,30 %	66,29 %	71,18 %
	5	64,40 %	66,47 %	77,53 %
Best-fit	1	74,47 %	75,77 %	74,91 %
	3	75,69 %	77,40 %	77,41 %
	5	76,61 %	78,89 %	80,75 %

Table 5: Mean efficiency values for the six scheduling strategies tested.

5 Conclusions and Future Work

This paper presented Generational Scheduling with Task Replication (GSTR), and adaptive scheduling algorithm specially suited to work in large, heterogeneous and possibly unstable environments like grids. Derived from an algorithm known as Generational Scheduling (GS), GSTR introduces task replication and a combination of static and dynamic scheduling as mechanisms to adapt it to the constantly changing conditions found in grids.

Since GSTR has to be used in conjunction with a static algorithm, two options for task distribution were tested: trivial and best-fit. The results were compared with GS using the same two options and the trivial and best-fit algorithms used alone, for a total of six different scheduling strategies. GSTR, when combined with best-fit, proved to be the approach with better results, specially when the heterogeneity of the system increased. Even when for some cases other approaches produced slightly better results, the fault tolerant and adaptive capabilities of GSTR make this algorithm the best choice.

Future work includes extensive testing of GSTR in larger clusters and including some other popular scheduling strategies in the comparison. A key issue to be studied is the impact of the static-dynamic relation in GSTR. Which portion of the tasks should be scheduled statically and which part must be left to be dynamically scheduled later? Does this relationship vary

with grid performance characteristics? If so, how?

Another issue to be studied is how the extension proposed for GSTR to work with several groups compares with other alternatives, such as splitting the task graph beforehand and making each group execute a subset of the original application independently.

References

[1] Marco Aurlio Amaral Henriques. A proposal for java based massively parallel processing on the web. In *Proceedings of The First Annual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing*, pages 59–66, Rhodes, Greece, June 1999.

[2] D. Fernandez Baca. Allocation modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, SE 15(11):1427–1436, 1989.

[3] Christopher A. Bohn and Gary B. Lamont. Load balancing for heterogeneous clusters of PCs. *Future Generation Computer Systems*, 18:389–400, 2002.

[4] Brent R. Carter, Daniel W. Watson, Freund Richard F., Keith Elaine, Mirabile Francesca, and Howard Jay Siegel. Generational scheduling for dynamic task management in heterogeneous computing systems. *Journal of Information Sciences*, 106:219–236, 1998.

[5] J. H. Conway and Guy R. K. *The Book of Numbers*. Springer-Verlag, 1996.

[6] D.H.J Epenema, M. Livny, R. Van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load sharing among Workstations Clusters. *Journal of Future Generation Computer Systems*, (12):53–65, 1996.

[7] Ian Foster, Carl Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002.

[8] Andrew S. Grimshaw and Wm. A. Wulf. Legion - A View From 50,000 Feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, Agosto 1996. IEEE Computer Society Press.

6.2 Sumário do capítulo

O fato de que o poder de cômputo e a velocidade dos enlaces de rede que formam um grid variem de forma imprevisível dificulta o desenvolvimento de uma estratégia de escalonamento cujo desempenho seja consistentemente bom. Este capítulo apresentou a combinação de escalonamento estático e dinâmico em uma proposta capaz de se adaptar a mudanças de desempenho – e até falhas – nos computadores e enlaces que formam o grid. Esta proposta, chamada GSTR (Generational Scheduling with Task Replication) baseia-se no escalonamento geracional (GS: Generational Scheduling) e introduz a possibilidade de replicar tarefas como mecanismo de recuperação ante falhas ou mudanças bruscas no desempenho

Embora as dificuldades citadas no Capítulo 5 impeçam a condução de testes com um grande número de computadores, os resultados apresentados aqui mostram que o algoritmo proposto tem desempenho superior em sistemas heterogêneos e com alta probabilidade de falhas – como os grids – quando comparado com algoritmos puramente estáticos (trivial, *best-fit*) ou com o algoritmo geracional puro (GS).

A Fault Tolerance Mechanism for Scalable Grids *

Eduardo J. Huerta Yero†, Francisco S. Sambatti‡,
Fabiano de O. Lucchese, and Marco A. A. Henriques
DCA - FEEC - UNICAMP
[huerta,sambatti,flucches,marco]@dca.fee.unicamp.br

Abstract

Grid environments are intended to execute for long periods of time. During these intervals the grid must deal with failures, both in computers executing parallel tasks and in those managing the grid. In any case it is not desirable to lose the work the application completed before the failure. This paper proposes a scalable fault tolerance mechanism useful for grids. The mechanism is based on replication of parallel tasks to deal with failures in the components executing them and on a checkpoint/log/replay mechanism to protect the components that manage the grid. Preliminary results obtained from the current implementation show that the solution proposed provides fault tolerance at a low overhead cost.

1 Introduction

Research efforts have been conducted recently to allow the execution of parallel applications on grid environments. Grids formed by a large number of general purpose computers connected by the Internet offer a low-cost, already deployed processing power that remains unused most of the time.The idea to use this power to solve large instances of parallel applications – that would otherwise remain unsolved – has gained strength with the recent success of projects like SETI@Home [**?**].

Due to their nature, parallel applications will execute on grids for long periods of time before producing the final result. Since grids are dynamic, weakly coupled environments it is highly probable that during this period of time some network and computer failures will occur. Even when the computer does not fail the owner may decide to pull it out of the grid while it is processing, which from the grid software's point of view is the same as a failure. It is the grid's responsibility to guarantee that none of these events prevent the application from progressing. In other words, the grid must be tolerant to failures, guaranteeing the application success in spite of them.

*Work partially supported by grant 98/04305-9 of the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

†On leave from University of Havana, Cuba

‡Western Paraná State University

Building a fault tolerant grid is not an easy task. Fault detection, the starting point of any fault tolerance mechanism, is compromised by the fact that there is no totally reliable scheme to detect if a computer connected to the Internet is working properly [**?**]. Hence, timeout-based systems are the common choice, suppported by the idea that it is better to assume that a component has failed after a period of time and move on than to wait forever for an absolutely correct information [**?**]. As a consequence, solutions based on timeouts have to deal with false positives, i.e., computers considered as failed by the system suddenly coming back to life.

Furthermore, fault tolerance mechanisms usually require a degree of communication among the system components which can have a negative impact on the already slow communications available for the grid. Scalability of the fault tolerance algorithm and the overhead in time and physical storage introduced are also problems whose impact is greater in larger systems like grids.

Common solutions to fault tolerance in distributed systems can be classified in two large groups: replication, on which more than one copy of key system components are maintained, and rollback/recovery, on which the system is rolled back after a failure to a consistent state and restarted from there.

Replication creates several copies – or replicas – of sensitive components in the system so that if one fails the other ones can continue to perform the component's job. The replication scheme must keep this fact transparent to other system entities, all of which should see only one replica of every component. For this to be accomplished all replicas should have their states synchronized. As noted by Alvisi and Marzullo in [**?**] this approach may not be appropriate over slow, unreliable networks like those forming the grid.

Rollback/recovery mechanisms aim at restarting the system after a failure from a consistent state, preferably the one closest to the point of failure to minimize the amount of processing lost. These mechanisms can be divided in two classes: checkpoint/rollback and log/replay [**?**].

Checkpoint/rollback mechanisms depend on the system periodically saving its state to persistent storage. When a failure occurs the system is rolled back to the most recent consistent checkpoint. For a checkpoint to be consistent it has to represent a valid state of the system. When distributed components in a system checkpoint their states independently it is not guaranteed that any set of those checkpoints form a consistent system checkpoint. On the other hand, coordinated algorithms for checkpointing a distributed system tend to be slow and communication-intensive.

Mechanisms based on log/replay assume that all non-deterministic events in the life of a distributed component can be stored and replayed later in the case of a failure. These mechanisms also assume that component execution between two non-deterministic events is deterministic (Piecewise Determinism, or PWD [**?**]). The main problem faced by this approach is the size of the log when the system executes for long periods of time.

This paper proposes a fault tolerance mechanism which leverages the advantages of the techniques commonly used while avoiding their disadvantages. For the parallel tasks executing in the system it is used a simple replication technique. For sensitive system components it is proposed a checkpoint/log/replay strategy which is based on log/replay with the addition of checkpoints to reduce the size of the log.

The rest of the paper is organized as follows. Section 2 formalizes the grid, application and fault models upon which the solution is constructed. Section 3 explains the strategy used to detect failures. Section 4 presents the fault tolerance approach proposed in this work.

Section 5 shows results obtained by an implementation of the mechanism proposed. Section 6 gives an overview of similar approaches found in the literature. Finally, Section 7 presents the conclusions of this work.

2 Models

In order to establish the framework upon which the proposed solution is built this section presents the grid, application and fault models considered.

2.1 Grid model

The grid considered by this approach divides the computers in *groups*, each formed by a *coordinator* and several *workers*. The coordinator manages the workers in its group, synchronizing their efforts to execute parallel applications. Applications are only executed in the workers. A central component called *server* acts as the first point of contact to enter the grid and as the general manager of the groups (Fig. 1).

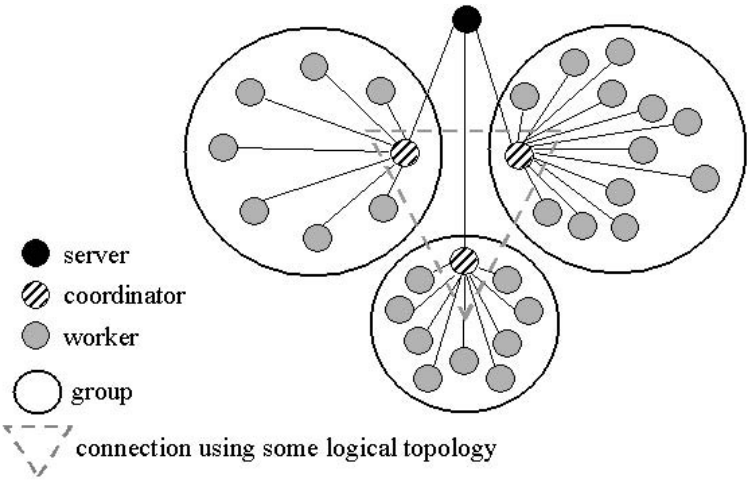

Figure 1: The grid model.

Each component in the system has the same internal structure. They are composed by a *ServiceManager* and a collection of independent *services*. Services are active entities, i.e., each one has its own thread of execution. Interactions between services are monitored by a proxy layer (Fig. 2).

2.2 Application model

The application model is assumed to be a general task graph, with nodes representing the tasks and links representing the data flow and precedence relationships (Fig. 3).

Tasks execute in a sandbox. They do not produce any collateral effects (e.g. sending messages, creating a file) while executing.

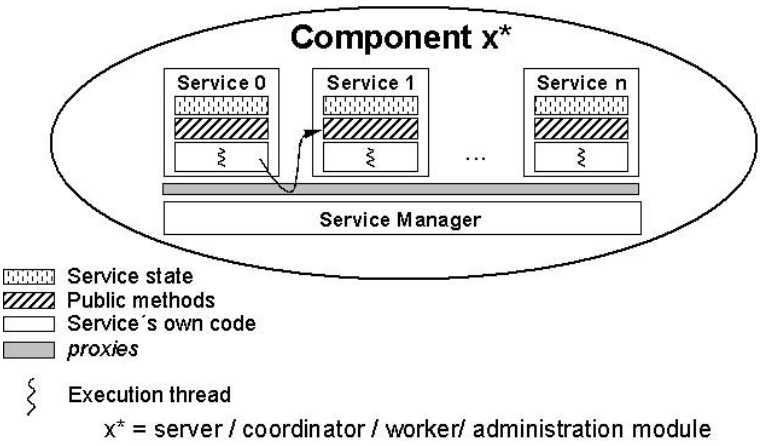

Figure 2: The internal structure of each component.

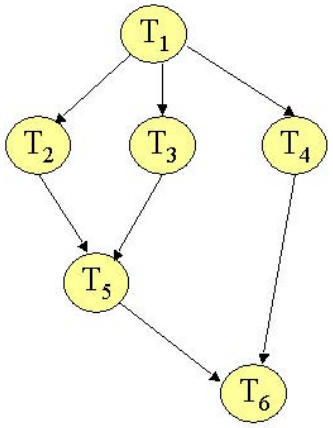

Figure 3: An example of a task graph.

2.3 Fault model

Failures are assumed to follow the *fail-stop* model, that is, it is not considered the case where a malfunctioning component keeps executing. Once a failure occurs the component is assumed to halt. Coordinators and server are assumed to have a reliable secondary storage protected by a separate fault tolerance mechanism.

3 Fault detection

As discussed in the Introduction it is not possible to implement a reliable fault detector over the Internet. The approach used here is commonly found in distributed monitoring systems like the Globus Heartbeat Monitor [**?**]. Each component sends a *heartbeat* to its manager (e.g. workers to coordinator, coordinators to server) in a regular rythm. If the receiving end stops listening to the heartbeats of any component under its control for a timeout period it assumes

that the component failed. There are two issues that must be carefully studied in this approach: (a) how to specify the timeout period in order to minimize the number of false positives and (b) what to do when a false positive occurs.

Determining the timeout is tricky. If it is too short it will produce a large number of false positives. If it is too long it will compromise the usefulness of the fault detector. A study presented by Stelling et. al.[**?**], in an environment similar to those found in grids, showed that the rate of false positives for a timeout of 240 seconds with heartbeats every 10 seconds was less than 0.001%. Although these results seem good enough the timeout period should be adjusted experimentally according to the characteristics of the grid. With the continuous improvements in network technology the timeout period is expected to reduce without losing accuracy.

Even with the best possible timeout there is still a chance that a false positive occurs. In the solution proposed here a unique *ticket* is handled to every component upon entering the system. If that component is considered failed at some point the ticket is invalidated. Any attempt made by the component to reinitiate contact with the system will result on a refusal from the system, hence forcing the component to reenter as if it were arriving for the first time. Although this technique is admittedly pessimistic, it is extremely simple to implement and it has proven effective in experiments conducted on a long-running grid.

4 Fault tolerance

The proposal presented here divides the possible failures in the grid in two sets: failures in the workers and failures in the coordinators and server. The reason for this division lies in the fact that the characteristics of these components are different. Workers exist in large numbers and their failures can be tolerated by the system, since they do not execute any vital role. Furthermore, their availability is uncertain: workers may come and go randomly, so no solution should be constructed relying on access to their secondary storages.

Coordinators and server, on the other hand, are limited in number. Even a large grid is expected to have a few dozen coordinators and a server. These components are assumed to execute on more reliable computers, on which access to secondary storage is possible. They perform vital administrative tasks, so their failure will have a strong impact in the system, particularly if the status information they hold is lost.

In this section we explore the fault tolerance mechanism proposed in this work for both sets of failures.

4.1 Fault tolerance for workers

The only responsibility of a worker is to execute parallel tasks. When they fail the tasks they are executing are lost. Fault tolerance techniques that depend on secondary storage are discarded, following the assumption that workers are not dependable components, so other approach is needed.

The solution proposed here uses replication for parallel tasks. When a worker takes too long to produce an answer the tasks it was executing are replicated in other workers. This scheme helps both when a worker fails and when it is slowed down by a sudden increase in its workload. In any case the tasks are replicated and the application continues to progress.

Notice that replicating parallel tasks does not require further efforts to synchronize states since each task executes in a sandbox with no contact with the execution environment. The only care that must be taken by the coordinator is to discard the results received from a replicated task when one of its replicas already finished. Notice also that with a simple extension this mechanism may be used to detect Byzantine failures, on which a task finishes but produce incorrect results. The only modification needed is to compare the results of the replicas instead of discarding the ones arriving late.

4.2 Fault tolerance for coordinators and server

The fault tolerance mechanism proposed for coordinators and server is an extension of the log/replay mechanism. Before presenting the extension let us introduce how log/replay is used in the context under study.

Since services are assumed to be piecewise deterministic the only difference between the original execution and a reexecution come from its interactions with other services. These interactions can take the form of invoking a method provided by (or processing an invocation made by) another service.

The log/replay mechanism works as follows. Every method invocation made between services is logged by the proxy layer in stable storage before completion (Fig. 4a). When a failure occurs and a service is reexecuted it will make the same invocations; however *the proxy layer will return the values stored in the log instead of actually reexecuting the call* (Fig. 4b). This mechanism, together with the fact that services are PWD, guarantees that every service reaches the same state it had before the failure. The recovery phase is called *replay*, since inter-service method invocations are forced to produce the same results.

The code segment in Fig. 5 illustrates the information stored in the log as a service progresses.

The status of the log in points `A` and `B` during the first iteration of the `while()` instruction is shown in Table 1.

Point	Log contents
A	- Result for `comm.receiveMessage(...)` - Result for `tidm.createAppNumber(...)`
B	- Result for `comm.receiveMessage(...)` - Result for `tidm.createAppNumber(...)` - Result for `comm.sendMessage(...)` - Result for `comm.receiveMessage(...)`

Table 1: Log status in points `A` and `B`

If a failure occurs in point `B` during the first iteration of the `while(...)` none of the service calls registered in the log will be reexecuted. The first actual service call that will be made is the `comm.sendMessage(...)` following point `B`.

The log/replay mechanism is simple, transparent to the services and guarantees full recovery. However, the main drawback of this scheme makes it unusable in environments with long expected execution times like grids. The size of the log will increase over time until it becomes

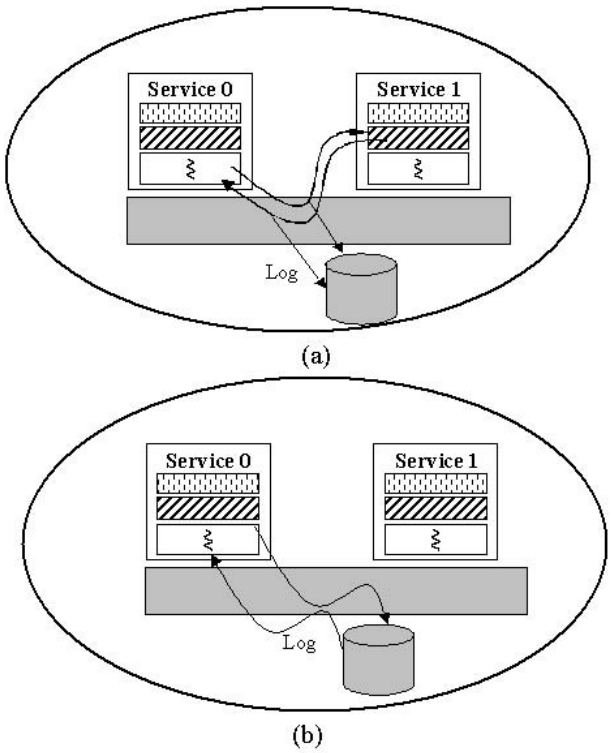

Figure 4: The log/replay mechanism. (a) A method call is logged with its result. (b) During recovery the result is obtained from the log, instead of reexecuting the call.

unmanageable. As it can be noted from Table 1 the log entries in point `B` contain those in point `A` plus a few more. As the service progresses the log will continue to grow.

Another problem with log/replay is derived from the fact that instructions in the services must be actually reexecuted during recovery. If a service was running for several months all the instructions it executed will be repeated, with the exception of the calls it made to other services. As a consequence the recovery phase may take an excessively long time to complete.

Resuming, log/replay alone is not enough to provide fault tolerance for a grid. An extension is needed to shorten the size of the log and speed up the recovery phase.

4.3 Saving the service state

The main problem with the log/replay solution is that since services are restarted from their initial status no entry can be removed from the log. If the state of the service is saved periodically and the service is restarted after a failure from its last saved state then some log entries will not be needed. Depending on the strategy used to discard entries the size of the log can be bounded. In the code sample presented in Fig. 5 all log entries could be discarded if the state is saved after point `B`.

The main problem with this approach is that there is no universal mechanism to save the state of a software component and restart it from the same point it was before the failure. Platform specific approaches – such as using a core dump of the process – are not extensible to heterogeneous environments like grids. Platform independent approaches like Java do not

```
//Service that provides message-based
// communications
Communicator comm = ...

...  //Service that manages application numbers
TIDManager tidm = ...
...
Message m = comm.receiveMessage(...);
...
long appNumber = tidm.createAppNumber(...);
...
while(...)
...  <---------------------- A
comm.sendMessage(...);
...
m = comm.receiveMessage(...);
...  <---------------------- B
comm.sendMessage(...);
...
```

Figure 5: A code fragment to illustrate the log/replay mechanism.

provide checkpointing mechanisms for the virtual machine.

The same problem was faced by the Object Management Group (OMG) when defining CORBA [**?**], a standard for developing multi-platform, multi-language distributed object systems. The solution adopted by CORBA leaves the responsibility of saving and restoring the state to the objects, providing only a set of standard interfaces. Likewise, the solution presented here relies on the services to save and restore state, offering interfaces similar to those in CORBA. These interfaces are presented in Fig. 6.

The `Checkpointable` interface supports full checkpoints, while the `Updateable` interface supports incremental checkpoints. The service must choose between them based on the size of the checkpoint and the frequence on which the state is saved, among other factors. The functions in the interfaces will be invoked by the `ServiceManager` (Fig. 2) when the state of the service must be saved or restored.

4.4 Reexecution

Since it is not possible to restart service execution from an arbitrary point in its code it is imperative to structure the code so that some parts of it are not executed during recovery. The proposal made in this work divides service code in *reexecution blocks*, each one initiated with a `begin()` call and ended either with `end()` or `endSaveState()`. These calls are provided by the `StateManager` service, whose interface is presented in Fig. 7.

Reexecution blocks can be nested. Each block has an associated *level*, an integer number

```
public interface Checkpointable {
  Serializable getInitialState();
  Serializable getState();
  void setState(Serializable state);
}

public interface Updateable{
  Serializable getUpdate();
  void setUpdate(Serializable update);
}
```

Figure 6: The `Checkpointable` and `Updateable` interfaces implemented by services to save and restore their states.

```
public interface StateManager{
  void begin();
  void end();
  boolean endSaveState();
  boolean saveStateAndReturn(Serializable returnValue);
}
```

Figure 7: The StateManager service interface.

that increases from outer to inner blocks. Instructions outside all blocks are considered to belong to level 0. Figure 8 illustrates these concepts.

At the beginning of each block a new section is created in the log file to store service interactions within the block. The new section is identified with the level number. At the end of the block all entries of the level are discarded. As an example consider the code segment shown in Fig. 9 and the corresponding logs in points `A`, `B` and `C` shown in Table 2.

The status of a service is not only affected by the calls it makes to other services but by the invocations it receives as well. If processing a request changes the internal state of a service it should use the `saveStateAndReturn()` primitive provided by the `StateManager` (Fig. 7). A simple example is presented in Fig. 10, which shows the `createAppNumber()` method provided by the `TIDManager` service.

It is worth noticing that under this scheme the service programmer is responsible for avoiding the reexecution of a block once its execution ends. A simple yet effective solution based on a boolean condition is presented in Fig. 11.

This example finishes the presentation of the checkpoint/log/replay mechanism proposed in

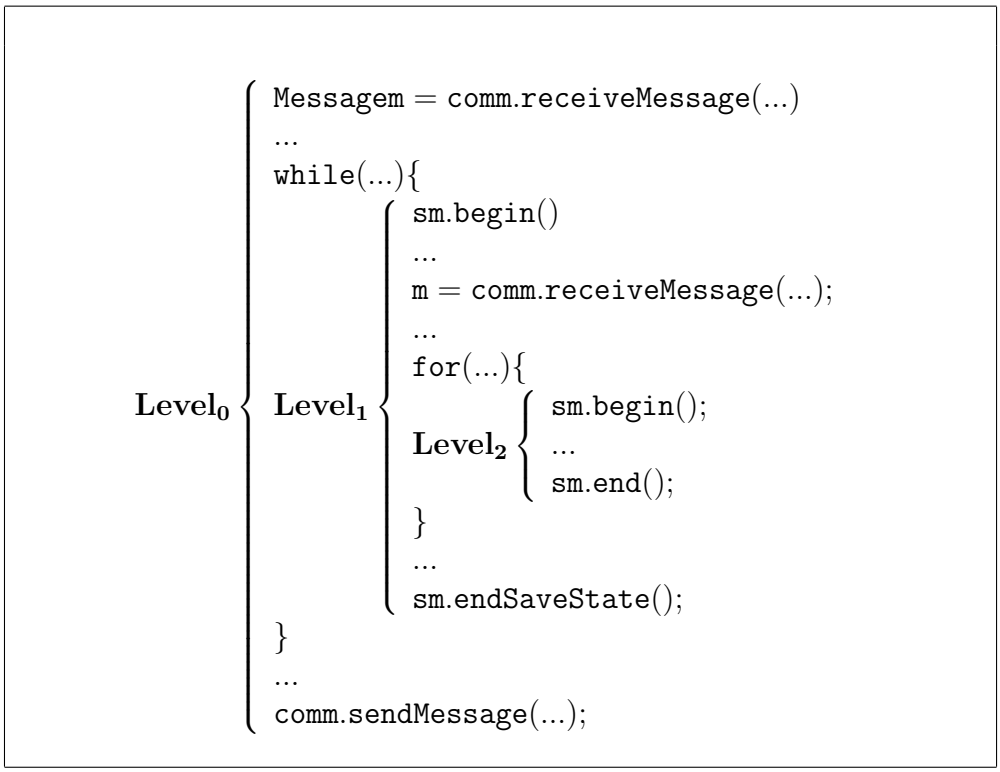

Figure 8: Nesting of reexecution blocks.

Point	Level	Contents
A	0	- comm.receiveMessage(...) - tidm.createAppNumber(...)
	1	- comm.sendMessage(...)
B	0	- comm.receiveMessage(...) - tidm.createAppNumber(...)
	1	- comm.sendMessage(...)
	2	- comm.receiveMessage(...)
C	0	- comm.receiveMessage(...) - tidm.createAppNumber(...)

Table 2: Log status in points A, B and C of Fig. 9.

this paper to recover from failures in grids. The next section presents results obtained from its implementation.

5 Results

The checkpoint/log/replay mechanism proposed here was implementated in a system called JoiN[?], a Java-based grid developed by the authors. A set of tests was conducted to evaluate the overhead imposed by the logging mechanism, which is the most invasive technique used in

```
StateManager sm = ...
Communicator comm = ...
TIDManager tidm = ...
...
Message m = comm.receiveMessage(...);
...
long appNumber = tidm.createAppNumber(...);
...
while(...){
   sm.begin();
   ...
   comm.sendMessage(...);
   ...  <------------------------- A
   sm.begin();
   ...
   m = comm.receiveMessage(...);
   ...  <------------------------- B
   sm.endSaveState();
   ...
   comm.sendMessage(...);
   ...
   sm.endSaveState();
   ...  <------------------------- C
}
...
```

Figure 9: A code segment to illustrate log status in different execution points.

```
long createAppNumber(...){
// determines the first free application number
// and reserves it, thus altering the state
// of the service
...
sm.saveStateAndReturn(nextAppNumber);
return nextAppNumber;
}
```

Figure 10: An example of the use of `saveStateAndReturn`.

the solution proposed. The impact of logging is the only one that can be accurately measured,

```
X = true; // allows execution first time
...
while (...){
  sm.begin();
  ...
  if(X) {
    sm.begin();
    ...
    X = false; // avoids reexecution
    sm.endSaveState();
  }
  ...
  X = true; // allows execution in next iteration
  sm.endSaveState();
}
```

Figure 11: Avoiding reexecution of a block.

since the overhead generated by storing the state of the service largely depends on how the service programmer uses the mechanism. Frequent full checkpoints will generate a large overhead, while unfrequent incremental checkpoints will not impact performance as much.

The test environment was formed by 8 computers, ranging from a 360 MHz Sun Ultra 60 workstation with 512 Mb of RAM to a 110 MHz Sun SparcStation 4 with 32 Mb of RAM. The computers were connected by a 10 Mbps Ethernet network. The grid was formed by a server, a coordinator and 6 workers. To measure the impact of logging we executed a parallel application that searches for large prime numbers. The application was formed by 17 parallel tasks. The first task divided the search interval in 15 parts, which are delivered to 15 tasks that perform the search on each subinterval. Each of these tasks handle their results to a final task that composes the output of the application. The application was executed 5 times with and without logging. The results are shown in Table 3.

Execution	Without Logging	With Logging	Overhead (%)
1	98612	100550	1.97
2	85160	88784	4.26
3	85604	86504	1.06
4	85063	88000	3.44
5	85307	86796	1.75
Mean	87949	90126	2.48

Table 3: Execution times of the application with and without logging.

The mean overhead imposed by logging was $2,48\%$, which is rather small. Even considering that the actual overhead is larger when the state of the service is stored these results indicate that a careful implementation of this mechanism can provide fault tolerance for a grid at an acceptable overhead cost.

A simple extension may further improve the results obtained here. During the tests we identified that 14% of the method calls in the server and 38% in the coordinator were idempotent, i.e., would return the same result if called N times. Most of these operations had short execution times so allowing them to reexecute instead of logging their results to replay them later would introduce a performance gain. We are currently working to introduce this extension to the proposal.

6 Related work

Several approaches to fault tolerance for parallel distributed systems can be found in the literature. They are mainly based on replication, checkpointing, message logging, transactions or a combination of the above.

MIST [**?**] uses coordinated checkpointing to add fault tolerance and allow task migration in PVM [**?**]. Although useful for small systems, coordinated checkpointing is not a scalable technique and thus it is not suitable for grids.

DOME [**?**] is a C++ library that provides explicit checkpointing support for PVM. DOME forces the programmer to be aware of the mechanism in order to minimize the size of the checkpoint.

Li and Tsay [**?**] propose a checkpointing technique for MPI [**?**] that uses coordinated checkpoints for tasks in the same computer and independent checkpointing with message logging for tasks in different computers.

Charlotte [**?**] is a Java-based grid that uses task replication to obtain fault tolerance and load balancing. A distributed shared memory mechanism provides the consistency needed for replication. The weak point of the solution is the existence of a manager that constitutes a single point of failure.

XtremWeb [**?**] is an Internet-based framework for parallel computing. It uses task replication for load balancing and fault tolerance. Fault tolerance in the server is obtained using transactional database support.

SETI@Home [**?**] uses task replication to implement fault tolerant applications. Servers are also replicated to allow for failures. Nguyen-Tuong et al. [**?**] also propose a technique based on replication of idempotent tasks.

In general the solutions found in the literature are either incomplete (do not support failures in the manager), not scalable (coordinated checkpointing, single manager) or need extra support (transactional database system). The solution presented here is scalable, self-sufficient and protects all components in the system.

7 Conclusions

This paper introduced a fault tolerance mechanism for grids based on replication for parallel tasks and checkpoint/log/replay for sensitive system components. The replication of parallel

tasks allows for failures in worker components - those that execute the applications – to be masked by executing the same task in several workers. This process is simplified by the fact that parallel tasks are assumed to execute in a sandbox, producing no side-effects such as creating files or sending messages.

Coordinators and the server – the components that manage the grid according to the model proposed – use a checkpoint/log/replay technique to recover from failures. The technique is an extension of the log/replay, on which all interactions among services – the basic elements composing any grid component – are recorded to be replayed in the case of a reexecution. Since services are assumed to be piecewise deterministic this technique guarantees a safe recovery. Checkpointing is introduced to reduce the size of the log and to shorten the recovery time, since services are restarted from the last saved state instead of from the beginning.

The paper presented results obtained from JoiN, a grid platform based on Java developed by the authors. The overhead introduced by logging service interactions was found to be small, which indicates that this mechanism may be an interesting solution for the fault tolerance issue in other grids as well.

7.2 Sumário do capítulo

Este capítulo mostrou uma solução ao problema de falhas no servidor e coordenadores. A solução, baseada em técnicas de log/replay e checkpoint/rollback, permite armazenar o estado de cada um dos serviços que constituem o servidor e os coordenadores de forma tal que eles possam ser recuperados após a ocorrência de uma falha. O mecanismo de log/replay, junto à premissa de que os serviços são *piecewise-deterministic*, permite recuperar um serviço garantindo que sua reexecução após uma falha seja uma réplica exata da execução original. Para atingir este objetivo o mecanismo armazena no log o resultado de todas as interações entre serviços, o que permite reproduzi-las em uma reexecução. Como entre quaisquer duas interações os serviços são determinísticos, este mecanismo garante que um serviço atingirá na reexecução o mesmo estado que tinha antes da falha.

O mecanismo de checkpoint/rollback permite controlar o tempo necessário para recuperar um serviço, controlando o tamanho do arquivo de log. A plataforma armazena periodicamente o estado de cada serviço e utiliza o log apenas para registrar as interações realizadas pelo serviço entre um checkpoint e outro. Assim, o processo de recuperação após uma falha consiste em carregar o último estado salvo e reexecutar o serviço a partir dele, utilizando para as interações entre serviços os resultados armazenados no log até que ele seja esvaziado.

Os testes conduzidos apresentaram resultados satisfatórios relacionados com a robustez e a sobrecarga imposta pelo mecanismo durante o funcionamento normal do sistema.

Cabe destacar que, por faltar em Java mecanismos que suportem checkpoints em sua máquina virtual, a solução proposta depende da colaboração dos programadores de serviços da plataforma. Isto implica que o uso incorreto destas técnicas por um dos serviços pode deixar o sistema vulnerável a falhas. Embora esta característica não seja desejável, acreditamos que seu impacto não é grande, uma vez que a plataforma é implementada por pessoal especializado. Por outro lado, se algum mecanismo de checkpoint fosse disponibilizado por Java no futuro – o que acreditamos acontecerá mais cedo ou mais tarde – a solução proposta poderá ser facilmente adaptada e esta desvantagem desaparecerá.

Com este capítulo encerra-se a Parte III, dedicada a mostrar uma proposta de projeto e implementação de grid. A Parte IV será dedicada a apresentar as conclusões deste trabalho e as direções futuras para as quais podem ser encaminhadas as pesquisas em grids.

Parte IV

Conclusões

Capítulo 8

Conclusões e trabalhos futuros

As pesquisas na área de grids estão apenas começando. Embora seja difícil estabelecer o ponto exato de início, os primeiros trabalhos importantes datam de 1988, quando o projeto Condor começou a ser implementado. As intensas pesquisas desenvolvidas neste curto período de tempo já mostram seus resultados. Sistemas que implementam grids começam a ser conhecidos pelo público e aos poucos passam a formar parte da vida digital cotidiana. Os pesquisadores de áreas com grandes necessidades de processamento estão começando a perceber o enorme potencial dos grids e orientando suas demandas nesta direção.

O objetivo primário desta tese foi contribuir com o desenvolvimento de grids mais eficientes e confiáveis. Para tanto, encaminhamos nossas pesquisas em duas direções: a fundamentação teórica necessária para a computação paralela em grids e o projeto e implementação de uma plataforma de grid onde muitas propostas poderiam ser avaliadas.

8.1 Fundamentação teórica

O Capítulo 3 propôs um mecanismo para modelar a execução de aplicações paralelas em grids. O modelo proposto tem a vantagem de ser estático e, portanto, sua avaliação é rápida quando comparada a abordagens baseadas em simulações. Mesmo sendo estático o modelo é capaz de representar características dinâmicas do sistema, tais como a contenção pelo uso simultâneo de recursos.

O Capítulo 4 apresentou uma análise detalhada da execução de uma aplicação Mestre-Escravo em um grid. Usando uma expressão analítica para o tempo de execução da aplicação (implicitamente derivada dos trabalhos apresentados no Capítulo 3) , o Capítulo 4 estudou as características de desempenho de tais aplicações. Dentre as características estudadas estão os limites para o ganho em velocidade (*speedup*) e as condições necessárias e suficientes para uma aplicação Mestre-Escravo ser escalável quando executada em um grid. Ainda, o capítulo estudou o impacto da contenção pelo uso da rede nestas medidas de desempenho.

O estudo concluiu que é possível executar aplicações Mestre-Escravo de forma escalável em grids arbitrariamente grandes desde que tanto a parte seqüencial da aplicação quanto a sobrecarga imposta pela comunicação entre tarefas sejam de ordem menor que a parte paralelizável da aplicação.

8.2 Implementação de um grid

Como contribuição prática a tese propôs no Capítulo 5 uma implementação de um grid chamado JoiN. Totalmente baseada em Java, a abordagem proposta em JoiN tem em sua simplicidade um de seus pontos fortes. O projeto garante ao sistema flexibilidade suficiente para introduzir novas funcionalidades sem interferir com as existentes mediante a estruturação da plataforma como um conjunto de serviços independentes. Esta arquitetura, que foi instanciada em JoiN com um conjunto particular de serviços, é uma das contribuições do trabalho, visto que ela pode ser implementada por outros sistemas. Serviços para comunicação, segurança, gerenciamento de aplicações, atualização do código da plataforma, tolerância a falhas e entrada-saída em arquivos, entre outros, encontram-se já operacionais ou em fase final de testes.

Por serem de vital importância para o sucesso de um grid, a tese destacou as propostas feitas para o escalonamento de tarefas e tolerância a falhas. O Capítulo 6 propôs um algoritmo de escalonamento capaz de lidar eficientemente com as constantes variações de disponibilidade de recursos em um grid e com as eventuais falhas de computadores durante a execução de uma aplicação. O algoritmo caracteriza-se por sua simplicidade e pelos bons resultados obtidos, particularmente quando a heterogeneidade do sistema testado foi maior.

O Capítulo 7 apresentou uma nova abordagem ao problema de tolerância a falhas em um grid. A abordagem combina a replicação de tarefas da aplicação paralela com o uso de *logs* e *checkpoints* para proteger o estado do grid. Com o uso adequado destes mecanismos pelo programador de novos serviços para o sistema é possível garantir que um coordenador ou o servidor se recuperem de falhas sem perder o trabalho feito pelas aplicações que estão sendo executadas. Testes preliminares indicam que o mecanismo proposto não introduz uma sobrecarga significativa ao funcionamento normal do sistema.

Apesar de que ainda resta muito trabalho a ser feito, acreditamos que os resultados apresentados aqui incrementam os conhecimentos existentes sobre grids, tanto do ponto de vista teórico quanto prático. As soluções propostas consolidam o início de um esforço de pesquisa em grids. Os avanços obtidos até o momento indicam direções para as quais as pesquisas futuras podem ser encaminhadas. A seção seguinte mostra algumas delas.

8.3 Trabalhos Futuros

Por ser uma área nova existem ainda muitos aspectos a serem pesquisados na computação paralela em grids, tanto técnicos quanto humanos. A falta de resultados sólidos em várias áreas vitais é atualmente o limitante fundamental para a popularização dos grids.

Desde o ponto de vista técnico o principal problema que enfrentam hoje os pesquisadores é a falta de um modelo matemático consistente para os recursos que formam o grid. É necessário desenvolver análises estatísticas detalhadas que produzam um perfil consistente do comportamento dos recursos que conformam um grid. Alguns aspectos cuja compreensão é ainda insuficiente são listados a seguir.

Poder computacional disponível. Um grid que tenha informações precisas sobre o poder computacional disponível no momento e o previsto para o futuro poderá tomar decisões de escalonamento mais eficientes. No entanto, mesmo as ferramentas mais invasivas não

conseguem disponibilizar ao escalonador informações de qualidade quando o número de computadores aumenta durante o processamento.

Padrão de uso dos computadores. Relacionado com o item anterior, procura modelar o comportamento de um colaborador que participa no grid. Quando ele disponibiliza seu computador? Por quanto tempo? Enquanto o computador participa no grid, quais outras atividades ele desenvolve?

Topologia e desempenho da rede. Dado um grid formado por computadores espalhados pela Internet é necessário obter uma visão a mais exata possível dos enlaces de rede que os unem, tanto no referente à topologia física das conexões quanto ao desempenho das mesmas. Esta informação pode ser usada para estruturar o grid da forma mais eficiente e tolerante a falhas possível.

Padrão de uso da rede. Para prever o desempenho dos enlaces de rede no futuro é necessário modelar da forma mais exata possível os padrões de uso dos mesmos, desde as redes locais até a Internet.

Padrões de falhas. As informações sobre as falhas, tanto de computadores quanto de enlaces de redes, são importantes para o mecanismo de tolerância a falhas e para o escalonador, que podem usá-las para tomar providências mais eficazes ante sua ocorrência.

Com uma melhor compreensão destes aspectos será possível desenvolver algoritmos melhores para escalonamento e tolerância a falhas em um grid. Ainda, será possível entender melhor quais aplicações podem tirar melhor proveito deste tipo de plataforma. A extensão do tipo de aplicações executadas em um grid de Mestre-Escravo para estruturas mais gerais colaborará também com a expansão desta abordagem.

Dentre os aspectos mais práticos destaca-se a necessidade de produzir mais e melhores softwares. Escalabilidade, robustez e eficiência são campos nos quais ainda existe muito caminho a percorrer. É necessário estudar quais são as arquiteturas de software mais apropriadas para um grid. As interfaces de programação usadas devem ser padronizadas. É preciso desenvolver ferramentas que auxiliem na programação e teste de grids. É necessário mapear estruturas para concorrência em linguagens de alto nível com implementações práticas em grids, de forma tal que seja possível programar aplicações usando estas linguagens ao invés de usar diretamente os APIs fornecidos pelos grids.

Na arquitetura do sistema devem ser dedicados esforços especiais para determinar a melhor topologia lógica de interconexão de componentes. É preciso estudar também, dado um conjunto grande de computadores, como distribuí-los em subconjuntos gerenciáveis (ou grupos, na terminologia usada na tese) de forma a maximizar a eficiência e a tolerância a falhas do grid.

Desenvolvimentos recentes em áreas como servidores de aplicações (e.g. implementações do padrão J2EE — Java 2 Platform, Enterprise Edition) e serviços Web fornecem uma infraestrutura flexível, robusta e eficiente para a implementação de aplicações distribuídas complexas. Estas tecnologias podem e devem ser exploradas por implementações comerciais de grids, como mostram os esforços iniciais conduzidos pela Globus Alliance na definição da arquitetura padrão OGSA (Open Grid Service Architecture). A arquitetura OGSA define o conceito de *serviço grid* como extensão de um serviço Web para introduzir facilidades necessárias para um grid. Esta abordagem está sendo rapidamente adotada pelas maiores empresas e centros de pesquisa

na area de grids e já é praticamente um padrão de fato. Portanto, impõe-se estudar meios de estender as propostas deste trabalho de forma a atender os padrões existentes e assim aumentar sua utilidade.

Finalmente, é necessário estudar melhor o perfil dos donos de computadores e determinar a melhor forma de motivá-los a participar em um grid. Por terem características diferentes, os grids devem ser entendidos antes de serem aceitos pelos colaboradores. Apesar do estímulo econômico ter um potencial de sucesso grande, as preocupações com segurança e com a falta de privacidade dos computadores conectados à Internet são barreiras a serem vencidas antes que o processamento em grid em larga escala torne-se uma realidade.

Acreditamos, no entanto, que já existem as condições necessárias para se obter resultados importantes usando grids no futuro próximo e que estes grids ainda influenciarão muito no desenvolvimento da ciência e da tecnologia, trazendo benefícios concretos para a as áreas científica e tecnológica.

Apêndice A

Exemplo de execução de JoiN

Nos capítulos pertencentes à Parte III desta tese descrevemos o projeto e implementação de JoiN, um sistema baseado em Java que permite formar um grid usando computadores espalhados pela Internet. Neste capítulo apresentaremos um conjunto de passos para formar um grid de teste, que terá um servidor, um coordenador e 2 trabalhadores. Mostraremos também como utilizar este grid para instalar e executar uma aplicação paralela.

Os passos a serem seguidos são listados a seguir.

- Executar o servidor.
- Executar o coordenador.
- Executar os trabalhadores.
- Instalar a aplicação.
- Submeter a aplicação.

Nas seguintes seções apresentaremos cada passo com mais detalhes.

A.1 Pré-requisitos

JoiN não exige a configuração de variáveis de ambiente para ser executado. Basta que cada computador tenha instalado o Java Runtime Environment (JRE) 1.4 ou superior e uma cópia do código da plataforma. Esta cópia precisa ser feita apenas uma vez. Modificações subseqüentes no código da plataforma serão feitas automaticamente no momento de entrada do computador no sistema.

A.2 Execução

JoiN é executado através do arquivo JAR executável `join.jar` que se encontra no diretório `bin` da instalação. Todos os comandos apresentados a seguir pressupõem que `JOIN_HOME/bin` é o diretório atual (substitua `JOIN_HOME` pelo diretório em que o código foi instalado).

O formato básico da instrução é:

```
java -jar join.jar -ct type [-cc] [-sa address]
```

onde os parâmetros `pars` podem ser:

-ct *type* Indica o tipo de componente que será executado (ct: component type). O argumento *type* pode ser `server`, `col` ou `jack`, indicando se o computador agirá como servidor, colaborador (que pode ser coordenadores ou trabalhador) ou módulo de administração.

-cc Para o caso em que se escolha `-ct col` é possível indicar se o computador pode ou não agir como coordenador (cc: can coordinate). Se for necessário, o sistema usará um dos componentes executados com este parâmetro como coordenador.

-sa *address* Este parâmetro indica o endereço do servidor (sa: server address) a ser contatado. Ele não é necessário para o caso em que se escolha `-ct server`.

A seguir mostra-se um exemplo prático.

Servidor : `buzios.dca.fee.unicamp.br`

Coordenador : `grumari.dca.fee.unicamp.br`

Trabalhador 1 : `itapua.dca.fee.unicamp.br`

Trabalhador 2 : `dunas.dca.fee.unicamp.br`

JACK : `laguna.dca.fee.unicamp.br`

A.2.1 Execução do servidor

Para iniciar o servidor basta executar na máquina `buzios.dca.fee.unicamp.br` o comando a seguir.

```
cd JOIN_HOME/bin
java -jar join.jar -ct server
```

Isto produzirá uma série de mensagens na tela indicando o progresso da inicialização. Uma parte delas é mostrada na Fig. A.1

A.2.2 Execução de um coordenador

O comando para executar um coordenador é

```
cd JOIN_HOME/bin
java -jar join.jar -ct col -cc -sa buzios.dca.fee.unicamp.br
```

e deve ser executado em `grumari.dca.fee.unicamp.br`. O resultado pode ser visto na Fig. A.2.

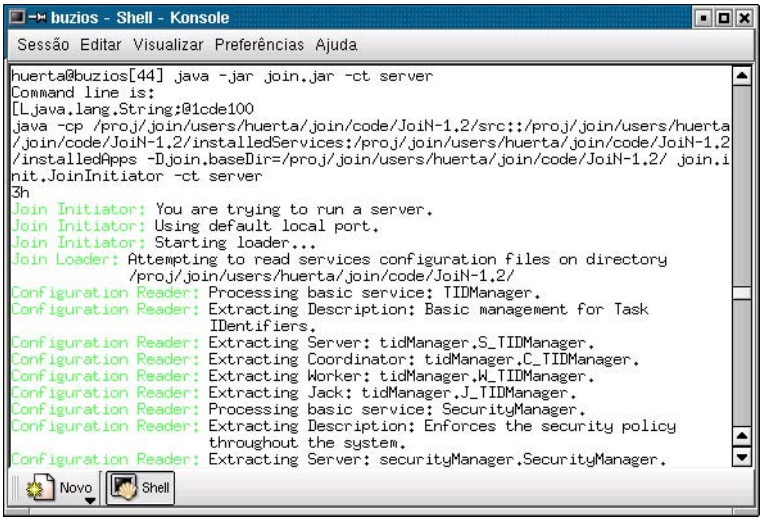

Figura A.1: Execução de um servidor.

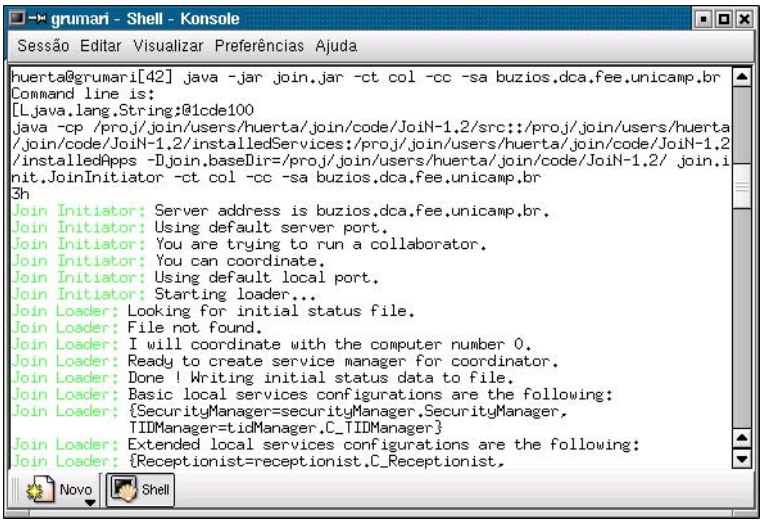

Figura A.2: Execução de um coordenador.

A.2.3 Execução dos trabalhadores

Para executar um trabalhador basta digitar o comando a seguir.

```
cd JOIN_HOME/bin
java -jar join.jar -ct col -sa buzios.dca.fee.unicamp.br
```

Os resultados obtidos após executar o comando em `dunas` e `itapua` são mostrados nas Figs.A.3 e A.4 respectivamente.

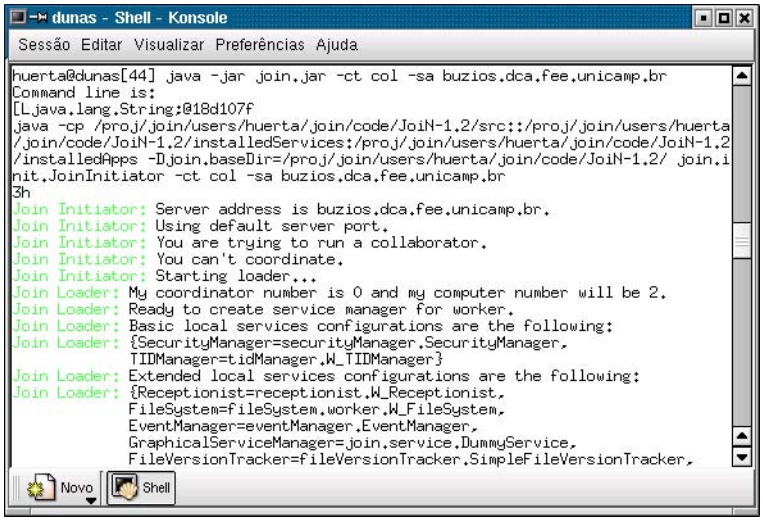

Figura A.3: Execução de um trabalhador em `dunas`.

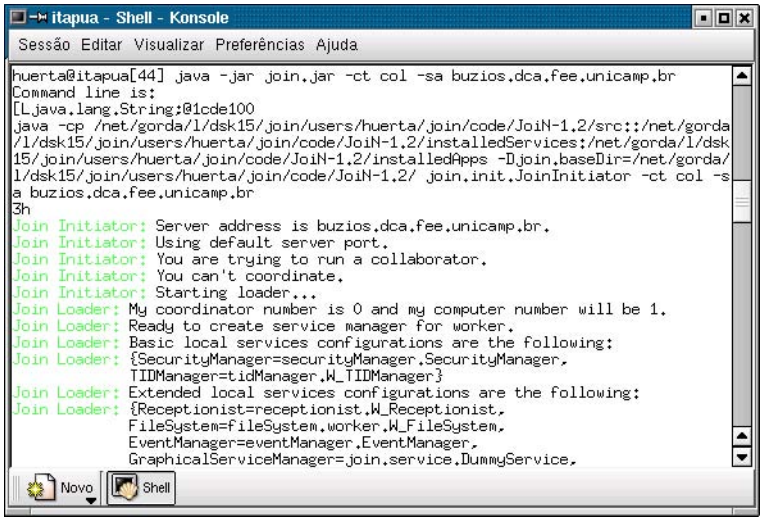

Figura A.4: Execução de um trabalhador em `dunas`.

A.2.4 Execução de um JACK

Para executar o JACK na máquina `laguna` digitou-se o comando abaixo.

```
cd JOIN_HOME/bin
java -jar join.jar -ct jack -sa buzios.dca.fee.unicamp.br
```

Como resultado o terminal mostra algumas mensagens indicando o progresso (Fig. A.5).
No entanto, o resultado mais importante da execução de um JACK é a apresentação de uma interface gráfica com o usuário, mostrada na Fig. A.6.

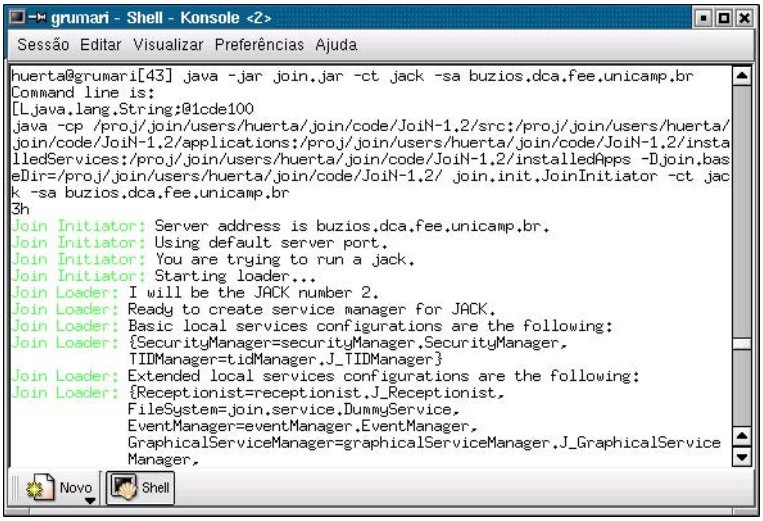

Figura A.5: Execução de um JACK em `laguna`.

Figura A.6: Interface gráfica do JACK.

A.2.5 Instalação e execução de aplicações

Para instalar e executar (submeter) aplicações é necessário usar a interface gráfica associada ao serviço `ApplicationManager`. Clicando duas vezes no item correspondente (ver Fig.A.6) será

aberta a janela mostrada na Fig. A.7.

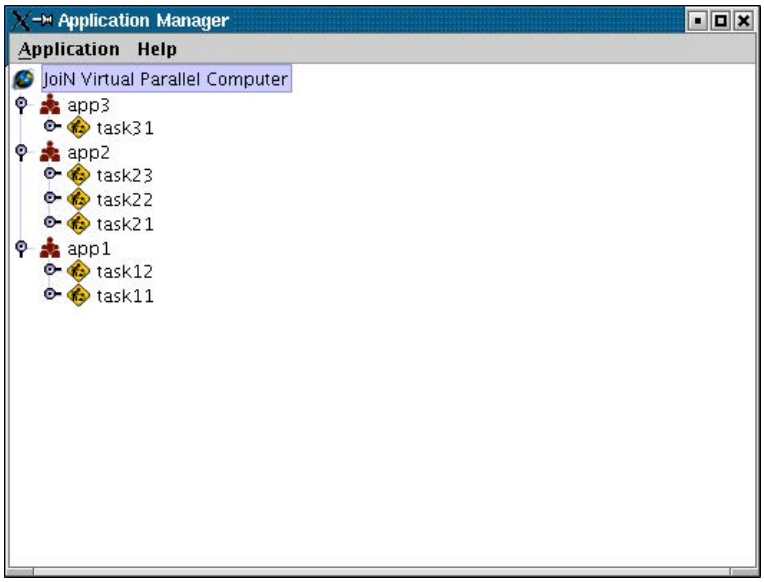

Figura A.7: Interface gráfica do `ApplicationManager`.

Instalação de aplicações

Selecionando o item `Install App...` no menu o sistema oferecerá uma oportunidade para selecionar o arquivo PAS da aplicação que se deseja instalar. Uma vez selecionado, aparecerá a janela mostrada na Fig. A.8. Neste caso está sendo instalada a aplicação `PrimeSearch`, que procura por números primos grandes.

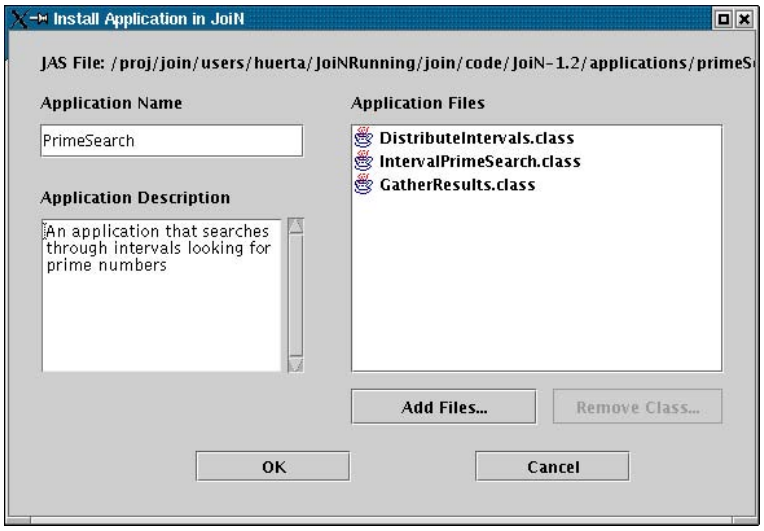

Figura A.8: Instalação de uma aplicação.

Ao selecionar `OK` o sistema executará as operações necessárias para a instalação. Se tudo correr bem, uma mensagem de aviso como a mostrada na Fig. A.9 aparecerá na tela.

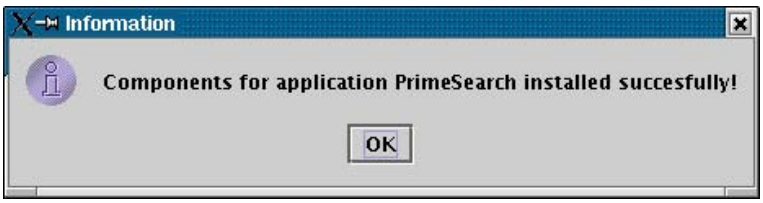

Figura A.9: Indicação de sucesso na instalação de uma aplicação.

A.2.6 Execução de uma aplicação

Para selecionar uma aplicação para ser executada no sistema é necessário escolher o item `Submit App...` no menu `Application` da janela mostrada na Fig. A.7. Como conseqüência aparecerá na tela a janela mostrada na Fig. A.10.

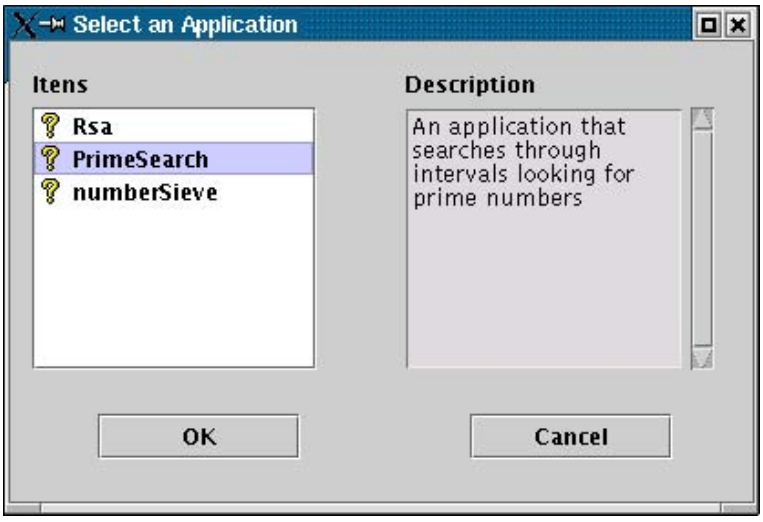

Figura A.10: Submissão de uma aplicação.

Se tudo correr bem e a aplicação começar a ser executada, a mensagem de sucesso mostrada na Fig. A.11 aparecerá na tela.

Para saber quando a aplicação terminou basta verificar as mensagens que aparecem na tela do coordenador, como mostra a Fig. A.12.

Figura A.11: Indicação de sucesso na submissão de uma aplicação.

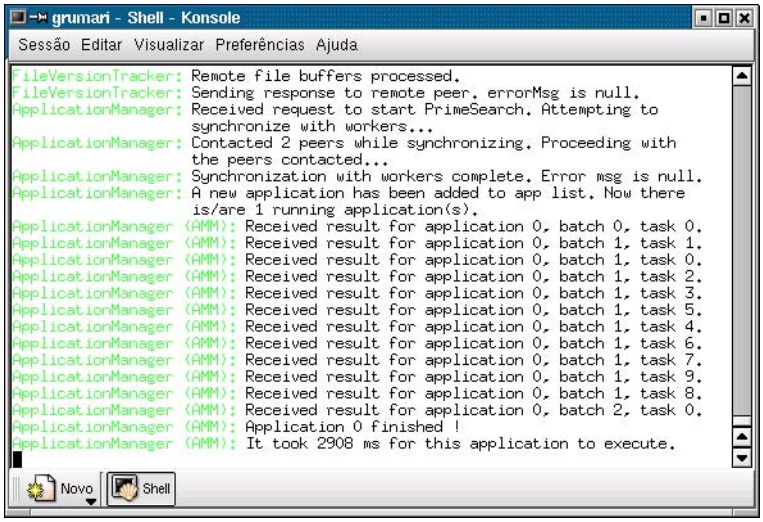

Figura A.12: Indicação do fim da execução de uma aplicação.

Apêndice B

Artigos derivados da tese

Neste apêndice mostram-se os artigos derivados da tese, tanto os publicados quanto submetidos para publicação.

- Eduardo Javier Huerta Yero and Marco A. Amaral Henriques. *"A Method to Solve the Scalability Problem in Massively Parallel Processing on the Internet"* . The 7th Euromicro Workshop on Parallel and Distributed Processing, Funchal, Portugal, Fev. 1999.

- Eduardo Javier Huerta Yero, Marco A. Amaral Henriques, Javier R. García and Alina C. Leyva. *"JOINT: An Object Oriented Message Passing Interface for Parallel Programming in Java"*. Java for High Performance Computing Workshop, High Performance Computing Networks 2001, Amsterdã, Holanda, Junho 25-27, 2001.

- Eduardo J. Huerta Yero and Marco A. Amaral Henriques. *"Contention-Sensitive Static Performance Prediction for Parallel Distributed Applications"*. Submetido a Performance Evaluation, Elsevier Science.

- Eduardo J. Huerta Yero and Marco A. Amaral Henriques. *"Speedup and Scalability Analysis of Master-Slave Applications on Grid Environments"*. Submetido a IEEE Transactions on Parallel and Distributed Processing, IEEE Computer Society Press.

- Eduardo J. Huerta Yero, Fabiano Oliveira Lucchese, Francisco S. Sambatti, Miriam von Zuben and Marco A. Amaral Henriques. *"*JoiN*: The Implementation of a Java-based Massively Parallel Grid"*. Submetido a Future Generation Computing Systems, Elsevier Science.

- Eduardo J. Huerta Yero, Fabiano Oliveira Lucchese, Francisco S. Sambatti and Marco A. Amaral Henriques. *"An Adaptive and Fault Tolerant Scheduler for Grids"*. Submetido a Journal of Grid Computing, Kluwer Academic Publishers.

- Eduardo J. Huerta Yero, Francisco S. Sambatti, Fabiano Oliveira Lucchese and Marco A. Amaral Henriques. *"A Fault Tolerance Mechanism for Scalable Grids"*. Submetido a Distributed Computing, Springer-Verlag.