# InteGrade: Middleware para Computação em Grade Oportunista*

**Andrei Goldchleger**†, **Fabio Kon**
Departamento de Ciência da Computação
Universidade de São Paulo
{andgold,kon}@ime.usp.br
http://gsd.ime.usp.br/integrade

**Abstract**

Grid Computing allows for the integration of distributed computing resources, providing seamless access to their combined computing power. However, the existing Grid solutions targeted at low-end computing resources, such as desktop machines, have a number of limitations that hinder their usability in many application scenarios. This paper describes the core design of the InteGrade Grid Computing System, the implementation of various modules and tools, and the architecture and implementation of a parallel programming library that allows parallel applications to be executed on InteGrade grids.

**Keywords:** Grid Computing, Distributed Computing, Parallel Computing.

**Resumo**

A Computação em Grade permite a integração de recursos computacionais distribuídos, provendo acesso transparente à capacidade combinada de múltiplos recursos. Entretanto, as soluções para Computação em Grade voltadas a recursos computacionais de baixo custo apresentam uma série de limitações que impedem sua utilização por várias categorias de aplicação. Esse artigo descreve a arquitetura central do InteGrade, a implementação de diversos módulos e ferramentas, e a arquitetura e implementação de uma biblioteca de programação paralela que permite que aplicações paralelas sejam executadas em grades InteGrade.

**Palavras-chave:** Computação em Grade, Computação Distribuída, Computação Paralela.

## 1 Introdução

A computação se tornou uma ferramenta indispensável para as mais diversas atividades humanas. As ciências biológicas dependem de poder computacional para realizar tarefas como simulações, avaliação de modelos e mineração de dados. Físicos analisam grandes quantidades de dados geradas em experimentos realizados em aceleradores de partículas. A indústria cinematográfica utiliza aglomerados de computadores para gerar efeitos visuais cada vez mais realistas. A exploração de petróleo requer amplos estudos para determinar a probabilidade de existir petróleo em uma determinada localização. O mercado financeiro realiza simulações mercadológicas e análise de risco, o que requer uma grande capacidade de processamento. A cada dia, novas aplicações são concebidas, as quais requerem quantidades crescentes de computação.

A Computação em Grade [17] ajuda as instituições a lidar com a crescente necessidade de poder computacional: como é impossível obter capacidade de processamento adicional sem a aquisição de novos equipamentos, uma instituição pode implantar um sistema de Computação em Grade sobre os recursos que já possui, podendo utilizar sua capacidade combinada de processamento de maneira mais eficiente. Os sistemas de Grade facilitam as tarefas dos usuários e desenvolvedores de aplicações, pois provêem uma camada de abstração que encapsula a complexidade da infraestrutura distribuída, a qual potencialmente inclui recursos presentes em diferentes localizações geográficas e domínios administrativos.

Os sistemas de Computação em Grade podem ser divididos em duas grandes categorias. Alguns sistemas visam principalmente a integração de recursos computacionais de alto desempenho. Tais sistemas tendem a

*Este trabalho é financiado pelo CNPq, Brasil, processos 55.2028/2002-9 e 55.0094/2005-9.
†Andrei Goldchleger foi parcialmente financiado por uma bolsa de mestrado da CAPES, Brasil.

oferecer uma ampla gama de funcionalidades, como monitoramento de recursos, integração de computadores pertencentes a diversos domínios administrativos, e suporte a diversas categorias de aplicações paralelas. Entretanto, tais sistemas demandam uma implantação complexa, requerendo intervenções dos administradores de rede, além de equipamentos de alto custo. A segunda categoria objetiva integrar recursos computacionais de baixo custo, tais como computadores pessoais. Tais sistemas possuem implantação simplificada, porém oferecem funcionalidades limitadas. Em alguns casos, não possuem monitoramento de recursos, ou não permitem a execução de aplicações paralelas que demandam comunicação entre seus nós, o que impede sua utilização por determinadas categorias de aplicação.

O InteGrade [21] é um sistema de Computação em Grade que visa integrar recursos de baixo custo, tais como computadores pessoais. O InteGrade utiliza a capacidade ociosa de tais máquinas para executar aplicações da grade (daí o termo "Computação em Grade Oportunista"). O sistema oferece uma ampla gama de recursos que normalmente só estão presentes nos sistemas voltados a recursos de grande porte. Por exemplo, o sistema comporta a execução de diversas categorias de aplicação, incluindo aplicações seqüenciais (convencionais), *Bag-of-Tasks* (aplicações cujo domínio pode ser particionado sem que existam dependências entre cada partição), e paralelas com comunicação entre nós. A implementação e arquitetura do InteGrade são inteiramente orientadas a objetos, facilitando a integração de módulos e extensibilidade.

O trabalho realizado no contexto dessa dissertação de mestrado consistiu na definição da arquitetura básica do InteGrade, assim como na implementação dos seus principais módulos. Além disso, desenvolvemos uma biblioteca para programação paralela no InteGrade que permite a construção de aplicações que demandam comunicação entre seus nós. A organização deste artigo é a seguinte: a Seção 2 apresenta a arquitetura do InteGrade, sua implementação e avaliação de desempenho. A Seção 3 apresenta a biblioteca de programação paralela do InteGrade. Já a Seção 4 apresenta brevemente alguns trabalhos relacionados, e a Seção 5 apresenta as publicações produzidas durante o Mestrado. Finalmente, a Seção 6 apresenta nossas conclusões sobre o trabalho.

## 2 InteGrade

O Projeto InteGrade [33, 21] objetiva construir um middleware que permita a implantação de grades sobre recursos computacionais não dedicados, fazendo uso da capacidade ociosa normalmente disponível nos parques computacionais já instalados. É de conhecimento geral que grande parte dos computadores pessoais permanecem parcialmente ociosos durante longos períodos de tempo, culminando em períodos de ociosidade total: por exemplo, as estações de trabalho reservadas aos funcionários de uma empresa tradicional raramente são utilizadas à noite. Dessa maneira, a criação de uma infra-estrutura de software que permita a utilização efetiva de tais recursos que seriam desperdiçados possibilitaria uma economia financeira para as instituições que demandam grandes quantidades de computação. O InteGrade é um projeto desenvolvido conjuntamente por pesquisadores de três instituições: Departamento de Ciência da Computação (IME-USP), Departamento de Informática (PUC-Rio) e Departamento de Computação e Estatística (UFMS).

O InteGrade possui arquitetura orientada a objetos, onde cada módulo do sistema se comunica com os demais a partir de chamadas de método remotas. O InteGrade utiliza CORBA [27] como sua infra-estrutura de objetos distribuídos, beneficiando-se de um substrato elegante e consolidado, o que se traduz na facilidade de implementação, uma vez que a comunicação entre os módulos do sistema é abstraída pelas chamadas de método remotas. CORBA também permite o desenvolvimento para ambientes heterogêneos, facilitando a integração de módulos escritos nas mais diferentes linguagens, executando sobre diversas plataformas de hardware e software. Finalmente, CORBA fornece uma série de serviços úteis e consolidados, como os serviços de Transações [30], Persistência [29], Nomes [28] e *Trading*[1] [26], os quais podem ser utilizados pelo InteGrade, facilitando assim o desenvolvimento.

Desde a sua concepção, o InteGrade foi desenvolvido com o objetivo de permitir o desenvolvimento de aplicações para resolver uma ampla gama de problemas paralelos. Vários sistemas de Computação em Grade restringem seu uso a problemas que podem ser decompostos em tarefas independentes, como *Bag-of-Tasks* ou aplicações paramétricas. Alguns pesquisadores argumentam que sistemas de Computação em Grade não são apropriados para aplicações paralelas que possuem dependências entre seus nós, uma vez que tais dependências implicam em comunicações sobre redes de grande área, nem sempre robustas, e a aplicação

[1] O serviço de *Trading* definido em CORBA armazena *ofertas de serviços*, que contém características associadas a objetos CORBA. Através do uso da linguagem TCL (*Trader Constraint Language*), é possível realizar consultas de modo a obter referências a objetos que atendam a determinados requisitos.

inteira pode falhar por causa de um nó, caso não se adotem as medidas necessárias. De fato, algumas aplicações paralelas demandam as redes de interconexão proprietárias dos computadores paralelos, porém existe uma série de problemas que demandam comunicação e podem ser tratados por máquinas conectadas por redes convencionais. Além disso, nota-se a contínua evolução na capacidade de transmissão das redes locais e de grande área – o acesso do tipo "banda larga" já é uma realidade há anos, inclusive nas residências, e sua disseminação tende a aumentar, assim como a capacidade de transmissão de tais linhas.

O InteGrade é extremamente dependente dos usuários provedores de recursos, ou seja, usuários que compartilham a parte ociosa de seus recursos na Grade. Sem a colaboração de tais usuários, não existirão recursos disponíveis para as aplicações da Grade. Dessa maneira, o InteGrade pretende impedir que os usuários provedores de recursos sintam qualquer degradação de desempenho causada pelo InteGrade quando utilizam suas máquinas. Esse objetivo é atingido através da implantação de um módulo compacto nas máquinas provedoras de recursos, de maneira a consumir poucos recursos adicionais. Além disso, pretendemos utilizar o DSRT (*Dynamic Soft-Realtime CPU Scheduler*) [44], um escalonador em nível de aplicação para limitar a quantidade de recursos que pode ser utilizada pelas aplicações da Grade, permitindo assim que o usuário imponha políticas de compartilhamento de recursos, caso assim deseje. É importante notar que a aplicação de tais políticas é opcional, ou seja, mesmo que tais políticas não sejam aplicadas o InteGrade deve garantir a qualidade de serviço oferecida ao usuário provedor de recursos.

O InteGrade é um sistema muito dinâmico – uma vez que a maioria de seus recursos é compartilhada, a disponibilidade de recursos pode variar drasticamente ao longo do tempo e um recurso pode ser retomado por seu proprietário a qualquer momento. Tal ambiente é hostil às aplicações da Grade e prejudica o escalonamento, uma vez que as decisões de escalonamento podem ser invalidadas na prática devido à dinamicidade na disponibilidade dos recursos. Dessa maneira, o InteGrade pretende adotar um mecanismo para atenuar os efeitos do ambiente dinâmico: a Análise e Monitoramento dos Padrões de Uso, cujo objetivo é coletar longas séries de informações de maneira a permitir uma previsão probabilística da disponibilidade dos recursos compartilhados. Durante o escalonamento, a Análise e Monitoramento dos Padrões de Uso permitirá estimar por quanto tempo um recurso permanecerá ocioso, colaborando assim para melhores decisões de escalonamento.

Além das preocupações com desempenho, o InteGrade deve impedir que as máquinas da grade tenham sua segurança comprometida. Um usuário provedor de recursos deve estar seguro de que aplicações de terceiros submetidas através da Grade não comprometam seu sistema. Dessa maneira, o InteGrade deve tomar precauções para impedir que uma aplicação apague ou altere arquivos do usuário, ou tenha acesso a informações confidenciais. Uma abordagem a ser considerada é utilizar técnicas de *sandboxing* [20] para limitar as capacidades das aplicações de terceiros. Dessa maneira, por exemplo, podemos restringir o acesso ao sistema de arquivos a um determinado diretório, impedindo assim que as aplicações de terceiros obtenham acesso a dados confidenciais. Outro exemplo de uso é impedir que as aplicações da Grade acessem determinados dispositivos, como impressoras.

### 2.1 Arquitetura e Implementação

A arquitetura inicial do InteGrade foi inspirada no sistema operacional distribuído 2K [36]. Dessa maneira, o InteGrade herdou parte da nomenclatura de 2K. A unidade estrutural básica de uma grade InteGrade é o aglomerado (cluster). Um aglomerado é um conjunto de máquinas agrupadas por um determinado critério, como pertinência a um domínio administrativo. Tipicamente o aglomerado explora a localidade de rede, ou seja, o aglomerado contém máquinas que estão próximas uma das outras em termos de conectividade. Entretanto tal organização é totalmente arbitrária e os aglomerados podem conter máquinas presentes em redes diferentes, por exemplo. O aglomerado tipicamente contém entre uma e cem máquinas.

A Figura 1 apresenta os elementos típicos de um aglomerado InteGrade. Cada uma das máquinas pertencentes ao aglomerado também é chamada de nó, existindo vários tipos de nós conforme o papel desempenhado pela máquina. O Nó Dedicado é uma máquina reservada à computação em grade, assim como os nós de um aglomerado dedicado tradicional. Tais máquinas não são o foco principal do InteGrade, mas tais recursos podem ser integrados à grade se desejado. O Nó Compartilhado é aquele pertencente a um usuário que disponibiliza seus recursos ociosos à Grade. Já o Nó de Usuário é aquele que tem capacidade de submeter aplicações para serem executadas na Grade. Finalmente, o Gerenciador de Aglomerado é o nó onde são executados os módulos responsáveis pela coleta de informações e escalonamento, entre outros. Note que um nó pode pertencer a duas categorias simultaneamente – por exemplo, um nó que tanto compartilha seus recursos ociosos quanto é capaz de submeter aplicações para serem executadas na Grade.

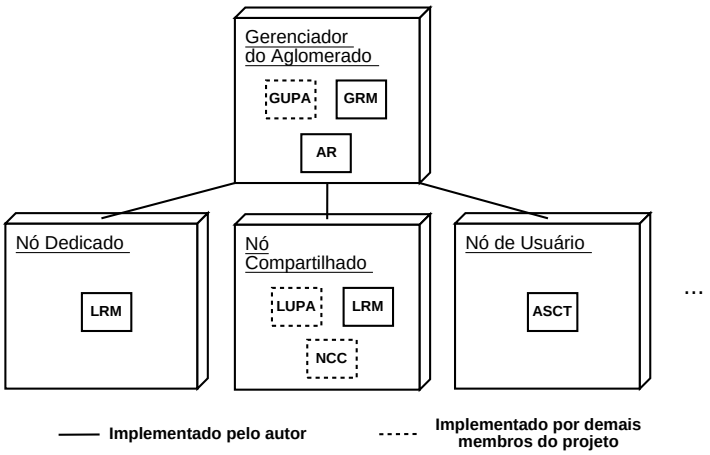


Figura 1: Arquitetura intra-aglomerado do InteGrade

Os módulos apresentados na Figura 1 são responsáveis pela execução de diversas tarefas necessárias à Grade. O **LRM** (Local Resource Manager) é executado em todas as máquinas que compartilham seus recursos com a Grade. É responsável pela coleta e distribuição das informações referentes à disponibilidade de recursos no nó em questão, além de exportar os recursos desse nó à Grade, permitindo a execução e controle de aplicações submetidas por usuários da Grade. Uma vez que o LRM é um módulo que deve consumir poucos recursos da máquina, escolhemos C++como a linguagem para seu desenvolvimento, aliada ao ORB compacto OiL [46], escrito na linguagem Lua [32].

O **GRM** (Global Resource Manager) tipicamente é executado no nó Gerenciador de Aglomerado e possui dupla função: (1) atua como o Serviço de Informações recebendo dos LRMs atualizações sobre a disponibilidade de recursos em cada nó do aglomerado e (2) atua como escalonador ao processar as requisições para a execução de aplicações na Grade. O GRM utiliza as informações de que dispõe para escalonar aplicações aos nós mais apropriados, conforme os requisitos das mesmas. A implementação do GRM utiliza a linguagem Java e o ORB JacORB [5].

O **NCC** (Node Control Center) é executado nas máquinas compartilhadas e permite que o proprietário da máquina controle o compartilhamento de seus recursos com a Grade. Atuando conjuntamente com o LRM, permite que o usuário defina períodos em que os recursos podem ou não ser utilizados (independente de estarem disponíveis), a fração dos recursos que pode ser utilizada (exemplo: 30% da CPU e 50% de memória) ou quando considerar a máquina como ociosa (por exemplo, quando não há de atividade de teclado por 5 minutos, ou quando o usuário encerra sua sessão). Convém lembrar que tais configurações são estritamente opcionais, uma vez que o sistema se encarregará de manter a qualidade de serviço provida ao proprietário dos recursos.

O **ASCT** (Application Submission and Control Tool) permite que um usuário submeta aplicações para serem executadas na Grade. O usuário pode estabelecer requisitos, como plataforma de hardware e software ou quantidade mínima de recursos necessários à aplicação, e preferências, como a quantidade de memória necessária para a aplicação executar com melhor desempenho. A ferramenta permite também que o usuário monitore e controle o andamento da execução da aplicação. Quando terminada a execução, o ASCT permite que o usuário recupere arquivos de saída de suas aplicações, caso existam, além do conteúdo das saídas padrão e de erro, úteis para diagnosticar eventuais problemas da aplicação. A implementação atual do ASCT consiste de uma aplicação gráfica escrita em Java utilizando JacORB.

O **LUPA** (Local Usage Pattern Analyzer) é responsável pela Análise e Monitoramento dos Padrões de Uso. A partir das informações periodicamente coletadas pelo LRM, o LUPA armazena longas séries de dados e aplica algoritmos de clustering [47] de modo a derivar categorias comportamentais do nó. Tais categorias serão utilizadas como subsídio às decisões de escalonamento, fornecendo uma perspectiva probabilística de maior duração sobre a disponibilidade de recursos em cada nó. Note que o LUPA só é executado em máquinas compartilhadas, uma vez que recursos dedicados são sempre reservados à computação em grade.

O **GUPA** (Global Usage Pattern Analyzer) auxilia o GRM nas decisões de escalonamento ao fornecer as informações coletadas pelos diversos LUPAs. O GUPA pode funcionar de duas maneiras, de acordo com

as políticas de privacidade do aglomerado – se as informações fornecidas pelo LUPA não comprometem a privacidade de um usuário, como é o caso de uma estação de trabalho de um laboratório, o GUPA pode agir como aglomerador das informações disponibilizadas pelos LUPAs. Entretanto, caso o perfil de uso gerado pelo LUPA comprometa a privacidade do proprietário de um recurso (como uma estação de trabalho pessoal), os padrões de uso nunca deixam o LUPA – nesse caso, o GUPA realiza consultas específicas ao LUPA, podendo assim funcionar como *cache* das respostas fornecidas sob demanda pelos diversos LUPA.

O **AR** (Application Repository) armazena as aplicações a serem executadas na Grade. Através do ASCT, o usuário registra a sua aplicação no repositório para posteriormente requisitar sua execução. Quando o LRM recebe um pedido para execução de uma aplicação, o LRM requisita tal aplicação ao Repositório de Aplicações. O Repositório de Aplicações pode fornecer outras funções mais avançadas, como por exemplo o registro de múltiplos binários para a mesma aplicação, o que permite executar uma mesma aplicação em múltiplas plataformas, a categorização de aplicações, facilitando assim a busca por aplicações no repositório, a assinatura digital de aplicações, o que permite verificar a identidade de quem submeteu a aplicação, e controle de versões. Assim como o GRM, o Repositório de Aplicações foi desenvolvido em Java e utiliza o JacORB para a comunicação com os demais módulos.

### *2.1.1 Protocolo de Disseminação de Informações*

O Protocolo de Disseminação de Informações do InteGrade permite que o GRM mantenha informações relativas à disponibilidade de recursos nas diversas máquinas do aglomerado. Tais informações incluem dados estáticos (arquitetura da máquina, versão do sistema operacional e quantidades totais de memória e disco) e dinâmicos (porcentagem de CPU ociosa, quantidades disponíveis de disco e memória, entre outros). Tais informações são importantes para a tarefa de escalonamento de aplicações; de posse de uma lista de requisitos da aplicação, o GRM pode determinar uma máquina adequada para executá-la. O Protocolo de Disseminação de Informações do InteGrade é o mesmo utilizado pelo sistema operacional distribuído 2K [36].

Uma questão importante associada ao Protocolo de Disseminação de Informações é a periodicidade com a qual as atualizações são feitas. Se atualizarmos as informações muito freqüentemente, o desempenho do sistema tende a degradar, uma vez que a rede será tomada por mensagens de atualização. Por outro lado, quanto maior o intervalo de atualização de informações, maior a tendência de tais informações não corresponderem à real disponibilidade de recursos nas máquinas do aglomerado. Dessa maneira utilizamos o conceito de dica (*hint*) [37], ou seja, as informações mantidas no GRM fornecem uma visão aproximada da disponibilidade de recursos no aglomerado.

O Protocolo de Disseminação de Informações é o seguinte: periodicamente, a cada intervalo de tempo $t_1$, o LRM verifica a disponibilidade de recursos do nó. Caso tenha havido uma mudança significativa entre a verificação anterior e a atual, o LRM envia tais informações ao GRM. A determinação do que é significativo é dada através de uma porcentagem para a qual uma mudança é considerada significativa (por exemplo, 10% de variação na utilização de CPU). Entretanto, caso não hajam mudanças significativas em um intervalo $t_2$ ($t_2 > t_1$), o LRM mesmo assim manda uma atualização das informações. Tal atualização serve como *keep-alive* e permite que o GRM detecte quedas dos LRM. Essa abordagem resulta em redução do tráfego na rede, uma vez que as mensagens só são enviadas no caso de mudanças significativas, ou em intervalos maiores, no caso de servirem como *keep-alive*.

### *2.1.2 Protocolo de Execução de Aplicações*

O Protocolo de Execução de Aplicações do InteGrade permite que um usuário da grade submeta aplicações para execução sobre recursos compartilhados. Assim como o Protocolo de Disseminação de Informações, o Protocolo de Execução é derivado do protocolo utilizado no sistema 2K, porém alterado para o InteGrade. A Figura 2 ilustra os passos do protocolo. Uma vez que a aplicação tenha sido registrada no Repositório de Aplicações através do ASCT, o usuário solicita a execução da aplicação (1). O usuário pode, opcionalmente, especificar requisitos para a execução de sua aplicação, como por exemplo a arquitetura para a qual a aplicação foi compilada, ou a quantidade mínima de memória necessária para a execução. Também é possível especificar preferências, como executar em máquinas mais rápidas, por exemplo.

Assim que a requisição de execução é enviada ao GRM, este procura um nó candidato para executar a aplicação (2). Utilizando os requisitos da aplicação informados pelo usuário e as informações sobre disponibilidade de recursos nos nós fornecidas pelo Protocolo de Disseminação de Informações já descrito, o GRM procura por um nó que possua recursos disponíveis para executar a aplicação. Caso nenhum nó satisfaça

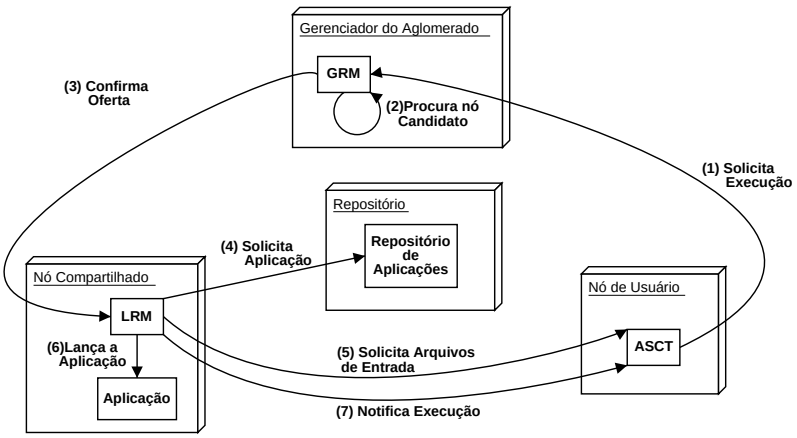


Figura 2: Protocolo de Execução de Aplicações

os requisitos da aplicação, o GRM notifica tal fato ao ASCT que solicitou a execução. Entretanto, caso haja algum nó que satisfaça os requisitos, o GRM envia a solicitação para o LRM da máquina candidata a executar a aplicação (3). Nesse momento, o LRM verifica se, de fato, possui recursos disponíveis para executar a aplicação – como já mencionado, o GRM mantém uma visão aproximada da disponibilidade de recursos nos nós do aglomerado, o que leva à necessidade de tal verificação. Caso o nó não possua os recursos necessários para a execução da aplicação, o LRM notifica o GRM de tal fato e o GRM retorna ao passo (2), procurando por outro nó candidato.

Entretanto, caso o nó candidato possa executar a aplicação, o LRM de tal nó solicita a aplicação em questão ao Repositório de Aplicações (4), solicita os eventuais arquivos de entrada da aplicação ao ASCT requisitante (5), e lança a aplicação (6), notificando ao ASCT que sua requisição foi atendida (7). O ASCT assim descobre em qual LRM sua aplicação está executando, podendo assim controlá-la remotamente.

Quando a requisição de execução envolve uma aplicação composta por múltiplos nós, por exemplo, uma aplicação paralela com $n$ nós, o protocolo é levemente alterado: no passo (2), o GRM procura até $n$ máquinas candidatas a executar a aplicação[2]. Posteriormente, no passo (3), o GRM realiza $n$ chamadas para executar a aplicação, cada uma destas solicitando a execução de um dos nós da aplicação.

Atualmente, a implementação do protocolo encontra-se incompleta: quando o GRM envia a requisição para o LRM, este ainda não verifica se tem recursos disponíveis para executar a aplicação, portanto, atualmente, um LRM nunca recusa um pedido de execução. Essa é uma deficiência que deve ser sanada no futuro.

### 2.2 Avaliação de Desempenho do LRM

O LRM é certamente o módulo mais crítico do InteGrade no tocante ao consumo de recursos. Como é executado em máquinas compartilhadas, o LRM não pode utilizar muitos recursos computacionais, sob pena de causar degradação na qualidade de serviço oferecida aos usuários que compartilham suas máquinas com a Grade. Assim, realizamos alguns experimentos com o objetivo de medir a utilização de CPU e memória do LRM. Os experimentos foram realizados em uma máquina com processador Athlon XP 2800+ e 1 GiB de memória RAM. A máquina utilizada possui o sistema operacional GNU/Linux (Debian 3.1 *testing*), kernel 2.6.10 e biblioteca de *threads* com suporte a NPTL (*Native POSIX Thread Library*) [14]. É importante notar que a combinação do kernel 2.6.10 com a biblioteca NPTL é fundamental [48] para permitir a correta medição da utilização de recursos de aplicações *multi-threaded* como o LRM. Utilizamos a ferramenta `top` para a medição do consumo de CPU e `pmap` para obter o consumo detalhado de memória de uma aplicação. Ambas ferramentas fazem parte do pacote Procps. Para a determinação do consumo de memória efetivo da aplicação (RSS – *Resident Set Size*)[3], inspecionamos o pseudo-diretório `/proc` através de *scripts*.

[2]Caso não consiga $n$ máquinas, o GRM pode escalonar dois ou mais nós da aplicação para uma mesma máquina.

[3]O RSS representa a quantidade de memória física que uma determinada aplicação ocupa em um determinado momento. É a soma do espaço ocupado pelo executável, pilha, bibliotecas compartilhadas e área de dados.

A Tabela 1 apresenta o uso de memória do LRM após sua iniciação[4]. Note que a maioria do espaço ocupado em memória pelo LRM se refere a bibliotecas compartilhadas. Do total de bibliotecas, apenas a `liblua` (80 KiB) e a `liblualib` (88 KiB) são bibliotecas de uso específico. As demais bibliotecas são de uso geral e muito provavelmente são utilizadas por aplicações do usuário, já em execução na mesma máquina e, portanto, não implicam em gastos adicionais. Assim, dos 2816 KiBs ocupados após a iniciação, apenas 576 KiB são exclusivos do LRM, o que é pouco quando consideramos que as estações de trabalho atuais possuem ao menos 256 MiB de memória RAM. Tais observações nos levam a concluir que a implementação do LRM atingiu o objetivo de gastar poucos recursos adicionais das máquinas compartilhadas.

| Consumo de Memória do LRM | |
|---|---|
| Tipo | Tamanho (KiB) |
| *Resident Set Size* | 2816 |
| Executável | 128 |
| Pilha | 52 |
| Bibliotecas | 2408 |

| Consumo de Memória por Biblioteca | | |
|---|---|---|
| Nome | Finalidade | Tamanho (KiB) |
| libc | Biblioteca padrão C | 1188 |
| libstdc++ | Biblioteca padrão C++ | 636 |
| libm | Biblioteca matemática C | 132 |
| ld | Carregador de bibliotecas dinâmicas | 88 |
| liblualib | Biblioteca Lua | 88 |
| liblua | Biblioteca Lua | 80 |
| libresolv | Resolução de nomes (DNS) | 60 |
| libpthread | Biblioteca de *threads* | 48 |
| libnss_files | Obtenção de informações de configuração do sistema | 36 |
| libgcc_s | Biblioteca de suporte a exceções para C++ | 32 |
| libnss_dns | Obtenção de informações de configuração do sistema | 12 |
| libdl | Biblioteca para carga dinâmica de bibliotecas | 8 |

Tabela 1: Consumo detalhado de memória do LRM

O primeiro experimento realizado refletiu a operação do LRM quando este não recebe nenhuma requisição para executar aplicações. Nesse cenário, o LRM apenas envia periodicamente para o GRM atualizações sobre a quantidade de recursos disponíveis no nó, além de verificar periodicamente a terminação de aplicações previamente iniciadas. Configuramos o LRM de maneira a verificar a variação da disponibilidade de recursos a cada segundo e enviar uma atualização ao GRM a cada dois segundos no máximo. Para realizar as medições, utilizamos um programa especialmente desenvolvido para coletar a utilização de CPU e memória (*Resident Set Size*) a cada três segundos. O programa `top` foi utilizado para fins de controle, permitindo a validação dos dados obtidos por nosso programa. O monitoramento foi realizado durante 5 minutos.

A Figura 3 apresenta os resultados obtidos no experimento. Podemos notar que o consumo de CPU se mantém extremamente baixo, em menos de 1%, sendo que na maioria do tempo a utilização de CPU permaneceu em 0%. Através do programa desenvolvido, verificamos que a quantidade total de CPU utilizada (`usertime + systemtime`) aumenta muito lentamente, sendo tais aumentos extremamente pequenos. Quanto ao uso de memória, podemos notar que sofre um pequeno aumento que não chega a ser significativo.

O segundo experimento reflete o consumo de recursos quando o LRM recebe requisições de execução. Nesse experimento, o LRM recebia uma requisição para a execução e lançava uma instância de uma aplicação seqüencial. Esse procedimento foi repetido cinqüenta vezes, com um intervalo de dez segundos entre cada requisição. As medições foram realizadas com as mesmas ferramentas utilizadas no primeiro experimento. A Figura 4 apresenta os gráficos dos resultados obtidos pelo experimento. Notamos que o consumo de CPU para lançar uma instância da aplicação é extremamente baixo, sempre permanecendo inferior a 2%. Já o consumo de memória cresce rapidamente após as primeiras requisições e fica sujeito a oscilações nas

[4]Apesar de incluído no RSS, o espaço destinado à área de dados não consta na tabela pois pode variar de tamanho ao longo da execução do LRM.

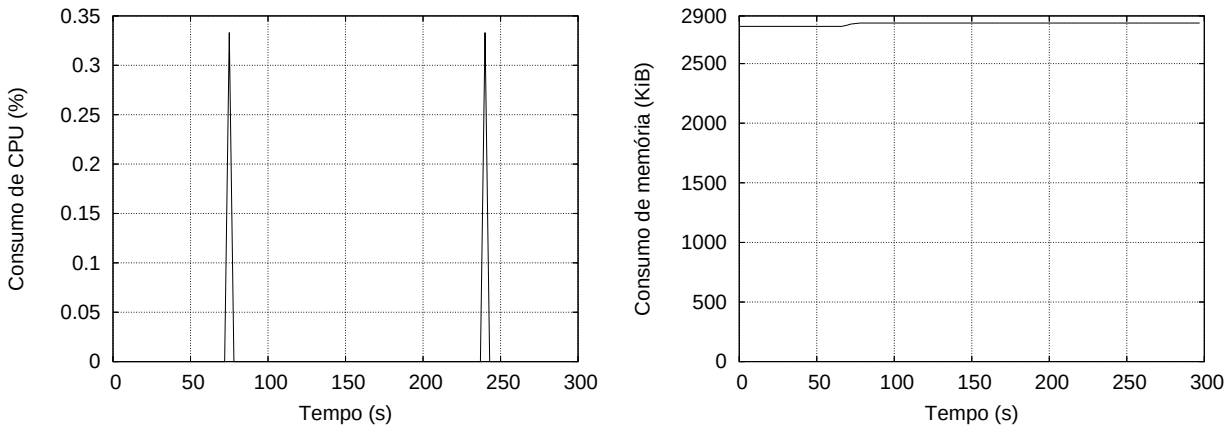


Figura 3: Experimento 1: Consumo de CPU e memória do LRM

execuções seguintes. Entretanto, o crescimento no consumo de memória do LRM não chega a consumir uma quantidade significativa de recursos da máquina. No futuro, experimentos adicionais podem ser conduzidos de maneira a determinar mais precisamente os motivos que levam ao aumento no consumo de memória.

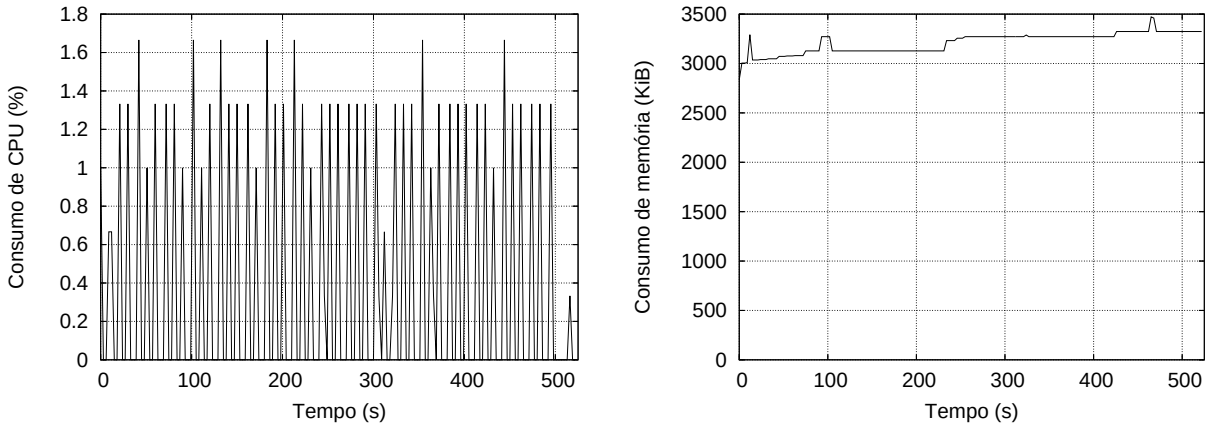


Figura 4: Experimento 2: Consumo de CPU e memória do LRM

## 3 A Biblioteca de Programação Paralela

Ambientes de Computação em Grade possuem diversas características que os tornam extremamente adequados à execução de aplicações paralelas. A grande disponibilidade de recursos sugere a possibilidade de executarmos várias aplicações paralelas sobre os mesmos, eliminando a necessidade de recursos dedicados. Porém, após uma análise inicial, nos deparamos com diversos problemas relacionados à execução de aplicações paralelas sobre as Grades:

- *comunicação*: algumas aplicações paralelas fortemente acopladas demandam grande quantidade de comunicação entre seus nós. Tais aplicações dificilmente podem se beneficiar de sistemas de Computação em Grade, a não ser em casos excepcionais onde seja possível a utilização de redes de altíssima velocidade. Ainda assim, existe uma ampla gama de aplicações que podem ser executadas em ambientes típicos de Grade;

- *tolerância a falhas*: executar uma aplicação paralela sobre uma grade cria dificuldades inexistentes quando trabalhamos com máquinas paralelas ou aglomerados dedicados. O ambiente de Grade, devido

a sua natureza altamente distribuída, é muito mais propenso a falhas. Dessa maneira, sistemas de Computação em Grade idealmente precisam prover mecanismos de tolerância a falhas adequados, de maneira a garantir que aplicações progridam em sua execução, mesmo em caso de falhas;

- *checkpointing*: a habilidade de salvar o estado da aplicação de maneira a garantir o seu progresso é uma característica importante que deve estar presente nos sistemas de Computação em Grade, especialmente em ambientes que fazem uso oportunista de recursos ociosos, como o InteGrade. Porém, *checkpointing* de aplicações paralelas é um problema significativamente mais difícil que *checkpointing* de aplicações convencionais [55];

- *variedade de modelos existentes*: existe uma ampla gama de modelos e implementações de bibliotecas para programação paralela, dentre as quais podemos citar MPI [25], PVM [51], BSP [54] e CGM [13], entre outros. Tais bibliotecas foram criadas anteriormente à existência de sistemas de Computação em Grade, portanto existe uma considerável quantidade de aplicações já escritas para as mesmas. Assim, é importante que os sistemas de Computação em Grade ofereçam suporte a tais bibliotecas, de maneira a permitir que aplicações pré-existentes possam ser executadas sobre as grades sem que sejam necessários grandes esforços para modificar tais aplicações.

Apesar dos problemas, o suporte a aplicações paralelas está presente nos principais sistemas de Computação em Grade. A abordagem mais usada é intuitiva: cria-se uma biblioteca que possua as mesmas interfaces de uma já existente fora do contexto de grade, como MPI. Internamente, tal biblioteca possui uma implementação específica para cada sistema de Computação em Grade, porém tais detalhes são ocultados do usuário, que enxerga apenas as mesmas interfaces com as quais estava acostumado a programar na biblioteca original. Tal abordagem é bem sucedida, uma vez que permite utilizar aplicações existentes sobre as Grades efetuando poucas ou até mesmo nenhuma alteração.

Legion provê suporte a aplicações MPI e PVM [45]. Aplicações pré-existentes precisam apenas ser recompiladas e religadas de maneira a permitir que se beneficiem das facilidades oferecidas pelo sistema. Uma alternativa possível é executar aplicações MPI e PVM sem recompilá-las, porém facilidades como *checkpointing* não estarão disponíveis para as aplicações.

Condor também provê suporte a MPI e PVM. O suporte a PVM de Condor não requer que a aplicação seja recompilada e religada, mas a aplicação PVM deve seguir o paradigma mestre-escravo. Dessa maneira, aplicações PVM existentes podem ser executadas diretamente sobre Condor, respeitando as restrições de arquitetura e sistema operacional. Já o suporte a MPI apresenta um inconveniente: as máquinas sobre as quais as aplicações executam devem ser designadas "reservadas", ou seja, quando começam a executar uma aplicação paralela tais máquinas o fazem até o fim, não podendo sofrer assim nenhum tipo de interrupção [55]. Tal inconveniente dificulta o uso de máquinas compartilhadas e computação oportunista na execução de aplicações paralelas.

Globus provê suporte para aplicações escritas em MPI, através da biblioteca MPICH-G2 [35], uma implementação de MPI específica para o Globus, porém compatível com as demais implementações de MPI. MPICH-G2 utiliza o Globus para a comunicação entre os nós das aplicações, provendo diversos protocolos de comunicação e conversão no formato de dados. Mais recentemente, Globus foi estendido para prover suporte ao modelo BSP [53, 52]. A arquitetura proposta é similar à disponível para execução de programas MPI.

### 3.1 O Modelo BSP

O modelo BSP (*Bulk Synchronous Parallelism*) [54] foi concebido como uma nova maneira de se escrever programas paralelos. Assim como ocorreu nos casos do MPI e PVM, o modelo BSP é genérico o suficiente de maneira que possa ser implementado sobre as mais diferentes arquiteturas, desde uma máquina paralela até um aglomerado de PCs conectados por meio de uma rede *Ethernet*. Entretanto, o modelo BSP e suas implementações se diferenciam de modelos como o MPI ou o PVM. Primeiramente, as bibliotecas que implementam o modelo BSP costumam ser enxutas: por exemplo, a BSPlib [31], uma das implementações do BSP, possui em sua biblioteca núcleo apenas 20 funções[5]. A característica fundamental do modelo BSP é o desacoplamento entre comunicação e sincronização entre os nós de uma aplicação paralela. Outra característica do modelo BSP é a possibilidade de se fazer uma análise prévia do custo de execução dos programas: o modelo BSP provê métricas para o cálculo do desempenho de programas em uma certa arquitetura. Baseando-se

[5] A biblioteca MPI é composta por mais de 100 funções.

em parâmetros como o número de processadores, o tempo necessário para sincronizar os processadores e a razão entre as vazões de comunicação e computação é possível calcular previamente o desempenho de um programa.

Teoricamente, o modelo BSP é composto por um conjunto de processadores virtuais, cada qual com sua memória local. Tais processadores virtuais são conectados por uma rede de comunicação. No caso de uma máquina paralela, os processadores virtuais são mapeados para os processadores da máquina e a rede de comunicação é a infra-estrutura específica de interligação dos processadores em tal arquitetura, podendo ser um barramento, por exemplo. No caso de um aglomerado de computadores, cada "processador virtual" é um computador e a rede de comunicação é a rede local *Ethernet*, por exemplo.

O mecanismo de comunicação entre os nós de uma aplicação BSP não é restringido pelo modelo. Exemplos de mecanismos presentes nas implementações do modelo são: memória distribuída compartilhada e troca de mensagens (*Message Passing*).

A estrutura de um programa BSP é composta por uma série de *superpassos* (*supersteps*). Cada superpasso é composto por 3 etapas: inicialmente, cada uma das tarefas do programa realiza computação com os dados que possui localmente. Posteriormente, todas as comunicações pendentes entre as tarefas são realizadas. Finalmente, ocorre uma barreira de sincronização que marca o fim de um superpasso e o início do seguinte. Uma característica importante é o fato da comunicação só ser efetivada *no final* do superpasso. Ou seja, se uma tarefa escreve um valor na memória de outra tarefa, tal escrita só se materializa de fato no próximo superpasso. Algumas implementações, como a BSPlib, oferecem métodos que permitem a escrita e leitura imediata de valores em outras tarefas, porém tais métodos devem ser usados com cuidado e seguindo uma série de restrições de maneira a garantir a correção da aplicação.

### 3.2 A Implementação

Um dos objetivos da implementação BSP no InteGrade é permitir que aplicações BSP existentes possam ser executadas sobre a Grade com um mínimo de alterações. Apesar de compartilharem o mesmo modelo, cada biblioteca que o implementa possui interfaces diferentes, prejudicando assim a portabilidade. Para minimizar tal inconveniente, decidimos utilizar em nossa implementação a interface C[6] da BSPlib, uma das implementações BSP disponíveis. Assim, a tarefa de portar uma aplicação que utiliza a BSPlib para o InteGrade consiste apenas em incluir um cabeçalho diferente, recompilar a aplicação e religá-la com a nossa biblioteca BSP. Tal facilidade no porte é muito importante, uma vez que aplicações existentes podem se beneficiar dos recursos de uma grade sem que seja necessário um processo de porte trabalhoso e caro.

Outra característica importante da implementação BSP no InteGrade é a sua relativa independência em relação ao resto do sistema. Como InteGrade é um sistema em desenvolvimento intenso, é importante que suas interfaces mantenham-se enxutas, descrevendo apenas a funcionalidade essencial do sistema. Dessa maneira, toda a funcionalidade relativa à implementação do BSP é descrita em interfaces IDL separadas das demais. As interfaces IDL dos módulos do InteGrade em sua maioria não foram alteradas: a única alteração, realizada no ASCT, consistiu na adição de um método.

A biblioteca BSP do InteGrade utiliza CORBA para a comunicação entre os nós da aplicação, facilitando o desenvolvimento, manutenção e extensão do código. O uso de CORBA pode ser questionado em uma aplicação de alto desempenho como uma biblioteca para programação paralela, porém no caso específico de nossa implementação existem alguns atenuantes: (1) InteGrade se beneficia de poder de processamento que estaria ocioso, portanto, desempenho neste caso não é a preocupação principal, uma vez que ele provavelmente vai ser comprometido por outros fatores, como a necessidade de migração resultante da indisponibilidade de recursos. (2) ORBs compactos são utilizados com êxito em ambientes altamente restritivos no tocante à disponibilidade de recursos, como sistemas embutidos. Além disso, experimentos com o UIC-CORBA [49] mostram uma perda de desempenho de apenas 15% quando comparado a soquetes, demonstrando que é possível combinar as vantagens de CORBA a um desempenho bastante razoável. Caso necessário, no futuro, poderemos utilizar apenas soquetes ao custo de aumentar a complexidade da biblioteca BSP do InteGrade.

As aplicações BSP no InteGrade normalmente são compostas por dois ou mais nós, também chamados de processos ou tarefas. Cada nó possui um identificador único dentro da aplicação, o *BSP PID*, utilizado por exemplo no endereçamento das comunicações para envio de dados – todas as funções da biblioteca para tal finalidade possuem como um de seus parâmetros o identificador da tarefa para a qual a chamada deve ser feita. É importante ressaltar que as aplicações BSP são do tipo SPMD (*Single Program, Multiple Data*), ou

[6] Existe também uma interface para FORTRAN na BSPlib, porém não a oferecemos no InteGrade.

seja, todos os nós da aplicação executam o mesmo programa. É comum, entretanto, que diferentes nós da aplicação executem diferentes trechos de código – por exemplo, é comum que apenas um nó realize tarefas de totalização de resultados. Os identificadores de processo são novamente úteis nesse cenário: pode-se por exemplo determinar que um dado trecho de código seja executado apenas pelo nó que possua um determinado identificador.

As aplicações BSP em muitos aspectos são tratadas pelo InteGrade de maneira similar às aplicações convencionais: por exemplo, o registro de uma aplicação BSP no Repositório de Aplicações se dá de maneira idêntica ao registro de uma aplicação convencional. As requisições de execução referentes a aplicações BSP são tratadas pelo GRM de maneira idêntica às aplicações paramétricas: para todos os efeitos, uma aplicação BSP é apenas uma aplicação que possui múltiplos nós. É bem provável que futuramente as diferentes classes de aplicações venham a ser tratadas de maneira diferente: por exemplo, como os nós das aplicações BSP comunicam-se entre si, é desejável que eles sejam alocados para máquinas com boa conectividade. Entretanto, ao minimizar a necessidade de mudanças para cada classe de aplicação contribuímos para a simplicidade do sistema, introduzindo novas características apenas quando necessário.

Cada uma das tarefas da aplicação BSP possui um *BspProxy* associado, um servente CORBA que recebe mensagens relacionadas à execução da aplicação BSP. O BspProxy é criado de maneira independente em cada nó, durante a iniciação da aplicação. O BspProxy representa o lado servidor de cada um dos nós da aplicação BSP, recebendo mensagens de outros processos, tais como escritas ou leituras remotas em memória, mensagens sinalizando o fim da barreira de sincronização, entre outras. A Figura 5 apresenta a interface IDL do BspProxy. A finalidade de cada método será descrita posteriormente, no contexto das funções da biblioteca BSP do InteGrade.

```
interface BspProxy{
  void registerRemoteIor(in long pid, in string ior);
  void takeYourPid(in long pid);
  void bspPut(in types::DrmaOperation drmaOp);
  void bspGetRequest(in types::BspGetRequest request);
  void bspGetReply(in types::BspGetReply reply);
  void bspSynch(in long pid);
  void bspSynchDone(in long pid);
};
```

Figura 5: Interface IDL do BspProxy

Em determinados momentos, as aplicações BSP necessitam de um coordenador central. As principais tarefas que demandam coordenação são:

- *difusão do endereço IOR das tarefas*: como cada uma das tarefas de uma aplicação BSP potencialmente comunica-se com as demais, é conveniente que cada uma das tarefas possa se comunicar diretamente com as outras. Dessa maneira, é necessário que o coordenador colete e posteriormente distribua os endereços IOR de todas as tarefas que compõem a aplicação;

- *atribuição de identificadores de processo*: cada nó da aplicação deve ser identificado unicamente. Assim, o coordenador deve ser responsável por atribuir identificadores a cada uma das tarefas que compõem a aplicação;

- *coordenação das barreiras de sincronização*: as barreiras de sincronização ao final de cada superpasso indicam que cada uma das tarefas atingiu um determinado ponto em sua execução. O coordenador é responsável por receber as mensagens das tarefas que atingiram a barreira, e quando todas a tiverem atingido, o coordenador deve notificar que cada uma das tarefas pode prosseguir em sua execução.

Em nossa implementação, decidimos que o coordenador seria um dos nós da própria aplicação, dispensando assim serviços externos de coordenação. Dessa maneira, elegemos um dos nós da aplicação para ser o coordenador da mesma. Esse nó é denominado *Process Zero* em alusão ao seu identificador de processo – tal nó é responsável por atribuir os identificadores de processo e ele sempre atribui zero a si próprio. Note que a escolha do coordenador é feita de maneira totalmente transparente ao programador da aplicação. Além

disso, o Process Zero não tem suas atividades restritas à coordenação, ele executa normalmente suas tarefas como os demais nós da aplicação.

A implementação dos métodos da BSPlib no InteGrade se encontra parcialmente completa. Até o presente momento, implementamos as funções necessárias para a iniciação de uma aplicação BSP, algumas funções de consulta sobre características da aplicação, as funções que permitem a comunicação entre processos através de memória compartilhada distribuída e a função de sincronização dos processos. As funções que permitem comunicação por troca de mensagens (*Bulk Synchronous Message Passing* (BSMP)) foram implementadas por Carlos Alexandre Queiroz, membro do projeto InteGrade; entretanto, tais funções não serão aqui apresentadas. Nas seções seguintes, apresentaremos os métodos implementados e o seu funcionamento interno, assim como as principais classes que compõem a biblioteca.

### *3.2.1 Funções de Iniciação e Consulta*

A Figura 6 apresenta parte das funções básicas presentes na BSPlib e que já se encontram implementadas na biblioteca BSP do InteGrade. A função `bsp_begin` determina o início do trecho paralelo de uma aplicação BSP. De acordo com as especificações da BSPlib, cada programa BSP deve ter apenas um trecho paralelo[7]. Nenhuma das demais funções da biblioteca pode ser chamada antes de `bsp_begin`, uma vez que esta realiza as tarefas de iniciação da aplicação, entre elas a eleição do Process Zero, o coordenador da aplicação.

```
void bsp_begin(int maxProcs)
int bsp_pid()
int bsp_nprocs()
void bsp_end()
```

Figura 6: Funções básicas da biblioteca BSP do InteGrade

A eleição de um nó coordenador para a aplicação implica em um conhecimento mútuo dos nós participantes da eleição. Entretanto, convém lembrar que como os nós de uma aplicação BSP são escalonados de maneira semelhante às demais aplicações, cada nó da aplicação não conhece os demais. Dessa maneira, tornou-se necessário utilizar um intermediário que seja conhecido por todos os nós da aplicação, de maneira que estes possam descobrir uns aos outros. Ao invés de implementarmos um serviço especial para tal tarefa, optamos por estender a interface do ASCT para realizar a tarefa de iniciação da aplicação. Uma vez que o ASCT é o requisitante das execuções, foi possível convertê-lo facilmente em um serviço conhecido por todos os nós da aplicação, de maneira a permitir a descoberta mútua.

Ao realizar uma requisição que envolva uma aplicação BSP, o ASCT adiciona um arquivo especial na lista de arquivos de entrada da aplicação, o `bspExecution.conf`. A Figura 7 apresenta um exemplo do conteúdo desse arquivo. O campo `appMainRequestId` contém um identificador da aplicação BSP emitido pelo ASCT. O campo `asctIor` contém o endereço IOR do ASCT que requisitou a execução e o campo `numExecs` indica quantos nós fazem parte da aplicação. Esse arquivo é transferido ao LRM que atendeu à requisição da mesma maneira que os demais arquivos de entrada.

```
appMainRequestId 3
asctIor IOR:00CAFEBA...
numExecs 8
```

Figura 7: Exemplo de arquivo bspExecution.conf

A primeira tarefa realizada por `bsp_begin` é a eleição do Process Zero. O processo de eleição é extremamente simples: a partir do endereço IOR do ASCT contido em `bspExecution.conf`, `bsp_begin` instancia um *stub* para o ASCT e realiza a chamada `registerBspNode`, contendo como parâmetros o identificador da aplicação no ASCT e o endereço IOR do nó BSP em questão[8]. Cada nó da aplicação realiza tal chamada

[7] O trecho paralelo da aplicação, delimitado pelas chamadas `bsp_begin` e `bsp_end`, pode ser precedido ou seguido por trechos seqüenciais de código, ou seja, código `C` que não utiliza as funções da biblioteca BSP.

[8] Esse endereço IOR é o endereço do BspProxy associado ao nó, criado no início de `bsp_begin`.

de maneira independente. O nó que tiver sua chamada completada primeiro é automaticamente eleito coordenador. Nessa situação, o valor de retorno de `registerBspNode` é o *struct* `BspInfo` com os campos preenchidos da seguinte forma: `isProcessZero` é verdadeiro, `processZeroIor` é o endereço IOR do próprio processo. Para os demais nós, `BspInfo` é devolvido com `isProcessZero` igual a falso e `processZeroIor` contendo o endereço IOR do nó que foi eleito coordenador.

Após a eleição do Process Zero, restam ainda dois passos de iniciação da aplicação: a distribuição de identificadores de processo BSP e a difusão dos endereços IOR de cada tarefa que compõe a aplicação. A Figura 8 apresenta um diagrama de seqüência de tais passos envolvendo o Process Zero e uma das demais tarefas[9]. Cada nó da aplicação realiza os seguintes passos: envia seu endereço IOR para o Process Zero, através do método `registerRemoteIor` em BspProxy, e bloqueia até receber o seu identificador de processo BSP e os endereços IOR das demais tarefas. Já o Process Zero, a cada endereço IOR recebido, atribui e envia um identificador de processo BSP para a tarefa em questão, através do método `takeYourPid`. Quando todas as tarefas realizaram tais passos, o Process Zero envia para *cada tarefa* os pares *(BSP PID, IOR)* de *todas* as tarefas que fazem parte da aplicação. Dessa maneira, nas operações subseqüentes, cada tarefa pode comunicar-se diretamente com as demais, dispensando intermediários.

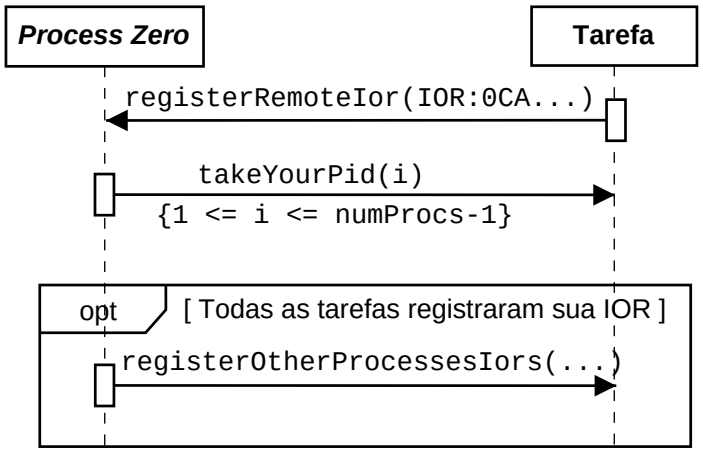


Figura 8: Diagrama de seqüência das tarefas de iniciação da aplicação BSP

A função `bsp_pid`, apresentada na Figura 6, fornece o identificador de processo BSP da tarefa em questão. Tal função é comumente usada para indicar que apenas uma determinada tarefa realize uma função na aplicação: por exemplo, pode-se assim determinar que apenas a tarefa de identificador $i$ gere arquivos de saída. Já a função `bsp_nprocs` devolve o número de tarefas que compõem a aplicação BSP. Finalmente, a função `bsp_end` delimita o final do bloco paralelo da aplicação. Em nossa implementação, esse método não precisa realizar nenhuma operação.

### *3.2.2 Distributed Remote Memory Access (DRMA)*

*Distributed Remote Memory Access* (DRMA) é um dos métodos disponíveis na BSPlib para a comunicação entre os processos de uma aplicação BSP. DRMA provê a abstração de memória compartilhada distribuída: determinadas áreas de memória de cada nó de uma aplicação podem ser acessadas pelos demais nós para operações de leitura e escrita.

Antes de realizar uma operação de leitura ou escrita na memória compartilhada, todos os nós da aplicação devem registrar a área de memória envolvida com a biblioteca BSP. Cada nó da aplicação BSP possui uma Pilha de Registros, como a exibida na Figura 9, onde são guardados registros descrevendo as áreas de memória que podem ser envolvidas em operações de escrita e leitura. Cada registro contém um endereço físico de memória e um tamanho em bytes referente ao tamanho da área de memória registrada. A posição do registro na Pilha de Registros representa o *endereço lógico* do registro. Uma determinada variável $X$ definida na aplicação BSP certamente terá diferentes endereços físicos em cada um dos nós da aplicação, porém possuirá o mesmo endereço lógico em todos os nós da aplicação.

[9]Ou seja, as atividades descritas no diagrama se repetem para cada um dos demais nós da tarefa.

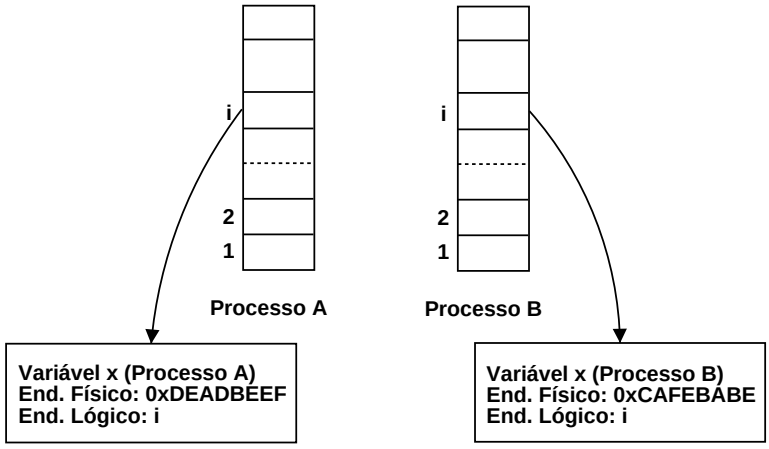


Figura 9: Exemplo da Pilha de Registros

A partir da Figura 9, podemos derivar um exemplo de como funciona a resolução de endereços com o auxílio da Pilha de Registros. Suponha que o processo $A$ deseja escrever ou ler a variável $X$ no processo $B$. Como se trata de uma aplicação distribuída, $A$ não tem como diretamente obter o endereço físico de $X$ em $B$. Assim, $A$ consulta sua Pilha de Registros à procura de algum registro que contenha a variável $X$, buscando pelo endereço físico local de $X$. Tal busca é feita a partir do topo da pilha, decrescendo eventualmente até a base. Quando $A$ acha algum registro para a variável $X$, obtém dessa maneira o endereço lógico de $X$, $i$, que é único para todos os processos. Dessa maneira, $A$ envia mensagem a $B$ indicando a escrita ou leitura na variável de endereço lógico $i$. $B$ consulta a posição $i$ sua Pilha de Registros, descobrindo assim o endereço físico local de $X$, podendo assim realizar a operação de escrita ou leitura. Note a importância de que o endereço lógico de uma variável na pilha seja o mesmo em todos os processos: caso contrário, uma escrita na variável $X$ poderia resultar em uma escrita na variável $Z$, causando erros no programa.

A Figura 10 apresenta o conjunto de funções para DRMA da BSPlib que já se encontram implementadas na biblioteca BSP do InteGrade. O método `bsp_pushregister` registra na pilha uma posição de memória de endereço `addr` e tamanho `size` em bytes. Note que é possível registrar um endereço de memória múltiplas vezes – a resolução de endereços é sempre realizada a partir do topo da pilha, dessa maneira, para um dado endereço físico de memória o registro devolvido será o mais recentemente registrado. Tal característica é útil quando são manipuladas estruturas compostas, como vetores, que podem ser registrados com diferentes comprimentos caso seja conveniente. Já o método `bsp_popregister` remove da pilha um registro referente ao endereço `addr`. Note que o método não recebe o tamanho do registro como parâmetro: a pilha é percorrida a partir do topo em direção a base e o primeiro registro encontrado referente a `addr` é removido. É importante ressaltar que a inserção e remoção de registros só será efetivada no final do superpasso.

```
void bsp_pushregister(const void * addr, int size)
void bsp_popregister(const void * addr)
void bsp_put(int pid, const void * src, void * dst, int offset, int nbytes)
void bsp_get(int pid, const void * src, int offset, void * dst, int nbytes)
```

Figura 10: Funções da biblioteca BSP do InteGrade para comunicação DRMA

O método `bsp_put` permite que um nó da aplicação realize uma escrita remota na memória de outra tarefa que compõe a aplicação. O parâmetro `pid` contém o identificador do processo no qual será realizado a escrita. `src` aponta para os dados a serem copiados para o processo remoto. `dst` é o endereço físico local da variável onde a escrita vai ser efetivada. Tal endereço será traduzido para o endereço lógico da variável pelo procedimento já descrito. Note que `dst` deve ser um endereço registrado na Pilha de Registros. Já o parâmetro `offset` representa um deslocamento a partir do endereço `dst`, ou seja, os dados serão escritos a partir do endereço `dst + offset`. Finalmente, `nbytes` indica o número de bytes que serão copiados. A

chamada remota (`bspPut` do BspProxy) é realizada no momento da chamada a `bsp_put`, porém a escrita no processo remoto só é efetivada ao final do superpasso.

De maneira oposta a `bsp_put`, o método `bsp_get` permite que um determinado nó da aplicação BSP leia um valor da memória de outra tarefa da aplicação. O método `bsp_get` possui uma assinatura semelhante à de `bsp_put`: o parâmetro `pid` contém o identificador do processo BSP do qual se lerão os dados. `src` aponta para os dados a serem copiados a partir do processo remoto. Tal endereço será traduzido para o endereço lógico da variável pelo procedimento já descrito. `offset` representa um deslocamento a ser aplicado a partir do endereço `src`, ou seja, os dados serão copiados a partir do endereço `src + offset`. É importante ressaltar que `src` deve estar registrado na Pilha de Registros. Já `dst` representa o endereço físico local para onde os dados serão copiados. Finalmente, `nbytes` indica o número de bytes que serão copiados a partir de `src + offset`. Assim como ocorre com `bsp_put`, a chamada remota (`bspGetRequest` do BspProxy) é realizada no momento da chamada a `bsp_get`, porém a leitura só é efetivada ao final do superpasso: após a barreira de sincronização, o processo alvo lê o valor requisitado e o envia ao requisitante através do método `bspGetReply`.

#### *3.2.3 A Barreira de Sincronização*

Conforme citado previamente, a computação no modelo BSP se desenvolve em termos da unidade básica chamada superpasso. Durante um superpasso, cada processo pode registrar e excluir posições da Pilha de Registros, assim como escrever ou ler resultados na memória dos demais, porém todas as operações só são efetivadas após todos os processos atingirem a barreira de sincronização. O método chamado em cada processo para indicar que uma barreira de sincronização foi atingida é `void bsp_sync()`.

Em nossa implementação, a barreira de sincronização é coordenada pelo Process Zero. Quando um processo executa a função `bsp_sync` presente no código de uma aplicação BSP, tal chamada desencadeia uma chamada remota `bspSynch` ao BspProxy associado ao Process Zero. Nesse momento, o processo que atingiu a barreira tem sua execução bloqueada, ficando apenas recebendo eventuais mensagens dos outros processos. Já o Process Zero, ao receber `bspSynch`, contabiliza o número de processos que já enviaram tal mensagem. Caso todos os processos tenham enviado tal mensagem, e inclusive o Process Zero tenha atingido a barreira, o Process Zero envia a mensagem `bspSynchDone` para todos os processos. Ao receberem tal mensagem, todos os processos são desbloqueados e passam a tratar as mensagens recebidas e os eventos ocorridos durante o superpasso, sempre na seguinte ordem:

1. `bsp_get`: todas as requisições de leitura efetuadas por outros processos são atendidas ao final do superpasso. Note que os valores são retornados antes de sofrer quaisquer eventuais alterações originadas pelos demais processos, uma vez que os `bsp_get` são tratados `antes` dos `bsp_put`;
2. `bsp_put`: as escritas remotas realizadas pelos demais processos são então efetivadas;
3. Registros de variáveis na pilha, ou seja, chamadas a `bsp_pushregister`;
4. Exclusões de variáveis da pilha, ou seja, chamadas a `bsp_popregister`.

## 4 Trabalhos Relacionados

Nos últimos anos, a pesquisa em Computação em Grade foi muito ativa, resultando em uma grande quantidade de sistemas para as mais diversas finalidades. A Tabela 2 apresenta uma comparação entre os diversos sistemas aqui apresentados.

O Globus [19, 16] é o projeto de maior impacto na área de Computação em Grade. O principal foco do projeto é a integração de recursos computacionais de alto desempenho. Baseado no conceito de caixa de ferramentas (*Globus Toolkit* – GT), permite que as Grades e suas respectivas aplicações sejam construídas de maneira incremental através da progressiva adição de serviços. A versão 3.0 (GT3) marcou uma grande mudança em relação a anterior (GT2), caracterizada pela adoção de Web Services como tecnologia de integração dos serviços da grade, chamados de *Grid Services*, além da definição e implementação de diversos padrões tais como OGSA [15], OGSI [18] e WSRF [8], que oferecem diretrizes para a construção de serviços da Grade.

O Legion [38, 24] objetivou a construção de um sistema de Computação em Grade totalmente orientado a objetos. Construiu uma infraestrutura baseada em Objetos Núcleo [40], modelo de objetos distribuídos que

proviam funcionalidades básicas aos demais serviços da grade. Apesar de possuir os mesmos objetivos do Globus, a arquitetura de Legion era radicalmente diferente: enquanto cada serviço Globus era construído de maneira independente, os serviços Legion eram todos construídos sobre seu arcabouço de objetos distribuídos. De certa maneira, a versão 3.0 de Globus adotou parte dos aspectos arquiteturais de Legion.

Condor [7, 41] é o pioneiro dos sistemas de Computação em Grade. Assim como o InteGrade, objetiva a utilização do poder de processamento ocioso de equipamentos de baixo custo. Inicialmente concebido como um sistema de gerenciamento de um aglomerado de máquinas, evoluiu de maneira a integrar recursos pertencentes a diversos domínios administrativos, permitindo inclusive a integração com grades Globus. Condor provê *checkpointing* para aplicações convencionais, permitindo que uma execução interrompida a qualquer momento seja retomada posteriormente. É possível utilizar Condor para executar aplicações paralelas MPI e PVM, porém estas últimas devem seguir o modelo de mestre-escravos. Entretanto, Condor não provê *checkpointing* de aplicações paralelas.

MyGrid [43, 6] permite que grades computacionais sejam implantadas facilmente pelo próprio usuário sobre recursos aos quais tem acesso. A implantação do sistema é simplificada, requerendo a instalação de alguns poucos módulos. MyGrid permite a execução de aplicações do tipo *Bag-of-Tasks*, mas não comporta aplicações que exigem comunicação entre nós. Devido a sua simplicidade, também não implementa alguns recursos típicos de uma grade, como monitoramento de recursos. OurGrid [3] é uma extensão de MyGrid que permite o compartilhamento de recursos computacionais em uma rede *Peer-to-Peer* [39]. Dessa maneira, a integração de recursos passa a ser maior, uma vez que é possível integrar recursos pertencentes a diversos indivíduos e instituições.

SETI@home [50, 2] consiste de uma aplicação distribuída com o objetivo de analisar ondas de rádio em busca de indícios que sugiram inteligência extra-terrestre. Distribuído no formato de um protetor de tela, a aplicação periodicamente se conecta a um complexo de servidores centralizados, os quais enviam um conjunto de dados a serem processados. Dessa maneira, o projeto utiliza a capacidade de processamento ociosa de milhares de computadores pessoais distribuídos ao redor do mundo. Apesar de suas limitações arquiteturais, o projeto alcançou grande êxito no seu objetivo de arregimentar colaboradores, possuindo mais de quatro milhões de usuários cadastrados e cerca de seiscentos mil usuários ativos.

BOINC (*Berkeley Open Infrastructure for Network Computing*) [4, 1] é um projeto sucessor de SETI@home, e objetiva sanar parte das limitações de seu antecessor. BOINC permite que a mesma aplicação cliente (o protetor de tela) realize processamento de dados para diversos projetos, cuja participação é definida pelo usuário que cede seus recursos. Apesar das melhorias, BOINC ainda limita seu uso à execução de aplicações do tipo *Bag-of-Tasks*.

| Característica \ Sistema | GT2 | GT3 | Legion | Condor | MyGrid | OurGrid | SETI@home | BOINC | InteGrade |
|---|---|---|---|---|---|---|---|---|---|
| Grade computacional tradicional | ✗ | ✗ | ✗ | | ✗ | ✗ | | | |
| Grade computacional oportunista | | | | ✗ | | | ✗ | ✗ | ✗ |
| Código aberto | ✗ | ✗ | | | ✗ | ✗ | | ✗ | ✗ |
| Binários gratuitos | ✗ | ✗ | | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Implementação orientada a objetos | | ✗ | ✗ | | ✗ | ✗ | | ✗ | ✗ |
| Comunicação baseada em padrões[a] | ✗[b] | ✗ | | | | | | | ✗ |
| Suporte a múltiplas aplicações | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | | ✗ | ✗ |
| Suporte a aplicações paralelas[c] | ✗ | ✗ | ✗ | ✗ | | | | | ✗ |
| Implementa o padrão OGSA | | ✗ | | | | | | | |

[a] Comunicação baseada em padrões da indústria tais como CORBA e Web Services.

[b] Apenas em alguns serviços, entre eles o MDS.

[c] Consideramos apenas aplicações paralelas que exigem comunicação entre seus nós, ou seja, que não sejam trivialmente paralelizáveis.

Tabela 2: Comparação entre diversos sistemas de Computação em Grade

## 5 Publicações

O trabalho aqui representado gerou oito publicações; em três delas participei como autor principal:

- *InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines* [22]: artigo curto publicado no *ACM/IFIP/USENIX Middleware'2003 International Workshop on Middleware for Grid Computing*, descreve a arquitetura e os primeiros esforços de implementação do InteGrade;
- *InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines* [21]: artigo publicado no periódico *Concurrency and Computation: Practice & Experience*, é uma versão estendida do artigo anterior;
- *Running Highly-Coupled Parallel Applications in a Computational Grid* [23]: artigo curto publicado no 22$^o$ *Simpósio Brasileiro de Redes de Computadores (SBRC)*, descreve a implementação da primeira versão da biblioteca BSP do InteGrade.

Participei como co-autor nas seguintes publicações:

- *Grid: An Architectural Pattern* [10]: artigo publicado na *11th Conference on Pattern Languages of Programs (PLoP'2004)*, é um padrão arquitetural que descreve os sistemas de Computação em Grade de maneira genérica, listando os principais serviços que estes oferecem;
- *Grid Middleware: Leveraging Distributed Processing Capabilities* [9]: versão estendida do padrão anterior, a ser publicado no livro *Pattern Languages of Program Design (PloPD)*;
- *Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware* [12]: artigo publicado no *ACM/IFIP/USENIX Middleware'2004 2nd International Workshop on Middleware for Grid Computing*, descreve a versão inicial do mecanismo de *checkpointing* para aplicações paralelas desenvolvidas com a biblioteca BSP do InteGrade;
- *Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware* [11]: artigo a ser publicado no periódico *Concurrency and Computation: Practice & Experience*, é uma versão estendida do artigo anterior;
- *InteGrade: A Tool for Executing Parallel Applications on a Grid for Opportunistic Computing* [34]: Artigo publicado no Salão de Ferramentas do 23$^o$ *Simpósio Brasileiro de Redes de Computadores (SBRC)*, descreve em detalhes as ferramentas do InteGrade que permitem a execução e monitoramento de aplicações.

## 6 Conclusão

O trabalho aqui apresentado e desenvolvido durante este mestrado consistiu na definição da arquitetura básica e implementação da fundação do sistema InteGrade. O resultado de tal trabalho produziu um sistema totalmente funcional que pode ser utilizado para a execução de diferentes classes de aplicações. A biblioteca de programação paralela permite a execução de aplicações paralelas que demandam comunicação entre nós, aumentando assim a gama de problemas que podem se beneficiar da grade.

A infra-estrutura aqui descrita serve como fundação para a pesquisa de outros membros do projeto, que resultará em novas funcionalidades para o sistema. As áreas de pesquisa do InteGrade incluem segurança, tolerância a falhas, escalonamento, agentes móveis e algoritmos paralelos. Os resultados estão sendo progressivamente incorporados à base de código do InteGrade.

Além de servir como substrato para a pesquisa dos demais membros do projeto, o trabalho aqui desenvolvido é utilizado como ponto de partida para dois projetos independentes: o projeto MAG [42] utiliza agentes móveis para a resolução de problemas que demandam grandes quantidades de computação; já o projeto FlexiGrid visa investigar diversos aspectos referentes à construção de grades computacionais, tais como qualidade de serviço, mobilidade e adaptação dinâmica.

O InteGrade é um projeto de código aberto hospedado na Incubadora Virtual da FAPESP, no endereço `http://incubadora.fapesp.br/projects/integrade`. A incubadora permite que qualquer interessado obtenha o código fonte do sistema e sua documentação. O texto completo da dissertação de mestrado está disponível no endereço `http://gsd.ime.usp.br/publications/thesis/MestradoAndrei.pdf`.

## Referências


[1] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10. IEEE Computer Society, 2004.

[2] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky e Dan Werthimer. SETI@home: an Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro e Paulo Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862, pages 61–86. Springer Verlag, Junho de 2003. Lecture Notes in Computer Science.

[4] BOINC. `http://boinc.berkeley.edu`. Último acesso em Julho/2005.

[5] Gerald Brose. JacORB: Implementation and Design of a Java ORB. In *Proceedings of the DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems*, pages 143–154, Setembro de 1997.

[6] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauvé, Fabrício A. B. Silva, Carla O. Barros e Cirano Silveira. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *Proceedings of the 2003 International Conference on Parallel Processing*, pages 407–416, Outubro de 2003.

[7] Condor. `http://www.cs.wisc.edu/condor`. Último acesso em Julho/2005.

[8] Karl Czajkowski, Don Ferguson, Ian Foster, Jeff Frey, Steve Graham, Tom Maguire, David Snelling e Steve Tuecke. *From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution*, Março de 2004. Version 1.1.

[9] Raphael Y. de Camargo, Andrei Goldchleger, Marcio Carneiro e Fabio Kon. Grid Middleware: Leveraging Distributed Processing Capabilities. A ser publicado no livro *Pattern Languages of Program Design 5 (PloPD'5)*.

[10] Raphael Y. de Camargo, Andrei Goldchleger, Marcio Carneiro e Fabio Kon. Grid: An Architectural Pattern. In *The 11th Conference on Pattern Languages of Programs (PLoP'2004)*, Monticello, Illinois, USA, Setembro de 2004.

[11] Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon e Alfredo Goldman. Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware. A ser publicado no periódico *Concurrency and Computation: Practice and Experience*.

[12] Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon e Alfredo Goldman. Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware. In *Proceedings of the ACM/IFIP/USENIX Middleware'2004 2nd International Workshop on Middleware for Grid Computing*, pages 35–40, Toronto, Ontario, Canada, Outubro de 2004.

[13] F. Dehne. Coarse grained parallel algorithms. *Algorithmica Special Issue on "Coarse grained parallel algorithms"*, 24(3–4):173–176, 1999.

[14] Ulrich Drepper e Ingo Molnar. The Native POSIX Thread Library for Linux. White Paper, Red Hat, Fevereiro de 2003.

[15] I. Foster, C. Kesselman, J. Nick e S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, Junho de 2002. Global Grid Forum, Open Grid Service Infrastructure Working Group.

[16] Ian Foster e Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 2(11):115–128, 1997.

[17] Ian Foster e Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 2003.

[18] Global Grid Forum. *Open Grid Services Infrastructure (OGSI) Version 1.0*, Junho de 2003. GFD-R-P.15 (Proposed Recommendation).

[19] Globus. `http://www.globus.org`. Último acesso em Julho/2005.

[20] Ian Goldberg, David Wagner, Randi Thomas e Eric A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th Usenix Security Symposium*, pages 1–13, San Jose, CA, USA, Julho de 1996.

[21] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger e Germano Capistrano Bezerra. InteGrade: object-oriented Grid middleware leveraging the idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, Março de 2004.

[22] Andrei Goldchleger, Fabio Kon, Alfredo Goldman vel Lejbman e Marcelo Finger. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. In *Proceedings of the ACM/IFIP/USENIX Middleware'2003 1st International Workshop on Middleware for Grid Computing*, pages 232–234, Rio de Janeiro, Junho de 2003.

[23] Andrei Goldchleger, Carlos Alexandre Queiroz, Fabio Kon e Alfredo Goldman. Running Highly-Coupled Parallel Applications in a Computational Grid. In *Proceedings of the 22th Brazilian Symposium on Computer Networks (SBRC' 2004)*, Gramado-RS, Brazil, Maio de 2004. Short paper.

[24] Andrew S. Grimshaw, Wm. A. Wulf e The Legion Team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, 1997.

[25] W. Gropp, E. Lusk, N. Doss e A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Setembro de 1996.

[26] Object Management Group. *Trading Object Service Specification*, Junho de 2000. OMG document formal/00-06-27, version 1.0.

[27] Object Management Group. *CORBA v3.0 Specification*. Needham, MA, Julho de 2002. OMG Document 02-06-33.

[28] Object Management Group. *Naming Service Specification*, Setembro de 2002. OMG document formal/02-09-02, version 1.2.

[29] Object Management Group. *Persistent State Service Specification*, Setembro de 2002. OMG document formal/02-09-06, version 2.0.

[30] Object Management Group. *Transaction Service Specification*, Setembro de 2003. OMG document formal/03-09-02, version 1.4.

[31] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas e Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[32] Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes Filho. Lua-an extensible extension language. *Software: Practice & Experience*, 26:635–652, 1996.

[33] InteGrade. `http://gsd.ime.usp.br/integrade`. Último acesso em Julho/2005.

[34] José Braga Pinheiro Jr., Raphael Y. de Camargo, Andrei Goldchleger e Fabio Kon. InteGrade: a Tool for Executing Parallel Applications on a Grid for Opportunistic Computing. In *Proceedings of the 23th Brazilian Symposium on Computer Networks (SBRC Tools Track)*, Fortaleza-CE, Brasil, Maio de 2005.

[35] N. Karonis, B. Toonen e I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, Maio de 2003.

[36] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt e Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC '9)*, pages 201–208, Pittsburgh, Agosto de 2000.

[37] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 33–48. ACM Press, 1983.

[38] Legion. `http://www.cs.virginia.edu/~legion`. Último acesso em Julho/2005.

[39] Bo Leuf. *Peer to Peer: Collaboration and Sharing over the Internet*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[40] Michael J. Lewis e Andrew Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC '96)*, pages 551–561, Los Alamitos, California, Agosto de 1996. IEEE Computer Society Press.

[41] Michael Litzkow, Miron Livny e Matt Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, Junho de 1988.

[42] MAG. `http://www.lsd.ufma.br/mag`. Último acesso em Julho/2005.

[43] MyGrid/OurGrid. `http://www.ourgrid.org`. Último acesso em Julho/2005.

[44] Klara Nahrstedt, Hao hua Chu e Srinivas Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal of High-Speed Networks, Special Issue on Multimedia Networking*, 7(3,4):229–257, Março de 1999.

[45] Anand Natrajan, Michael Crowley, Nancyc Wilkins-Diehr, Marty Humphrey, Anthony D. Fox, Andrew S. Grimshaw e Charles L. Brooks III. Studying Protein Folding on the Grid: Experiences Using CHARMM on NPACI Resources under Legion. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC '10)*, pages 14–21. IEEE Computer Society, 2001.

[46] OiL. `http://oil.luaforge.net`. Último acesso em Julho/2005.

[47] Lawrence J. Hubert Phipps Arabie e Geert De Soete, editors. *Clustering and Classification*. World Scientific, 1996.

[48] Procps. the Procps FAQ: `http://procps.sourceforge.net/faq.html`. Último acesso em Julho/2005.

[49] Manuel Román, Fabio Kon e Roy Campbell. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online*, 2(5), Julho de 2001.

[50] SETI@home. `http://setiathome.ssl.berkeley.edu`. Último acesso em Julho/2005.

[51] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[52] Weiqin Tong, Jingbo Ding e Lizhi Cai. A Parallel Programming Environment on Grid. *Lecture Notes in Computer Science*, 2657:225–234, 2003.

[53] Weiqin Tong, Jingbo Ding e Lizhi Cai. Design and Implementation of a Grid-Enabled BSP. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003. Disponível em: `http://ccgrid2003.apgrid.org/online_posters/index.html`. Último acesso em Fevereiro/2005.

[54] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[55] Derek Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor. In *Proceedings of the Linux Clusters: The HPC Revolution Conference*, Champaign - Urbana, IL, Junho de 2001.