UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

MARCELO DE CASTRO CARDOSO

MPICH-IG

Uma Infra-estrutura de Execução de Aplicações Paralelas do Tipo MPI em Grades Computacionais Oportunistas

Goiânia
2008

MARCELO DE CASTRO CARDOSO

MPICH-IG

Uma Infra-estrutura de Execução de Aplicações Paralelas do Tipo MPI em Grades Computacionais Oportunistas

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Fábio Moreira Costa

Goiânia
2008

MARCELO DE CASTRO CARDOSO

MPICH-IG

Uma Infra-estrutura de Execução de Aplicações Paralelas do Tipo MPI em Grades Computacionais Oportunistas

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Ciência da Computação, aprovada em 11 de Abril de 2008, pela Banca Examinadora constituída pelos professores:

Prof. Dr. Fábio Moreira Costa
Instituto de Informática – UFG
Presidente da Banca

Prof. Dr. Fabio Kon
Departamento de Ciência da Computação – IME-USP

Prof. Dr. Vagner José do Sacramento Rodrigues
Instituto de Informática – UFG

Marcelo de Castro Cardoso

Graduou-se em Ciência da Computação na UFG – Universidade Federal de Goiás. Durante sua graduação, foi monitor no Instituto de Informática da UFG. Após a graduação foi bolsista DTI do CNPq em um projeto envolvendo Grades Computacionais. Atualmente conclui o programa de mestrado do INF/UFG com bolsa da CAPES, desenvolvendo este trabalho voltado à programação paralela em grades. É professor substituto do Instituto de Informática da UFG e trabalha com sistemas distribuídos e soluções para a Internet.

Dedico este trabalho a minha esposa, ao meu filho e aos meus pais.

Agradecimentos

Agradeço primeiramente a Deus pela oportunidade e sabedoria na realização deste trabalho.

Aos meus pais, Nelson e Helaine, por acreditarem em mim.

À minha esposa, Sheila, pelo apoio incondicional e por fazer de mim uma pessoa melhor.

Ao meu orientador, Prof. Dr. Fábio Moreira Costa, pela orientação e parceria imprescindível na realização deste trabalho.

Aos meus colegas e amigos do GEApIS e LAVIS, pelos momentos de descontração e auxílio que impulsionaram o desenvolvimento deste trabalho.

Ao CNPq e à CAPES, pelo apoio financeiro.

A todos da secretaria do INF/UFG, que estão sempre dispostos a ajudar quando preciso.

Ao programa de mestrado do INF/UFG, pela oportunidade.

A todos que direta ou indiretamente contribuíram para a realização deste.

A melhor forma de lidar com a mudança é ajudar a criá-la.

Robert Doyle .

Resumo

de Castro Cardoso, Marcelo. **MPICH-IG: Uma Infra-estrutura de Execução de Aplicações Paralelas do Tipo MPI em Grades Computacionais Oportunistas**. Goiânia, 2008. 100p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Ambientes de grade oportunista que oferecem o serviço de execução remota devem ser preparados para executar aplicações do tipo MPI (Interface de Passagem de Mensagem), uma vez que MPI é considerado hoje um padrão de fato para construir programas paralelos. Este trabalho tem por objetivo descrever MPICH-IG, uma implementação eficiente do padrão MPI que possibilita a execução de aplicações paralelas desse tipo no InteGrade, um middleware de grade oportunista. Grades oportunistas introduzem alguns problemas não existentes em outros ambientes distribuídos, devendo a MPICH-IG ser preparada para lidar com a baixa conectividade entre os provedores de recursos, além da heterogeneidade da infra-estrutura física subjacente e do ambiente de software dos diferentes domínios. A MPICH-IG define também uma infra-estrutura tolerante a falhas utilizando *checkpointing* para garantir uma execução segura de aplicações paralelas no ambiente altamente dinâmico das grades oportunistas.

Palavras–chave

Grades oportunistas, MPI, tolerância a falhas, *checkpointing*.

Abstract

de Castro Cardoso, Marcelo. **MPICH-IG: A Infrastructure for MPI Parallel Application Performed in Scavenging Grids**. Goiânia, 2008. 100p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Scavenging grids enviroments that offer the remote execution service must be prepared to execute programs written using MPI (Message Passing Interface), which is considered today a *de facto* standard to construct parallel programs. This work discusses MPICH-IG, an eficient implementation of the MPI stadard which makes possible to execute MPI application in InteGrade grid middleware. Scavenging grids introduce some problems which do not exist in other distributed environments, such as requiring the MPICH-IG implentation to deal with unstable connectivity between resource providers, as well as the heterogeneity of both the underlying physical infrastructure and the software environment at direrent domains. MPICH-IG also defines an infrastructure for fault-tolerance using checkpointing to guarantee the safe execution of parallel applications in the very dynamic environment of scavenging grids.

Keywords

Scavenging grids, MPI, fault-tolerance, checkpointing.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

Atualmente, os computadores são utilizados para resolver diversos problemas nas mais diferentes áreas do conhecimento humano. Problemas de domínio científico e industrial, tais como, simulações geológicas e biológicas, otimizações, análise e mineração de dados, criação de efeitos especiais para a indústria de filmes, entre outros, necessitam de grande poder computacional para serem solucionados.

Há algumas décadas essas diferentes atividades eram somente realizadas em poderosos computadores dotados de grande capacidade de processamento e armazenamento. Essas máquinas eram extremamente caras, estando, portanto, restritas a algumas universidades, grandes corporações e alguns centros de pesquisa. Entretanto, mesmo essas máquinas caras um dia têm sua utilidade reduzida, principalmente devido à necessidade de maior poder computacional à medida que são realizados novos avanços no conhecimento. A solução mais evidente é a substituição do equipamento antigo por um mais avançado e, é claro, extremamente caro.

Com o crescimento de redes de computadores nas últimas décadas, principalmente em instituições públicas, originaram-se enormes parques computacionais interconectados, formados, principalmente, por tecnologias ligadas à Internet. No entanto, essas redes são geralmente utilizadas de forma simplista e ineficiente, na maioria das vezes, devido à pouca disponibilidade de técnicas que permitam o uso compartilhado e transparente de recursos na realização de computações mais complexas. A conseqüência disto é que os recursos do sistema ficam muitas vezes ociosos, ao mesmo tempo em que os usuários não obtêm uma qualidade de serviço satisfatória [66]. Além disso, desde o início da década de 90, vem sendo observado que o custo de uma máquina de alto desempenho é superior ao de uma rede local com computadores de pequeno porte interconectados, mas com poder computacional agregado equivalente.

O desenvolvimento das tecnologias de sistemas distribuídos tem permitido a exploração de diferentes soluções para a obtenção de alto poder computacional, aproveitando esses enormes parques computacionais interconectados e subutilizados. A pesquisa levou a modelos de computação distribuída chamados de Computação em Aglomerados (*Cluster Computing*) [17], onde vários computadores dedicados são interconectados por

uma rede local e trabalham em conjunto para resolver um determinado problema. Esta solução se tornou economicamente viável, uma vez que esses computadores podem ser de pequeno porte, ou seja, mais baratos.

Recentemente, surgiram novas tecnologias que passaram a interligar computadores de diferentes localizações, não se restringindo a redes locais. No final de 1995, às vésperas da conferência *Supercomputing'95* foi estabelecida a I–WAY [34], uma infra-estrutura experimental conectando 17 supercomputadores e dispositivos de armazenamento de diferentes instituições. Esta iniciativa impulsionou pesquisas em Computação em Grade (*Grid Computing*) [37].

Um sistema de computação em grade [37, 40], também chamado de *middleware*[1] de grade, pode ser definido de maneira bem abrangente como uma infra-estrutura de software capaz de interligar e gerenciar diversos recursos computacionais, possivelmente distribuídos por uma grande área geográfica, de maneira a oferecer ao usuário acesso transparente a tais recursos, independente da localização dos mesmos. Na maioria dos casos, o principal recurso das grades é a capacidade de processamento. Porém, alguns sistemas incluem outros recursos como: dispositivos de armazenamento de grande capacidade, instrumentos científicos e até componentes de software, como bancos de dados.

Sistemas de grade que combinam o processamento de várias máquinas para prover um alto poder computacional, são chamados de **grades computacionais (*computational grids*)** [23]. Estes sistemas nasceram da comunidade de Processamento de Alto Desempenho, motivados pela idéia de se utilizar computadores independentes e amplamente dispersos como plataforma de execução de aplicações paralelas [37]. As grades são ambientes propícios para a execução de aplicações paralelas, uma vez que disponibilizam uma infinidade de recursos computacionais agregados e muitas vezes subutilizados. Isto reduz a necessidade de ambientes dedicados com o fim de se conseguir poder computacional suficiente para a realização da tarefa. Sistemas de grades computacionais que aproveitam recursos ociosos para prover, por exemplo, processamento para aplicações paralelas são classificados como grades computacionais oportunistas (*scavenging grids ou high throughput computing grids* [65]).

A utilização de um ambiente computacional distribuído, onde se enquadram as grades computacionais oportunistas, não é uma tarefa totalmente transparente para o desenvolvedor de aplicações paralelas. Este deve estar consciente da distribuição sem se preocupar, entretanto, com detalhes de mais baixo nível, como comunicação entre os componentes da. Para facilitar o desenvolvimento de programas paralelos em ambientes

[1] *Middleware* é uma infra-estrutura de software que se interpõem entre o núcleo do sistema operacional e a aplicação com o objetivo de abstrair complexidades advindas da distribuição e que pouco interessam para a lógica da aplicação [25, 94, 8].

distribuídos faz-se uso de modelos de programação paralela [6, 52], tais como: MPI [49, 85], PVM [42], BSP [91], P4 [16], PARMACS [18], entre outros.

Esses modelos são implementados como bibliotecas de programação paralela, normalmente voltadas para ambientes de Computação em Aglomerados, sendo assim, inadequadas ao suporte de ambientes de grade. As grades são sistemas dinâmicos e heterogêneos, que aumentam a complexidade do ambiente e introduzem problemas como: localização e escalonamento dos recursos utilizados pela aplicação, recuperação em caso de falha de algum recurso, segurança dos dados da aplicação ao transpor diversos domínios administrativos, gerenciamento da heterogeneidade do ambiente de execução, falhas devido à interconexão por redes não dedicadas e ambientes de execução compartilhados. Uma solução para transpor estas complexidades para permitir a execução de aplicações paralelas na grade é a modificação ou a reimplementação das bibliotecas de programação paralela.

1.1 Motivação

A tecnologia de redes permitiu a interconexão de várias máquinas a fim de se conseguir um poder computacional equivalente ao de supercomputadores. Entretanto, a tecnologia de *Cluster* é inadequada quando o contexto são máquinas heterogêneas e ambientes de redes de alta escala geográfica. O uso de sistemas de grades oportunistas é bastante interessante quando o objetivo é a utilização de parques computacionais já existentes e de infra-estrutura heterogênea e não-dedicada.

O InteGrade [44] é um sistema de grade oportunista que visa atender os requisitos descritos acima: integrar computadores pessoais compartilhados de maneira a permitir a utilização de sua capacidade ociosa em tarefas de computação de alto desempenho. A vantagem do InteGrade com relação a outros sistemas de grade é o foco em estações de trabalho comuns, não necessitando de modificações drásticas no sistema pré-instalado das máquinas para se obter um ambiente de processamento paralelo. O InteGrade permite que programas de computação paralela já existentes possam usufruir de ambientes não dedicados de forma transparente aos programadores das aplicações. Isto é possível através de adaptações nas bibliotecas de programação paralela para permitir ao InteGrade escalonar e controlar a execução das aplicações.

Existe uma variedade de modelos de programação para aplicações paralelas. Entre as principais se encontram os modelos de Passagem de Mensagem (*Message Passing* [42, 49]). Modelos de passagem de mensagem provêem uma visão de mais alto nível da comunicação do que aquele usado, por exemplo, em soquetes TCP/IP, enquanto preservam, para o programador, certo nível de controle sobre como e quando ocorrerá a comunicação [33].

A biblioteca MPI (*Message Passing Interface* [85]) permite aos programadores escreverem programas de passagem de mensagem sem necessitarem conhecimento de detalhes de mais baixo nível de comunicação, tais como tipo de máquina, estrutura de rede, protocolos de baixo nível (camada de transporte), entre outros. Essa biblioteca está entre as mais completas e utilizadas na atualidade, sendo de extrema importância que projetos como o InteGrade possam prover suporte para a mesma, uma vez que outros projetos de grade há mais tempo consolidados já oferecem esse suporte [59, 75, 99].

Porém, a utilização de programas paralelos em ambientes de grade pode gerar uma diversidade de problemas para os quais as implementações comerciais das bibliotecas de programação paralela não oferecem suporte, principalmente por serem voltadas a ambientes de aglomerados [44]:

- **Comunicação**: Algumas aplicações paralelas demandam uma grande quantidade de comunicação entre os nós da aplicação, necessitando assim de redes de alta velocidade.
- **Tolerância a falhas**: O ambiente de grade é muito mais propenso a falhas que ambientes onde se trabalha com máquinas paralelas ou aglomerados dedicados. Faz-se, portanto, necessária a existência de métodos que permitam que aplicações paralelas mantenham sua execução mesmo perante falhas advindas da rede.
- ***Checkpointing***: A habilidade de salvar o estado da aplicação para garantir o seu progresso em outra máquina é uma característica importante. Isto deve estar presente em ambientes que fazem uso oportunista de recursos ociosos, como uma forma de minimizar o re-trabalho perante as constantes falhas de execução[2] das aplicações nesse ambientes.
- **Suporte a aplicações legadas**: Existem vários modelos de bibliotecas para o desenvolvimento de aplicações paralelas. Os sistemas de grade devem fornecer suporte ao maior número possível dessas bibliotecas de forma que aplicações paralelas que façam uso das mesmas não necessitem de sérias modificações para que possam rodar na grade.

Tendo em vista esta diversidade de problemas, faz-se necessária uma implementação mais robusta das bibliotecas de programação paralela, incorporando as soluções para esses problemas através do uso dos serviços de gerenciamento fornecidos pela grade. As aplicações paralelas podem então utilizar essa nova implementação da biblioteca de forma transparente, necessitando de nenhuma ou poucas alterações em seu código fonte.

[2]Para mais detalhes veja o Capítulo 4.

1.2 Objetivos

Este trabalho tem como objetivo geral possibilitar que aplicações paralelas do tipo MPI (*Message Passing Interface* [85, 49]) possam ser executadas em uma grade computacional oportunista, permitindo que aplicações que foram desenvolvidas para computação em aglomerados possam ser executadas em uma grade sem necessidade de alteração do código-fonte. Outro objetivo é a criação de um mecanismo de tolerância a falhas que permita que as aplicações se recuperem de falhas comuns em ambientes de grade oportunista, por exemplo, quando um dos processos da aplicação é interrompido durante sua execução pelo fado de o nó onde este estava executando não estar mais disponível. Isto permitirá que as aplicações não necessitem reiniciar sua execução a partir do início, melhorando a qualidade do serviço oferecido ao usuário pela grade. Estas funcionalidades estão contidas na biblioteca MPICH–IG, objeto deste trabalho.

Mais especificamente, os objetivos são:

- Definir e implementar uma arquitetura de suporte eficiente para aplicações paralelas do tipo MPI no *middleware* de grade InteGrade.
- Facilitar a execução de aplicações MPI legadas no InteGrade, dispensando alterações no código-fonte das aplicações MPI já existentes.
- Definir estratégias de tolerância a falhas das aplicações MPI para transpor problemas decorrentes do ambiente oportunista, reaproveitando ao máximo a arquitetura de recuperação presente no InteGrade.
- Avaliar estratégias elaboradas através de uma implementação para recuperação de aplicações MPI na presença de falhas através do uso de *checkpointing*.

1.3 Organização da Dissertação

Os Capítulos 2, 3 e 4 tratam de conceitos fundamentais para o entendimento do trabalho, abordando as definições e características de programação paralela e bibliotecas utilizadas para esse fim. Esses capítulos trazem, ainda, uma discussão sobre computação em grade e tolerância a falhas, contextualizando os problemas relacionados com a construção de aplicações paralelas nesse ambiente.

O Capítulo 5 aborda os trabalhos relacionados, fazendo breves comentários a respeito destes em relação à proposta apresentada nesta dissertação, considerando as diferenças e as semelhanças dos aspectos mais relevantes.

O Capítulo 6 apresenta o MPICH–IG, abordando todas as características da infraestrutura, desde uma visão geral da sua arquitetura, definição dos seus componentes, protocolos utilizados e modificados e alterações realizadas no InteGrade, até a sua imple-

mentação efetiva. O capítulo faz ainda uma breve reflexão sobre o trabalho desenvolvido, apresentado as vantagens e características que tornam a arquitetura possível de ser utilizada em grades computacionais oportunistas.

Finalmente, o Capítulo 7 traz as conclusões do trabalho desenvolvido nesta dissertação e discute trabalhos futuros que podem contribuir para uma complementação dos resultados e da própria infra-estrutura da MPICH–IG.

CAPÍTULO 2

Computação em Grade

As grades computacionais surgiram com a promessa de viabilizar a execução de aplicações variadas através da integração e gerenciamento de recursos geograficamente dispersos. Grades são sistemas desenvolvidos a partir de tecnologias de sistemas distribuídos, uma vez que devem resolver problemas ocasionados pela dispersão de recursos, tais como: comunicação transparente, localização dos recursos, falha de parte das aplicações durante a execução, sincronização das aplicações, coordenação do estado global, entre outros. De modo geral, grades são mais distribuídas, diversas e complexas que outras plataformas de sistemas distribuídos. Os aspectos que mais fortemente evidenciam esta distribuição, diversidade e complexidade são [23]:

- **Heterogeneidade**: Os componentes que constituem as grades são extremamente heterogêneos. A infra-estrutura das grades deve lidar com uma diversidade de dispositivos de hardware e com plataformas de software de vários tipos e versões.
- **Alta dispersão geográfica**: As grades tendem a fornecer serviços em uma escala global.
- **Compartilhamento**: A infra-estrutura de grade não é dedicada à execução de uma aplicação exclusiva, ou seja, há compartilhamento de recursos entre várias aplicações ao mesmo tempo.
- **Múltiplos domínios administrativos**: Pode haver diferentes políticas de acesso e uso dos serviços da Grade, segundo as diretrizes de cada instituição conectada ao sistema.
- **Controle distribuído**: Em conseqüência da alta distribuição dos componentes, as grades tendem a ter um controle descentralizado, ou seja, uma única instituição ou entidade não mantém controle total sobre todos os serviços da grade.

Note que estas características não propõem uma definição e sim um conceito para sistemas de grade. Plataformas de software que apresentam estas características certamente podem ser classificadas como grades. Contudo, a falta de alguma das características listadas acima não descaracteriza um sistema como grade.

No núcleo de um sistema de grade está o Sistema Gerenciador de Recursos (*Resource Management System* – RMS) [17], que tem a função de gerenciar os recursos de um sistema de computação distribuído [61]. O RMS é responsável por gerenciar problemas comuns em ambientes de computação distribuída como: extensibilidade, adaptabilidade, autonomia, qualidade de serviço, além de outros problemas que são mais comuns em ambientes de grade como: escalabilidade, tolerância a falhas, instabilidade dos recursos e privilégios de utilização. *Krauter et al* [61] definiram uma taxonomia para classificar os sistemas de grade conforme a atividade principal à qual se destinam:

- **Grade Computacional (*Computing Grid*)**: São sistemas de alto poder computacional que provêem serviços de processamento combinando o poder de cada máquina que compõe a grade. Esses sistemas são também chamados de grades de alto desempenho [23].
- **Grade de Dados (*Data Grid*)**: São sistemas que provêem uma infra-estrutura de armazenamento, gerenciamento e acesso a dados. Os dados são distribuídos por vários repositórios que compõem a grade, os quais são conectados por uma rede de grande área (a Internet, por exemplo).
- **Grade de Serviços (*Service Grid*)**: São sistemas que têm como foco prover uma infra-estrutura que viabilize serviços sob demanda, permitindo uma maior colaboração entre várias instituições através do compartilhamento dos seus serviços e recursos e utilizando mecanismos que viabilizem a interoperabilidade.

Uma vez que este trabalho trata da execução de aplicações paralelas e uso de processamento das máquinas conectadas à grade, vamos nos restringir a definições e trabalhos relacionados a grades de alto desempenho.

2.1 Grades de Alto Desempenho

A característica que mais evidencia as grades de alto desempenho é o fornecimento do serviço de execução remota. Este serviço permite que usuários da grade possam submeter aplicações para serem executadas utilizando os recursos disponibilizados pela grade. Uma vez que a aplicação é submetida, a grade fica responsável por escalonar os recursos a serem utilizados, gerenciar a execução da aplicação, coletar as informações geradas pela execução e enviá-las ao cliente solicitante.

Prover este serviço passa a ser uma tarefa bastante complicada, uma vez que uma série de fatores devem ser levados em consideração, tais como: disponibilidade de recursos suficientes para executar as aplicações, segurança contra aplicações maliciosas nas máquinas fornecedoras de recursos, tratamento de falhas na execução das aplicações, entre outros.

Além disso, serviços complementares devem ser implementados, como, por exemplo, o escalonamento eficiente das máquinas que executarão a aplicação. O escalonador deve escolher quais recursos serão utilizados na execução, estabelecer quais tarefas cada um destes recursos realizará e submeter solicitações aos provedores de recursos da grade apropriados para que as aplicações sejam executadas. A Figura 2.1 ilustra um cenário geral de uma grade de alto desempenho que fornece o serviço de execução remota.

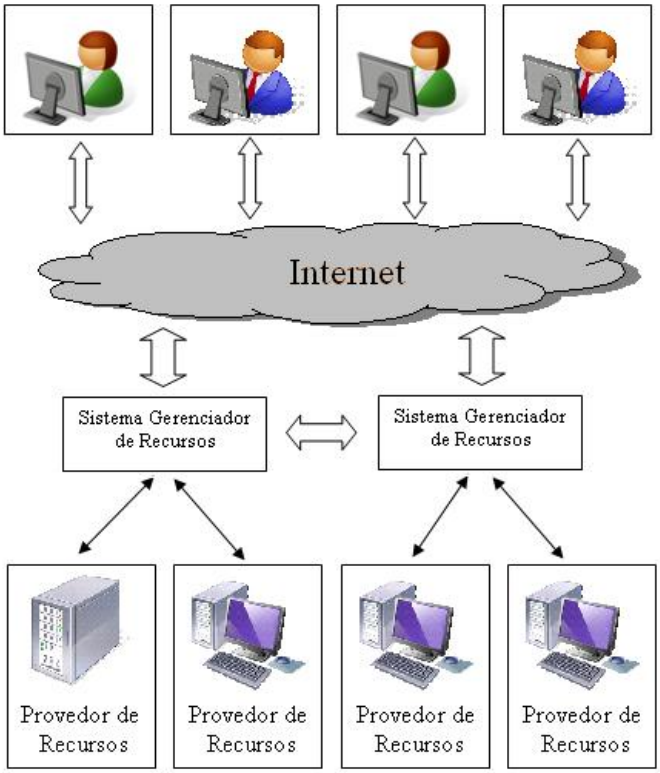


Figura 2.1: *Cenário típico de uma grade de alto desempenho*

Outro serviço importante é a previsão de desempenho antes de executar uma aplicação, tornando o escalonamento de recursos mais eficiente. Este serviço deve auxiliar o escalonador de forma a escolher recursos que minimizem o tempo de processamento ou o tempo de comunicação entre nós da aplicação, no caso de aplicações paralelas.

As grades de alto desempenho têm seu foco na execução de aplicações que exigem alto poder computacional. Grades que fazem uso oportunista dos recursos (utilizando máquinas não dedicadas) são classificadas como grades computacionais oportunistas. No ambiente de grades oportunistas os recursos podem ficar indisponíveis a qualquer momento, o que pode ocasionar falhas na execução das aplicações. Faz-se necessário, então, algum mecanismo de tolerância a falhas que não perca a computação já realizada pela aplicação, uma vez que em grades oportunistas as máquinas podem ficar indisponíveis várias vezes em um único dia.

Esses serviços fornecidos pela grade fazem parte do que denominamos meta-computação (*metacomputing* [45, 94, 35]) e, em resumo, realizam operações que dão suporte à execução das aplicações em um sistema distribuído (a grade). Entretanto, esses serviços não têm papel no resultado lógico da aplicação. Dentre as diversas iniciativas no desenvolvimento de grades de alto desempenho, pode-se destacar: Globus [43], por ser o mais conhecido e utilizado; Condor [24], pela característica oportunista na exploração de recursos; e o projeto InteGrade [56], que está diretamente ligado à implementação da MPICH–IG.

2.2 Globus

O projeto Globus [43, 35, 36] é atualmente o projeto de maior destaque na área de Computação em Grade. O sistema de computação em grade desenvolvido no contexto do projeto Globus é denominado *Globus Toolkit* [40, 32, 38] e provê um conjunto de bibliotecas e programas que oferecem suporte à arquitetura e ao desenvolvimento de aplicações para o Globus. O *Globus Toolkit* compreende um conjunto de componentes que implementam serviços básicos para segurança de acesso, descoberta, alocação e gerência de recursos, comunicação, entre outros [36].

Atualmente em sua versão 4.0, o *Globus Toolkit* sofreu várias mudanças arquiteturais. A principal evolução nas últimas versões foi o uso abrangente de *Web Services* [10] na definição de interfaces dos componentes da arquitetura e utilização de OGSA (Open Grid Services Archtecture) [38] como padrão para criação dos serviços. *Web Services* definem um mecanismo de comunicação flexível, extensível e largamente adotado, baseado em documentos XML, a qual é utilizada em serviços de comunicação, descoberta e RPCs (*Remote Procedure Calls*) [51].

2.2.1 Arquitetura

A arquitetura do *Globus Toolkit* 4 é ilustrada na Figura 2.2, descrevendo três conjuntos de componentes [32]:

- Um conjunto de serviços úteis que compõem a infraestrutura básica. Estes serviços são destinados à gerência de execução (GRAM), acesso e movimentação de dados (GridFTP, RFT, OGSA–DAI), gerenciamento de réplicas (RLS, DRS), monitoramento e descoberta (Index, Trigger, WebMDS), gerenciamento de credenciais (MyProxy, Delegation, SimpleCA), gerenciamento de instrumentos e simulações (GTCP). A maioria destes componentes são implementados em Java.

- Três ***containers***[1] que podem ser usados como hospedeiros para serviços desenvolvidos pelo usuário. Estes fornecem implementação de segurança, descoberta e gerenciamento, entre outros, nas linguagens Java, C e Python.
- Um conjunto de bibliotecas que permitem a programas clientes em Java, C ou Python utilizarem serviços do *Globus Toolkit* ou outros serviços implementados pelo usuário.

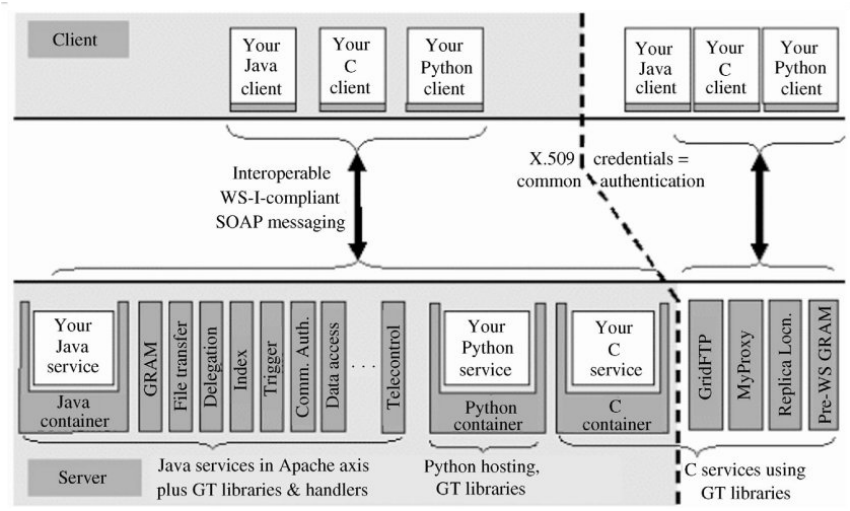


Figura 2.2: *Arquitetura do Globus Toolkit 4: interação entre os componentes [32]*

Os componentes do conjunto de ferramentas podem ser usados, tanto independentemente quanto em conjunto, no desenvolvimento de aplicações e de ferramentas para grades. Os componentes mais utilizados são:

- GRAM – *Grid Resource Allocation Manager*: é o serviço responsável por iniciar, monitorar e gerenciar a execução de aplicações nos computadores remotos. O GRAM fornece uma interface baseada em *web services* para o cliente, a qual permite informar parâmetros e restrições de execução.
- RFT – *Reliable File Tranfer*: é o serviço responsável pelo gerenciamento e tolerância a falhas na transferência de arquivos entre nós da grade usando o GridFTP.
- WebMDS – *Web Monitoring and Discovery Service*: o Globus mantém um conjunto de serviços e ferramentas que fazem a descoberta, publicação e acesso a informações sobre os recursos da grade. Essas informações podem ser coletadas através de uma interface de *web services* disponibilizada pelo WebMDS.

[1] *Containers* são objetos de software que permitem armazenar outros objetos de um mesmo tipo a fim de fornecerem uma interface comum para acesso dos mesmos.

- GSI – *Grid Security Infrastructure* [62, 96, 39]: é uma infra-estrutura de componentes que fornece serviços de segurança para o Globus. Os serviços são oferecidos através de: mensagens de autenticação, criação de CAS (*Community Authorization Service*) para controle de acesso dinâmico, funcionalidades de segurança fornecidas pelo padrão OGSA [38], padrões de autorização XACML (*eXtensible Access Control Markup Language*) e SAML (*Security Assertion Markup Language*) [74].

Atualmente, o Globus é o projeto de maior impacto na pesquisa em grades computacionais, principalmente devido à utilização de padrões no contexto de grades, tornando-o uma importante referência de estudo. Na prática, esta ferramenta é amplamente utilizada na integração de *clusters* através da Internet. Entretanto, por ser uma ferramenta bastante robusta, necessita que vários pacotes e serviços sejam instalados para disponibilizar os recursos da grade. Além disso os recursos gereciados pelo Globus geralmente são dedicados, o que significa que o *toolkit* não possui uma API que forneça suporte nativo para grades oportunistas.

2.3 Condor

Condor [24] é um sistema desenvolvido na Universidade de *Wisconsin–Madison/USA*, o qual tem como foco fornecer grande poder computacional através da integração de diversos computadores em uma infra-estrutura de aglomerados. Este sistema foi desenvolvido para permitir a execução de aplicações de alto desempenho que necessitam de alto poder computacional e podem ficar várias horas, ou até dias, executando [63].

O sistema Condor faz uso dos recursos ociosos das máquinas conectadas à infra-estrutura da grade, o que caracteriza esse sistema como grade computacional oportunista. No escalonamento oportunista das aplicações, o sistema assume que os recursos não são dedicados e podem não estar disponíveis por todo o tempo de execução da aplicação [99]. Condor faz uso extensivo de *checkpointing*, onde a aplicação é suspensa e pode ser reiniciada em outra máquina caso o recurso atual fique indisponível[2].

Quando uma aplicação é submetida para execução no Condor, o sistema escolhe quando e onde executar essa aplicação baseando-se em alguma política pré-definida. O sistema então monitora todos os processos gerados nas máquinas provedoras de recursos para informar o usuário quando a tarefa estiver completada [7]. Este recurso de submissão de aplicações é formado por serviços de gerenciamento de aplicações, políticas de escalonamento, planejamento de prioridades, monitoramento e gerenciamento de recursos [88].

[2]Para mais detalhes sobre *checkpointing* veja o Capítulo 4.

2.3.1 Arquitetura

A infra-estrutura do sistema Condor é organizada como um conjunto de aglomerados, onde cada aglomerado é estruturado em um conjunto de *daemons* chamado de *Condor pool* (ver Figura 2.3). Em um *Condor pool* todos os provedores de recursos devem enviar informações ao *daemon* chamado *Collector*. O *Collector* é o repositório central de informações do sistema Condor, para onde quase todos os *daemons* enviam informações de atualização periodicamente. Essas informações são enviadas na forma de uma mensagem composta de uma estrutura chamada *ClassAd*, definindo atributos específicos da entidade remetente. O *daemon Collector* deve, portanto, executar no nó de gerenciamento central (*Central Manager*).

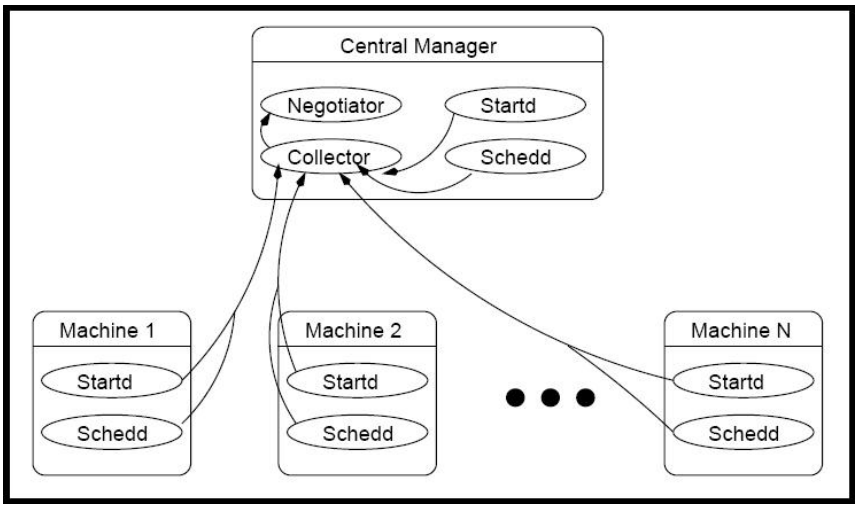


Figura 2.3: *Arquitetura do Condor pool [99]*

No *Central Manager* também é executado o *daemon Negotiator*, que periodicamente executa um ciclo de negociação. Nesse ciclo, o *Negotiator* procura entre os vários *ClassAds*, particularmente os de solicitação de recursos e disponibilidade de recursos, quais são compatíveis entre si. Ao encontrar um par solicitação/disponibilidade, são enviadas notificações aos envolvidos.

No provedor de recursos é executado o *daemon startd*, que tem a função de monitorar as condições dos recursos locais e publicar recursos e políticas de uso através do envio de *ClassAds*. As aplicações dos usuários são representadas no sistema Condor como um *ClassAd* e podem ser submetidas para execução ao *daemon schedd*. Este *daemon* mantém uma lista de tarefas a serem executadas, publica requisições de recursos *ClassAds* e negocia por recursos disponíveis. O *schedd* recebe as notificações do *Negotiator* e envia *ClassAds* aos *startd* para controle temporário dos seus recursos.

2.4 InteGrade

O projeto InteGrade [56] 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 [44]. A criação de uma infra-estrutura de software que permita utilizar efetivamente esses recursos ociosos possibilitaria uma economia financeira para as instituições que demandam grandes quantidades de computação.

O InteGrade é, portanto, considerado um sistema de grade oportunista e tem como principais objetivos:

- Fazer uso intensivo de Orientação a Objetos e padrões de projeto: permite modularizar a implementação do sistema.
- Dar suporte a diversos tipos de aplicações paralelas: possibilita a diversificação do uso do sistema, principalmente para aplicações de alto desempenho.
- Manter o desempenho das aplicações locais executando nas máquinas fornecedoras de recursos: viabiliza o uso oportunista de recursos nas máquinas conectadas à grade.
- Analisar e monitorar os padrões de uso das máquinas provedoras de recursos: possibilita ao sistema fazer o escalonamento mais eficiente das aplicações, uma vez que contará com estimativas de quanto uma máquina da grade permanecerá ociosa.

O InteGrade foi desenvolvido para permitir a implementação de aplicações para resolver uma ampla gama de problemas paralelos. Aplicações paralelas tradicionais demandam redes proprietárias de interconexão entre os computadores paralelos que estão executando a aplicação, possuindo, portanto, dependências entre seus nós. Essas aplicações não estão preparadas para ser executadas em ambientes dinâmicos, necessitando de mecanismos de correção de erros ou tolerância a falhas fornecidos pela infra-estrutura do InteGrade.

2.4.1 Arquitetura

A arquitetura do InteGrade é orientada a objetos e cada módulo do sistema se comunica com os demais a partir de chamadas de método remotos, utilizando CORBA [76] como sua infra-estrutura de objetos distribuídos.

A utilização de CORBA se justifica por uma série de vantagens [44, 76]:

- Facilita a implementação da comunicação entre os módulos, uma vez que abstrai a interação entre os objetos através de chamadas de método remotos.

- Permite o desenvolvimento para ambientes heterogêneos, facilitando a integração de módulos escritos nas mais diferentes linguagens, executando sobre diversas plataformas de hardware e software.
- Fornece serviços de transações, persistência, nomes e *trading*, facilitando a implementação dos módulos.

A unidade estrutural básica de uma grade InteGrade é o aglomerado (*cluster*), cujas máquinas podem ou não pertencer a uma mesma rede ou a um mesmo domínio administrativo. O InteGrade pode ser composto por vários aglomerados, formando uma federação de aglomerados interconectados. A Figura 2.4 apresenta os elementos de um aglomerado InteGrade, onde cada máquina pertencente ao aglomerado é chamada de nó. A seguir, são descritos os quatro tipos de nós de um aglomerado:

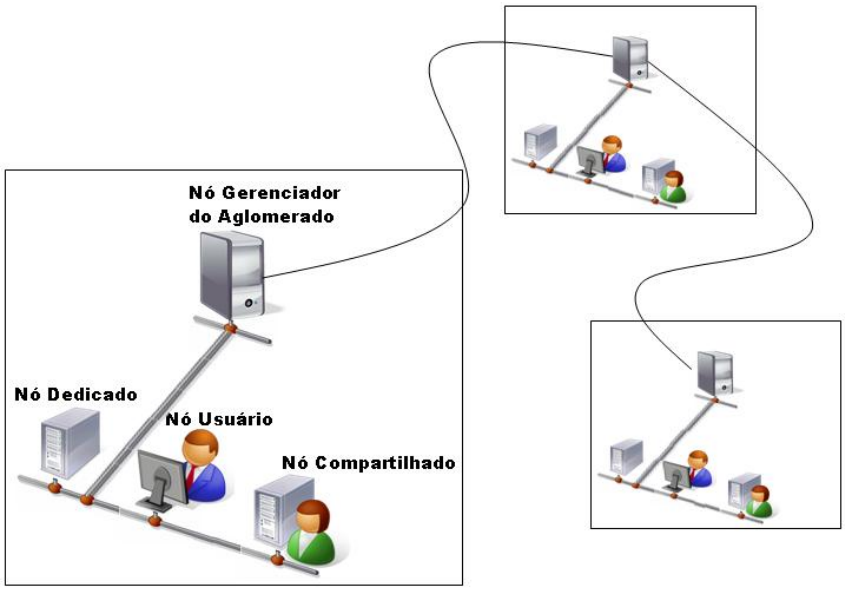


Figura 2.4: *Federação de Aglomerados do InteGrade*

- **Nó Dedicado**: uma máquina reservada exclusivamente para a computação em grade.
- **Nó Compartilhado**: uma máquina pertencente a um usuário que disponibiliza seus recursos ociosos à grade. Nós compartilhados e dedicados são chamados de nós provedores de recursos.
- **Nó de Usuário**: uma máquina que tem a capacidade de submeter aplicações para serem executadas na grade.
- **Nó Gerenciador de Aglomerado**: uma máquina onde são executados os módulos responsáveis pela concentração de informações sobre recursos disponíveis, pelo escalonamento de atividades e pelo gerenciamento de recursos do aglomerado.

Em cada nó do aglomerado InteGrade são executados módulos do sistema responsáveis pela execução de diversas tarefas necessárias à grade. A Figura 2.5 ilustra os componentes de software existentes em cada nó, os quais têm as seguintes funções [44]:

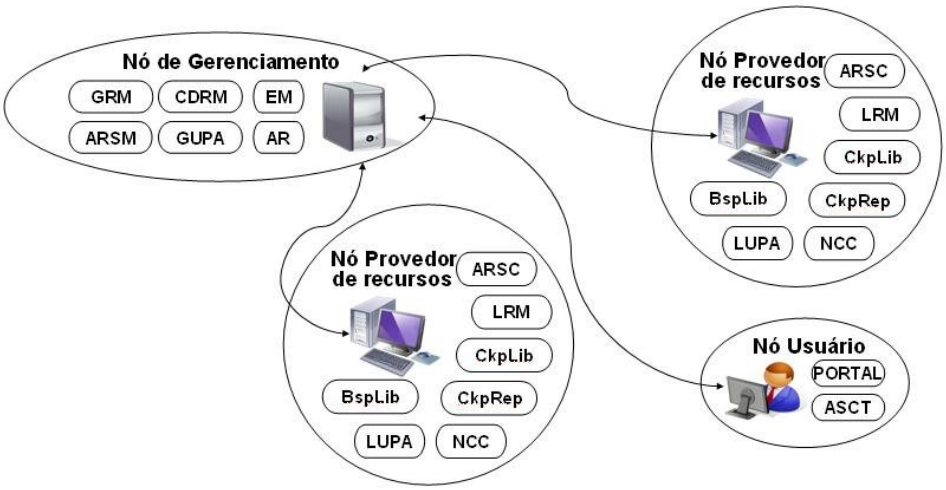


Figura 2.5: *Componentes de software de um aglomerado InteGrade*

- LRM (*Local Resource Manager*): tem a função de disponibilizar informações sobre os recursos ociosos da máquina para a grade, além de aceitar requisições para a execução de aplicações da grade na máquina. Ao iniciar, o LRM se conecta ao gerenciador do aglomerado e espera por requisições de execução.
- GRM (*Global Resource Manager*): tem a função de gerenciar os recursos do aglomerado, escalonar aplicações e manter uma lista dos LRMs ativos. O GRM recebe informações periódicas dos LRMs sobre recursos disponíveis e escalona aqueles que atendem as solicitações de execução de aplicações específicas. Este módulo é executado no nó gerenciador do aglomerado.
- AR (*Application Repository*): tem a função de armazenar as aplicações que serão executadas na grade. O LRM faz o *download* da aplicação a partir do AR após a solicitação para sua execução [78].
- ARSM (*Application Repository Security Manager*): faz o gerenciamento de segurança das aplicações que executam em um aglomerado através de assinaturas digitais do código das aplicações armazenados no AR, encriptação de dados, autenticação e autorização [56].
- ARSC (*Application Repository Security Client*): implementa a segurança na comunicação entre os componentes do InteGrade; em particular, permite ao LRM fazer acesso seguro ao AR para obter o binário das aplicações submetidas [56].

- CDRM (*Checkpointing Data Repository Manager*): mantém informações sobre os repositórios de *checkpoints* do aglomerado. Esses dados são fornecidos à biblioteca de *checkpointing* no momento de armazenar ou recuperar *checkpoints* de aplicações [19].
- CkpLib (*Checkpoint Library*): possibilita a realização de *checkpoints* das aplicações que executam em um nó provedor de recursos. Este se comunica com o CDRM para obter a lista de repositórios disponíveis para armazenar ou recuperar *checkpoints* [19].
- CkpRep (*Checkpoint Repository*): armazena os dados de *checkpoint* e arquivos de saída gerados pela aplicação que são enviados e solicitados pelo CkpLib. Os vários CkpRep em um aglomerado são gerenciados pelo CDRM. Este módulo executa em nós provedores de recursos [19].
- LUPA (*Local Usage Pattern Analyzer*): utiliza as informações de uso coletadas pelo LRM, derivando categorias comportamentais de cada nó da grade. Os resultados da análise possibilitam estimar padrões de uso dos nós da grade, auxiliando no escalonamento eficiente das aplicações.
- GUPA (*Global Usage Pattern Analyzer*): é responsável por armazenar as informações enviadas pelo LUPA e auxiliar o GRM no escalonamento.
- EM (*Execution Manager*): é responsável por gerenciar uma aplicação através do monitoramento de todos os nós que a executam. Este módulo mantém uma lista de todas as aplicações rodando no aglomerado, faz notificações quanto ao término da execução e toma ações de recuperação em caso de falhas [19].
- BSPLib (*BSP Library*): biblioteca que fornece uma API para execução de aplicações paralelas do tipo BSP (*Bulk Synchronous Parallelism*) no InteGrade.
- NCC (*Node Control Center*): este módulo permite ao usuário criar políticas de disponibilidade de recursos [56].
- ASCT (*Application Submission and Control Tool*): este módulo permite ao usuário submeter aplicações para execução na grade, monitorar sua execução e coletar as informações resultantes dessa execução. Este módulo é executado no nó de usuário.
- PORTAL: permite realizar as mesmas operações que o ASCT, com a diferença de fornecer este serviço através de um *portal web* [19].

O InteGrade utiliza dois ORBs que interoperam para prover comunicação entre módulos escritos em linguagens diferentes. O OIL [50], um ORB CORBA desenvolvido em linguagem LUA, permite a interoperabilidade de componentes desenvolvidos em linguagem C (LRM e módulos do provedor de recursos), uma vez que LUA provê *binding*

para a linguagem C. O outro ORB é o JacORB [57], desenvolvido em linguagem Java, que trabalha com componentes escritos em Java (ASCT e componentes do nó gerenciador do aglomerado).

O InteGrade permite a execução de três tipos de aplicações: aplicações seqüenciais, aplicações paramétricas e aplicações paralelas do tipo BSP. As aplicações seqüenciais são aplicações comuns, com um único arquivo executável, sendo necessário somente uma máquina para a execução. Aplicações paramétricas são aplicações onde um mesmo binário é executado em várias máquinas com parâmetros diferentes. As aplicações paralelas do tipo BSP são classificadas como SPMD (*Single Program, Multiple Data*), em que um mesmo binário é executado em várias máquinas com os mesmos parâmetros, mas havendo comunicação entre os processos para coordenação. A execução e recuperação de aplicações do tipo BSP é discutida em seguida, contextualizando o uso de aplicações paralelas no InteGrade.

2.4.2 Protocolo de Execução de Aplicações BSP

O protocolo de execução de aplicações do InteGrade, exibido na Figura 2.6, define como uma aplicação paralela do tipo BSP é submetida para execução. Aplicações paralelas necessitam localizar os nós que participam da execução da aplicação, de forma a estabelecer comunicação entre os diversos processos paralelos.

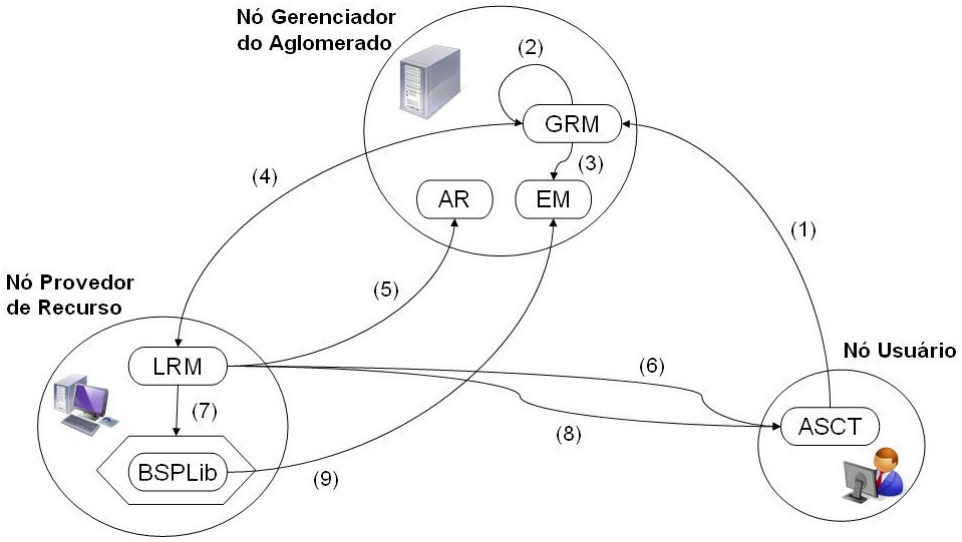


Figura 2.6: *Protocolo de execução de aplicações BSP no InteGrade*

Após o registro prévio da aplicação BSP no AR pelo ASCT, o cliente da grade pode solicitar a execução dessa aplicação (1); o GRM então procura por nós que atendam

aos requisitos desta aplicação[3], tomando como base as informações coletadas dos LRMs (2); após definir quais são os recursos necessários, o GRM aciona o EM para coordenar a aplicação que será iniciada (3) e envia uma mensagem aos LRMs confirmando a disponibilidade dos recursos (4); após aceitar a execução, os LRMs fazem o *download* da aplicação BSP a partir do AR (5); em seguida, os LRMs fazem a solicitação dos arquivos de entrada ao ASCT (6) e lançam a aplicação (7), notificando a execução ao ASCT (8); no início da execução, a BSPLib se encarrega de registrar cada nó junto ao EM (9), possibilitando a sincronização entre todos os nós (a figura foi simplificada para ilustrar apenas um nó provedor de recursos). A partir deste ponto a BSPLib se encarrega de coordenar a comunicação dos processos paralelos [44].

2.4.3 Protocolo de Recuperação de Aplicações BSP

O InteGrade provê suporte à recuperação de aplicações em caso de falhas ocasionadas pela perda de algum dos recursos. Essa perda pode ter ocorrido por falha do provedor de recursos ou porque o dono da máquina solicitou os recursos de volta. Para não perder a computação já realizada, o mecanismo de recuperação de aplicações foi desenvolvido. Esse mecanismo trabalha com recuperação por retrocesso utilizando *checkpointing* (Figura 2.7).

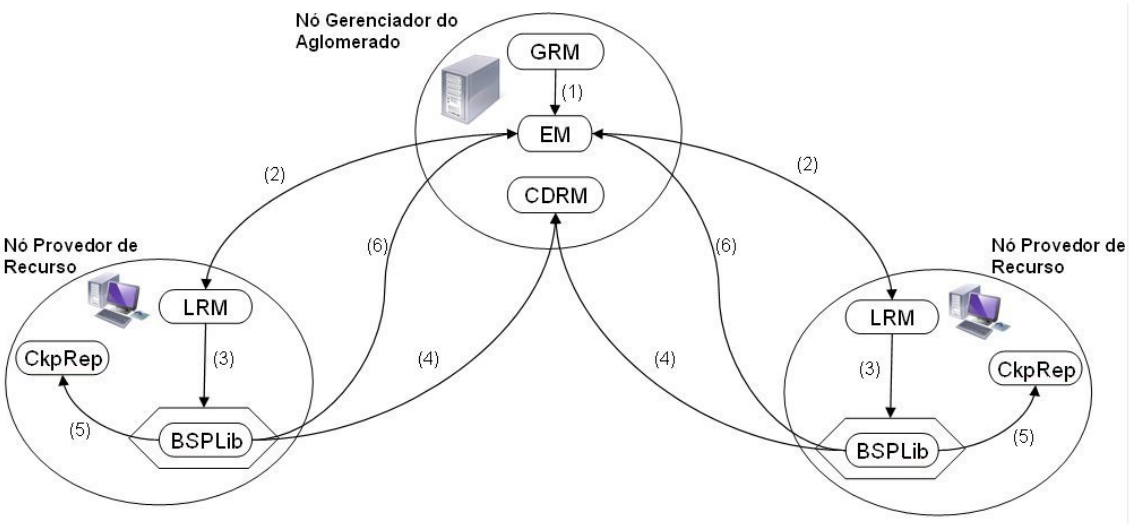


Figura 2.7: *Protocolo de recuperação de aplicações BSP no InteGrade*

Após o EM identificar a falha de algum processo da aplicação (1), este envia uma mensagem aos LRMs solicitando o reinício de todos os processos (2 e 3); a BSPLib envia uma mensagem ao CDRM solicitando uma lista dos repositórios de *checkpointing*

[3]Um contrato de solicitação e provisão de recursos entre a aplicação e a grade.

(CkpRep) (4); em seguida a BSPLib busca no CkpRep o *checkpoint* para recuperar o processo (5) e registra-se novamente junto ao EM (6), como acontece no passo (9) da Figura 2.6. A figura ilustra a busca por *checkpoints* apenas no CkpRep local, mas esta pode ser realizada em qualquer nó da lista obtida no passo (4) [19].

2.5 Comentários

O Globus é atualmente a grade mais completa e de maior impacto na pesquisa de tecnologias de grade. Entretanto, tem o foco na integração de ambientes dedicados diferentemente dos outros projetos discutidos: Condor e InteGrade. O InteGrade diverge do Condor, entre outras coisas, por fazer análise de uso dos provedores de recurso de forma a melhorar o escalonamento perante a execução de aplicações de larga escala. Esta característica permite ao InteGrade fornecer ao usuário uma melhor qualidade de serviço na execução de aplicações em grades oportunistas. O Condor fornece serviços de execução remota para aplicações paralelas do tipo MPI e PVM, e o InteGrade somente para BSP. Esses tipos de aplicações paralelas são discutidos no próximo capítulo.

Um considerável esforço de pesquisa tem sido dedicado à solução de problemas relacionados a grades computacionais e aplicações paralelas, bem como à integração entre ambos. A maioria dos sistemas de grade existentes disponibilizam o serviço de execução remota de aplicações paralelas. O que diverge entre estes projetos são os modelos de programação paralela suportados e os recursos fornecidos pela grade para obter um melhor desempenho na execução destas aplicações. No próximo capítulo são apresentados conceitos relevantes sobre programação paralela, descrevendo alguns projetos que envolvem o uso de grades computacionais.

CAPÍTULO 3

Programação Paralela

Programação paralela consiste, basicamente, em criar aplicações que executam partes de um mesmo programa simultaneamente em processos diferentes [6]. Este tipo de programação tornou-se viável com o surgimento de sistemas operacionais multitarefa, *multi-threaded* e paralelos, sendo que os programas podem ser executados em ambientes com um processador, ou em máquinas multiprocessadas, ou até mesmo em um grupo de máquinas interligadas por redes públicas ou privadas (sistemas de computação distribuída, por exemplo: *clusters* e grades).

A execução dessas aplicações em máquinas com um único núcleo de processamento é denominada pseudoparalelismo ou concorrência. Nesse tipo de ambiente não há um paralelismo real, mas sim, uma troca de contexto da aplicação, provida pelo sistema operacional, de forma a executar os vários processos ou *threads* por divisão de tempo. Em máquinas multiprocessadas o ambiente de execução provê um paralelismo real através de hardware paralelo específico [6].

Um sistema de computação distribuído consiste de múltiplos processadores autônomos que não compartilham uma memória principal, mas cooperam enviando mensagens de comunicação pela rede para se coordenarem [87, 26]. O ambiente de sistemas distribuídos é caracterizado pelo fraco acoplamento, imprevisibilidade devido a falhas e atrasos de comunicação, e controle distribuído. Neste contexto, um programa paralelo deve implementar mecanismos que coordenem sua execução através da troca de mensagens pela rede.

A Figura 3.1 exemplifica a arquitetura MIMD (*Multiple Instruction stream, Multiple Data stream*), que representa ambientes de computação distribuída. Cada processador tem uma memória local e executa parte do programa, acessando dados na memória de outras máquinas indiretamente através de comunicação pela rede. Este processo de comunicação pode se dar de duas maneiras [84]:

- Passagem de Mensagem (*Message passing* – MP): a obtenção de dados em memórias remotas é realizada através do envio explícito de mensagens de solicitação.

- Acesso Direto à Memória Remota (*Direct remote memory access* – DRMA): implementa a funcionalidade de memória compartilhada distribuída, em que os processos determinam locais de memória que podem ser acessados por outros processos remotos. O acesso é coordenado implicitamente de forma a sincronizar acessos concorrentes.

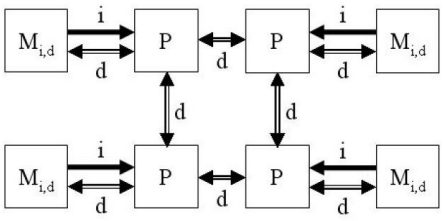


Figura 3.1: *Arquitetura MIMD: M=memória, P=processador, i=instruções e d=dados*

Implementar e manter este tipo de coordenação no desenvolvimento de aplicações paralelas pode ser uma tarefa bem difícil. Para resolver este problema faz-se necessário o uso de modelos de programação paralela, que definem uma interface de programação com operações de mais alto nível. Estes modelos definem uma API que elimina complexidades de implementação na coordenação do sistema paralelo, simplificando a estrutura do software e reduzindo a dificuldade de implementação [84].

Os modelos de programação paralela utilizados em arquiteturas MIMD são os baseados em SPMD (*Single Program Multiple Data*) e MPMD (*Multiple Program Multiple Data*). Os modelos de programação SPMD e MPMD mais encontrados na literatura são: PVM [42], BSP [91] e MPI [49, 85]. Estes modelos são implementados na forma de bibliotecas, existindo, portanto, várias implementações de um mesmo modelo para diferentes arquiteturas de hardware e software. Programas SPMD são utilizados, principalmente, pela simplicidade de depuração e desenvolvimento. Neste modelo, um mesmo programa é executado em máquinas diferentes. Entretanto, partes diferentes do programa são executadas simultaneamente em cada máquina, manipulando também, dados diferentes.

3.1 *Bulk Synchronous Parallelism*

O modelo *Bulk Synchronous Parallelism* (BSP) [91] foi desenvolvido com o objetivo de criar programas paralelos de propósito geral nas mais diferentes arquiteturas. Isto é possível uma vez que sua interface é simplificada, contendo algumas poucas funções que podem facilmente serem implementadas em plataformas e arquiteturas variadas.

A característica fundamental do modelo BSP é o desacoplamento entre comunicação e sincronização dos nós da aplicação paralela. Isto é realizado através de uma

seqüência de super-passos, onde cada supre-passo é constituído de três fases ordenadas (Figura 3.2): (1) computações simultâneas de processos em cada máquina, utilizando somente valores gravados na memória local; (2) comunicação entre os processos para troca de valores; (3) barreira de sincronização que espera que todas as comunicações sejam terminadas para então iniciar outro super-passo [54].

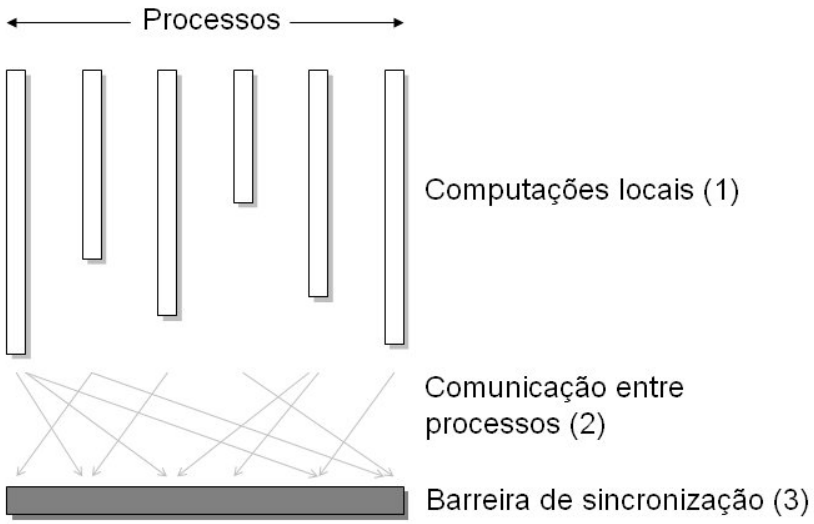


Figura 3.2: *Super-passo do modelo BSP*

Outra característica do modelo BSP é a previsibilidade quanto ao desempenho da aplicação, que pode ser calculado através de métricas obtidas de certa arquitetura: número de processadores, tempo necessário para realizar a barreira de sincronização entre eles, e a taxa entre as velocidades da rede e processador [83]. Esta previsibilidade é possível em arquiteturas de sistemas distribuídos estáticas. Ambientes altamente distribuídos, como grades computacionais, tornam esta previsibilidade mais complicada.

A interface do modelo BSP define as funções *bsp_begin* e *bsp_end*, que são responsáveis por iniciar e finalizar, respectivamente, uma aplicação BSP. As barreiras de sincronização são definidas explicitamente através da função *bsp_sync*. A interface do modelo BSP especifica dois modos de comunicação, discutidos anteriormente: *Direct Remote Memory Acess* (DRMA) e *Message Passing* (MP) [54].

Para utilizar o DRMA, os processos devem publicar endereços de memória local como endereços virtuais compartilhados. Isto é feito com a chamada da função *bsp_push_reg*. Outros processos então podem acessar esse espaço de memória compartilhado através das funções *bsp_put* e *bsp_get* para ler e escrever, respectivamente, no endereço remoto. A função *bsp_pop_reg* faz o desregistro de uma memória previamente compartilhada. O Código 3.1, em linguagem C, exemplifica o uso das funções DRMA utilizando a interface BSP. Este programa calcula a média de valores contidos em um arquivo, sendo que cada processo fará o cálculo em parte do arquivo.

Código 3.1: *Programa BSP utilizando DRMA*

```
1  #include<bsp.h>
2  int main( int argc, char **argv ){
3      int n_processos = 4, pid;
4      double media, *lista_de_medias;
5
6      lista_de_medias = (double*) malloc( sizeof(double)*n_processos );
7      bsp_begin( n_processos );
8      pid = bsp_pid();
9      bsp_push_reg( &lista_de_medias, sizeof(double)*n_processos );
10     media = calcularMediaDeParteDoArquivo( pid, n_processos );
11
12     if ( pid != 0 )
13        bsp_put( 0, &media, &lista_de_medias, pid, sizeof(double) );
14     else
15        lista_de_medias[0] = media;
16
17     bsp_sync(); // barreira de sincronização
18     if ( pid == 0 ) {
19        media = calculaAMediaFinal(lista_de_medias, n_processos);
20        printf("A média dos valores do arquivo é: %d\n", media);
21     }
22     bsp_end();
23 }
```

Este programa é um exemplo de SPMD em BSP. A identificação de quais trechos de código serão executados por quais processos é definida pela função *bsp_pid* (linha 8), que devolve um número seqüencial e diferente para cada processo. O uso de DRMA é indicado quando na maior parte do tempo são realizadas computações locais e, eventualmente alguma sincronização, uma vez que esse mecanismo demanda muita comunicação para sincronizar dados de memórias remotas.

No caso de programas que demandam muita comunicação, a melhor opção é utilizar MP através de funções chamadas de *Bulk Synchronous Message Passing* (BSMP), em que as mensagens a serem trocadas são definidas de forma explícita. A função não-bloqueante *bsp_send* é responsável por enviar uma mensagem para outro processo. Esta mensagem é transportada para o processo de destino durante a fase de comunicação do super-passo. A mensagem pode ser acessada no destino através da função *bsp_move*.

Foram desenvolvidas diversas implementações do modelo BSP, entre as quais se destacam: BSPLib de Oxford [54], PUB [9], BSP–G [95], implementada para o Globus (Seção 2.2), e BSPLib implementada para o InteGrade [44] (Seção 2.4).

A barreira de sincronização imposta pelo modelo facilita a implementação da biblioteca em diferentes arquiteturas, inclusive a criação de protocolos coordenados de

checkpointing, uma vez que a barreira de sincronização é realizada periodicamente, o que facilita a obtenção de estados consistentes. Entretanto, esta sincronização periódica também apresenta desvantagens. Sincronizações muito freqüentes podem levar à perda de desempenho da aplicação, ainda mais acentuada em ambientes de rede como a Internet, onde os atrasos de comunicação são significativos. A barreira de sincronização impõe uma restrição ao desenvolvimento de aplicações que possuem muita dependência entre os processos paralelos, ou seja, demandam muita comunicação.

3.2 *Message Passing Interface*

O modelo *Message Passing Interface* (MPI) [49, 85] é bastante aceito na comunidade e tornou-se um padrão *de facto* para computação paralela. O objetivo principal de MPI é definir uma interface que seja amplamente utilizada no desenvolvimento de programas paralelos baseados em passagem de mensagem. O desenvolvedor não deve se preocupar com detalhes de mais baixo nível, tais como tipo de máquina, estrutura da rede, protocolos de baixo nível, entre outros [33].

A interface de MPI deve fornecer ao programador praticidade, portabilidade, eficiência e flexibilidade. Para tanto, foram estabelecidas as seguintes metas, as quais o padrão deveria alcançar [70]:

- permitir o uso em ambientes heterogêneos;
- permitir comunicação eficiente: evitar cópias de memória-para-memória e possibilitar computação e comunicação simultaneamente;
- criar uma interface portável para várias plataformas, sem a necessidade de sérias modificações;
- criar uma interface que seja independente da linguagem hospedeira; e
- criar uma interface que seja *thread–safe*[1].

A primeira versão do padrão MPI foi publicada em 1995, chamada de MPI–1 [70]. Esta versão disponibiliza um conjunto extenso de funções de comunicação na tentativa de atingir a meta de flexibilização no desenvolvimento de aplicações paralelas. Para tanto, define funções para:

- Comunicação ponto-a-ponto: possibilita o envio de mensagens entre dois processos, podendo ser bloqueantes ou não-bloqueantes, o que determina se o processo receptor ou remetente ficará bloqueado enquanto a operação de comunicação não terminar. A comunicação não-bloqueante permite ganho de desempenho da aplicação,

[1] Ser *Thread–safe* significa que estão sendo utilizados mecanismos para controlar o acesso simultâneo de *threads* a um mesmo espaço de memória, evitando inconsistência dos dados [92].

uma vez que os processos podem continuar realizando suas computações enquanto a comunicação é realizada. A comunicação ponto-a-ponto permite ao usuário definir o modo de comunicação, que especifica quando uma operação de comunicação terminou:

  - *default*: necessita que o receptor execute uma operação de recebimento para determinar o término.
  - *synchronous*: espera uma confirmação de que o receptor processou a mensagem.
  - *ready*: envio mais eficiente onde o receptor já espera a mensagem sem necessidade de confirmação.
  - *buffering*: a mensagem é colocada em um *buffer* e o sistema define quando é conveniente enviá-la.

- Definição de topologias e grupos de processos: permite hierarquizar e definir grupos de processos, aos quais são atribuídos identificadores diferentes (*comunicators*), restringindo o escopo de operações de sincronização e comunicação.
- Comunicação coletiva: possibilita o envio de mensagens para um grupo de processos (*comunicator*). Essas mensagens podem ser de vários tipos: *broadcast* - envio de dados de um para todos os processos do grupo; *gather* - envio de dados de todos para um; *scatter* - dados diferentes em um processo são enviados para todos; *allgather* - dados de todos são enviados para todos; *alltoall* - dados diferentes de todos são enviados para todos; *barrier* - mensagem de sincronização entre os processos; *reduction* - permite a combinação de dados de vários processos.
- Consulta: operações para obter informações sobre o estado da execução dos vários processos.

A segunda versão do padrão MPI foi publicada em 1997, chamada de MPI–2 [71]. Esta versão incluiu extensões substanciais, onde as principais são:

- DRMA: rotinas de comunicação com memória compartilhada distribuída, utilizando operações do tipo *get* e *put*, para ler e escrever, respectivamente, em memória remota.
- *Spawned process*: criação de processos dinâmicos, onde processos podem ser adicionados ou removidos convenientemente, durante a execução da aplicação.

O modelo MPI apresenta um conjunto extenso de funções, que objetiva flexibilizar o desenvolvimento. À primeira vista, pode parecer complexo o desenvolvimento de programas em MPI. Entretanto, a maioria dos problemas que demandam computação paralela necessita de apenas seis dessas funções, denominadas de conjunto básico

de operações MPI (Tabela 3.1). O Código 3.2, em linguagem C, exemplifica o uso destas funções através de um programa SPMD simplificado que envia uma mensagem de um processo para outro através de operações bloqueantes ponto-a-ponto (*MPI_Send* e *MPI_Recv*). Estas funções utilizam um manipulador de comunicação como argumento (*MPI_COMM_WORLD*), o qual identifica o grupo de processos e o contexto das operações a serem executadas.

Tabela 3.1: *Conjunto Básico de Funções MPI*

Função	**Descrição**
MPI_Init	Inicia os mecanismos responsáveis pela coordenação do programa MPI.
MPI_Finalize	Libera recursos utilizados pelo MPI.
MPI_Comm_rank	Obtem o *rank* que identifica o processo dentre um grupo de processos.
MPI_Comm_size	Obtem o número de processos do grupo.
MPI_Send	Envia uma mensagem, identificada por uma *flag*, para um determinado processo.
MPI_Recv	Recebe uma mensagem, identificada por uma *flag*, de um determinado processo.

Código 3.2: *Programa MPI com send e receive bloqueante*

```
1  #include <mpi.h>
2  main( int argc, char **argv ) {
3      char message[20];
4      int myrank,n_processos, cont, tag = 1;
5      MPI_Status status;
6      MPI_Init( &argc, &argv );
7      MPI_Comm_size( MPI_COMM_WORLD, &n_processos );
8      MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
9      if (myrank == 0) {     /* codigo para processo zero */
10         sprintf(message,"Ola, sou o processo 0.");
11         for (cont = 1; cont < n_processos; cont++)
12             MPI_Send(message, strlen(message), MPI_CHAR, cont, tag,
                   MPI_COMM_WORLD);
13     }
14     else {                 /* codigo para processo um */
15         MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
16             &status);
17         printf("processo %d recebeu: %s:\n", myrank, message);
18     } MPI_Finalize();
19 }
```

Existem várias implementações públicas e comerciais da biblioteca MPI, dentre elas: Verari MPI/Pro e ChaMPIon/Pro [93], Scali MPI Connect [81], CHIMP/MPI [1],

LAM/MPI [14], OpenMPI [90] e MPICH [48]. A maioria das implementações citadas são comerciais e específicas para uma plataforma de execução. A MPICH se destaca por ser pública e por permitir maior facilidade quanto à portabilidade para outras plataformas. A MPICH é discutida em maiores detalhes na próxima seção.

3.3 MPICH

A biblioteca de programação paralela MPICH [48] foi desenvolvida pelo *Argonne National Laboratory* em conjunto com *Missisipi State University* e implementa todas as funcionalidades especificadas no padrão MPI. Um dos objetivos da MPICH foi criar uma biblioteca eficiente que, ao mesmo tempo, pudesse ser facilmente portada para outras plataformas. O "CH"em MPICH vem de "*Chameleon*", símbolo desta adaptabilidade.

O desenvolvimento da MPICH começou paralelamente com a definição da especificação MPI, como uma forma de avaliar o cumprimento das metas já estabelecidas para o padrão. A MPICH2 [4] é a última versão desta implementação, refletindo a especificação MPI–2, versão mais recente do padrão.

3.3.1 Arquitetura

O padrão MPI é bastante extenso, dificultando a tarefa de portá-lo diretamente para várias plataformas. Uma solução viável foi definir um arcabouço que minimize o impacto ao portar uma implementação para outra plataforma. A MPICH alcançou as metas de portabilidade e eficiência com a criação de uma arquitetura estruturada em camadas. Na arquitetura da MPICH todas as funções MPI são implementadas em termos de macros e funções definidas nas camadas de mais baixo nível. O mecanismo principal desta estrutura em camadas é uma interface chamada de *Abstract Device Interface* (ADI) [47].

A ADI especifica as operações necessárias para criar os protocolos de comunicação utilizados pelas funções definidas no padrão MPI. Através de múltiplas implementações desta interface é possível portar a biblioteca para diferentes plataformas de hardware e software, chamadas de *devices*. Uma vez que a implementação da ADI é específica para uma determinada arquitetura de comunicação, espera-se um ganho natural de desempenho com a utilização de recursos específicos da plataforma. A ADI também facilita a portabilidade, uma vez que sua implementação é pequena comparada com a implementação completa do padrão MPI. A Figura 3.3 ilustra a estrutura em camadas da MPICH2, incluindo a ADI–3, versão mais recente da ADI [52, 4].

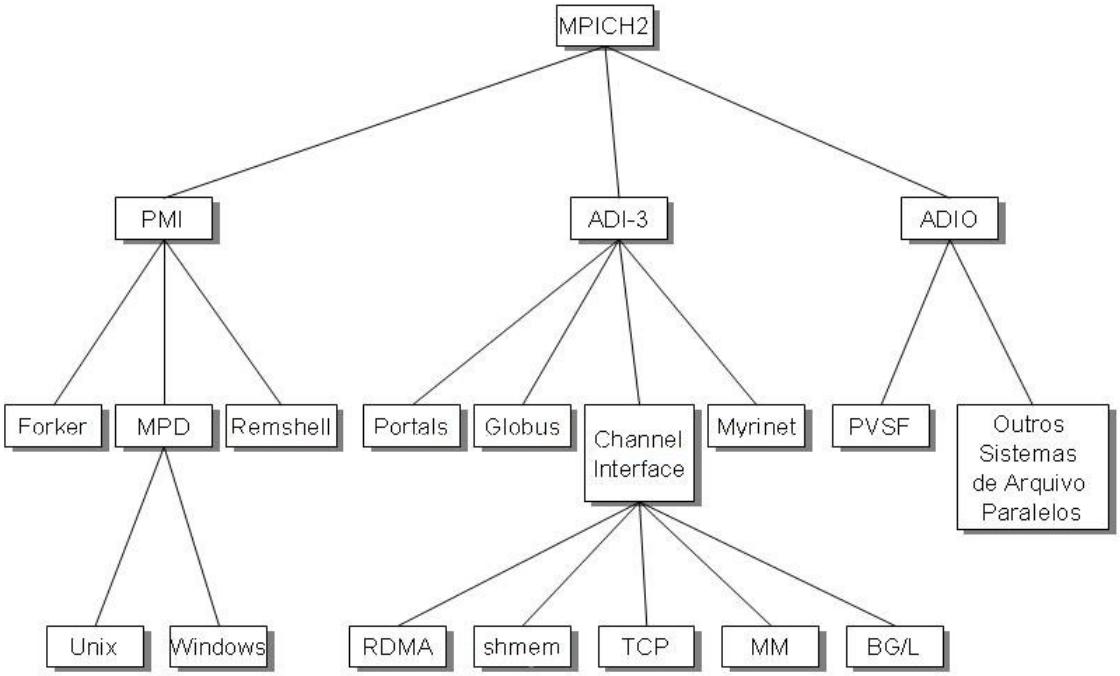


Figura 3.3: *Estrutura em camadas da MPICH2*

Uma das implementações da ADI–3, que acompanha a implementação padrão da MPICH2, é chamada de CH3 e foi desenvolvida com o objetivo de minimizar ainda mais o conjunto de operações a serem implementadas para portar a biblioteca. Esta implementação define outra camada de mais baixo nível denominada *Channel Interface* (CI), que especifica funções que fornecem capacidades básicas de envio de dados de um processo para outro, tais como: *read*, *write* e *select*. A CI possibilita portar rapidamente a MPICH2 para diversas plataformas com a implementação de poucas funções [5, 46].

Outra camada que provê facilidades de portabilidade para a MPICH2 é a *Process Manager Interface* (PMI) [3]. A PMI é uma interface que fornece definições para possibilitar o gerenciamento de comunicação entre os processos paralelos de um programa SPMD. O gerenciamento de comunicação tem como objetivos definir a localização dos processos participantes, estabelecer um ambiente de compartilhamento de dados necessário para a comunicação (por exemplo, endereço IP e porta no caso de comunicação via TCP/IP), negociar a criação de processos dinâmicos, coletar resultados da computação da aplicação, entre outros.

A implementação da interface PMI, que acompanha a MPICH2 padrão, é o *Multipurpose Daemon* (MPD) [15]. Deve haver haver uma instância do MPD executando em cada máquina onde serão executadas aplicações MPI. No momento da solicitação de execução (utilizando geralmente o comando *mpiexec*) estes *daemons* fazem o compartilhamento de dados necessário para iniciar o processo de comunicação da aplicação e coletam os fluxos de saída que são enviados para um único processo, o que solicitou a execução. A

PMI é utilizada de forma vertical entre as camadas de comunicação da MPICH2, podendo ser acessada quando necessária por qualquer um dos níveis de comunicação.

Finalmente, a última camada de destaque na arquitetura MPICH2 é a interface *Abstract I/O Device* (ADIO) [89], que define operações para abrir, fechar, ler e escrever em arquivos paralelamente (compartilhamento de arquivos). A Figura 3.4(a) ilustra a organização das camadas (*interfaces*) da MPICH2, comparando-as com suas respectivas implementações padrão (*default*), Figura 3.4(b).

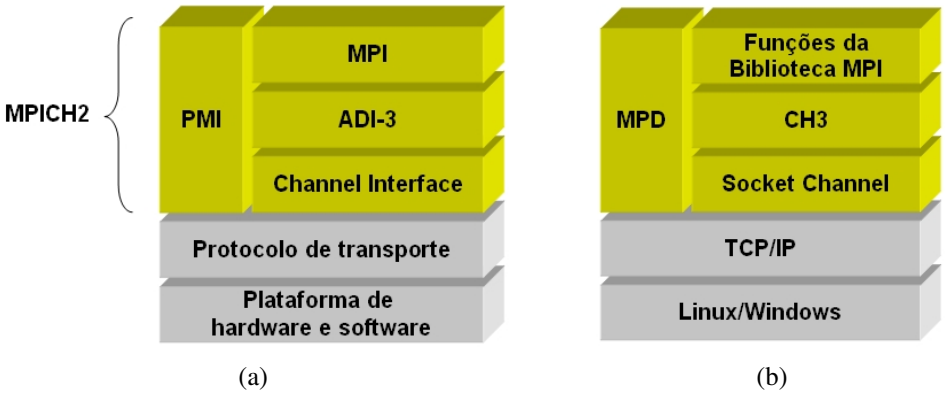


Figura 3.4: *Disposição das interfaces da MPICH2* (a) *e suas respectivas implementações* (b)

Existem implementações de MPICH para várias plataformas, como por exemplo: MPICH–GM para *Myrinet* [73], MVAPICH para *InfiniBand* [72], MPICH–G2 para o Globus [59, 68], MPI–SCTP para SCTP[2] [67], entre outros. Também existem implementações da MPICH para prover outros serviços, tais como NEXUS [33], para gerenciamento de heterogeneidade em redes de grande área, e MPICH–V [69, 13], que define uma implementação tolerante a falhas utilizando *checkpointing*. As implementações de maior relevância para este projeto são a MPICH–G2 e a MPICH–V, apresentadas na Seção 5.1.1, por se tratarem de implementações específicas para grades computacionais e tolerantes a falhas.

3.4 Aplicações Paralelas em Grades Computacionais

As grades computacionais são ambientes adequados para a execução de aplicações paralelas, uma vez que interconectam uma infinidade de dispositivos com recursos de processamento. Além disso, uma grade cria um ambiente transparente para o usuário submeter aplicações e coletar resultados das execuções (ver Seção 2.4.2). Neste contexto,

[2] *Stream Control Transmission Protocol*

grades computacionais oportunistas possibilitam o aproveitamento de recursos de processamento já existentes nas organizações para computação de alto desempenho. Esta característica resulta em economia financeira, uma vez que não necessitam de parques computacionais dedicados para executar as aplicações paralelas.

Outra vantagem é a aplicabilidade no caso de programas paralelos escaláveis[3], que não têm resultados satisfatórios em ambientes limitados, como é o caso da maioria dos *clusters*, pois para entradas muito grandes as aplicações não podem usufruir extensivamente de recursos como forma de manter o desempenho. Neste contexto, o ambiente de grade disponibiliza recursos em larga escala, possibilitando um uso extensivo de recursos. Os maiores problemas quanto à execução de aplicações paralelas no contexto de grades computacionais oportunistas, no qual se enquadra o projeto InteGrade, são:

- localização: uma vez que as grades tendem a ser bastante distribuídas, fazem-se necessários mecanismos de publicação e descoberta de dados para iniciar a comunicação entre os nós de uma aplicação;
- comunicação: necessidade de mecanismos que permitam o estabelecimento de comunicação entre diversos domínios administrativos, inclusive transpondo *firewalls* e serviços de NAT, os quais são bastante comuns; e
- tolerância a falhas: a instabilidade e a alta dispersão dos recursos, que freqüentemente ficam indisponíveis, causam falhas constantes na execução das aplicações.

Para transpor estes problemas, os sistemas de passagem de mensagem devem ser capazes de negociar com diversos gerenciadores de recursos, gerenciadores de processos, e serviços de segurança para executar programas que integrem diversos domínios administrativos. Entretanto, a implementação padrão da MPICH embora seja portável, eficiente e forneça algum suporte para ambientes heterogêneos, provê suporte limitado para ambientes de metacomputação em larga escala, como grades computacionais [33].

Um programa desenvolvido a partir de um modelo de programação paralela utiliza uma biblioteca que implementa o modelo, limitando o programa ao uso de serviços definidos pela biblioteca. Por exemplo, um programa que utiliza o modelo MPI e a biblioteca MPICH está limitado aos serviços fornecidos pela MPICH. Sendo assim, para adicionar novos serviços necessários ao programa que irá executar na grade faz-se necessário modificar ou reimplementar a biblioteca. Essa estratégia é ilustrada na Figura 3.5.

A nova biblioteca implementa a especificação do modelo de programação original (1), muitas vezes sem alterações, mantendo, para o desenvolvedor de aplicações

[3]Um programa é dito escalável quando sua eficiência (tempo de execução T) se mantêm constante à medida que a entrada do problema (N) e o número de processadores (P) crescem proporcionalmente, ou seja, $T = f\left(\frac{P}{N}\right)$, onde f é uma função linear.

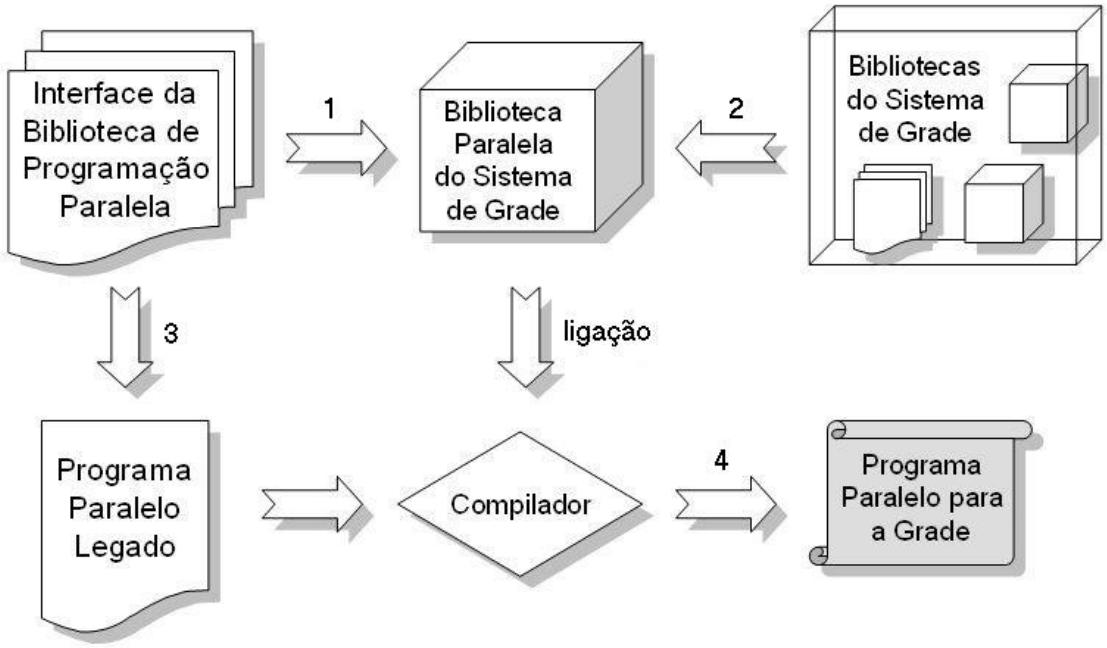


Figura 3.5: *Estratégia para utilizar programas paralelos legados (implementados para ambientes de cluster) em ambientes de grade*

paralelas, transparência no uso da grade. A implementação utiliza também as bibliotecas e serviços fornecidos pela grade (2) para resolver problemas decorrentes da execução do programa em larga escala. Programas paralelos legados desenvolvidos a partir do modelo original (3) podem ser recompilados e/ou ligados à nova biblioteca, gerando programas paralelos para a grade computacional (4). Esta estratégia é a base da proposta deste trabalho, sendo discutida em maiores detalhes no Capítulo 6.

3.5 Comentários

A biblioteca MPICH fornece uma implementação portável e de alto desempenho do padrão MPI para ambientes heterogêneos. Entretanto, ela é limitada para uso em ambientes distribuídos como as grades computacionais. Redes de grande área introduzem uma série de novos problemas para a implementação de sistemas de passagem de mensagem, necessitando novas implementações das bibliotecas de programação paralela. A MPICH pode ser utilizada como base de desenvolvimento para muitas outras implementações, com diferentes plataformas e arquiteturas.

A MPICH–IG, proposta neste trabalho, é uma implementação que usa como base a MPICH2, reimplementando alguns módulos para possibilitar a execução de aplicações MPI legadas em um ambiente de grade computacional oportunista, o InteGrade. A MPICH–IG prevê também estratégias de tolerância a falhas como forma de melhorar

o desempenho das aplicações MPI, um requisito importante especialmente em ambientes oportunistas. No próximo capítulo, são discutidos alguns fundamentos de tolerância a falhas, em especial a recuperação por retrocesso, que é utilizada para execução confiável de aplicações MPI no ambiente de grade.

CAPÍTULO 4

Tolerância a Falhas

Um sistema de grade oportunista é caracterizado pelo ambiente instável quanto à comunicação e disponibilidade de recursos. As aplicações não têm garantias quanto à interrupção da execução ou quanto a falhas na comunicação dos processos paralelos. Se uma única falha ocorre, todos os outros processos são finalizados, necessitando ser re-executados e perdendo toda a computação já realizada [97]. Aplicações paralelas em ambientes de grade oportunista podem falhar devido à interrupção na execução de algum processo paralelo (falha parcial), devido ao usuário local requisitar de volta os recursos fornecidos à grade, devendo a aplicação ser migrada para outro provedor de recurso, ou ainda devido a falhas de comunicação ou à baixa conectividade em redes de grande área.

Para aumentar a confiança do sistema de grade, mecanismos de tolerância a falhas são indispensáveis. Um sistema é dito tolerante a falhas se ainda continua funcionando corretamente na presença de falha de algum processo da aplicação, permitindo ao programa distribuído continuar sua execução e aos usuários continuarem usando o sistema [6]. As principais estratégias de tolerância a falhas são: recuperação através de réplicas e recuperação por retrocesso.

A recuperação através de réplicas é fundamentada na criação de várias cópias de um mesmo processo paralelo, onde, no caso de falha do processo principal, uma das cópias se torna o novo processo principal. Esta estratégia demanda uma maior quantidade de recursos disponíveis, além de um sistema de coordenação sofisticado para monitorar as mensagens trocadas entre os processos paralelos, a fim de evitar duplicatas. O mecanismo de recuperação por retrocesso, que é a estratégia adotada neste trabalho, é discutido a seguir.

4.1 Recuperação por Retrocesso em Aplicações Paralelas

No contexto de aplicações paralelas de passagem de mensagem, um estado global é uma coleção de estados individuais de todos os processos participantes e dos estados dos canais de comunicação. Um estado global consistente é o conjunto de estados dos processos que definem um ponto alcançável da execução, o qual pode ser utilizado

para recuperar consistentemente a aplicação, por exemplo, após a ocorrência de falhas na computação distribuída. O estado de um processo contém os valores de memória necessários para a execução do processo, a pilha de chamadas de funções, o ponteiro para próxima instrução, os recursos utilizados pela aplicação, como sockets e arquivos abertos, e as mensagens em trânsito no canal de comunicação entre os processos paralelos. Esse conjunto de dados é denominado *checkpoint*, e o conjunto de *ckeckpoints* determina uma linha de recuperação, como observado na Figura 4.1.

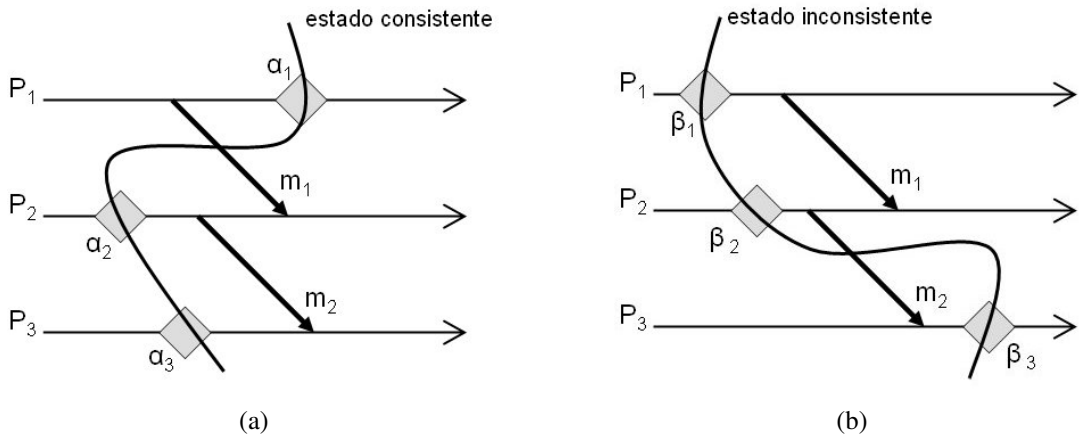


Figura 4.1: *Definição de estado consistente* (a) *e estado inconsistente* (b) *na obtenção do estado global de uma aplicação paralela*

A linha de recuperação é consistente se define um estado de execução alcançável pela aplicação, e inconsistente caso contrário. A Figura 4.1(a) ilustra um estado global consistente, que é determinado pelos estados individuais $\alpha_1, \alpha_2, \alpha_3$ e m_1, em que a mensagem m_1 não foi recebida por P_2, devendo ser enviada no momento de uma possível recuperação. Na Figura 4.1(b), por outro lado, o estado global é definido por $\beta_1, \beta_2, \beta_3$ e m_2, onde, segundo a linha de recuperação, a mensagem m_2 foi recebida pelo processo P_3 mas não foi enviada pelo processo P_2, o que causará inconsistência durante uma possível recuperação [28].

O objetivo fundamental de protocolos de recuperação por retrocesso é trazer o sistema para uma linha de recuperação consistente quando inconsistências ocorrerem devido a alguma falha. As técnicas de recuperação por retrocesso baseadas em *checkpoint* podem ser classificadas em: não-coordenadas, induzidas por comunicação e coordenadas [28].

4.1.1 Protocolos não-coordenados

Um protocolo de *checkpointing* não-coordenado (ou assíncrono) permite que processos tenham a máxima autonomia para decidir quando fazer seus *checkpoints*. A

principal vantagem desta autonomia é a escolha de momentos convenientes, por exemplo, quando as informações a serem salvas forem as menores possíveis. Mas esta abordagem também tem sérias desvantagens. Primeiro, devido à dependência entre os processos paralelos, pode não ser possível determinar a linha de recuperação tão facilmente, causando um retrocesso muito grande e perdendo muito trabalho já realizado. Segundo, por não haver um limite quanto ao número de *checkpoints* necessários para recuperar a aplicação até o último estado global consistente, mecanismos de coleta de lixo devem ser executados periodicamente para diminuir a quantidade de dados armazenados.

4.1.2 Protocolos induzidos por comunicação

Um protocolo de *checkpointing* induzido por comunicação (ou quase-síncrono), tem o mesmo objetivo dos protocolos não-coordenados: permitir que os processos realizem o *checkpoint* quando acharem conveniente. Entretanto, os processos realizam *checkpoints* extras para garantir um menor retrocesso em caso de falhas. Esses *checkpoints* adicionais são realizados no momento do envio ou recebimento de mensagens, que são a causa das dependências entre os processos. Uma desvantagem desses protocolos é a grande quantidade de *checkpoints* gerados, inviabilizando seu uso para aplicações que demandam muita comunicação.

4.1.3 Protocolos coordenados

Os protocolos de *checkpointing* coordenados (ou síncronos) têm como característica principal a coordenação entre os processos para realizar *checkpoints* consistentes. Essa coordenação se dá através do envio de mensagens pela rede como uma forma de sincronizar a geração dos *checkpoints* em todas as máquinas. Uma vantagem desses protocolos é o pequeno retrocesso necessário no caso de falhas, utilizando sempre o último *checkpoint* gerado. Outra vantagem é que qualquer *checkpoint* que não seja o último pode ser descartado, minimizando as operações de coleta de lixo.

A maior desvantagem desta abordagem é o *overhead* adicional para sincronizar a geração dos *checkpoints*, uma vez que o mesmo tende a crescer com o aumento no número de processos paralelos. Nesse sentido, são definidas duas estratégias para protocolos de coordenação, que podem ser bloqueantes ou não-bloqueantes.

Protocolo coordenado bloqueante

O processo de sincronização nos protocolos coordenados ocorre quando um dos processos (eleito o coordenador) envia uma mensagem aos demais solicitando a realização de um *checkpoint*. Nesse momento, os processos bloqueiam o envio de mensagens

e esperam dos outros processos uma mensagem que determina o momento de realizar o *checkpoint*. Esta estratégia é denominada bloqueante porque impede o envio de mensagens durante a sincronização. Isto pode causar degradação no desempenho da aplicação, uma vez que um processo pode ficar em espera devido a uma mensagem bloqueada em outro processo durante o período de sincronização.

Uma vantagem desta estratégia é a facilidade quanto à implementação do protocolo, uma vez que não é necessário realizar *checkpoint* do canal de comunicação. Outra vantagem é observada quando o protocolo é aplicado em ambientes com tecnologias de rede local de alta velocidade, em que o processo de sincronização é bem rápido, não causando perda de desempenho significativa à aplicação.

Protocolo coordenado não-bloqueante

O problema fundamental na coordenação de *checkpointing* é prevenir que um processo receba mensagens de outro processo durante a sincronização, o que poderia gerar inconsistências. Isto acontece quando um processo que já gerou o *checkpoint* envia uma mensagem para outro processo que ainda não gerou seu *checkpoint* (este é o caso da mensagem m_2 da Figura 4.1(b)).

Uma alternativa aos protocolos bloqueantes é a utilização de protocolos não-bloqueantes, que evitam a inconsistência do estado global através do monitoramento do canal de comunicação. Um exemplo de protocolo coordenado não-bloqueante e que trabalha sobre um canal de comunicação FIFO (*First In, First Out*) é o *distributed snapshot* [21]. Neste protocolo, um inicializador grava um *checkpoint* e envia uma mensagem (*marker*) a todos os outros processos. Quando um processo recebe a mensagem, ele realiza o *checkpoint* e a repassa novamente para todos os outros processos. O processo de sincronização só termina quando um processo recebe de volta a mensagem de todos os outros processos.

Durante o período de sincronização, o processo não fica proibido de enviar mensagens. Entretanto, ele deve armazenar todas as mensagens recebidas, as quais também são incluídas no *checkpoint* (*checkpoint* do canal de comunicação). Esta estratégia não bloqueia a comunicação durante a sincronização e tem como desvantagem somente o *overhead* de comunicação da mensagem de *marker*. Um fator agravante neste protocolo é a implementação um tanto complexa, pois ele deve executar um mecanismo de monitoramento do canal de comunicação para realizar o *checkpoint* das mensagens recebidas, além de determinar em que momentos essas mensagens devem ser repassadas para o processo durante a recuperação (*message logging protocol*).

4.2 Abordagens para *Checkpointing*

Basicamente, existem dois tipos de abordagem de tolerância a falhas usando técnicas de *checkpoint*: o *checkpoint* no nível de sistema, onde o *checkpoint* é criado a partir de uma cópia do espaço de memória do processo; e o *checkpoint* no nível de aplicação, onde o programa determina que dados precisam ser armazenados para garantir um estado recuperável.

4.2.1 *Checkpoint* no nível de sistema

Checkpoint no nível de sistema geralmente é implementado como um módulo do núcleo do sistema operacional, o que possibilita o acesso direto à memória do processo e a valores que determinam o estado do sistema [102, 53, 77]. O acesso ao núcleo facilita amplamente a realização de *checkpoint* de forma otimizada, gerando menos sobrecarga, além de fornecer uma recuperação automática de aplicações transparentemente ao programador da aplicação.

A maior vantagem desta abordagem é que a biblioteca de *checkpoint* não conhece a estrutura semântica das aplicações, que são vistas como uma "caixa–preta", não necessitando conhecimento sobre características internas da aplicação. Como resultado, é possível utilizar a mesma biblioteca para gerar *checkpoints* de qualquer aplicação, sem que o programador se preocupe com o mecanismo.

A maior desvantagem desta abordagem é que, geralmente, a biblioteca precisa ser desenvolvida para uma versão específica do sistema operacional, o que impossibilita a portabilidade do *checkpoint* gerado. Algumas bibliotecas de *checkpoint* no nível de sistema não fazem uso de privilégios do modo protegido para acessar as estruturas do sistema operacional [64, 79], o que aumenta a portabilidade da biblioteca para diferentes sistemas operacionais. Em contrapartida, esta técnica limita o conjunto de chamadas de sistema que a aplicação pode realizar, restringido o uso para algumas aplicações específicas [82].

Um exemplo de biblioteca bastante utilizada é a *Berkeley Lab Checkpoint/Restart* (BLCR) [53] implementada para Linux, cuja principal meta é ser utilizada para aplicações de computação de alto desempenho, em particular aplicações MPI. BLCR é implementada para as versões 1.4.x e 2.6.x do núcleo do Linux em arquiteturas x86 e x86–64, cobrindo a maioria das plataformas utilizadas atualmente.

4.2.2 *Checkpoint* no nível de aplicação

Nesta abordagem, a aplicação é responsável por fornecer os dados necessários para criar o *checkpoint*, além recuperar a aplicação a partir de um *checkpoint* gerado [12].

A aplicação deve fornecer mecanismos que possibilitem obter seu estado, como: persistência de dados, *logging* e monitoramento.

A maior vantagem desta abordagem é a portabilidade do *checkpoint* gerado. Uma vez que a aplicação fornece os dados necessários para o *checkpoint*, ela pode também fornecer informações semânticas sobre os mesmos. Essas informações podem ser utilizadas como forma de portar os dados para outros sistemas, tornando o *checkpoint* independente de plataforma.

Uma desvantagem desta abordagem é que prover os mecanismos de obtenção e recuperação do estado da aplicação pode ser uma tarefa bastante complicada para o programador e bastante sujeita a erros. Uma solução para este problema é utilizar pré-compiladores que realizam instrumentação de código, inserindo o código necessário para a obtenção e recuperação do estado da aplicação de forma transparente para o programador [58]. Entretanto, o uso do pré-compilador limita a aplicação quanto à linguagem utilizada e impossibilita a realização de *checkpoints* em pontos arbitrários da execução do programa, sendo estes definidos pelo pré-compilador [19].

4.3 Comentários

Neste capítulo foram apresentados os princípios gerais de tolerância a falhas, e os principais conceitos relacionados com a recuperação por retrocesso de aplicações paralelas em ambientes distribuídos. A possibilidade de manter a aplicação executando perante a falhas, sem a interferência do usuário, é necessária para fornecer uma melhor qualidade de serviço em ambientes de grades oportunistas.

A MPICH–IG apresenta uma arquitetura que permite a implementação de estratégias de tolerância a falhas através de *checkpoints* em nível de sistema. A implementação de recuperação por retrocesso utilizando *checkpoints* está fundamentada em um protocolo de coordenação de *checkpoint* bloqueante e em uma biblioteca responsável por gerar e recuperar o estado da aplicação. O próximo capítulo descreve alguns trabalhos relacionados com a implementação da MPICH–IG.

CAPÍTULO 5

Trabalhos Relacionados

Neste capítulo são discutidos trabalhos encontrados na literatura que relacionam MPI, grades computacionais e mecanismos de recuperação por retrocesso. São abordadas, também, definições e características que contextualizam este trabalho em meio aos demais.

5.1 MPI em grades computacionais

5.1.1 MPICH–G2

A MPICH–G2 [68] é uma implementação do modelo MPI que permite aos usuários executarem programas MPI em múltiplos computadores localizados em diferentes domínios administrativos e conectados por uma infra-estrutura de grade. Essa biblioteca estende a implementação da MPICH, desenvolvida sobre o padrão MPI–1, e usa serviços do *Globus Toolkit* (Sessão 2.2) para autenticação, autorização, alocação de recursos e E/S, além da criação, monitoramento e controle de processos [59].

A MPICH–G2 é uma implementação do módulo ADI da arquitetura MPICH, representada atualmente na MPICH2 pela ADI–3 (Figura 3.4(a)). Como discutido na Sessão 3.3, a ADI define funções de comunicação em mais baixo nível, sendo que as funções da biblioteca MPI são desenvolvidas como macros sobre esse módulo. A MPICH–G2 reimplementa as funções de comunicação para prover um suporte eficiente e transparente na execução de aplicações sobre o ambiente heterogêneo da grade. Para atingir a meta de eficiência em ambientes heterogêneos, o canal de comunicação deve utilizar, quando possível, algum mecanismo de comunicação específico fornecido pela arquitetura de cada aglomerado (por exemplo, IBM SP, *shared memory*, InfiniBand, SCTP, BG/L, etc). Em casos onde não há uma arquitetura de comunicação específica um canal de soquetes TCP/IP é utilizado como padrão (*default*).

A Figura 5.1 ilustra o mecanismo de execução de aplicações MPI utilizando a MPICH–G2. A infra-estrutura da MPICH–G2 utiliza os mecanismos e serviços já existente no *Globus Toolkit* para gerenciar e monitorar os problemas advindos da utilização

de ambientes de grade. No início da execução, o usuário obtém junto ao *Grid Security Infrastructure* (GSI), uma chave pública que autentica o usuário para uso dos serviços remotos. O usuário então utiliza o *Monitoring and Discovery Service* (MDS) para selecionar as máquinas que farão a execução da aplicação. O usuário usa o comando padrão da MPICH *mpirun* para realizar a autenticação e descrever os requisitos da aplicação, que são mapeados para o módulo *globusrun*, que escalona a aplicação entre os computadores selecionados utilizando o *Dynamically–Updated Request Online Coallocator* (DUROC). O DUROC fica responsável por negociar com os vários *Grid Resource Allocation and Management* (GRAM) a alocação de recursos dentro dos aglomerados. Após definidas as máquinas, a MPICH–G2 determina que métodos de comunicação são mais adequados para as mesmas. O GRAM obtém a aplicação e arquivos de entrada através do *Global Access to Secondary Storage* (GASS) [59].

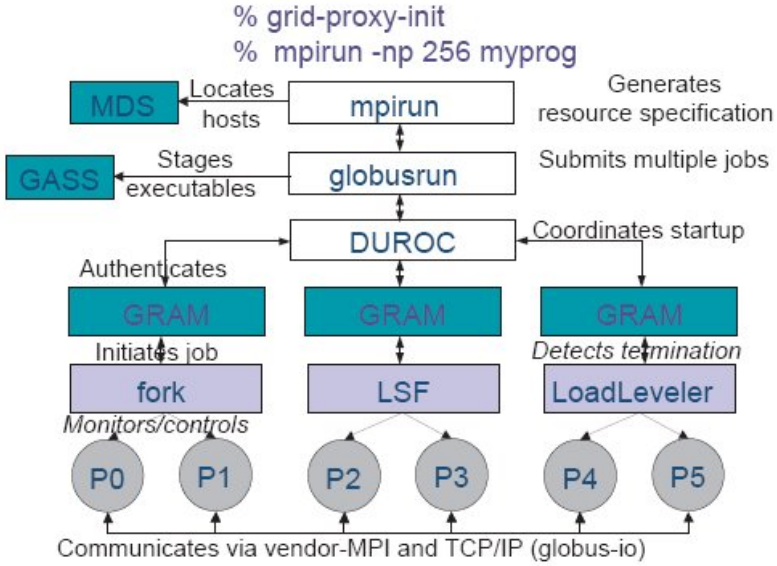


Figura 5.1: *Uso da MPICH–G2 em coordenação com vários componentes do Globus Toolkit e escalonadores de recursos locais [59]*

Após iniciada a aplicação, o DUROC e o GRAM interagem para monitorar a execução da aplicação. O DUROC coordena também a sincronização entre os processos da aplicação através de uma barreira que só libera os processos após todos terem sido iniciados. A MPICH–G2 também permite a criação de multi-níveis de aglomerados (*multilevel clustering*) baseados na topologia de rede subjacente, que permite melhorar o desempenho no uso de operações coletivas. A descrição dos multi-níveis é realizada pelo programador da aplicação através de funções que estendem a biblioteca padrão MPI, ou seja, o programador define rotinas para organizar grupos de processos que devem ficar dentro de um aglomerado, visando melhorar o desempenho da aplicação (configuração em nível de usuário).

A principal vantagem do uso da MPICH–G2 é possibilitar que aplicações paralelas do tipo MPI executem em ambientes de grade com um desempenho razoável se comparadas a ambientes de aglomerados. Isto é possível uma vez que utiliza implementações específicas para as plataformas de comunicação que executam a aplicação. Outra vantagem é permitir que o programador defina topologias de comunicação de grupo, melhorando a portabilidade de programas para o ambiente de grade. Uma desvantagem é a falta de um mecanismo de recuperação em caso de falhas, tendo o usuário que reiniciar a aplicação quando da ocorrência de falhas de comunicação ou execução.

5.1.2 MPICH–GF

A MPICH–G2 foi estendida para prover tolerância a falhas através de recuperação por retrocesso utilizando *checkpointing* em um trabalho chamado de MPICH–GF [97, 98]. A MPICH–GF tem como objetivo principal realizar o *checkpoint* de forma transparente para o programador de aplicações MPI. A Figura 5.2 ilustra a arquitetura da MPICH–GF, a qual é composta de três módulos:

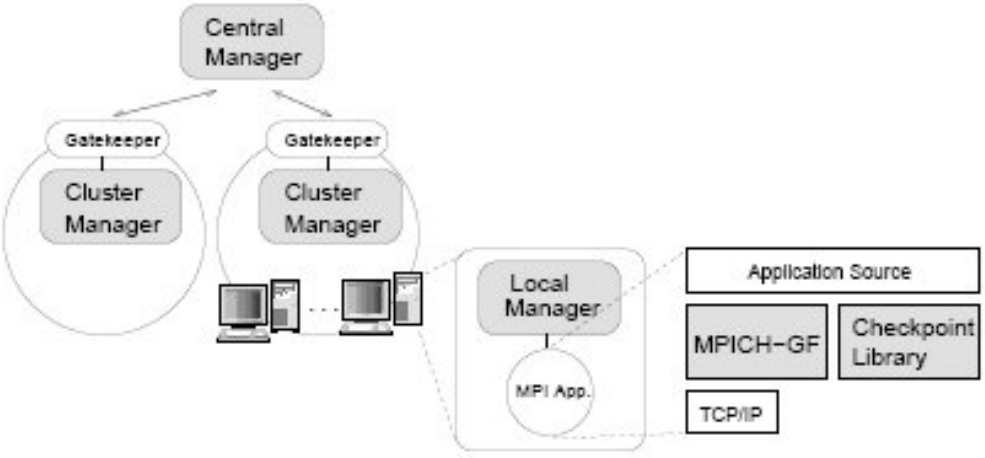


Figura 5.2: *Arquitetura da MPICH–GF [97]*

- A biblioteca MPICH–GF: implementa a biblioteca MPI, o modelo de comunicação entre os processos através de um canal de soquetes TCP/IP, e um protocolo coordenado bloqueante para realizar o *checkpoint* dos processos e recuperação em caso de falhas.
- A biblioteca de *checkpoint*: define funções em nível de sistema para realizar cópia de memória do processo em execução.
- A hierarquia do sistema de gerenciamento de processos: compreende os gerenciadores da aplicação providos pelo *Globus Toolkit*, que fornecem o serviço de coordenação do processo de armazenamento dos *checkpoints* e de recuperação da apli-

cação e dos *checkpoints* armazenados (*Central Manager*, *Cluster Manager* e *Local Manager*).

Esta implementação fornece recuperação através de um canal de comunicação por soquetes TCP/IP e gerando *checkpoints* em nível de sistema utilizando a biblioteca CKPT [100] para núcleo Linux 2.4 e 2.6. Uma desvantagem é que a implementação de *checkpointing* é específica para comunicação por soquetes TCP/IP o que restringe a utilização da MPICH-GF, além da perda de desempenho quando usando hardware de comunicação específico. Outra desvantagem é que *checkpointing* em nível de sistema limita a também a plataforma de execução.

Na próxima seção são descritos outros trabalhos que tratam especificamente da implementação de *checkpointing* para MPI, não necessáriamente para uso em ambientes de grade.

5.2 MPI com mecanismos de *checkpointing*

5.2.1 MPICH–V

A MPICH–V [69] é uma implementação do modelo MPI para execução de aplicações tolerantes a falhas. Essa biblioteca usa como base a MIPCH, e implementa duas versões principais: a MPICH–Vcl e a MPICH–Pcl. Ambas as versões implementam a ADI, assim como a MPICH–G2. A principal diferença é que a MPICH–G2 foi desenvolvida para permitir a execução de uma aplicação em uma grade computacional, utilizando os serviços fornecidos pela grade para realizar coordenação entre os diversos domínios administrativos e ambientes heterogêneos. A obtenção do estado dos processos é fornecido pela biblioteca de *checkpointing* BLCR. Esta biblioteca é responsável por criar *checkpoints* no nível do sistema operacional, portanto, os *checkpoints* não são portáveis. A MPICH–V foi desenvolvida para aglomerados homogêneos e implementa seus próprios serviços para garantir a interconexão dos processos, coleta de resultados da aplicação, além de armazenar e recuperar *checkpoints*.

MPICH–Vcl

A MPICH–Vcl [11] implementa um mecanismo de *checkpoint* utilizando um protocolo de coordenação não-bloqueante, o algoritmo *distributed snapshots* de Chandy/-Lamport [21], onde a comunicação entre os processos não é interrompida durante a obtenção do estado global. O uso de um protocolo não-bloqueante minimiza o *overhead* causado durante a criação do *checkpoint*. Outra característica é que todos os processos da aplicação devem ser reiniciados em caso de falha de algum dos processos, ao invés de reiniciar apenas os processos falhos.

A Figura 5.3(a) ilustra a arquitetura geral dos componentes da MPICH–V. A MPICH–Vcl foi desenvolvida a partir da primeira versão da MPICH, e implementa a CI[1] através de um dispositivo genérico de comunicação por soquetes TCP/IP. Os *daemons* implementam a estratégia de tolerância a falhas que será utilizada para gerar os *checkpoints*. No caso da implementação da MPICH–Vcl, o protocolo de coordenação não-bloqueante é implementado pelo *CL Daemon*. O *CL Daemon* funciona como um gerenciador de comunicação entre os nós da aplicação, enviando, recebendo, reordenando e restabelecendo a comunicação, para possibilitar o *checkpoint* do canal de comunicação. Este serviço é necessário uma vez que o uso de um protocolo não-bloqueante implica na realização de *checkpoint* das mensagens em trânsito no canal de comunicação, diferentemente de um protocolo bloqueante, que garante que o canal de comunicação estará vazio no momento de realizar o *checkpoint* da aplicação [11].

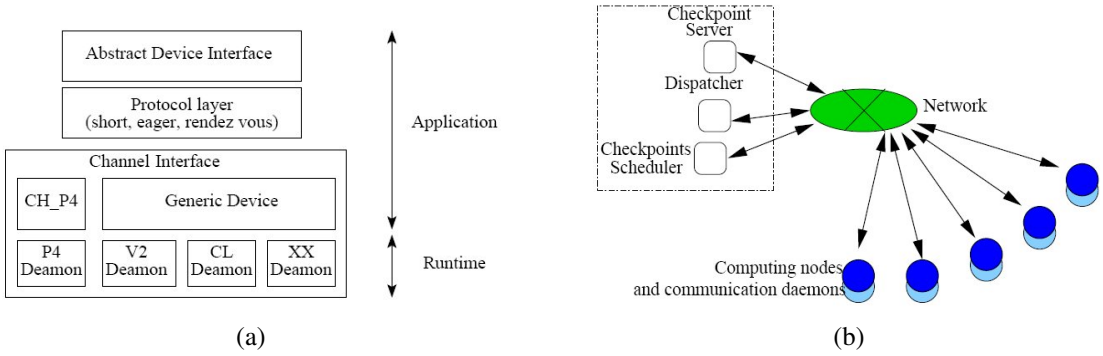


Figura 5.3: *Arquitetura geral da MPICH–Vcl* (a) *e a organização típica dos componentes* (b) *[11]*

A Figura 5.3(b) ilustra os componentes complementares da implementação da MPICH–Vcl. O *Dispacher* é responsável por iniciar a execução da aplicação, detectar a ocorrência de falhas e coordenar o reinício das aplicações. O *Checkpoints Scheduler* é responsável por criar **ondas de *checkpointing***[2], definindo o intervalo em que devem ser realizados os *checkpoints*. O *Checkpoint Server* define a interface responsável por realizar checkpoints em nível de sistema utilizando bibliotecas proprietárias. É responsável também por coordenar o armazenamento dos *checkpoints* gerados por todos os processos [13].

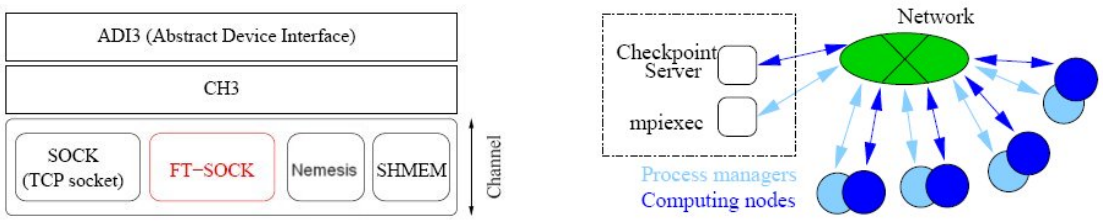
[1] *Channel Interface* (ver Sessão 3.3)

[2] Uma onda de *checkpointing* determina o mecanismo pelo qual o algoritmo cria os *checkpoints* distribuídos trocando mensagens de controle entre os processos da aplicação.

MPICH–Pcl

A MPICH–Pcl [13] implementa um mecanismo de *checkpoint* através de um protocolo de coordenação bloqueante, utilizando o mesmo algoritmo de Chandy/Lamport que a MPICH–Vcl. Entretanto, os canais de comunicação devem permanecer bloqueados durante uma onda de *checkpointing* como forma de eliminar mensagens em trânsito no canal de comunicação, não necessitando, portanto, realizar *checkpoint* dessas mensagens.

A Figura 5.4(a) ilustra a arquitetura da MPICH–Pcl, implementada utilizando a MPICH2. Esta arquitetura prevê a substituição da CI por uma implementação do canal de comunicação utilizando soquetes TCP/IP, provendo também serviços de tolerância a falhas. A infra-estrutura prevê ainda a implementação de um gerenciador de processos através da interface PMI, que gerencia a publicação de informações entre os processos paralelos. Como ilustrado na Figura 5.4(b) a MPICH–Pcl também utiliza o serviço fornecido pelo *Checkpoint Server* para criação e armazenamento de *checkpoints* em nível de sistema. Outra facilidade é a criação do programa *mpiexec*, que possibilita a especificação de máquinas a serem utilizadas na execução da aplicação, bem como as máquinas que realizarão o armazenamento dos *checkpoints* [13].



(a) (b)

Figura 5.4: *Arquitetura geral da MPICH–Pcl* (a) *e a organização típica dos componentes* (b) *[13]*

Em experimentos realizados por *Buntinas* [13], a utilização de MPICH–Vcl obteve um melhor desempenho em ambientes de grade, uma vez que o protocolo de coordenação não-bloqueante não introduz sobrecarga quanto ao processo de sincronização de *checkpoints*. Diferentemente, a MPICH–Pcl obteve um desempenho pior, já que ambientes de grade podem levar a atrasos maiores de comunicação. O uso de um protocolo de coordenação bloqueante em ambientes de grade pode manter processos bloqueados esperando o uso do canal de comunicação por períodos de tempo razoáveis. Em contrapartida, a implementação de protocolos bloqueantes é mais simples e de mais fácil portabilidade. Além disso, em ambientes de aglomerados a MPICH–Pcl apresentou desempenho semelhante ao da MPICH–Vcl devido à alta velocidade de sincronização entre os processos.

5.3 Outros trabalhos

5.3.1 LAM/MPI

LAM/MPI [55, 14], desenvolvida na Universidade de Indiana, é uma implementação de alto desempenho do modelo de programação MPI. LAM/MPI provê um mecanismo transparente de *checkpoint*, utilizando um arcabouço de recuperação por retrocesso (CRR) que facilita a portabilidade dos mecanismos de *checkpoint*. Isto é possível devido à existência de um arcabouço de componentes de comunicação modular (SSI) que permite adicionar implementações específicas de uma determinada infra-estrutura na forma de *plug-ins* [80, 101].

5.3.2 Cocheck

Cocheck [86] é um mecanismo de *checkpointing* que fornece o serviço de recuperação por retrocesso para aplicações paralelas que utilizam bibliotecas de passagem de mensagem. Esse mecanismo é uma extensão do protocolo síncrono para fazer *checkpointing* de aplicações no Condor com somente um processo. Um processo controlador é usado para enviar mensagens para todos os outros processos inicializando o *checkpoint*. Essa mensagem bloqueia a comunicação até que um estado global consistente seja conseguido e o *checkpoint* realizado. Cocheck provê um suporte limitado para MPI e também fornece um mecanismo primário de migração, caracterizando-se pelo alto *overhead* de sincronização.

5.4 Discussão

A Tabela 5.1 apresenta uma comparação entre a MPICH–IG e os demais sistemas apresentados neste capítulo. São comparadas algumas características que devem ser fornecidas para permitir a execução de aplicações paralelas em uma grade oportunista. A MPICH–IG deve ter estas características, mantendo para o desenvolvedor de aplicações, transparência quanto ao uso da grade, como por exemplo, a necessidade de um mínimo de alterações nas aplicações MPI para utilizar a MPICH–IG.

A MPICH–IG foi desenvolvida para a grade oportunista InteGrade, possibilitando a execução de aplicações MPI–2 em ambientes não dedicados. A MPICH–G2, por sua vez, foi proposta para o Globus, uma grade que utiliza geralmente recursos dedicados. O MPICH–G2 e o MPICH–IG apresentam alguns aspectos comuns, como por exemplo, a utilização dos serviços da grade como um *toolkit* para possibilitar a comunicação e o gerenciamento entre os nós da aplicação. Um exemplo, é a utilização de um módulo da grade com a função de gerenciar e sincronizar a comunicação de uma aplicação MPI:

Tabela 5.1: *Comparação entre os sistemas apresentados*

Característica \ Sistema	MPICH2	MPICH–G2	MPICH–GF	Cocheck	MPICH–Vcl	MPICH–Pcl	LAM/MPI	MPICH–IG
Suporte a MPI–2	✘			✘		✘	✘	✘[a]
Suporte ao ambiente de grade		✘	✘	✘				✘
Suporte a vários canais de comunicação	✘	✘					✘	✘[b]
Suporte a recuperação por retrocesso			✘	✘	✘	✘	✘	✘
Geração de *checkpointing* portável								✘[b]
Armazenamento de *checkpoints*			✘	✘	✘	✘		✘[b]
Mecanismo portável de geração de *checkpointing*				✘	✘	✘	✘	✘

[a] Parcialmente suportado uma vez que não permite a criação de processos dinâmicos.
[b] Suportado pela arquitetura mas ainda não implementado.

o módulo DUROC no caso da MPICH–G2 e o módulo EM no caso da MPICH–IG. A MPICH–G2 possibilita ainda explorar a topologia dos aglomerados para utilizar modelos de comunicação proprietárias de forma eficiente. Esta funcionalidade não está presente na MPICH–IG, devendo ser estudada a possibilidade de sua inclusão em trabalhos futuros.

Em grades computacionais oportunistas, os ambientes tendem a ser mais heterogêneos e complexos que os comumente utilizados com o Globus. Sendo assim, a infraestrutura da MPICH–IG permite o uso de mecanismos de gerencia de comunicação e recuperação disponibilizados pela grade para contornar os problemas encontrados em um ambiente oportunista. Um exemplo, são os serviços de escalonamento eficiente de aplicações e de recuperação em caso de falhas, que podem armazenar *checkpoints* portáveis, possibilitando que máquinas heterogêneas possam executar a aplicação a partir do mesmo *checkpoint*.

Um serviço importante necessário para execução de aplicações MPI em uma grade computacional é o serviço de *checkpointing*. A MPICH–V é uma implementação que disponibiliza o serviço de *checkpoint* para aplicações MPI executando em ambientes de aglomerados. A arquitetura da MPICH2, base da MPICH–V, permite que os módulos de coordenação e de comunicação dos processos sejam reimplementados para caracterizar diversos ambientes de execução, como, por exemplo, o de grades computacionais. A arquitetura da MPICH–V possibilita ainda o desacoplamento da comunicação entre os processos e a coordenação do mecanismo de *checkpoint*, permitindo que se utilize estratégias de *checkpointing* mais adequadas ao ambiente de execução. Um exemplo é a utilização de *checkpointing* portável em nível de aplicação, ao mesmo tempo que se utiliza modelos de comunicação específicos do ambiente local, melhorando o desempenho da aplicação.

A MPICH–IG estende a MPICH–V, mais especificamente a MPICH–Pcl, para fornecer o serviço de *checkpointing* de aplicações MPI–2 no InteGrade, uma vez que a arquitetura separa a implementação do canal de comunicação do protocolo de *checkpoint*, possibilitando uma implementação portável do mecanismo de tolerância a falhas para outros canais de comunicação. A maioria dos trabalhos encontrados na literatura descreve mecanismos de realização de *checkpoint* em nível de sistema, gerando *checkpoints* não portáveis. Isto ocorre porque são projetados para ser utilizados em ambientes de aglomerados homogêneos. Para utilizar aplicações MPI de forma eficiente em ambientes de grade computacional oportunista, uma arquitetura mais robusta deve possibilitar a criação de *checkpoints* portáveis para várias plataformas, utilizando, por exemplo, *checkpoint* em nível de aplicação e serviços de armazenamento de *checkponits*, serializados segundo alguma representação de dados independente.

A MPICH–IG, descrita no próximo capítulo, consiste de uma infra-estrutura para execução de aplicações MPI–2 no InteGrade. Esta implementação, a exemplo da MPICH–G2, utiliza a MPICH como forma de facilitar a implementação da biblioteca MPI em uma grade computacional, pois sua arquitetura minimiza a quantidade de funções a serem implementadas. A MPICH–IG também adapta a arquitetura e códigos desenvolvidos pela MPICH–Pcl como forma de favorecer o desenvolvimento do serviço portável de recuperação por retrocesso para aplicações MPI–2. A arquitetura da MPICH–IG visa atender as necessidades das aplicações MPI no contexto de grades oportunistas, tais como, localização e sincronização dos processos, heterogeneidade de comunicação e tolerância a falhas do ambiente de execução.

CAPÍTULO 6

MPICH–IG

Este capítulo descreve a arquitetura e implementação da MPICH–IG, que atende os requisitos necessários para permitir a execução de aplicações MPI na grade oportunista InteGrade. Ao final, são apresentados e discutidos os resultados obtidos com o trabalho.

6.1 Visão Geral

O modelo MPI define uma especificação bastante completa e bem aceita na comunidade de computação de alto desempenho. Entretanto, MPI especifica apenas os mecanismos e operações que são necessários para criar programas paralelos, não tratando a forma como estas funcionalidades devem ser implementadas. Outra característica é a flexibilidade quanto ao desenvolvimento de aplicações paralelas, o que difundiu seu uso na comunidade de processamento paralelo, sendo de extrema importância que projetos de grades computacionais de alto desempenho forneçam suporte para MPI.

Desde sua concepção, o InteGrade oferece suporte para a execução de aplicações paralelas do tipo BSP (veja Seção 3.1), implementando assim alguns requisitos necessários para executar aplicações paralelas no ambiente de grade. O InteGrade possibilita escalonar várias máquinas simultaneamente para compor os nós que executarão a aplicação paralela. Outro serviço é o de gerenciamento da aplicação paralela, que fornece subsídios para localizar e difundir informações de conexão entre os diversos nós que compõem a execução. Este serviço permite também reescalonar novos recursos caso tenha ocorrido alguma falha durante a execução da aplicação, como por exemplo, o usuário provedor de recursos requisitar de volta os recursos fornecidos para a grade, ou ainda alguma falha durante a comunicação entre os diversos nós que compõem a aplicação. Este serviço é apoiado por um armazenamento distribuído de *checkpoints*, permitindo que aplicações possam re-iniciar sua execução utilizando informações de estado previamente salvas em outras máquinas conectadas à grade.

Estes serviços fornecidos pelo InteGrade funcionam como um *toolkit*, contemplando parte dos requisitos necessários para executar aplicações paralelas na grade. A outra parte é composta pela implementação de como utilizar estes serviços. Em geral, a

aplicação, durante a execução, deve fazer uso destes serviços quando necessário, como por exemplo, para definir o momento de gerar *checkpoints* e para armazená-los, para identificar a ocorrência de erros durante a comunicação e solicitar ao módulo de gerenciamento da grade que recupere a aplicação, ou ainda para especificar os dados necessários para estabelecer conexão entre os vários nós que compõem a execução, como por exemplo o endereço IP e número de porta para conexões TCP/IP.

Uma forma de se conseguir que a aplicação faça uso destes serviços de forma transparente para o programador é reimplementar a biblioteca paralela para utilizar os serviços da grade e recompilar a aplicação com essa nova biblioteca para que possa ser utilizada na grade (veja a Seção 3.4). As metas de flexibilização do modelo MPI o tornaram extenso quanto ao número de funções e mecanismos de comunicação, sendo uma tarefa árdua e complexa implementá-lo em diferentes plataformas, como a de grade.

A biblioteca de programação MPICH foi desenvolvida para resolve parcialmente este problema, tornando o trabalho de implementação de MPI em diferentes plataformas uma tarefa mais fácil reduzindo a quantidade de funções que devem ser implementadas, ao mesmo tempo em que consegue manter a eficiência das aplicações. A MPICH é discutida em maiores detalhes na Seção 3.3. Apoiados nestas características de facilidade de implementação e garantias quanto ao desempenho [4], optamos por utilizar a MPICH2 (versão mais recente da biblioteca) como base para atingir os objetivos deste trabalho. Outro fator de motivação foi a quantidade de projetos relacionados e bem sucedidos que tratam da implementação da MPICH em diversas plataformas, inclusive em ambientes de grade computacional. Os trabalhos mais relevantes foram descritos no Capítulo 5.

A biblioteca MPICH–IG permite a execução de aplicações paralelas baseadas em MPI no InteGrade. A MPICH–IG tem como base a MPICH2 e objetiva implementar o processo de comunicação entre os nós de uma aplicação MPI rodando no InteGrade, transpondo problemas de escalonamento de recursos, localização, sincronização, tolerância a falhas e coleta de resultados na execução das aplicações.

A Figura 6.1 ilustra a disposição da MPICH–IG junto ao InteGrade. Pode-se observar que as aplicações têm como plataforma de execução a MPICH–IG, abstraindo os mecanismos de comunicação e possibilitando que aplicações MPI legadas necessitem apenas ser recompiladas para executar no InteGrade. Outra característica é a utilização de CORBA como plataforma de comunicação entre a MPICH–IG e os módulos do InteGrade, reaproveitando interfaces já implementadas e mantendo desacoplados os serviços fornecidos pela grade.

A implementação padrão da MPICH2 não provê recursos para tratar os diversos problemas enfrentados na execução de aplicações em diferentes plataformas. Entretanto, sua arquitetura especifica onde esses recursos devem ser implementados, de forma a atingir as metas de portabilidade e desempenho. Segundo este modelo, a MPICH–IG deve

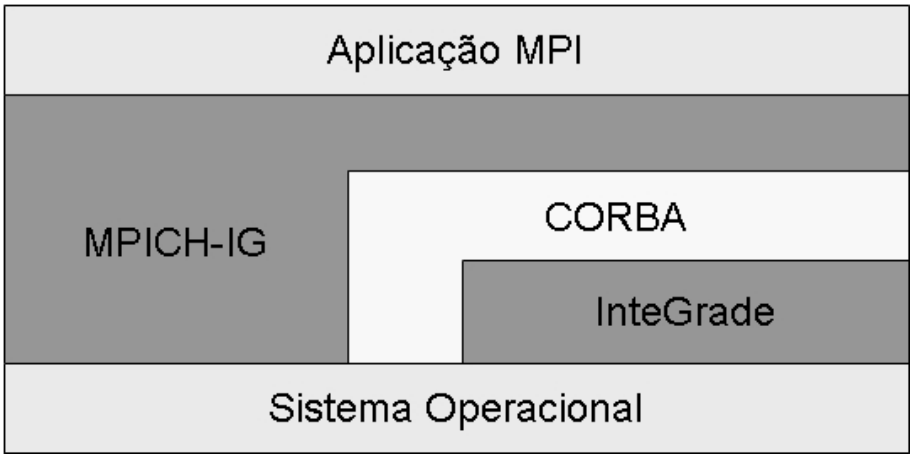


Figura 6.1: *Disposição em camadas da MPICH–IG e InteGrade*

reimplementar duas interfaces da MPICH2 (maiores detalhes na Seção 3.3):

- ***Process Manager Interface* (PMI)**: especifica funções para gerência de comunicação entre os processos, tais como: localização dos nós participantes da execução, sincronização dos processos, identificação de ocorrência de falhas, e armazenamento e recuperação de *checkpoints*.
- ***Channel Inteface* (CI)**: especifica funções para o uso eficiente de um canal de comunicação específico, possibilitando o monitoramento de comunicação para prover outros serviços, como por exemplo, tolerância a falhas de comunicação.

As interfaces PMI e CI são implementadas pelos módulos da MPICH–IG, IG–PM e IG–Sock, respectivamente, substituindo os módulos correspondentes na implementação padrão da MPICH2. Nas Seções 6.2 e 6.3 são discutidos detalhes dos componentes que compõem a infra-estrutura da MPICH–IG e do mecanismo de tolerância a falhas. Na Seção 6.4, é descrita a implementação desses módulos e as modificações realizadas no InteGrade. Na Seção 6.5 e 6.6 são levantadas algumas considerações e avaliações analíticas sobre o trabalho desenvolvido.

6.2 Arquitetura da MPICH–IG

Para fornecer o serviço de execução remota de aplicações paralelas, o InteGrade define dois protocolos para tratar algumas questões associadas a esse serviço: o protocolo de submissão de aplicações e o protocolo de recuperação por retrocesso em caso de falhas. Esses protocolos foram definidos para auxiliar na execução de aplicações do tipo BSP (Seção 2.4). Sua definição resultou no desenvolvimento de módulos responsáveis por gerenciar problemas de localização, sincronização, detecção de falhas, armazenamento de *checkpoints* e gerenciamento de consistência das aplicações.

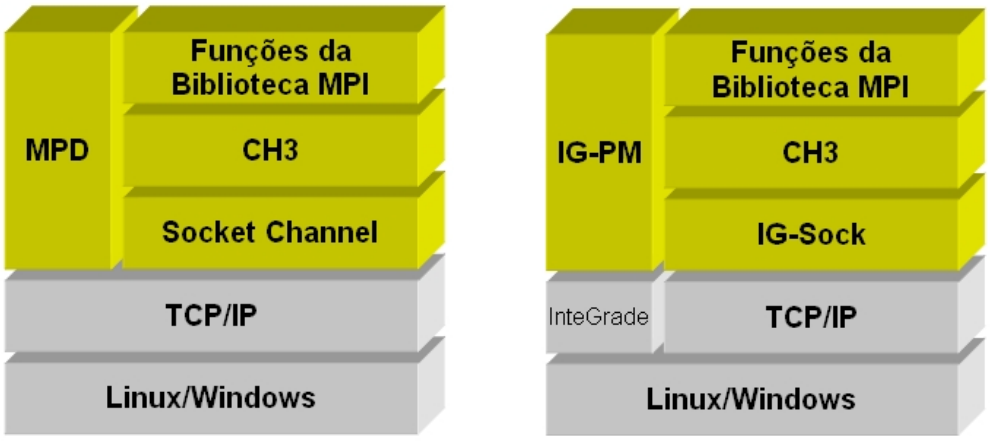
No contexto de aplicações MPI, a MPICH–IG, especifica adaptações nesses protocolos de forma a integrar os módulos do InteGrade com os componentes da MPICH2. A maior preocupação desta adaptação foi prover um mecanismo que permita a qualquer aplicação MPI legada ser executada no InteGrade com um mínimo ou nenhuma alteração do código fonte pelo desenvolvedor. Outra preocupação foi reaproveitar o máximo de código já implementado de ambos os sistemas, de forma a minimizar o impacto nos respectivos projetos. Isto possibilita, por exemplo, a utilização de serviços complementares que possam vir a ser desenvolvidos em implementações futuras de ambos os projetos, como por exemplo, a criação de novas estratégias de *checkpoint* e transposição de *firewalls* no contexto da grade, e a implementação de novas CI[1] para comunicação específica da plataforma de execução.

A Figura 6.2(b) ilustra a disposição das camadas da MPICH–IG, comparadas com a implementação padrão da MPICH2, exibida na Figura 6.2(a). A MPICH2 é organizada em várias camadas como forma de desacoplar funcionalidades necessárias para executar uma aplicação paralela, tornado mais simples o trabalho de portar a implementação para outra plataforma. O módulo MPD (*Multipurpose Daemon*) da MPICH2 define funções de comunicação para gerenciamento da aplicação, criando uma rede de interconexão entre várias máquinas que podem executar a aplicação e difundindo informações necessárias para conexão entre os processos que fazem parte da aplicação. Esse módulo foi substituído pelo IG–PM da MPICH–IG, que provê as mesmas funções. O módulo IG–PM é responsável por gerenciar a aplicação dentro do contexto da grade, para tanto faz uso do serviço de gerenciamento de aplicações do InteGrade, fornecido pelo módulo *Execution Manager* (EM). Diferentemente do MPD, que utiliza soquetes TCP/IP, o IG–PM utiliza como base de comunicação a infra-estrutura do InteGrade, permitindo que as funções de gerenciamento possam transpor problemas advindos da distribuição em larga escala, como por exemplo, *firewalls*, NAT, políticas de comunicação entre domínios administrativos, comunicação com serviços de armazenamento etc.

O módulo *Socket Channel* da MPICH2 define as funcionalidades necessárias para realizar a comunicação entre os nós de uma aplicação MPI utilizando um canal de soquetes TCP/IP. Todas as funções da biblioteca MPI (operações ponto–a–ponto, coletivas, DRMA, CH3, etc) são definidas como macros sobre as funções do *Socket Channel*, que é constituído de poucas funções. O módulo IG–Sock da MPICH–IG reaproveita o módulo *Socket Channel* e introduz uma adaptação que permite monitorar o canal de soquetes TCP/IP com o fim de criar o mecanismo de *checkpointing* coordenado e prover o serviço de recuperação por retrocesso das aplicações MPI no InteGrade. O IG–Sock utiliza funcionalidades do IG–PM para obter informações de como realizar o

[1] *Channel Interface*.

armazenamento e a recuperação dos *checkpoints* criados.



(a) (b)

Figura 6.2: *Disposição das camadas da MPICH2 padrão* (a) *e as respectivas implementações para MPICH–IG* (b)

Como descrito na Seção 3.3 a MPICH2 é uma biblioteca que implementa a especificação MPI–2, e deve ser ligada a programas que utilizam a interface do modelo de programação. A MPICH–IG substitui alguns módulos da implementação da MPICH2, de forma a utilizar também componentes do InteGrade para coordenar o processo de execução. Sendo assim, a MPICH–IG não gera uma dependência para o InteGrade e será utilizada somente para compilar os programas MPI. Detalhes de como utilizar a MPICH–IG são descritos na Seção 6.5. Em seguida são discutidos detalhes de projeto do componente IG–PM, associado ao serviço de sincronização e localização, e do componente IG–Sock, associado ao serviço de recuperação por retrocesso.

6.2.1 Serviço de Sincronização e Localização

O desenvolvimento de aplicações paralelas requer a utilização de um serviço de localização que facilite a comunicação entre os processos que compõem a aplicação. Em grades computacionais os processos paralelos podem estar geograficamente dispersos, necessitando de mecanismos para publicar as informações necessárias para o estabelecimento de comunicação entre os processos.

Além de obter informações para realizar a conexão, os processos necessitam sincronizar a execução da aplicação, definindo um momento para estabelecer a conexão entre os diversos nós paralelos. Na MPICH2 padrão, este serviço é fornecido pelo MPD. O módulo IG–PM substitui o MPD, encapsulando a função de gerenciamento local da aplicação MPI em execução no InteGrade. O trabalho de determinar o término da aplicação e coletar os resultados da execução é desempenhado pelo módulo LRM do

InteGrade, serviço que também é implementado pelo MPD na MPICH2. As principais funções do IG–PM são:

- **Carregar informações de execução**: o serviço de execução remota do InteGrade define vários parâmetros necessários para configurar o ambiente de execução, os quais são passados ao LRM no momento da solicitação de execução da aplicação. Entre esses parâmetros são definidos: o identificador da aplicação, o identificador único de cada processo (*rank*), a quantidade de processos que participam da execução e um valor numérico que define um intervalo em segundos para realizar os *checkpoints* da aplicação.
- **Sincronizar e localizar os processos da aplicação**: para estabelecer a comunicação com os outros processos, um processo local utiliza o IG–PM para obter as informações de conexão, como endereço IP e número de porta. O IG–PM em cada nó deve enviar para o módulo EM do InteGrade as informações de conexão do processo local e esperar como retorno as informações para conexão com os outros processos.
- **Armazenar e recuperar *checkpoints***: para realizar o armazenamento ou recuperação de *checkpoints* utilizando alguma estratégia implementada no InteGrade, o IG–PM utiliza a CkpLib. Este módulo define estratégias de armazenamento local ou de uso de um serviço de repositório distribuído, provendo seus serviços através de uma API [19].

Arquitetura do IG–PM

O protocolo de execução de aplicações paralelas MPI, exibido na Figura 6.3, reaproveita todos os módulos já desenvolvidos no InteGrade. Os passos ilustrados são os mesmos descritos na Figura 2.6 para o processo de execução de aplicações BSP. Somente o passo 9, que descreve o serviço de localização e sincronização das aplicações MPI foi alterado. Aplicações BSP utilizam este passo para obter a IOR do processo com identificador 0 (zero), que é o gerente do processo de comunicação. A IOR (*Interoperable Object Reference*) é um identificador utilizado por um ORB para prover acesso a um objeto remoto, segundo a especificação CORBA [76]. Na implementação da MPICH–IG o passo 9 é utilizado para publicar informações de conexão entre os processos da aplicação MPI descritos a seguir.

Na MPICH2 as informações necessárias para realizar a comunicação são definidas pela CI através de uma estrutura chamada **espaço chave–valor** (*key–value space* – KVS). Essa estrutura possibilita o desacoplamento entre gerenciamento e comunicação, onde o serviço de gerenciamento necessita compartilhar as informações contidas no KVS de cada processo. A Figura 6.4 mostra um diagrama de seqüência da execução de uma

aplicação MPI com dois processos no InteGrade, envolvendo o serviço de localização e sincronização descrito no passo 9 da Figura 6.3.

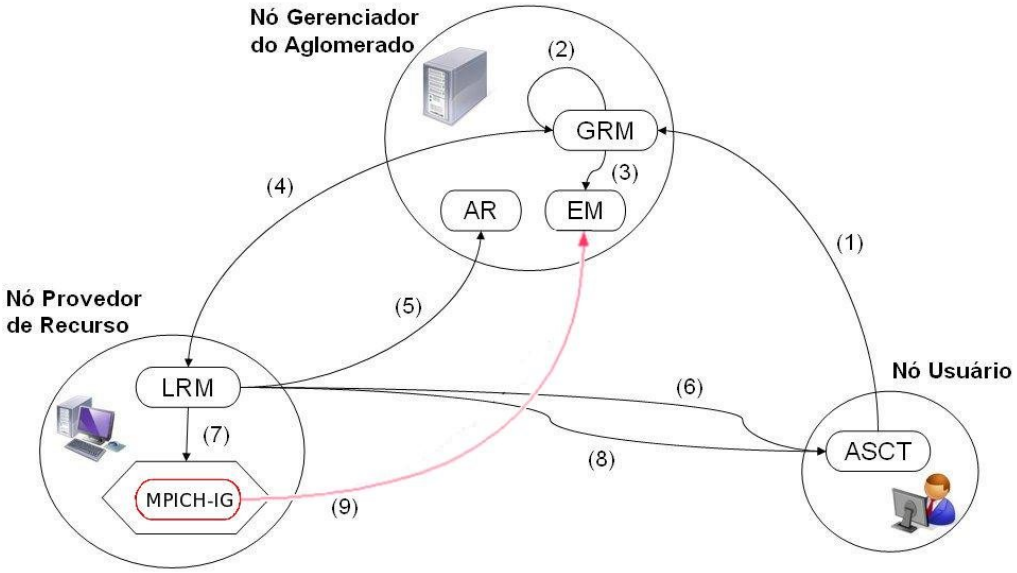


Figura 6.3: *Protocolo de execução de aplicaçõs MPI no InteGrade*

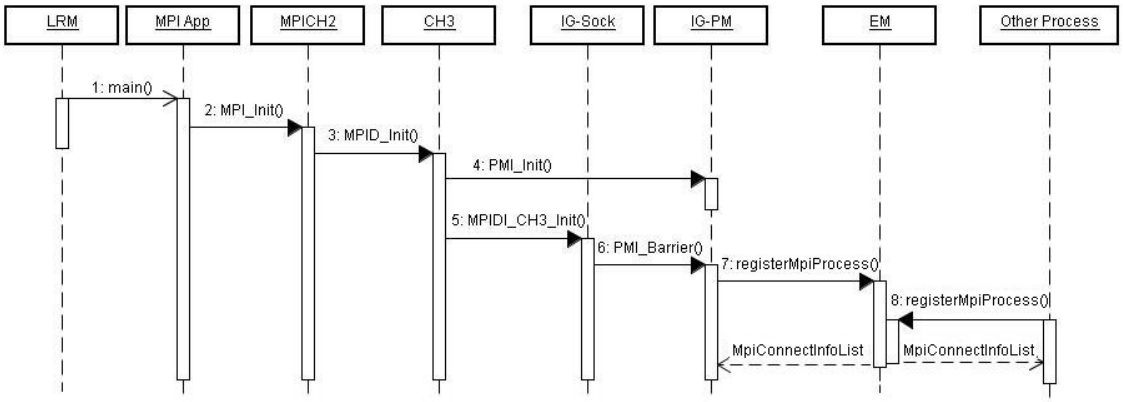


Figura 6.4: *Execução de uma aplicação MPI com dois processos no InteGrade*

Após o LRM lançar a aplicação (1) o processo de localização e sincronização é iniciado com a chamada da função *MPI_Init* de cada processo participante (2), onde a MPICH2 inicializa variáveis de controle da biblioteca. A chamada *MPID_Init* (3) inicia os *buffers* e estruturas de dados necessários para criar as estratégias de comunicação. A CH3 faz uma chamada à *PMI_Init* do IG–PM (4) para carregar as informações de execução da aplicação. Em seguida a CH3 faz uma chamada à *MPIDI_CH3_Init* do IG–Sock (5), que inicializa as estruturas específicas para comunicação através de soquetes TCP/IP. O IG–Sock inicia um soquete servidor e publica em seu KVS as informações necessárias para receber conexões (endereço IP e porta). Após definir o KVS local, o IG–Sock faz uma chamada à *PMI_Barrier* do IG–PM (6) para sincronizar a execução com

os outros processos. Através de um ORB CORBA OIL o IG–PM publica o KVS local no módulo EM do InteGrade através da chamada *registerMpiProcess* (7 e 8). O EM espera até que todos os processos se registrem (sincronização) para então compartilhar uma lista contendo os KVS de todos os processos. Detalhes de implementação da extensão do EM para suportar o registro de aplicações MPI são descritos na Seção 6.4.

O método *registerMpiProcess* do EM foi desenvolvido para dar suporte ao serviço de sincronização e localização no contexto das aplicações MPI, caracterizando uma extensão do código já existente no InteGrade. Esta adaptação segue o padrão de desenvolvimento de software orientado a objetos, estendendo o serviço de gerenciamento de aplicações através de herança do serviço padrão fornecido pelo EM. Esta alteração dirigida a um modelo garante o não comprometimento do código já existente. Os detalhes de implementação são fornecidos na Seção 6.4.

6.2.2 Serviço de Recuperação por Retrocesso

Como discutido no Capítulo 4, fornecer garantias quanto a falhas em uma aplicação paralela é muito importante, ainda mais quando executadas em uma grade. Os provedores de recursos de uma grade computacional oportunista podem perder conectividade com a grade várias vezes em um único dia, o que prejudica aplicações paralelas que podem executar por horas ou até dias.

Neste sentido o serviço de recuperação por retrocesso com *checkpoints* é a alternativa mais utilizada para evitar que a aplicação tenha que ser executada novamente a partir do início. O módulo IG–Sock foi desenvolvido a partir da implementação da MPICH–Pcl (discutida na Seção 5.2.1), utilizando o algoritmo *distributed snapshot* [21] para criação de *checkpoints* de aplicações distribuídas.

A MPICH–Pcl possibilita a realização de *checkpoints* em um canal de comunicação por soquetes TCP/IP (*UNIX sockets* e *WIN sockets*), utilizando como base a implementação da MPICH2. Este mecanismo de *checkpointing* pode também ser facilmente portado para outro canal de comunicação diferentemente, dos outros sistemas discutidos no Capítulo 5, justificando seu uso para o desenvolvimento do IG–Sock. O IG–Sock prevê a extensão de algumas funcionalidades da MPICH–Pcl para:

- **Definir quando gerar *checkpoints***: uma das variáveis do ambiente de execução carregadas pelo módulo IG–PM é o intervalo de tempo em que devem ser criados os *checkpoints*. Esse valor é obtido pelo IG–Sock, que deve lançar uma *thread* em um dos processos (processo coordenador) a fim de iniciar a criação dos *checkpoints* de forma coordenada.

- **Armazenar os *checkpoints***: após a criação do *checkpoint* o IG–Sock deve armazená-lo através do mecanismo de armazenamento distribuído disponível no InteGrade (ver Seção 2.4).
- **Recuperar a aplicação na ocorrência de falhas**: após ter ocorrido uma falha, o IG–Sock determina o reinício, fazendo a leitura dos dados de *checkpoint* armazenados e iniciando o mecanismo de recuperação da aplicação, discutido em maiores detalhes na Seção 6.3.

O módulo IG–Sock utiliza um protocolo coordenado bloqueante para obter o *checkpoint* global, base da implementação MPICH–Pcl. Esse protocolo foi utilizado devido à maior facilidade quanto à implementação de mecanismos para criar o protocolo bloqueante para *checkpoints* em MPI. A MPICH–Pcl utiliza como base a MPICH2, refletindo o padrão MPI–2, possibilitando que aplicações que utilizam esse padrão possam ser executadas no InteGrade através do módulo IG–Sock.

Como discutido por *Buntinas et al.* [13], em ambientes de grades computacionais o uso de protocolos bloqueantes pode causar uma perda de desempenho das aplicações, uma vez que redes de grande área podem gerar atrasos significativos durante o processo de coordenação. Esta implementação deve ser modificada no futuro para adicionar mecanismos que permitam utilizar uma abordagem não-bloqueante (veja Seção 4.1.3), ou seja, realizando o *checkpoint* do canal de comunicação. Isto é possível, por exemplo, fazendo uso de *log* das mensagens trocadas para gravar e restaurar as mensagens durante a criação e recuperação do *checkpoint*, respectivamente.

Arquitetura do IG–Sock

A infra-estrutura da MPICH–IG prevê futuramente a implementação de outras estratégias para geração de *checkpoint*, bem como a utilização de outras bibliotecas de *checkpointing*. Para tanto, os serviços fornecidos pelo IG–Sock, são desacoplados através de interfaces independentes, possibilitando que as implementações sejam alteradas de forma mais fácil e dirigida por um modelo. A Figura 6.5 ilustra em alto nível o desacoplamento entre os três componentes fundamentais: o canal de comunicação (FT–Sock), o protocolo de coordenação de *checkpoint* (PCL) e a biblioteca de geração de *checkpoint* (MpiCkpLib), que são descritos a seguir:

- **FT–Sock**: implementação da *Channel Interface* que é utilizada para realizar a comunicação por soquetes TCP/IP. A implementação atualiza as estruturas criadas nas camadas acima (funções de comunicação da MPI–2 e CH3) gerando notificações quanto ao recebimento e envio de mensagens, ao estabelecimento de comunicação e aos erros provenientes do canal. Estas notificações são enviados para o módulo PCL para coordenar o mecanismo de criação de *checkpoints*.

- **PCL**: implementação do protocolo de *checkpoint* que coordena os eventos provenientes do FT–Sock para execução do protocolo *distributed snapshot* de Chandy / Lamport. Este utiliza o módulo IG–PM para obter informações sobre onde armazenar e recuperar *checkpoints* e o intervalo em que devem ser gerados os *checkpoints*.
- **MpiCkpLib**: define a biblioteca que possibilita a criação de *checkpoints* do processo, ou seja, obtém os dados necessários para criar um estado local consistente. Também é responsável por recuperar a aplicação a partir de um *checkpoint* fornecido. A Seção 6.3 discute, em maiores detalhes, duas abordagens: *checkpoint* em nível de aplicação e *checkpoint* em nível de sistema.

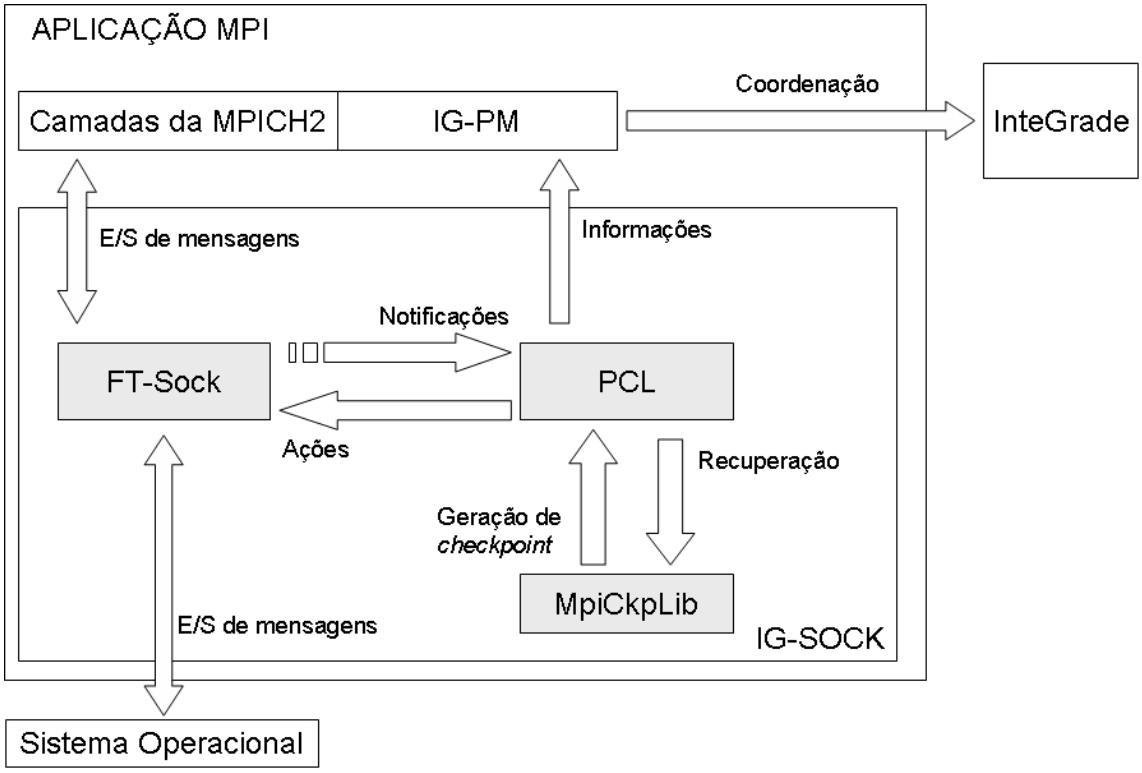


Figura 6.5: *Visão geral do módulo IG–Sock*

Os módulos FT–Sock e PCL são adaptações do código da MPICH–Pcl. Estas adaptações permitem melhorar o encapsulamento de cada um possibilitando que no futuro novos canais de comunicação, além do FT–Sock, possam ser incluídos sem que para isso necessite de sérias alterações no PCL ou MpiCkpLib.

Checkpointing coordenado bloqueante

O protocolo coordenado bloqueante *distributed snapshoot* (Figura 6.6) define a sincronização dos processos de forma a eliminar as dependências geradas pelas mensa-

gens em um canal de comunicação do tipo FIFO[2]. Essa estratégia é implementada pelo módulo PCL e evita a criação de *checkpoints* inconsistentes eliminando dependências entre os *checkpoints* gerados, que são ocasionados por mensagens em trânsito no canal de comunicação.

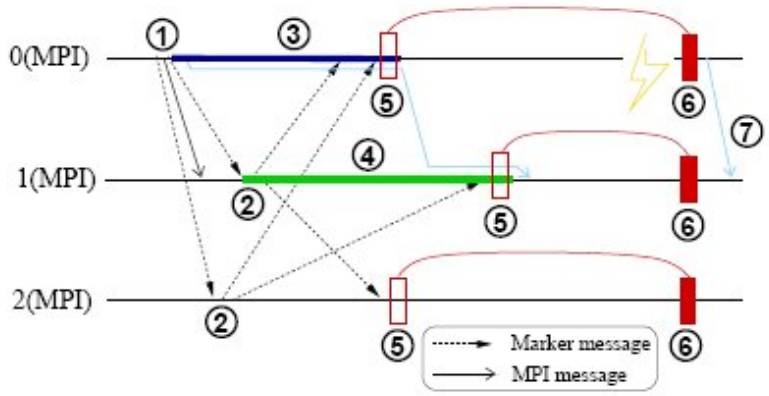


Figura 6.6: *Uma execução do protocolo Chandy/Lamport bloqueante [13]*

A sincronização é realizada pela troca de uma mensagem de marca (*marker*), que determina o início de uma onda de *checkpoint*. Esta mensagem é enviada pelo processo zero a todos os outros processos (1) determinando o início da onda. No momento em que um processo recebe a marca, ele a reenvia a todos os outros processos (2), além de bloquear o envio de mensagens a outros processos (3). Esse bloqueio garante que o canal pelo qual o processo recebeu a marca irá ficar vazio (4). Todos os processos esperam receber a marca de todos os outros processos para então realizar o *checkpoint* (5). Caso um processo receba uma mensagem de um canal limpo antes de realizar o seu *checkpoint*, este deve bloquear as mensagens recebidas até realizar o *checkpoint*.

As mensagens que foram bloqueadas pelo processo remetente devem ser incluídas no estado local pela biblioteca de *checkpoint*, e os processos receptores que bloquearam as mensagens devem descartá-las do *checkpoint*. Isso é necessário uma vez que, no momento de uma recuperação a partir dos *checkpoints* gerados (6), as mensagens bloqueadas nos remetentes devem ser enviadas (7).

A mensagem de marca é constituída como uma nova estrutura de dados dentro da CI e são utilizadas as rotinas da CI para enviá-las. Essas mensagens são interceptadas pelo módulo PCL (coordenador do protocolo), que realiza as operações necessárias para a coordenação. A arquitetura da MPICH–Pcl prevê a criação de servidores para armazenar os *checkpoints*. Na implementação da MPICH–IG, o IG–PM deve utilizar a interface do

[2]First in first out.

módulo CkpLib do InteGrade (Seção 2.4), para armazenar e recuperar os *checkpoints* criados.

6.3 Checkpoint de aplicações MPI no InteGrade

A utilização de *checkpoints* no InteGrade tem como principais dificuldades a implementação de estratégias para criar os *checkpoints* e definir como estes serão armazenados e posteriormente recuperados. Quanto ao problema de armazenamento e recuperação, o protocolo de recuperação por retrocesso, definido para aplicações BSP e discutido na Seção 2.4, possibilita o armazenamento de *checkpoints* portáveis. Os módulos CDRM, CkpRep e CkpLib do InteGrade fornecem um serviço de armazenamento genérico de *checkpoints* [19]. A implementação do mecanismo de recuperação da MPICH-IG encontra-se em desenvolvimento, não utilizando ainda estes módulos de armazenamento devendo gravar os *checkpoints* localmente. Contudo, a versão final da implementação prevê a gravação dos *checkpoints* em repositórios remotos, uma vez a falha do nó local pode impossibilitar a recuperação da aplicação. Em seguida são definidas considerações sobre as estratégias a serem utilizadas para realizar *checkpoints* através da infra-estrutura proposta.

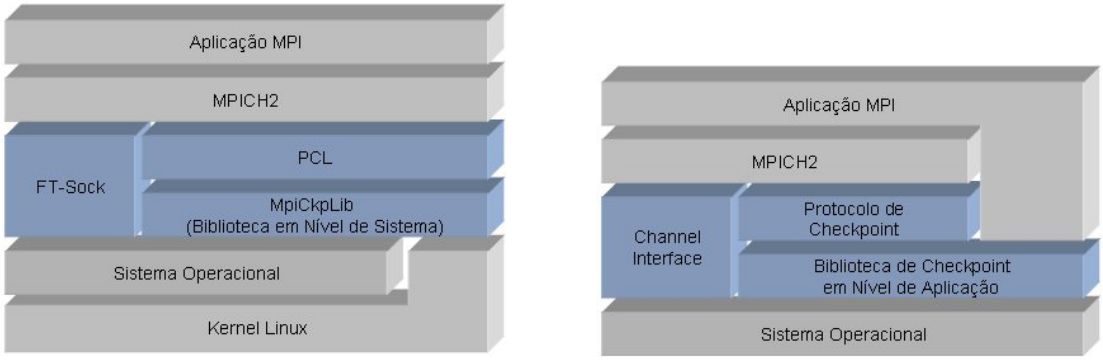
6.3.1 *Checkpointing* em nível de sistema e em nível de aplicação

As duas principais estratégias para criar *checkpoints* de aplicações são: *checkpointing* em nível de aplicação e *checkpointing* em nível de sistema. A principal diferença entre estas estratégias é que *checkpointing* em nível de sistema pode gerar *checkpoints* a qualquer momento durante a execução. Já *checkpointing* em nível de aplicação deve explicitar chamadas para realizar os *checkpoints*. Isto implica, geralmente, que o protocolo de coordenação de *checkpoints* é diferente dependendo da estratégia adotada.

O problema de gerar o estado de um processo da aplicação MPI envolve gravar os valores necessários para recuperar a aplicação: geralmente, valores de registradores e variáveis, pilha de execução e recursos do sistema, como arquivos abertos. Outros dados necessários estão ligados à implementação da biblioteca MPI utilizada. Esses dados se enquadram em três categorias [12]: *buffers* de mensagens da biblioteca; objetos opacos da MPI, que são acessados pela aplicação através de *handles*[3], como objetos de requisição, comunicação, grupos, tipos de dados, erros, etc; estado interno da biblioteca, como filas de mensagens e temporizadores. Esses dados não são possíveis de serem obtidos em nível de aplicação.

[3]Identificador que fornece um meio de acessar um conjunto de dados, embora escondendo informações quanto à sua estrutura.

O componente MpiCkpLib, ilustrado na Figura 6.7(a), implementa a estratégia de *checkpointing* em nível de sistema para processos no Linux. Como discutido na Seção 4.2, *checkpointing* em nível de sistema deve fazer acesso a dados específicos de uma plataforma, gerando *checkpoints* não portáveis. Entretanto, esta abordagem resolve o problema quanto ao acesso dos dados da biblioteca MPI de forma transparente.



(a) (b)

Figura 6.7: *Componentes do módulo IG–Sock com uma biblioteca de checkpoint em nível de sistema* (a) *e em nível de aplicação* (b).

A estratégia de *checkpointing* em nível de aplicação (Figura 6.7(b)) deve ter como auxílio o uso de um pré-compilador que instrumente o código fonte da aplicação, para possibilitar ao programador transparência na obtenção e recuperação do estado do processo. Quanto ao estado da biblioteca MPI, o componente MpiCkpLib tem acesso direto aos *buffers* e objetos opacos da biblioteca, pois está no nível mais baixo da arquitetura da MPICH2. Uma implementação desta estratégia deve coordenar os dados obtidos de ambos os níveis, aplicação e biblioteca, de forma a obter um estado consistente.

A estratégia de *checkpoint* em nível de sistema foi implementada nesta primeira versão da MPICH–IG. A justificativa desta escolha foi pela maior simplicidade quanto à implementação, uma vez que a utilização de uma abordagem em nível de aplicação necessitaria da adaptação de um pré-compilador para instrumentar a biblioteca MPI, o que inviabilizaria, neste momento, uma implementação efetiva do serviço de recuperação por retrocesso de aplicações MPI no InteGrade. Outra justificativa é quanto ao objetivo central deste trabalho, que é definir e implementar uma infra-estrutura robusta para executar aplicações MPI legadas no InteGrade, não sendo o foco até o momento a implementação de *checkpoints* portáveis ou resolver problemas quanto à integridade de comunicação entre ambientes heterogêneos.

6.3.2 Recuperação de aplicações MPI utilizando *checkpoints* em nível de sistema

O protocolo de recuperação de aplicações MPI é ilustrado na Figura 6.8. Uma aplicação MPI que deve ser recuperada é reescalonada pelo GRM (1) e executada pelos LRMs (2). Ao iniciar a aplicação, o módulo IG–PM identifica, através da chamada *PMI_Init* (discutida na Seção 6.2.1), que se trata de uma recuperação. A inicialização termina com a chamada *FT_init* do módulo FT–Sock para iniciar o canal de comunicação. O componente PCL identificará a necessidade de restauração através de uma chamada *PMI_KVS_Get* (3) ao módulo IG–PM. O PCL, por sua vez, faz uma chamada à biblioteca de *checkpoint* para recuperar a aplicação através dos dados obtidos pelo IG–PM (4). A *MpiCkpLib* por sua vez irá utilizar a interface *CkpLib* para recuperar os dados do *checkpoint* (5). Caso a estratégia de recuperação esteja utilizando um repositório distribuído, a *CkpLib* fará uma chamada ao módulo CDRM do gerenciador de aglomerados (6). Esta chamada irá retornar uma lista contendo os repositórios possíveis dos quais se obtém os dados de *checkpoints*. A *CkpLib* então comunica-se com os *CkpReps* definidos na lista (7) para obter os dados de *checkpoint*.

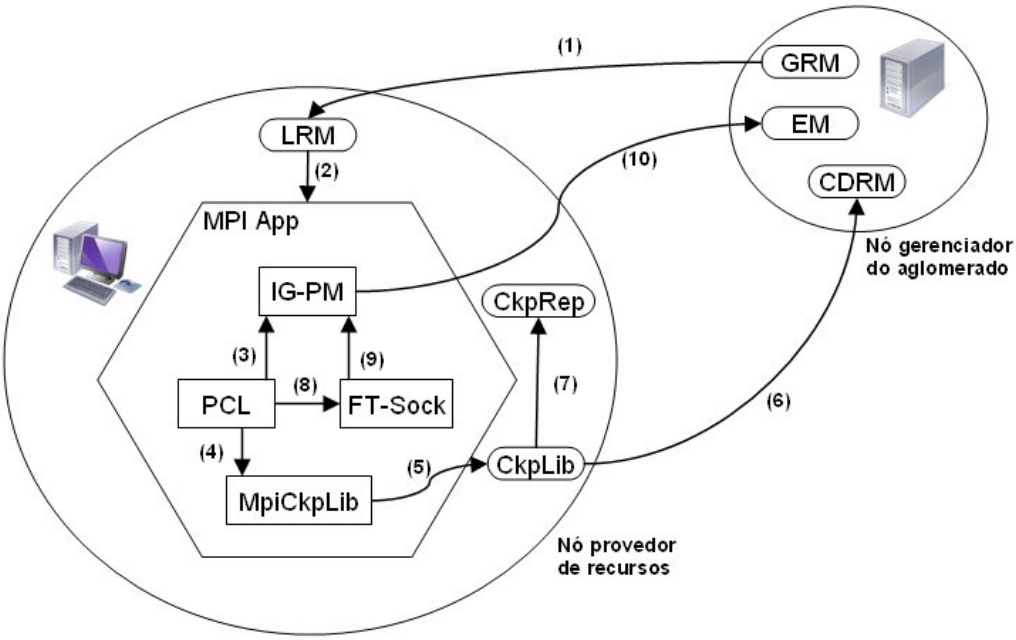


Figura 6.8: *Protocolo de recuperação de aplicações MPI no InteGrade*

Após obter os dados a MpiCkpLib realiza a recuperação do processo através da biblioteca de *checkpointing* BLCR. A BLCR faz a leitura dos dados e restaura o processo. Após a recuperação o PCL deve restabelecer o canal de comunicação com os outros processos. Para tanto, uma chamada *ft_netrestore* ao módulo FT–Sock (8) irá criar um novo soquete TCP/IP servidor e, posteriormente, realizar uma chamada a *PMI_Barrier*

(9) para obter os dados de conexão com os novos processos junto ao EM (10). O EM tem o papel de disseminar as informações de conexão criando uma nova barreira de sincronização e permitindo que todos os processos tenham iniciado um soquete servidor antes de tentar fazer qualquer conexão com outro processo. Finalmente, os canais de comunicação podem ser restabelecidos através do componente FT–Sock.

6.4 Implementação da MPICH–IG

A implementação da MPICH–IG estende a implementação da MPICH2 e MPICH–Pcl para permitir que aplicações MPI possam executar no InteGrade. A biblioteca foi desenvolvida usando a linguagem C e usa a biblioteca de *checkpoint* de nível de sistema BLCR, que se limita a plataformas *Linux* e exige uma uniformidade do hardware.

As únicas alterações necessárias nos componentes do InteGrade foram realizadas nos módulos ASCT e EM. O ASCT teve que ser alterado para possibilitar a execução de aplicações do tipo MPI. No momento da chamada *requestRemoteExecution*[4] do módulo GRM, é enviado um parâmetro que determina o tipo da aplicação como sendo MPI. A Figura 6.9 exibe a tela do ASCT para submissão de aplicações, que foi alterada para possibilitar a execução de aplicações MPI.

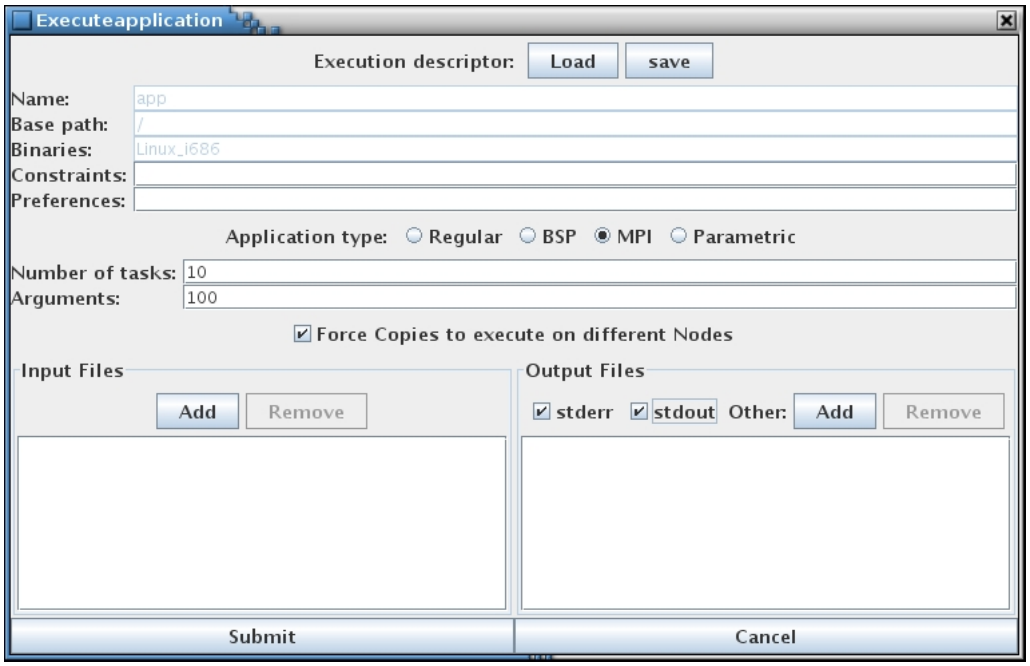


Figura 6.9: *Tela do ASCT para submissão de aplicações MPI no InteGrade*

[4]Solicita a execução de uma aplicação no InteGrade.

O EM sofreu alterações quanto à coordenação de aplicações MPI. O EM funciona como uma fachada para chamadas advindas de outros módulos. Ao receber mensagens, ele as repassa para coordenadores específicos criados no momento da solicitação de coordenação, realizada pelo GRM através da chamada *setExecutionScheduled*. O Código 6.1 mostra parte das definições em IDL da interface pública do EM. O método *registerMpiProcess* foi adicionado para permitir ao EM coordenar o serviço de sincronização e localização dos processos da aplicação MPI. Cada execução e gerenciada por um objeto coordenador específico para cada tipo de aplicação e que é identificado pelo parâmetro *executionRequestId*. Para aplicações MPI, esse coordenador é implementado pela classe *MpiRestartCoordinator*.

Código 6.1: *Interface IDL CORBA do módulo EM do InteGrade*

```
1  module dataTypes {
2          struct ExecutionRequestId {
3                  string requestId;
4                  string processId;
5          };
6          struct MpiConnectInformation {
7                  string kvs;
8                  string processId;
9          };
10         typedef sequence<dataTypes::MpiConnectInformation>
               MpiConnectInformationSequence;
11         struct BspProcessZeroInformation {
12                 boolean isProcessZero;
13                 string processZeroIor;
14         };
15
16         ...
17 };
18
19 module clusterManagement {
20         interface ExecutionManager {
21                 dataTypes::MpiConnectInformationSequence
                       registerMpiProcess(in dataTypes::ExecutionRequestId
                        executionRequestId, in string kvs, in long rank);
22                 dataTypes::BspProcessZeroInformation registerBspProcess
                       (in dataTypes::ExecutionRequestId
                       executionRequestId, in string bspProxyIor);
23                 void setProcessExecutionStarted(...);
24                 long setProcessExecutionFinished(...);
25                 void setExecutionScheduled(...);
26                 void setExecutionRefused(...);
27         };
```

```
28 };
```

A classe *MpiRestartCoordinator*, ilustrada na Figura 6.10, implementa o mecanismo de sincronização descrito na Seção 6.2.1, o qual espera que todos os processos paralelos façam a chamada ao método *registerMpiNode* de uma mesma instância do objeto coordenador para então liberar a barreira de sincronização. Esta classe herda de *RestartCoordinator*, que implementa funcionalidades genéricas, como por exemplo, o processo de recuperação em caso de falhas. O Código A.1, no Apêndice A, exibe a implementação desta classe.

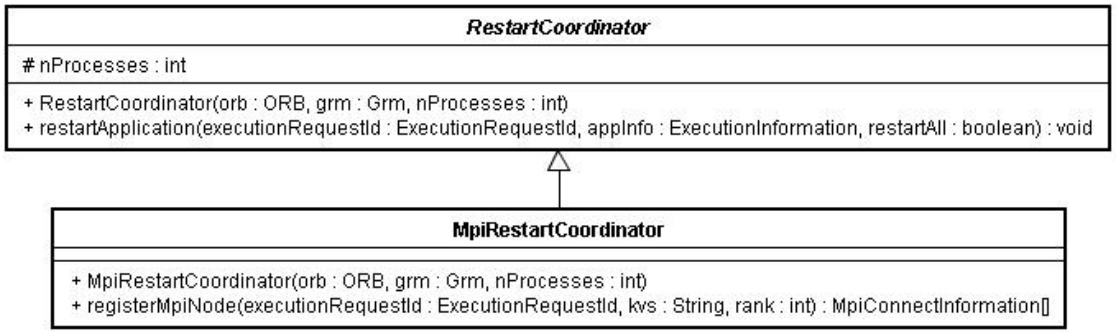


Figura 6.10: *Diagrama de classes do coordenador de aplicações MPI no módulo EM*

6.4.1 IG–PM

O Código 6.2 exibe a *interface* do IG–PM. A implementação desse módulo tem como principais funções:

- *PMI_Init*: carrega os dados de configuração de dois arquivos chamados *execution.conf* e *ckp.conf*, criados pelo LRM antes da execução da aplicação. São obtidos dados quanto ao rank da aplicação, a quantidade de processos participantes, o identificador da aplicação (usado para distinguir a aplicação junto ao EM), dados quanto à conexão com o serviço de nomes do InteGrade, e o intervalo de tempo em que devem ser gerados os *checkpoints*.
- *PMI_Get_size*: informa a quantidade de processos participantes da execução da aplicação.
- *PMI_Get_rank*: fornece um identificador único do processo local, chamado de *rank*.
- *PMI_KVS_Put*: adiciona valores ao KVS (*key–value space*) do processo local.
- *PMI_KVS_Get*: obtém o KVS de algum processo especificado, ou de algum outro KVS especificado pelo *kvsname*.

- *PMI_Barrier*: sincroniza o processo local com os demais através de uma chamada ao método *registerMpiProcess* do módulo EM do InteGrade, utilizando para isso uma biblioteca em C para instanciar o ORB CORBA OIL. Essa biblioteca instancia um ORB como *singleton*, sendo reutilizado em outras chamadas como, por exemplo, a obtenção de referências remotas junto ao serviço de nomes. A referência para o EM é obtida utilizando o serviço de nomes do aglomerado. Após obtida a referência, uma chamada para o método *registerMpiProcess* é realizada, passando como parâmetro o valor do KVS do processo local. Essa chamada permanece bloqueada até o EM sincronizar todos os processos.
- *PMI_Spawn_multiple*: esta função permite que aplicações possam criar novos processos dinamicamente, introduzindo mais nós na execução. Para tanto, faz-se necessário o escalonamento de outros LRMs junto ao GRM, utilizando, por exemplo, um novo protocolo de requisição de execução. Entretanto, este trabalho não trata da elaboração deste protocolo ou do recurso de criação dinâmica de processos provido pela MPICH2, ficando como trabalho futuro sua implementação na MPICH-IG.

Código 6.2: *Interface do módulo IGPM*

```
1  typedef struct PMI_keyval_t
2  {
3      char * key;
4      char * val;
5  } PMI_keyval_t;
6
7  int PMI_Init( int *spawned );
8
9  int PMI_Get_size( int *size );
10
11 int PMI_Get_rank( int *rank );
12
13 int PMI_Barrier( void );
14
15 int PMI_KVS_Put( const char kvsname[], const char key[], const char
       value[]);
16
17 int PMI_KVS_Get( const char kvsname[], const char key[], char value[],
       int length);
18
19 int PMI_Spawn_multiple(int count,
20                        const char * cmds[],
21                        const char ** argvs[],
22                        const int maxprocs[],
```

```
23                          const int info_keyval_sizesp[],
24                          const PMI_keyval_t * info_keyval_vectors[],
25                          int preput_keyval_size,
26                          const PMI_keyval_t preput_keyval_vector[],
27                          int errors[]);
28
29  ...
```

A implementação do módulo IG–PM não fornece, ainda, o mecanismo de comunicação com o módulo de armazenamento de *checkpoints* do InteGrade. A arquitetura prevê a comunicação com o módulo CkpLib para armazenamento e recuperação de *checkpoints*, devendo essa comunicação ser implementada em trabalhos futuros. Na implementação atual o armazenamento de *checkpoints* é local garantindo tratar erros ocasionados pela interrupção da execução, por exemplo, pela perda de comunicação. Entretanto, esta abordagem é inadequada em caso de falhas do provedor de recursos, que ocasiona interrupção na execução da aplicação. A abordagem atual trata-se de uma implementação temporária como forma de contornar a necessidade de armazenar os *checkpointings* gerados, devendo ser substituída pelo armazenamento distribuído de *checkpoints* em um futuro próximo possibilitando, por exemplo, a migração do processo falho para outro provedor de recursos.

6.4.2 IG–Sock

O módulo IG–Sock é definido em três submódulos, que cooperam para atingir a meta de fornecer um ambiente tolerante a falhas para execução de aplicações MPI no InteGrade. A infra-estrutura da MPICH–IG prevê o desacoplamento de implementação entre estes módulos através do uso de *interfaces* escritas em C. Essas *interfaces* definem as operações que são possíveis de serem utilizadas por outros módulos, podendo ser estendidas se necessário. A Figura 6.11 ilustra um diagrama de componentes do IG–Sock, definindo as interfaces utilizadas por cada componente para compor o serviço de *checkpoint*.

FT–Sock

Este componente foi portado a partir da MPICH–Pcl, implementando as *interfaces mpidi_ch3_impl.h* (Código A.2) e *mpidi_ig_restore.h*. A *interface mpidi_ig_restore.h* define a função

```
void ft_netrestore(int myrank);
```

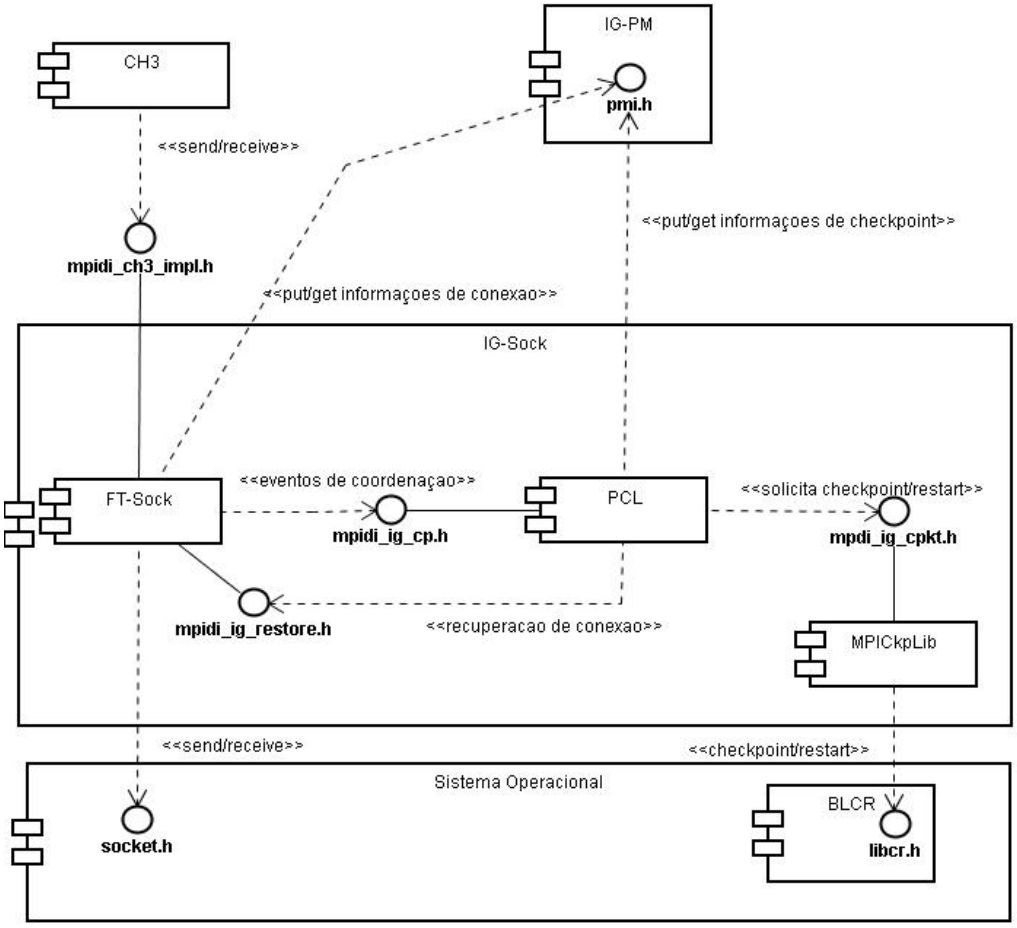


Figura 6.11: *Diagrama de componentes do IG–Sock*

que realiza a operação de reiniciar a conexão durante a recuperação do processo. Esta operação inicia um novo *socket* servidor e restabelece a conexão com os outros processos. A *interface mpidi_ch3_impl.h* tem como principais funções

```
int MPIDI_CH3_Init(int has_parent,
                   MPIDI_PG_t *pg_ptr,
                   int pg_rank );
```

que inicializa as variáveis específicas de comunicação, como a criação dos soquetes e estruturas que mantêm as conexões virtuais utilizadas pelos níveis mais acima para abstrair a coleta de informações sobre o estado do canal. A função

```
int MPIDI_CH3_iSend(MPIDI_VC_t * vc,
              MPID_Request * sreq,
              void * hdr,
    MPIDI_msg_sz_t hdr_sz);
```

envia dados contidos em *hdr* para outro processo, definido em *sreq*. A função de envio é não-bloqueante. Caso não seja possível enviar neste momento, a mensagem é retornada,

podendo ser reenviada posteriormente. As mensagens enviadas por outros processos que chegam são colocadas em uma fila de mensagens de entrada e são acessadas através da variável *msg_reorder_queue* dentro da estrutura *MPIDI_VC*. Essa estrutura é acessível pela estrutura *MPIDI_PG*, que funciona como um *wrapper* dos dados referentes a todos os processos de um grupo.

PCL

Este componente foi portado a partir da MPICH–Pcl e define a implementação do protocolo coordenado Chandy/Lamport (Seção 6.2.2). As principais funções fornecidas pela *interface mpidi_ig_cp.h* (Código A.3) são:

- *int pcli_init*: inicializa variáveis e estados do protocolo, além de realizar a chamada de inicialização da biblioteca de *checkpoint*.
- *int FT_can_send*: define se alguma mensagem pode ser enviada para o processo, especificado por uma parâmetro.
- *int FT_msg_recv*: informa ao PCL o recebimento de uma mensagem, devendo esta informar se a mensagem passará ou não para a camada superior, ou seja deve ser colocada da fila de entrada se o canal estiver livre, e não deve se colocada na fila de entrada se o canal estiver bloqueado.

Algumas funções necessitam, como parâmetros, funções de *callback* para executar algumas operações referentes ao protocolo, no caso o protocolo PCL, para enviar mensagens bloqueadas ou mensagens que não foram entregues durante a onda de *checkpoint*. O componente PCL é responsável por iniciar uma *thread* de sistema que periodicamente começa uma onda de *checkpoint*, caso seja possível. O intervalo é obtido a partir de um KVS criado pelo módulo IG–PM, o qual pode ser obtido pela chamada da função *PMI_KVS_Get*.

MPICkpLib

A estratégia de *checkpoint* implementada nesta versão da MPICH–IG foi a de *checkpointing* em nível de sistema, utilizando a biblioteca *Berkeley Lab Checkpoint/Restart* (BLCR) [53, 41]. Uma vantagem da utilização da BLCR é a capacidade de realizar *checkpoint* das mais variadas aplicações, inclusive recuperando recursos como arquivos abertos e *threads* do sistema, além de ser implementada para *kernel Linux* 1.4.x e 2.6.x em arquiteturas x86 e x86–64, cobrindo a maioria das plataformas *Linux* utilizadas atualmente.

O módulo PCL foi implementado em C e utiliza a biblioteca BLCR através de uma fachada que define o uso dessa biblioteca pelo protocolo de coordenação. Essa fachada é composta pelas seguintes funções:

- *int checkpoint_init(void)*: inicia aspectos específicos da biblioteca, como *buffers* e variáveis de controle.
- *int checkpoint_create(void)*: função chamada pelo protocolo de *checkpoint*, determinando o momento para realizar o *checkpoint* e armazená-lo para posterior recuperação. A implementação atual prevê o armazenamento local, embora seja possível fazer o armazenamento remoto enviando para o módulo CkpLib que é utilizado para armazenar os dados de *checkpoint*.
- *int checkpoint_restore(void)*: após o protocolo de *checkpoint* carregar os dados do *checkpoint*, esta função é chamada para recuperar o estado da aplicação. Os dados são obtidos localmente, embora possam também ser obtidos através do módulo CkpLib.

Os dados de *checkpoint* serão compostos possivelmente, por variáveis e pela pilha de chamadas de funções em nível da aplicação, que será monitorada pela biblioteca através de funções incluídas por um pré-compilador. Também devem ser armazenados, como parte do *checkpoint*, *buffers* e estruturas da MPICH2, acessíveis pela biblioteca através de adaptações na CI.

6.5 Portando uma aplicação MPI para executar no InteGrade

O InteGrade fornece, através do ASCT, o serviço de submissão de aplicações. O primeiro passo para submeter uma aplicação é enviar o binário para o repositório de aplicações do InteGrade. O binário de uma aplicação MPI deve ser gerado utilizando a MPICH–IG antes de ser submetido ao repositório de aplicações. A Figura 6.12 ilustra o mecanismo utilizado para portar uma aplicação MPI para o ambiente de grade.

O primeiro passo é gerar a MPICH–IG a partir da MPICH2 e incluir os módulos IG–Sock e IG–PM. O pacote MPICH2 consiste de um compilador e uma biblioteca que é ligada aos programas MPI. A MPICH–IG não necessita ser instalada nas máquinas que rodam a aplicação. Sendo assim, não são criadas novas dependências para o InteGrade. Entretanto, a biblioteca de *checkpointing* BLCR utilizada pelo MPICH–IG deve ser instalada em todas as máquinas que executam a aplicação. Esta dependência limita a execução de aplicações MPI a ambientes Linux homogêneos. A compilação da MPICH–IG só é

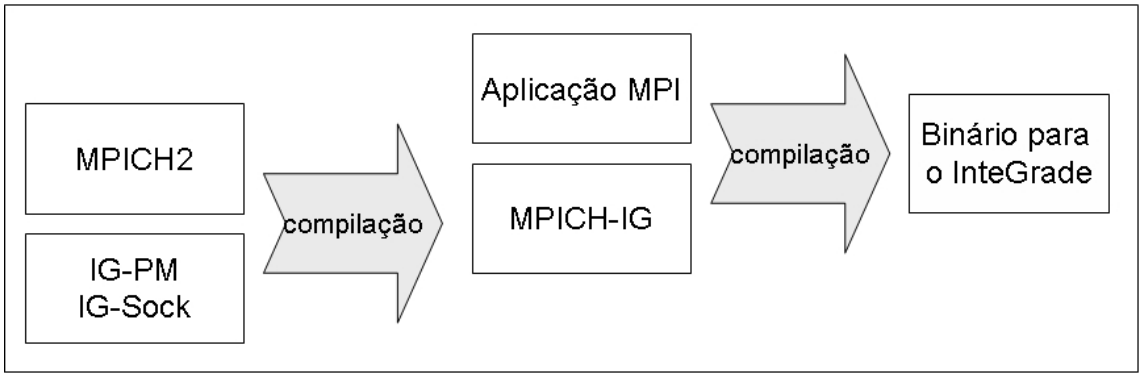


Figura 6.12: *Portando uma aplicação MPI para o InteGrade*

necessária uma vez possibilitando assim compilar as aplicações MPI que executarão no InteGrade.

A MPICH-IG não garante portabilidade total de aplicações MPI para o ambiente de grade. Uma vez que o ambiente de grade é mais instável que o comumente utilizado com a MPICH2, aplicações que necessitem, por exemplo, de velocidade de comunicação para executarem corretamente podem ter que ser reprojetadas e reimplementadas.

6.6 Considerações sobre a Abordagem Proposta

Objetivando uma avaliação da infra-estrutura proposta para execução de aplicações MPI no InteGrade, nas próximas seções serão apresentadas algumas considerações sobre o trabalho.

6.6.1 A Utilização de MPI no InteGrade

Devido ao uso do canal de comunicação por soquetes implementado pela MPICH–IG, não se espera uma perda significativa de desempenho quanto à execução da aplicação na grade, a não ser quando a comunicação ultrapassar os limites de um domínio administrativo, o que geralmente reduz a capacidade das linhas de conexão podendo causar um atraso maior na comunicação entre os processos. Não foi possível realizar testes entre redes de longa distância, principalmente devido à existência de *firewalls* que restringem a comunicação entre as redes disponíveis. Para tanto, mecanismos de travessia de *firewall* devem ser implementados para possibilitar a comunicação entre os objetos CORBA que compõem os módulos do InteGrade, bem como para possibilitar a comunicação entre os processos da aplicação MPI que executam nos diversos provedores de recursos.

Uma vantagem evidente nesta implementação é a facilidade que os usuários de programas MPI terão para a execução das aplicações na grade. Serviços como o repositó-

rio de aplicações e o escalonamento eficiente de recursos automatizam a distribuição e a execução do binário transparentemente. Este fato não ocorre comumente na execução de aplicações em outros ambientes de grade ou com outras implementações de MPI, uma vez que o usuário deve indicar ou definir quais recursos serão utilizados, além de se preocupar com toda a infra-estrutura de execução, como, por exemplo, copiar o binário para todas as máquinas que farão parte da execução da aplicação.

A implementação da arquitetura não gera dependências no InteGrade com relação à biblioteca MPI, mas sim o inverso. A implementação utiliza os serviços da grade como um *toolkit*, permitindo minimizar as dependências do InteGrade em relação ao ambiente de execução. A inclusão da BLCR como forma de realizar *checkpoint* é uma dependência ligada somente à implementação da MPICH–IG, podendo ser substituída por qualquer biblioteca de *checkpoint*, desde que implemente a interface descrita na Seção 6.4.2.

O processo de submissão de aplicações MPI não gerou alterações no módulo LRM do InteGrade, que coordena o processo de alocação local de recursos e execução da aplicação. Esta característica demonstra que a implementação manteve o tipo da aplicação que deverá ser iniciada transparente para o provedor de recursos, tornando o processo de portabilidade do LRM para outros ambientes, como o Windows, independente da MPICH–IG.

Quanto ao suporte às funções da MPI–2, a MPICH–IG possibilita a utilização de quase todas as funções, exceto aquelas relacionadas com a criação de processos dinâmicos (*spawned process*), uma vez que estas necessitam de um novo protocolo no InteGrade para escalonar novos processos junto ao GRM e associá-los à aplicação em execução.

6.6.2 A Portabilidade da Infra-estrutura de *Checkpoint*

A infra-estrutura apresentada propõe uma forma de lidar com os aspectos referentes à execução de aplicações MPI no InteGrade, definindo para isso os componentes e operações necessárias para localizar, sincronizar e salvar os estados de execução de uma aplicação MPI executando em vários provedores de recurso. A separação entre o canal de comunicação e o protocolo de *checkpointing* facilita a portabilidade do mecanismo de tolerância a falhas já desenvolvido para ser utilizado com outros canais de comunicação a fim de melhorar o desempenho em ambientes específicos.

A implementação de *checkpoints* não portáveis (isto é, em nível de sistema), estado atual da MPICH–IG, não impossibilita a migração, embora limite as máquinas que poderão ser utilizadas na execução àquelas cuja arquitetura seja compatível com os *checkpoints* gerados. Em contrapartida, o uso de uma biblioteca em nível de sistema pode garantir a recuperação de uma ampla quantidade de aplicações, diferentemente de uma biblioteca em nível de aplicação, já que o pré-compilador geralmente não prevê

a recuperação de diversos recursos do sistema, como por exemplo, *threads* e arquivos abertos. De qualquer forma, uma implementação que crie *checkpoints* portáveis seria necessária caso se pretenda fazer um uso mais amplo dos recursos disponibilizados pela grade.

6.6.3 Desempenho

O suporte de MPI no InteGrade teve como base a utilização da MPICH2, que tem como característica a possibilidade de utilizar implementações específicas para comunicação com algumas plataformas, melhorando a eficiência das aplicações, além de possibilitar uma adaptação mais fácil do mecanismo de *checkpointing* nestas novas implementações.

A Figura 6.13 exibe um gráfico de *tempo* × *número de processos* comparando o desempenho da MPICH–IG e da MPICH2 sem a utilização do mecanismo de *checkpointing*. As medida foram realizadas em um ambiente composto pelas máquinas descritas na Tabela B.1 conectadas através de uma rede local de alta velocidade, onde é possível fazer comparações com a MPICH2. Para o calculo de desempenho foi utilizado um programa de multiplicação de matrizes (Código B.1) que tem uma maior escalibilidade[5] e permite fazer comparações variando o tamanho da entrada e do número de processos.

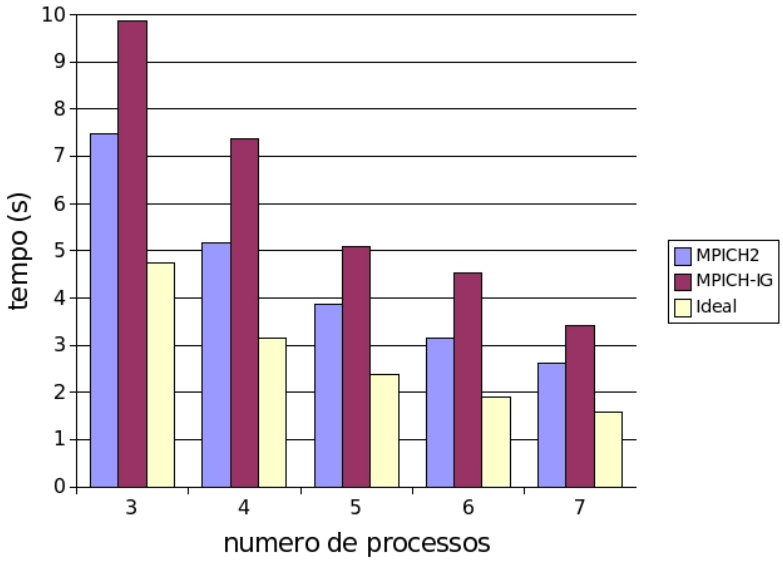


Figura 6.13: *Comparação entre MPICH2 e MPICH–IG para uma matriz de entrada 1000×1000*

Como demonstrado no gráfico o tempo de execução da MPICH–IG comparado a MPICH2 é maior, contudo esta diferença se mantém próxima a medida que o número de

[5] Veja Seção 3.4.

nós aumenta. Esta característica pode ser observada em outros gráficos de desempenho no Apêndice B. Pode-se concluir que a implementação da MPICH–IG não representa perdas significativas quanto ao desempenho. A MPICH–IG realiza a transferência da aplicação através do repositório de aplicações do InteGrade para os provedores de recurso, enquanto na MPICH2 supõe-se que a aplicação já se encontra disponível para execução em cada máquina. Outra diferença é o processo de localização e sincronização, que na MPICH2 é realizado através de comunicação direta através de soquetes TCP/IP, enquanto que na MPICH–IG cada nó deve realizar uma chamada remota através do ORB, gerando um maior *overhead*. Entretanto, ambos os fatores citados não implicam em perda significativa na execução de aplicações de alto desempenho, uma vez que as perdas de transferência da aplicação e sincronização entre os nós geralmente são irrelevantes se comparadas ao tempo total da execução. Outro fator é que o canal de comunicação utilizando soquetes TCP/IP não foi alterado, não influenciando no desempenho de aplicações que demandam muita comunicação.

A Figura 6.13 ilustra ainda um comparativo entre os sistemas e o ideal, caso em que a aplicação é perfeitamente escalável. O programa de multiplicação de matrizes apresentado define comunicação entre os processos para distribuir partes da computação. Esta comunicação degrada o tempo de execução a medida que o tamanho da matriz aumenta. Portanto, diferenças entre os sistemas e o ideal são toleráveis. O medida do tempo ideal foi realizada através da execução do programa de matrizes utilizando somente o processamento local da máquina com maior poder computacional disponível (Tabela B.1). Para obter o tempo ideal na execução com vários processos o tempo ideal com um processo foi proporcionalmente utilizado.

A MPICH–IG permite, ainda, utilizar o ambiente de grade oportunista do InteGrade, que oferece entre outras vantagens, um escalonamento inteligente da aplicação em máquinas provedoras de recursos, fazendo análise dos padrões de uso das máquinas e melhorando o desempenho com um escalonamento eficiente e em uma escala bem maior que a oferecida por um aglomerado limitado por uma rede local.

A implementação apresentada neste trabalho é o passo inicial para desenvolver um ambiente de execução de aplicações MPI que use efetivamente os recursos disponibilizados por uma grade oportunista. Uma vez que o ambiente é bastante dinâmico e heterogêneo, uma arquitetura que utilize os serviços disponibilizados pela grade e, ao mesmo tempo, possibilite o uso de recursos específicos das máquinas provedoras (através de implementações especializadas da *channel interface*) pode garantir uma execução mais eficiente, compensando um pouco a perda de desempenho causada pela comunicação em redes de grande área.

Ainda não foi possível realizar testes efetivos quanto ao desempenho do mecanismo de *checkpoint* pois o mesmo se encontra em desenvolvimento. Entretanto, trabalhos

relacionados a MPICH–Pcl, no qual se baseia esta implementação, demonstram que o uso de um protocolo de *checkpointing* coordenado bloqueante não tem um bom desempenho em grades computacionais, devido à maior latência de sincronização entre as aplicações nas redes de grande área [13]. Contudo, em ambientes de aglomerado não há perdas significativas, justamente pela sincronização se dar em redes locais de alta velocidade. Esperamos concluir este mecanismo bloqueante e publicar resultados quantitativos como parte de trabalhos futuros.

CAPÍTULO 7

Conclusões

A MPICH–IG tem como principal objetivo definir uma infra-estrutura que possibilite aplicações MPI legadas executarem no InteGrade, usufruindo dos recursos e serviços fornecidos pela grade. O desacoplamento entre comunicação e gerenciamento da biblioteca de programação paralela, que reflete a arquitetura da MPICH2, possibilita o uso eficiente e bem estruturado dos recursos do ambiente de execução. Esta característica permitiu a definição de uma infra-estrutura para desenvolvimento de estratégias de tolerância a falhas que utilizam os serviços fornecidos pelo InteGrade.

A MPICH–IG possibilitou, efetivamente, que aplicações MPI desenvolvidas para ambientes de aglomerados possam ser submetidas à grade de forma transparente, não exigindo que o usuário defina quais provedores de recursos são necessários para executar a aplicação. Esta estratégia é diferente da elaborada pela MPICH–G2 onde o usuário tem que definir o conjunto de máquinas necessárias para a execução, ou pela MPICH–V, que necessita que o aglomerado já esteja configurado e que a aplicação se encontre em todas as máquinas para iniciar sua execução.

Esta infra-estrutura traz ainda a possibilidade de adaptar várias implementações existentes da *Channel Interface* (que define a comunicação na MPICH2) para trabalhar com algum protocolo de coordenação de *checkpoint* já implementado, uma vez que a infra-estrutura prevê o desacoplamento dos mecanismos de criação de *checkpoint*, coordenação de *checkpointing* e canal de comunicação.

7.1 Trabalhos Futuros

Destacamos aqui algumas modificações e sugestões que podem ampliar as funcionalidades da MPICH–IG, possibilitando uma melhor qualidade de serviço na execução de aplicações MPI.

7.1.1 Modificações no InteGrade para melhorar a execução de aplicações

- Utilizar uma biblioteca padrão em nível de sistema para realização de *checkpointing*: uma vez que o InteGrade permite a execução de aplicações variadas, a utilização de uma biblioteca em nível de sistema para realizar *checkpointing* poderia garantir a recuperação de aplicações para as quais não foi prevista uma implementação específica de *checkpointing*. A biblioteca BLCR utilizada na implementação atual da MPICH–IG poderia ser utilizada para prover um mecanismo de recuperação padrão para qualquer aplicação, não se restringindo somente a MPI.
- Modificar o escalonamento do InteGrade: ampliar o escalonamento do InteGrade para tratar o contexto de hardware específico para comunicação. Essas informações poderiam ser cruzadas com os requisitos específicos do binário MPI submetido. O binário pode ser compilado para trabalhar com uma arquitetura de comunicação específica, por exemplo, IBM SP, *shared memory*, InfiniBand, SCTP, BG/L, etc. Isto garantiria sua execução nestas máquinas e obtendo um melhor desempenho.

7.1.2 Modificações na implementação do módulo IG–Sock

- Modificar a IG–Sock para tratar falhas advindas da própria aplicação: O LRM do InteGrade só classifica uma falha como sujeita a recuperação quando a aplicação é encerrada abruptamente. Se a aplicação falha por um erro de comunicação, por exemplo, quando interrompida por queda momentânea da rede (muito comum em ambientes de grade oportunista), então a aplicação irá ser encerrada. Neste caso o LRM não detectará o erro como falha de rede e o interpretará como um erro interno da aplicação, o que pode ocasionar um atraso indesejável, onde o usuário terá que submeter a aplicação novamente. A implementação da IG–Sock pode ser condicionada para que, em caso de falhas, requisite ao módulo IG–PM que envie uma mensagem de solicitação de recuperação ao EM. Isso traria um maior grau de transparência para o usuário da grade ao executar aplicações MPI, além de um ganho de desempenho perante falhas de comunicação.
- Criar protocolos de recuperação de um único nó: uma vez identificada uma falha possível de recuperação, os protocolos de recuperação, geralmente, retrocedem todos os processos da aplicação para um estado consistente. Esta estratégia simplifica o protocolo de recuperação. Entretanto, perde-se as computações realizadas após o último *checkpoint* em processos que não sofreram falhas. A implementação de protocolos que utilizem estratégias de *loging* de mensagens poderia melhorar a eficiência da aplicação através da recuperação do nó falho somente. Esta abordagem

também poderia minimizar o tempo gasto com uma nova sincronização entre todos os processos durante a recuperação.

- Gerar novos processos: o padrão MPI–2 define operações que possibilitam a criação de novos processos dinamicamente. O módulo IG–PM poderia solicitar ao módulo GRM do InteGrade novas execuções do binário da aplicação em outros nós provedores de recursos, necessitando para isso definir um novo protocolo para solicitar este tipo de execução ao GRM. Possivelmente, uma nova sincronização para realizar a conexão com os novos nós se fará necessária. Essa implementação irá permitir que aplicações MPI–2 que utilizem rotinas *spawned* possam ser utilizadas no InteGrade.
- Criar uma CI com base no OIL: o ORB CORBA OIL, utilizado pelo InteGrade define um canal de comunicação que abstrai uma grande quantidade de problemas de comunicação que não necessitam ser tratadas em níveis superiores na grade, como por exemplo, recuperação de conectividade entre os nós, escolha de protocolos de comunicação eficientes, transposição de *firewalls* e NAT, re-estabelecimento de conexão com processos migrados, entre outros. Um canal de comunicação usando um ORB pode substituir a comunicação padrão em caso de se precisar tratar algum problema não previsto pela implementação padrão do canal de comunicação. A comunicação através de um ORB pode garantir também a interoperabilidade em ambientes extremamente heterogêneos, podendo estabelecer comunicação entre aplicações que disponibilizam múltiplos binários para ambientes diferentes. A comunicação através de um ORB pode ser menos eficiente, mas seria uma alternativa em situações onde há empecilhos para a comunicação padrão, como *firewall*, queda constante de comunicação com dispositivos móveis, NAT, etc.

7.2 Considerações Finais

O projeto InteGrade visa a construção de um ambiente de grade computacional que possa ser utilizado em instituições que possuem redes privativas e recursos a serem explorados, criando uma infra-estrutura que permite utilizar os recursos ociosos sem prejudicar os usuários locais. Uma forma de garantir a aplicabilidade de grades nesses ambientes é fornecer serviços úteis para computação de alto desempenho, como por exemplo, a possibilidade de execução de aplicações MPI, as quais são bastante utilizadas para este fim.

A MPICH–IG possibilitou que aplicações MPI, antes utilizadas somente em ambientes de aglomerados limitados, possam ser utilizadas em uma escala bem maior, além de continuar tendo um bom desempenho em ambientes oportunistas devido à caracterís-

tica de escalonamento eficiente do InteGrade. A utilização de aplicações paralelas de alto desempenho em grades apresenta outros problemas mais sofisticados e que não foram tratados aqui como, por exemplo, a latência de comunicação entre nós muito distantes, os quais só podem ser resolvidos através de um escalonamento que leve em consideração a distância entre os provedores de recursos. A MPICH–IG é um ponto de partida para o desenvolvimento de uma infra-estrutura robusta para execução de aplicações MPI no InteGrade.

Referências Bibliográficas

[1] ALASDAIR, R; BRUCE, A; MILLS, J. G; SMITH, A. G. **CHIMP/MPI user guide**. Technical Report, Edinburgh Parallel Computing Centre, Jun 1994.

[2] ALLCOCK, B; BESTER, J; BRESNAHN, J; CHERVENAK, A. L; FOSTER, I; KESSELMAN, C; MEDER, S; NEFEDOVA, V; QUESNEL, D; TUECKE, S. **Data management and transfer in high-performance computational grid environments**. Parallel Computing, 2002.

[3] ASHTON, D. **SMPD PMI wire protocol reference manual**. Draft, Mathematics and Computer Science Division, Argonne National Laboratory, Oct 2007.

[4] ASHTON, D; GROPP, W; LUSK, E; ROSS, R; TOONEN, B. **MPICH2 design document**. Draft, Mathematics and Computer Science Division, Argonne National Laboratory, Oct 2003.

[5] ASHTON, D; GROPP, W; THAKUR, R; TOONEN, B. **The CH3 design for a simple implementation of ADI-3 for MPICH with a TCP-based implementation**. Technical Report, Mathematics and Computer Science Division, Argonne National Laboratory, Sep 2003.

[6] BAL, H. E; STEINER, J. G; TANENBAUM, A. S. **Programming languages for distributed computing systems**. ACM Computing Surveys, New York, USA, 21(3), Aug 1989.

[7] BASNEY, J; LIVNY, M. **Managing network resources in Condor**. Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9), p. 298–299, Aug 2000.

[8] BERNSTEIN, P. A. **Middleware: A model for distributed system services**. Communications of the ACM, 39(2), Feb 1996.

[9] BONORDEN, O; JUURLINK, B; VON OTTE, I; RIEPING, I. **The paderborn university BSP (PUB) library – design, implementation and performance**. 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, IEEE Computer Society, p. 99–104, 1999.

[10] BOOTH, D; HAAS, H; MCCABE, F; NEWCOMER, E; CHAMPION, M; FERRIS, C; ORCHARD, D. **Web services architecture**. W3C Working Draft, Aug 2003.

[11] BOUTEILLER, A; LEMARINIER, P; KRAWEZIK, G; CAPPELLO, F. **Coordinated checkpoint versus message log for fault tolerant MPI**. In proceedings of The 2003 IEEE International Conference on Cluster Computing, Honk Hong China, Dec 2003.

[12] BRONEVETSKY, G; MARQUES, D; PINGALI, K; STODGHILL, P. **Automated application-level checkpointing of MPI programs**. In Principles and Practices of Parallel Programming, San Diego, CA, USA, Jun 2003.

[13] BUNTINAS, D; COTI, C; HÉRAULT, T; LEMARINIER, P; PILARD, L; REZMERITA, A; RODRIGUEZ, E; CAPPELLO, F. **Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols**. Future Generation Compumter Systems, 24(1):73–84, 2008.

[14] BURNS, G; DAOUD, R; VAIGL, J. **LAM: An Open Cluster Environment for MPI**. Proceedings of Supercomputing Symposium, p. 379–386, 1994.

[15] BUTLER, R. M; GROPP, W; LUSK, E. **Components and interfaces of a process management system for parallel programs**. Parallel Computing, 27(11):1417–1429, 2001.

[16] BUTLER, R. M; LUSK, E. L. **Monitors, messages, and clusters: the P4 parallel programming system**. Parallel Computing, 20(4):547–564, 1994.

[17] BUYYA, R. **High Performance Cluster Computing: Architectures and Systems**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[18] CALKIN, R; HEMPEL, R; HOPPE, H. C; WYPIOR, P. **Portable programming with the PARMACS message-passing library**. Parallel Computing, Elsevier Science Publishers, 20(4):615–632, 1994.

[19] CAMARGO, R. Y. **Armazenamento distribuído de dados e checkpointing de aplicações paralelas em grades oportunistas**. Doctoral Thesis, Department of Computer Science - University of São Paulo, May 2007.

[20] CASANOVA, H. **Distributed computing research issues in grid computing**. ACM SIGACT News, 33(3):50–70, 2002.

[21] CHANDY, K. M; LAMPORT, L. **Distributed snapshots: Determining global states of distributed systems**. ACM Transactions on Computer Systems, 3(1):63–75, Feb 1985.

[22] CHERVENAK, A. L; FOSTER, I; KESSELMAN, C; SALISBURY, C; TUECKE, S. **The data grid: Towards and architecture for the distributed management and analysis of large scientific data sets**. Journal of Network and Computer Applications, 23(3):187–200, 2000.

[23] CIRNE, W; SANTOS, E. **Grids computacionais de alto desempenho a serviços sob demanda**. Mini-curso no 23º Simpósio Brasileiro de Redes de Computadores, May 2005.

[24] CONDOR. **Condor - high throughput computing**. `http://www.cs.wisc.edu/condor/`, Jan 2008.

[25] COSTA, F. M; KON, F. **Novas tecnologias de middleware: Rumo a flexibilização e ao dinamismo**. Mini-curso no SBRC, 2005.

[26] COULOURIS, G; DOLLIMORE, J; KINDBERG, T. **Distributed Systems: Concepts and Design**. Addison Wesley, 2005.

[27] CZAJKOWSKI, K; FITZGERALD, S; FOSTER, I; KESSELMAN, C. **Grid information services for distributed resource sharing**. In Proceedings of the lOth IEEE International Symposium on High Performance Distributed Computing (HPDC-10), Aug 2001.

[28] ELNOZAHY, M; ALVISI, L; WANG, Y.-M; JOHNSON, D. B. **A survey of rollback-recovery protocols in message-passing systems**. ACM Computing Surveys, 34(3):375–408, Sep 2002.

[29] FAGG, G. E; DONGARRA, J. J. **Building and using a fault-tolerant MPI implementation**. Source International Journal of High Performance Computing Applications archive, Sage Publications, 18(3):353–361, Aug 2004.

[30] FITZGERALD, S; FOSTER, I; KESSELMAN, C; VON LASZEWSKI, G; SMITH, W; TUECKE, S. **A directory service for configuring high-performance distributed computations**. In Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC-6), Aug 1997.

[31] FOSTER, I. **The grid: A new infrastructure for 21st century science**. Physics Today, 55(2):42, Feb 2002.

[32] FOSTER, I. **Globus toolkit version 4: Software for service-oriented systems**. IFIP International Conference on Network and Parallel Computing, p. 2–13, 2006.

[33] FOSTER, I; GEISLER, J; GROPP, W; KARONIS, N; LUSK, E; THIRUVATHUKAL, G; TUECKE, S. **Wide-area implementation of the message passing interface**. Parallel Computing, 24(12–13):1735–1749, 1998.

[34] FOSTER, I; GEISLER, J; NICKLESS, B; SMITH, W; TUECKE, S. **Software infrastructure for the i-way high-performance distributed computing experiment**. Proceedings 5TH IEEE Symposium on High Performance Distributed Computing, IEEE Computer Society Press, p. 562–571, Aug 1997.

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

[36] FOSTER, I; KESSELMAN, C. **The Globus project: a status report**. Future Generation Computer Systems, 15(5–6):607–621, 1999.

[37] FOSTER, I; KESSELMAN, C. **The Grid: Blueprint for a New Computing Infrastructure**. Morgan Kaufmann Publishers, San Francisco, 1999.

[38] FOSTER, I; KESSELMAN, C; NICK, J. M; TUECKE, S. **The physiology of the grid: An open grid services architecture for distributed systems integration**. Global Grid Forum, 22, Jun 2002.

[39] FOSTER, I; KESSELMAN, C; TSUDIK, G; TUECKE, S. **A security architecture for computational grids**. 5th ACM Conference on Computer and Communications Security Conference, p. 83–92, 1998.

[40] FOSTER, I; KESSELMAN, C; TUECKE, S. **The anatomy of the grid: Enabling scalable virtual organizations**. Supercomputer Applications, 2001.

[41] FUTURE TECNOLOGIES GROUP. **Berkeley lab checkpoint/restart (BLCR)**. `http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml`, Feb 2008.

[42] GEIST, A; BEGUELIN, A; DONGARRA, J; JIANG, W; MANCHEK, R; SUNDERAM, V. S. **PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing**. The MIT Press, 1994.

[43] GLOBUS. **The globus alliance**. `http://www.globus.org/`, Jan 2008.

[44] GOLDCHLEGER, A. **Integrade: Um sistema de middleware para computação em grade oportunista**. Master's thesis, Department of Computer Science - University of São Paulo, Dec 2004.

[45] GRIMSHAW, A; FERRARI, A; LINDAHL, G; HOLCOMB, K. **Metasystems**. Communications of the ACM, 41(11), Nov 1998.

[46] GROPP, W; LUSK, E. **Create a new MPICH device using the channel interface**. Technical Memorandum, Mathematics and Computer Science Division, Argonne National Laboratory, Jul 1996.

[47] GROPP, W; LUSK, E; ASHTON, D; ROSS, R; THAKUR, R; TOONEN, B. **MPICH abstract device interface**. Reference Manual, Mathematics and Computer Science Division, Argonne National Laboratory, May 2003.

[48] GROPP, W; LUSK, E; DOSS, N; SKJELLUM, A. **High-performance, portable implementation of the MPI Message Passing Interface Standard**. Parallel Computing, 22(6):789–828, 1996.

[49] GROPP, W; LUSK, E; SKJELLUM, A. **Using MPI: Portable Parallel Programming with the Message Passing Interface**, volume 1. The MIT Press, 1994.

[50] GROUP OF DISTRIBUTED SYSTEMS – PUC-RIO. **OIL – the Lua object request broker**. `http://oil.luaforge.net/`, Feb 2008.

[51] GUDGIN, M; HADLEY, M; MENDELSOHN, N; MOREAU, J.-J; NIELSEN, H. F; KARMARKAR, A; LAFON, Y. **SOAP version 1.2**. W3C Recommendation, Apr 2007.

[52] GUPTA, R; KASHYAP, M; FANG, Y.-C; IQBAL, S. **Considering middleware options in high perfrmance computing clusters**. Dell Power Solutions, High-Performance Computing, Feb 2005.

[53] HARGROVE, P. H; DUELL, J. C. **Berkeley lab checkpoint/restart (BLCR) for linux clusters**. In Proceedings of SciDAC 2006, Jun 2006.

[54] HILL, J. M. D; MCCOLL, B; STEFANESCU, D. C; GOUDREAU, M. W; LANG, K; RAO, S. B; SUEL, T; TSANTILAS, T; BISSELING, R. H. **BSPlib: The BSP programming library**. Parallel Computing, 24(14):1947–1980, 1998.

[55] INDIANA UNIVERSITY. **LAM/MPI Parallel Computing**. `http://www.lam-mpi.org/`, Feb 2008.

[56] INTEGRADE. **Integrade home page**. `http://www.integrade.org.br`, Jan 2008.

[57] **JacORB – the free java implementation of the OMG's CORBA standard**. `http://www.jacorb.org/`, Feb 2008.

[58] KARABLIEH, F; BAZZI, R. A; HICKS, M. **Compiler-assisted heterogeneous checkpointing**. Proceedings. 20th IEEE Symposium, Reliable Distributed Systems, p. 56–65, 2001.

[59] KARONIS, N. T; TOONEN, B; FOSTER, I. **MPICH-G2: A grid-enabled implementation of the message passing interface**. Journal of Parallel and Distributed Computing, 63(5):551–563, May 2003.

[60] KON, F; CAMPBELL, R. H; MICKUNAS, D. M; NAHRSTEDT, K; BALLESTEROS, F. J. **2K: A distributed operating system for dynamic heterogeneous environments**. Aug 2000.

[61] KRAUTER, K; BUYYA, R; MAHESWARAN, M. **A taxonomy and survey of grid resource management systems for distributed computing**. Software – Practice and Experience, 32(2):135–164, Feb 2002.

[62] LANG, B; FOSTER, I; SIEBENLIST, F; ANANTHAKRISHNAN, R; FREEMAN, T. **A multipolicy authorization framework for grid security**. Accepted by the IEEE NCA06 Workshop on Adaptive Grid Computing (to appear in Proceedings 1th IEEE Symposium on Network Computing and Application), p. 24–26, Jul 2006.

[63] LITZKOW, M; LIVNY, M; MUTKA, M. **Condor – a hunter of idle workstations**. Proceedings of the 8th International Conference of Distributed Computing Systems, p. 104–111, Jun 1988.

[64] LITZKOW, M; TANNENBAUM, T; BASNEY, J; LIVNY, M. **Checkpoint and migration of UNIX processes in the Condor distributed processing system**. Computer Sciences Technical Report, Apr 1997.

[65] LIVNY, M; BASNEY, J; RAMAN, R; TANNENBAUM, T. **Mechanisms for high throughput computing**. SPEEDUP Journal, 11(1), Jun 1997.

[66] MARQUES, J. R; KON, F. **Gerenciamento de recursos distribuídos em sistemas de grande escala**. Proceedings of the 20th Brazilian Symposium on Computer Networks, p. 800–813, May 2002.

[67] MPI-SCTP. **Using the stream control transmission protocol (SCTP) for parallel T programs written using the message passing interface (MPI)**. `http://www3.niu.edu/mpi/`, Jan 2008.

[68] MPICH-G2. **MPICH-G2 for Globus Toolkit**. `http://www3.niu.edu/mpi/`, Jan 2008.

[69] MPICH-V. **MPICH-V: Mpi implementation for volatile resources**. http://mpich-v.lri.fr/, Feb 2008.

[70] MPIFORUM. **MPI: A message-passing interface standard**. International Journal of Supercomputer Applications, Apr 1995.

[71] MPIFORUM. **MPI-2: Extensions to the message-passing interface**. Technical Report, Apr 1997.

[72] MVAPICH. **MVAPICH: MPI over infiniband and iWARP**. `http://mvapich.cse.ohio-state.edu/`, Jan 2008.

[73] MYRICOM. **MPICH-GM software downloads and installation instructions**. `http://www.myri.com/scs/download-mpichgm.html`, Jan 2008.

[74] NAEDELE, M. **Standards for XML and web services security**. Computer, 36(4):96–98, Apr 2001.

[75] NATRAJAN, A; CROWLEY, M; WILKINS-DIEHR, N; HUMPHREY, M. A; FOX, A. D; GRIMSHAW, A. S; BROOKS, C. L. **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, p. 14–21, Aug 2001.

[76] OMG. **CORBA – common object request broker architecture**. `http://www.corba.org/`, Jan 2008.

[77] PINHEIRO, E. **Truly-transparent checkpointing of parallel applications**. Work Draft, 2002.

[78] PINHEIRO, J. R. B; VIDAL, A. C. T; KON, F. **Repositório seguro de aplicações baseado em GSS**. In Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais, SBC, Sep 2005.

[79] PLANK, J. S; BECK, M; KINGSLEY, G; LI, K. **Libckpt: Transparent checkpointing under unix**. In Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais, SBC, p. 213–224, Jan 1995.

[80] SANKARAN, S; SQUYRES, J. M; BARRETT, B; LUMSDAINE, A; DUELL, J; HARGROVE, P; ROMAN, E. **The LAM/MPI checkpoint/restart framework: System-initiated checkpointing**. International Journal of High Performance Computing Applications, 19(4):479–493, 2005.

[81] SCALI – HIGHER PERFORMANCE COMPUTING. **Scali MPI connect**. http://www.scali.com/content/view/35/, Jan 2008.

[82] SILVA, L. M; SILVA, J. G. **System-level versus user-defined checkpointing**. Reliable Distributed Systems, Proceedings in Seventeenth IEEE Symposium, p. 20–23, Oct 1998.

[83] SKILLICORN, D. B; HILL, J. M. D; MCCOLL, W. F. **Questions and answers about BSP**. Scientific Programming, IOS Press, 6(3):249–274, 1997.

[84] SKILLICORN, D. B; TALIA, D. **Models and languages for parallel computation**. ACM Computing Surveys, 30(2), Jun 1998.

[85] SNIR, M; OTTO, S; HUSS-LEDERMAN, S; WALKER, D; DONGARRA, J. **MPI – The Complete Reference: The MPI Core**, volume 1. The MIT Press, Massachusetts, 1998.

[86] STELLNER, G. **CoCheck: Checkpointing and process migration for MPI**. Proceedings of the 10th International Parallel Processing Symposium, p. 526–531, 1996.

[87] TANENBAUM, A. S; VAN STEEN, M. **Distributed Systems: Principles and Paradigms**. Prentice Hall, 2006.

[88] THAIN, D; TANNENBAUM, T; LIVNY, M. **Condor and the grid**. Grid Computing: Making The Global Infrastructure a Reality, John Wiley, 2003.

[89] THAKUR, R; GROPP, W; LUSK, E. **An abstract-device interface for implementing portable parallel-I/O interfaces**. Appeared in Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computing, IEEE, p. 180–187, 1996.

[90] THE OPEN MPI PROJECT. **Open mpi: Open source high performance computing**. http://www.open-mpi.org/, May 2008.

[91] VALIANT, L. G. **A bridging model for parallel computation**. Communications of the ACM, 33(8), Aug 1990.

[92] VENNERS, B. **Design for thread safety**. http://www.javaworld.com, Aug 1998.

[93] VERARI SYSTEM SOFTWARE. **Verari MPI/Pro**. http://www.cs.wisc.edu/condor/, Jan 2008.

[94] VINOSKI, S. **CORBA: integrating diverse applications within distributed heterogeneous environments**. IEEE Communications Magazine, 14(2), Aug 1997.

[95] WEIQIN, T; JINGBO, D; LIZHI, C. **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.

[96] WELCH, V; SIEBENLIST, F; FOSTER, I; BRESNAHAN, J; CZAJKOWSKI, K; GAWOR, J; KESSELMAN, C; MEDER, S; PEARLMAN, L; TUECKE, S. **Security for grid services**. Twelfth International Symposium on High Performance Distributed Computing (HPDC-12), IEEE Press, Jun 2003.

[97] WOO, N; CHOI, S; JUNG, H; MOON, J; YEOM, H. Y; PARK, T; PARK, H. **MPICH-GF: Providing fault tolerance on grid environments**. The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003), Tokyo, May 2003.

[98] WOO, N; JUNG, H; SHIN, D; HAN, H; YEOM, H. Y; PARK, T. **Performance Evaluation of Consistent Recovery Protocols Using MPICH-GF**, p. 167–178. EDCC, Lecture Notes in Computer Science, Springer, Nov 2005.

[99] WRIGHT, D. **Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor**. Proceedings of the Linux Clusters: The HPC Revolution conference, Champaign - Urbana, IL, Jun 2001.

[100] ZANDY, V. C. **CKPT - process checkpoint library**. `www.cs.wisc.edu/~zandy/ckpt`, May 2008.

[101] ZHANG, Y; WONG, D; ZHENG, W. **User-level checkpoint and recovery for LAM/MPI**. ACM SIGOPS Operating Systems Review, 39(3):72–81, Jul 2005.

[102] ZHONG, H; NIEH, J. **CRAK: Linux checkpoint/restart as a kernel module**. Technical Report, Department of Computer Science, Columbia University, 2001.

APÊNDICE A

Código da MPICH-IG – Trechos Representativos

Os trechos de código listados neste apêndice representam as principais interfaces implementadas para fornecer o serviço de execução de aplicações MPI no InteGrade através da MPICH-IG.

Código A.1: *Implementação da classe MpiRestartCoordinator que sincroniza os processos MPI no EM*

```
1  package grm.executionManager;
2
3  import grm.executionManager.ExecutionInformation;
4
5  import java.util.Iterator;
6  import java.util.Vector;
7  import java.util.concurrent.CyclicBarrier;
8  import java.util.concurrent.atomic.AtomicInteger;
9
10 import org.omg.CORBA.ORB;
11
12 import resourceProviders.Lrm;
13 import resourceProviders.LrmHelper;
14 import clusterManagement.Grm;
15 import dataTypes.ApplicationExecutionInformation;
16 import dataTypes.MpiConnectInformation;
17 import dataTypes.ExecutionRequestId;
18 import dataTypes.ProcessExecutionInformation;
19 import java.util.StringTokenizer;
20
21 /** A class is created for every restart
22 *
23 * IME/USP
24 *
25 * Changed by Marcelo de Castro
26 * INF/UFG
27 */
```

```
28 public class MpiRestartCoordinator extends RestartCoordinator {
29
30     private MpiConnectInformation[] resp;
31
32     public MpiRestartCoordinator(ORB orb, Grm grm, int nProcesses) {
33         super(orb, grm, nProcesses);
34         resp = new MpiConnectInformation[this.nProcesses];
35     }
36
37     public MpiConnectInformation[] registerMpiNode(ExecutionRequestId
           executionRequestId, String kvs, int rank)
38         {
39                 resp[rank] = new MpiConnectInformation(kvs,
                       executionRequestId.processId);
40
41                 try {
42                         barrier.await();
43                 } catch (Exception e) {
44                         e.printStackTrace();
45                 }
46
47                 // return kvs information for all process of the mpi
48                 return resp;
49
50     }
51 }
```

Código A.2: *Interface da Channel Inteface (mpidi_ch3_impl.h)*

```
1  typedef struct MPIDI_PG
2  {
3      ....
4
5      /* Next pointer used to maintain a list of all process groups known
           to
6         this process */
7      struct MPIDI_PG * next;
8
9      /* Number of processes in the process group */
10     int size;
11
12     /* Virtual Connection table. */
13     struct MPIDI_VC * vct;
14
15     /* Pointer to the process group ID. */
16     void * id;
17
```

```
18      /*  Channel-specific data connections */
19      MPIDI_CH3_PG_DECL
20
21      .....
22
23  }
24  MPIDI_PG_t;
25
26
27  typedef struct MPIDI_VC
28  {
29      .....
30
31      /* state of the VC */
32      MPIDI_VC_State_t state;
33
34      /* Process group to which this VC belongs */
35      struct MPIDI_PG * pg;
36
37      /* Rank of the process in that process group associated with this
            VC */
38      int pg_rank;
39
40      /* Local process ID */
41      int lpid;
42
43      /* Sequence number of the next packet we expect to receive */
44      MPID_Seqnum_t seqnum_recv;
45
46      /* Queue for holding packets received out of order. */
47      MPIDI_CH3_Pkt_send_container_t * msg_reorder_queue;
48
49      ....
50  }
51  MPIDI_VC_t;
52
53  ....
54
55
56  int MPIDI_CH3_Init(int has_parent, MPIDI_PG_t *pg_ptr, int pg_rank );
57
58  int MPIDI_CH3_Finalize(void);
59
60  int MPIDI_CH3_VC_Init( struct MPIDI_VC *vc );
61
62  int MPIDI_CH3_PG_Destroy( struct MPIDI_PG *pg );
```

```
63
64 int MPIDI_CH3_VC_Destroy( struct MPIDI_VC *vc );
65
66 int MPIDI_CH3_iSend(MPIDI_VC_t * vc, MPID_Request * sreq, void * hdr,
67                     MPIDI_msg_sz_t hdr_sz);
68
69 int MPIDI_CH3_iSendv(MPIDI_VC_t * vc, MPID_Request * sreq,
70                      MPID_IOV * iov, int n_iov);
71
72 int MPIDI_CH3_iStartMsg(MPIDI_VC_t * vc, void * hdr, MPIDI_msg_sz_t
       hdr_sz,
73                         MPID_Request ** sreq_ptr)
74
75 int MPIDI_CH3_iStartMsgv(MPIDI_VC_t * vc, MPID_IOV * iov, int n_iov,
76                          MPID_Request ** sreq_ptr);
```

Código A.3: *Interface da Protocolo de Checkpoint (mpidi_ig_cp.h)*

```
1 void FT_init(MPIDI_PG_t * pg_p, int nb_proc, int my_proc_rank);
2
3 int FT_can_send(unsigned int to);
4
5 int FT_msg_recv(MPIDI_CH3I_Connection_t * conn, MPID_Request * rreq);
```

APÊNDICE B

Testes de Desempenho – Resultados Obtidos

Neste apêndice são listados alguns dados sobre os testes de desempenho.

Tabela B.1: *Lista de máquinas utilizadas para os testes de desempenho*

Quantidade	CPU	Memória	Sistema Operacional
3	Intel Pentium 4 3.20GHz	1Gb	Linux 2.6.18-bs
1	Intel Pentium 4 3.20GHz	1Gb	Linux 2.6.21.5-smp
1	Intel Pentium 4 3.00GHz	1Gb	Linux 2.6.23.12-smp
1	Intel Core2 Quad Q6600 2.40GHz	2Gb	Linux 2.6.23.12-smp
1	Intel Core2 Duo 6400 2.13GHz	3Gb	Linux 2.6.23.12-smp

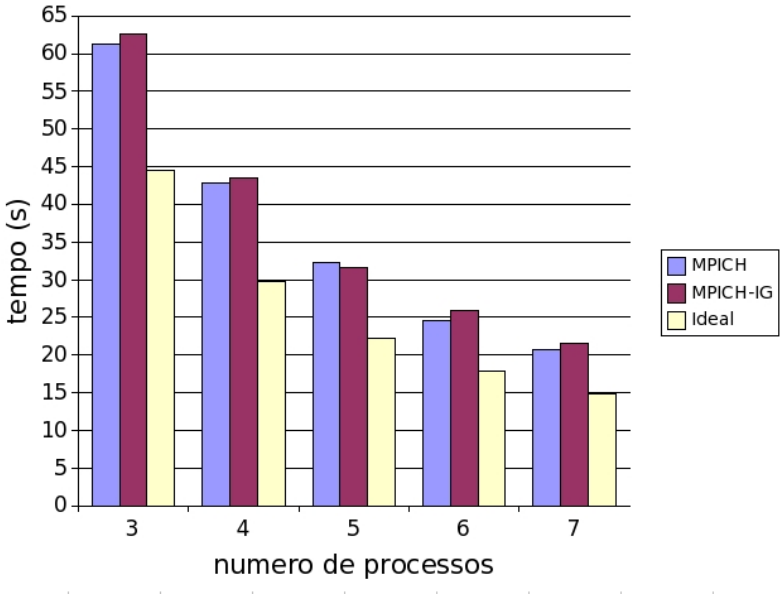


Figura B.1: *Comparação entre MPICH2 e MPICH–IG para uma matriz de entrada 2000×2000*

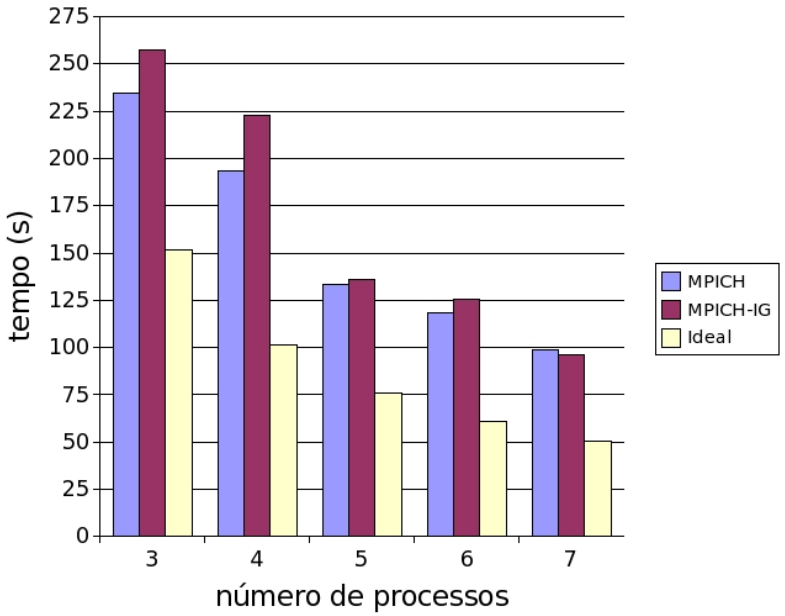


Figura B.2: *Comparação entre MPICH2 e MPICH–IG para uma matriz de entrada 3000×3000*

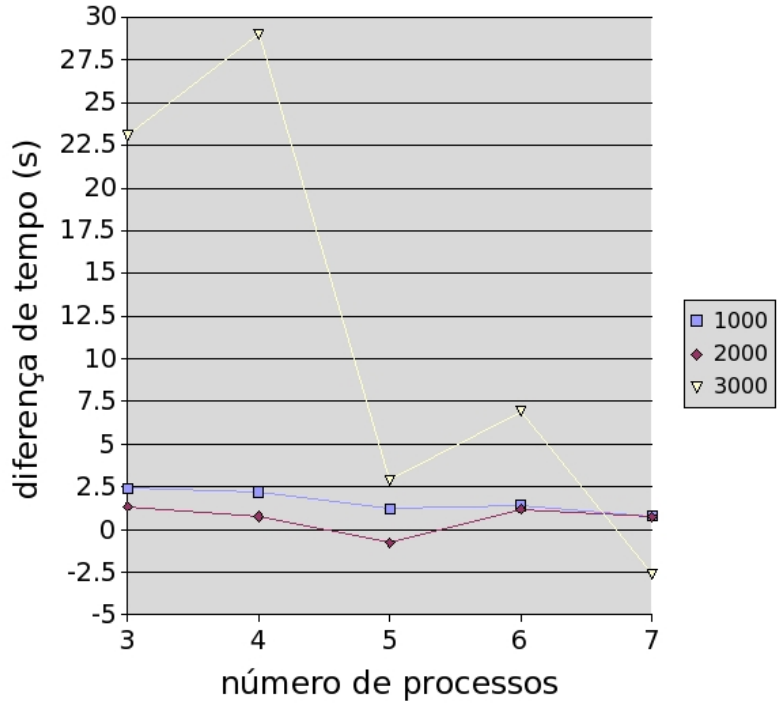


Figura B.3: *Diferença de desempenho entre MPICH2 e MPICH–IG para entradas 1000, 2000 e 3000*

Código B.1: *Implementação do programa de multiplicação de matrizes utilizado nos testes de desempenho*

```
1
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include"ftime.h"
5  #define limiteSuperior 4
6
7  #ifdef DIST
8
9          #include<mpi.h>
10         #define PARTITION_TAG 1
11         #define RETURN_RESULT 2
12         #define destino(a,b) a%(b-1)+1
13         int** distmultmat(int **mata, int **matb, int size);
14         int iniciarControle(int argc, char *argv[]);
15         int rank;
16         int numprocs;
17         MPI_Status status;
18
19 #else
20
21         int** multmat (int ** mata, int **matb, int size);
22
23 #endif
24
25 int* acoluna(int size);
26 int** alinha(int size);
27 int lcmult(int * linha, int ** mat, int coluna, int size);
28 void criaMatriz(int ***mata, int size);
29 void imprime(int **mat, int size);
30 void alocaMatriz(int ***mat, int size);
31
32 void imprime(int **mat, int size)
33 {
34         int i, j;
35
36         for (i = 0; i < size; i++)
37         {
38                 for (j = 0; j < size; j++)
39                         printf("%d ", mat[i][j]);
40                 printf("\n");
41         }
42 }
43
44 void criaMatriz(int ***mat, int size)
```

```
45 {
46         int i, j;
47
48         *mat = alinha(size);
49
50         for(i = 0; i < size; i++)
51         {
52                 (*mat)[i] = acoluna(size);
53                 for (j = 0; j < size; j++)
54                         (*mat)[i][j] = rand() % (limiteSuperior + 1);
55         }
56 }
57
58 void alocaMatriz(int ***mat, int size)
59 {
60         int i, j;
61
62         *mat = alinha(size);
63
64         for(i = 0; i < size; i++)
65                 (*mat)[i] = acoluna(size);
66 }
67
68 int lcmult(int * linha, int ** mat, int coluna, int size)
69 {
70         int i, soma = 0;
71
72         for (i = 0; i < size; i++)
73                 soma += linha[i]*mat[i][coluna];
74
75         return soma;
76 }
77
78 int** alinha(int size)
79 {
80         return (int**) malloc(sizeof(int*) * size);
81 }
82
83 int* acoluna(int size)
84 {
85         return (int*) malloc(sizeof(int) * size);
86 }
87
88 #ifdef DIST
89 /* DISTRIBUIDO
    ------------------------------------------------------------*/
```

```
90
91  int main(int argc, char * argv[])
92  {
93          int **mata = NULL, **matb = NULL, **matc = NULL, size = 0;
94      int start;
95
96      start = begintime();
97
98          size = atoi(argv[1]);
99
100         if (iniciarControle(argc, argv))
101         {
102                 srand ( time(NULL) );
103                 criaMatriz(&mata, size);
104                 criaMatriz(&matb, size);
105         }
106         else
107         {
108                 alocaMatriz(&matb, size);
109         }
110
111         matc = distmultmat(mata, matb, size);
112 #ifdef IMP
113         if (rank == 0)
114         imprime(matc, size);
115 #endif
116
117         //MPI_Barrier(MPI_COMM_WORLD);
118         MPI_Finalize();
119
120     printf("%d - %d : %s\n", numprocs, size, formattime(NULL, endtime_m
        (start)));
121
122         return 0;
123 }
124
125 int iniciarControle(int argc, char *argv[])
126 {
127         MPI_Init(&argc,&argv);
128     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
129     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
130
131     return (rank == 0);
132 }
133
134 int** distmultmat(int **mata, int **matb, int size)
```

```
135 {
136         int i, j, **matc = NULL;
137         MPI_Status status;
138         MPI_Request req;
139 
140         if (rank == 0)
141         {
142                 for (i = 0; i < size; i++)
143                         MPI_Bcast(matb[i], size, MPI_INT, rank,
                                MPI_COMM_WORLD);
144 
145                 for (i = 0; i < size; i++)
146                         MPI_Isend(mata[i], size, MPI_INT, destino(i,
                                numprocs), PARTITION_TAG, MPI_COMM_WORLD, &
                                req);
147 
148                 alocaMatriz(&matc, size);
149 
150                 for (i = 0; i < size; i++)
151                         MPI_Recv(matc[i], size, MPI_INT, destino(i,
                                numprocs), RETURN_RESULT, MPI_COMM_WORLD, &
                                status);
152         }
153         else
154         {
155                 int *linha = acoluna(size), *coluna = acoluna(size),
                        tam;
156 
157                 for (i = 0; i < size; i++)
158                         MPI_Bcast(matb[i], size, MPI_INT, 0,
                                MPI_COMM_WORLD);
159 
160                 tam = size / (numprocs-1);
161 
162                 if (rank <= size % (numprocs-1))
163                         tam++;
164 
165                 for (i = 0; i < tam; i++)
166                 {
167                         MPI_Recv(linha, size, MPI_INT, 0, PARTITION_TAG
                                , MPI_COMM_WORLD, &status);
168                         for (j = 0; j < size; j++)
169                                 coluna[j] = lcmult(linha, matb, j, size
                                        );
170                         MPI_Isend(coluna, size, MPI_INT, 0,
                                RETURN_RESULT, MPI_COMM_WORLD, &req);
```

```
171                     }
172
173             }
174
175             return matc;
176     }
177
178     #else
179
180     /* LOCAL _____________________________________________________________*/
181
182     int main(int argc, char * argv[])
183     {
184             int **mata = NULL, **matb = NULL, **matc = NULL, size = 0;
185             int start;
186
187         start = begintime();
188
189             size = atoi(argv[1]);
190             srand ( time(NULL) );
191
192             criaMatriz(&mata, size);
193             criaMatriz(&matb, size);
194
195             matc = multmat(mata, matb, size);
196     #ifdef IMP
197         imprime(matc, size);
198     #endif
199
200         printf("1 - %d : %s\n", size, formattime(NULL, endtime_m(start)));
201
202             return 0;
203     }
204
205     int** multmat (int ** mata, int **matb, int size)
206     {
207             int i, j;
208             int ** matc = alinha(size);
209
210             for (i = 0; i < size; i++)
211             {
212                     matc[i] = acoluna(size);
213                     for(j = 0; j < size; j++)
214                             matc[i][j] = lcmult(mata[i], matb, j, size);
215             }
216
```

```
217         return matc;
218 }
219
220 #endif
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo