**UNIVERSIDADE FEDERAL DE SANTA MARIA**
**CENTRO DE TECNOLOGIA**
**CURSO DE CIÊNCIA DA COMPUTAÇÃO**

# ESTUDOS COMPARATIVOS DE FERRAMENTAS DE PROGRAMAÇÃO PARA GRADES COMPUTACIONAIS

**TRABALHO DE GRADUAÇÃO**

**Tiago Scheid**

**Santa Maria, RS, Brasil**

**2007**

# ESTUDOS COMPARATIVOS DE FERRAMENTAS DE PROGRAMAÇÃO PARA GRADES COMPUTACIONAIS

**por**

**Tiago Scheid**

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de
**Bacharel em Ciência da Computação**

**Orientador: Prof. Dr. Benhur de Oliveira Stein**

**Trabalho de Graduação Nº 235**
**Santa Maria, RS, Brasil**

**2007**

**Universidade Federal de Santa Maria**
**Centro de Tecnologia**
**Curso de Ciência da Computação**

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

# ESTUDOS COMPARATIVOS DE FERRAMENTAS DE PROGRAMAÇÃO PARA GRADES COMPUTACIONAIS

elaborado por
**Tiago Scheid**

como requisito parcial para obtenção do grau de
**Bacharel em Ciência da Computação**

**COMISSÃO EXAMINADORA:**

**Prof. Dr. Benhur de Oliveira Stein**
(Presidente/Orientador)

**Profª Drª Iara Augustin (UFSM)**

**Profª Dr. Marcelo Pasin (UFSM)**

Santa Maria, 01 de Março de 2007.

# AGRADECIMENTOS

Primeiramente gostaria de agradecer Deus por me acompanhado nessa jornada de 4 anos que chega ao fim com este trabalho. Gostaria de agradecer aos meus pais por terem me apoiado todos esses anos longe de casa, e graças a eles eu consegui construir tudo o que eu tenho. Agradeço aos meus pais por formarem meu caráter e por serem os maiores exemplos da minha vida (quando eu crescer eu quero ser como eles). Gostaria de agradecer a minha irmã por ter me agüentado esses 5 anos aqui em Santa Maria, e por ter me ajudado sempre que eu precisei.

Gostaria de agradecer aos professores por terem me transmitido tamanho conhecimento. Gostaria de agradecer principalmente os professores Andrea e Benhur. À Andrea por ter me agüentado como orientando no LSC por 2 anos e ao Benhur por ter me agüentado como orientando durante o meu TG e por ter me aceito como seu orientando no mestrado. Ao professor Marcelo por ter me proporcionado uma experiência única na cooperação com o pessoal de Friburgo, o qual o meu TG faz parte. Agradeço à esses professores e aos professores Márcia, Iara e Caio por serem exemplos de professores.

Agradeço ao pessoal do LSC, o Schepke, Furlan, Veiga, Elton, Araújo, Reck, Geovani, Kreutz, Márcia, Lucas, Righi, Koslovski, Rodolfo, Reis, Boufleur, Matheus, por serem além de colegas de trabalho, amigos nas festas. Gostaria de agradecer especialmente ao Marcelo Veiga Neves por ser um grande amigo, tanto no trabalho do laboratório, o qual fazíamos sempre em conjunto, como nas festas do Aldeia e DCE. Ao Elton Nicoletti Mathias pelo trabalho antecessor a este TG e também por ser companheiro de festa. Ao Edmar Pessoa Araújo Neto pelas jogatinas de RPG e pelas festas, ainda mais agora que somos colegas no mestrado. Agradeço ao Redin, Rubens e o Pepe por serem amigos de festa, e por sempre me convidarem às concentrações da "nossa"turma.

Agradeço aos meus colegas de turma, especialmente Rafael, Fernando, Bruno, Scholz e Linck. Ao Rafael por me agüentar nos trabalhos em grupo, ao Fernando e ao Bruno pelos jogos de RPG, ao Scholz pelos jogos de RPG e por me agüentar nas festas e ao Linck por me agüentar nos trabalhos em grupo e por me agüentar nas festas.

:x!

*“Não tá morto quem luta e quem peleia!”* — LEOPOLDO RASSIER
*“ I believe in miracles they happen every day.”* — EDGUY
*“ Botemo! ”* — CÉLIO TROIS

# RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

**ESTUDOS COMPARATIVOS DE FERRAMENTAS DE PROGRAMAÇÃO PARA GRADES COMPUTACIONAIS**
Autor: Tiago Scheid
Orientador: Prof. Dr. Benhur de Oliveira Stein
Local e data da defesa: Santa Maria, 01 de Março de 2007.

O processamento paralelo e distribuído é freqüentemente utilizado para resolver problemas que demandam um grande poder computacional. Existem diversas plataformas que permitem executar programas paralelos e distribuídos. Uma dessas plataformas são as grades computacionais, as quais são compostas por vários computadores distribuídos geograficamente. Para o desenvolvimento de aplicações voltadas a grades computacionais, existem várias ferramentas que propõem abstrair do programador a complexidade da grade. Nesse contexto, o presente trabalho se propõe a realizar um estudo comparativo de algumas das ferramentas de programação para grades computacionais. Esta comparação visa fornecer dados que auxiliem programadores na escolha entre essas ferramentas, reduzindo a necessidade de fazer um teste prático. Para fazer essa comparação foram realizados testes práticos com implementações que usavam as ferramentas. Foi construída uma aplicação que resolve o problema das N-Rainhas utilizando ProActive e POP-C++. Dessa forma pode-se observar que as duas ferramentas possuem um bom desempenho, mas ambas carecem de melhorias. Também pode-se concluir que apesar de ProActive ser feito em Java, não apresentou uma perda de desempenho significativa, portanto ferramentas em Java podem ser uma boa escolha para se programar em grades computacionais.

**Palavras-chave:** Grades computacionais, Programação de grades, POP-C++, ProActive, N-Rainhas.

# ABSTRACT

Graduation Work
Graduate Program in Computer Science
Federal University of Santa Maria

**COMPARATIVE STUDIES FOR COMPUTATIONAL GRID PROGRAMMING TOOLS**
Author: Tiago Scheid
Advisor: Prof. Dr. Benhur de Oliveira Stein

Parallel and distributed computing is widely used to solve problems that demand high computational power. There are many platforms that allow executing parallel and distributed programs. One of these platforms is the computational grids, composed by many computers dispersed in many sites. There are many tools to help programmers to develop computational grid applications. The present work intended to compare studies of some of the computational grid programming tools. This comparison aim to help programmers to choose between these tools, decreasing the necessity for practical tests. To execute this comparison, practical tests with an implementation using these tools were run. An application that solve N-Queens problem was created using ProActive and POP-C++. Therefore, both tools have shown good performance, but both need to improve their features. ProActive is a Java tool, but this fact does not interfere on the application performance, so suggest that Java could be a choice for computational grid programming.

**Keywords:** computational grids, grids programming, POP-C++, ProActive, N-Queens.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| JVM | Java Virtual Machine |
| API | Aplication Programing Interface |
| POP-C++ | Parallel Object Programing C++ |
| INRIA | Institut National de Recherche en Informatique et en Automatique |

# SUMÁRIO

# 1 INTRODUÇÃO

O processamento paralelo é freqüentemente utilizado para resolver problemas que demandam um grande poder computacional. A paralelização é uma estratégia utilizada em computação para realizar operações mais rapidamente e/ou processar maiores volumes de dados.

Existem diversas plataformas que permitem a execução de programas paralelas e distribuídas. Uma dessas plataformas são as grades computacionais, que estão se tornando cada vez mais populares no meio acadêmico. Uma grade computacional pode ser composta por um grande número de computadores geograficamente distribuídos, podendo prover um alto poder computacional. Além disso, o uso de grades aumenta a colaboração entre diferentes instituições, através da agregação de recursos, e pode permitir que pesquisadores acessem um poder computacional maior do que o existente em suas instituições de origem (CIRNE et al., 2006).

A construção de grades computacionais, bem como o desenvolvimento de aplicações para este tipo de arquitetura, não é uma tarefa simples. Por serem compostas por recursos distribuídos em diferentes domínios administrativos, pode-se ter diferentes políticas de segurança e acesso aos recursos. Além disso, cada domínio administrativo pode ter sua própria infra-estrutura de *software*. Desta forma, faz-se necessário a utilização de ferramentas capazes de transpor esses (e outros) problemas de forma transparente ao usuário.

Atualmente, existem diversas ferramentas que se propõem a abstrair do programador a complexidade do grade como Boinc (ANDERSON, 2004), OurGrid (ANDRADE et al., 2003), ProActive (CAROMEL; KLAUSER; VAYSSIÈRE, 1998), Ibis (NIEUWPOORT et al., 2002), POP-C++ (NGUYEN; KUONEN, 2003), Globus Toolkit (I. Foster; C. Kesselman, 1997), entre outras. Um dos exemplos mais difundidos é o Globus Toolkit, que pode ser visto como uma solução completa para a construção e utilização de grades. Estas

ferramentas, de maneira geral, são compostas por um ambiente de execução (*middleware*) e uma API (*Aplication Programing Interface*) para o desenvolvimento de aplicações.

No entanto, ferramentas como OurGrid e Globus Toolkit possuem uma preocupação maior prover um *middleware* capaz de gerenciar os recursos, tarefas e usuários, deixando a parte de comunicação e portabilidade sob responsabilidade do programador. Dessa forma, para ter essas ferramentas na grade é necessária a cooperação dos domínios administrativos que fazem parte da grade, pois é necessário que todos tenham o *middleware*.

Já Ibis tem como maior preocupação abstrair do programador a complexidade de comunicação na grade e comunicação em grupo das mais diferentes maneiras. Porém Ibis não implementa o lançamento do programa em nós, o que pode ser muito complexo para o programador implementar em um ambiente como grade computacional.

Boinc e outras ferramentas semelhantes se preocupam no gerenciamento de tarefas que não tenham dependência entre si e que não tenham alta prioridade, pois ele utiliza o tempo ocioso de estações de trabalho. Em Boinc o lançamento do programa é realizado pelo usuário do nós da grade (estações de trabalho), o qual lança um cliente que se encarregará em calcular o problema quando o nó estiver ocioso.

Neste trabalho buscou-se por ferramentas que possuem um ambiente de execução mais simples de ser instalado e configurado, podendo ser instalado pelo usuário da grade sem necessitar o apoio dos administradores dos domínios administrativos para instalar em toda grade. Nesse caso foram escolhidas ProActive e POP-C++, pois são organizadas como biblioteca e possuem um ambiente de execução fácil de instalar e configurar.

A escolhas dessas ferramentas também está ligada a cooperação com a Universidade de Ciências Aplicadas de Friburgo Suíça no *Grid Crunching Day* (KUONEM, 2006). O *Grid Crunching Day* é um evento com o objetivo de mostrar o que está sendo desenvolvido para grades computacionais, motivar usuários a utilizarem grades computacionais e demonstrar aplicações sendo executada na grade.

Uma das motivações do trabalho é o fato de haver uma grande quantidade de ferramentas para programação para grades computacionais, o que torna difícil a escolha de uma delas por parte do programador. Alguns fatores que dificultam a escolha é o fato da documentação sobre as ferramentas pode ser tendenciosa, omitindo possíveis limitações das ferramentas. Atualmente, existem alguns trabalhos que se propõem a comparar algumas ferramentas (AL-JAROODI et al., 2002), porém esses estudos acabam sendo su-

perficiais, e não provém uma boa base para que o programador possa fazer a escolha da ferramenta que melhor se adapte a seu uso. Isso evidencia a carência por uma análise comparativa e imparcial das ferramentas.

Nesse contexto, o presente trabalho se propõem a realizar um estudo comparativo de algumas das ferramentas de programação para grades computacionais. Para isso será implementada uma aplicação para que possa fazer uma análise prática e, finalmente, traçar um comparativo entre elas.

Esta comparação visa fornecer dados complementares à documentação das ferramentas, para auxiliar programadores na escolha da ferramenta mais apta para resolver um dado problema, reduzindo, dessa forma, a necessidade de fazer um teste prático por parte do programador.

O texto deste trabalho está dividido em capítulos. Primeiramente, há uma explanação sobre grades computacionais, as ferramentas e sobre a aplicação utilizada. Em seguida há um capítulo sobre o projeto da aplicação e um capítulo sobre detalhes de implementação. Por fim há uma avaliação do trabalho e conclusões.

# 2 GRADES COMPUTACIONAIS: CONCEITOS E FERRAMENTAS

## 2.1 Grades computacionais

Atualmente, computadores têm sido usados para modelar e simular problemas complexos de ciência e engenharia, controle industrial de equipamentos, previsão do tempo, e vários outros problemas. Esse tipo de problema demanda um alto poder computacional, o qual é possível obter através da utilização de uma grade computacional formada pela agregação de vários *clusters* ou pela utilização de períodos ociosos de estações de trabalho.

As grades computacionais, segundo Foster et al. (FOSTER; KESSELMAN, 1999), são análogas ao uso da produção e transmissão de energia elétrica. Por volta de 1910, quem quisesse possuir os benefícios da energia elétrica, deveria construir e manter o seu próprio gerador. Com o tempo, as tecnologias de transmissão e distribuição permitiram criar uma grade de energia elétrica, a qual foi capaz de prover energia elétrica a baixo custo, e dessa forma, transformá-la em um recurso acessível a todos.

Analogamente, usa-se o termo **grade computacional** para a infra-estrutura que permitirá o crescimento da computação. Ainda não existe um consenso quanto a definição de computação em grande. Segundo Foster et al. (FOSTER; KESSELMAN, 1999), uma grade computacional é a infra-estrutura de *hardware* e *software* que provê dependabilidade, pervasividade, consistência e acesso barato a poder computacional de última geração. Já Buyya (BUYYA, 2003) define uma grade computacional como um tipo de sistema paralelo e distribuído que provê compartilhamento, seleção e agregação de recursos autônomos geograficamente distribuídos, levando em conta sua disponibilidade, capacidade, desempenho, custos e necessidades de qualidade de serviço dos usuários.

Hoje as grades computacionais consistem basicamente na aglomeração de vários *clus-*

*ters*, ou seja, algumas instituições agregam seus recursos de forma colaborativa, para obter maior poder computacional para executarem suas aplicações. Entre as aplicações típicas para grades estão aquelas que demandam um grande poder de processamento (*Distributed supercomputing*), processam um grande volume de dados (*Data intensive*), que possuem um grande número de tarefas independentes (*High throughput*), entre outras (FOSTER; KESSELMAN, 1999). Este trabalho tem como foco aplicações que demandam um grande poder de processamento.

As grades computacionais atuais possuem uma série de problemas que dificultam a criação de programas para elas. Um desses problemas é heterogeneidade de *hardware* e *software*. Esse problema ocorre pois é difícil, tendo em vista que uma grade pode ser formada por um número bastante grande de recursos, agregá-los de diversas instituições com as mesmas características de *software* e *hardware*.

Outro problema em agregar recursos de diversas instituições é o fato de os *clusters* continuarem sob a administração de cada instituição. Como cada instituição tem a sua própria política de segurança e acesso, há uma série de problemas para o funcionamento da grade computacional como, por exemplo, o gerenciamento de contas de acesso para usuários, *firewall*, acessibilidade de computadores individuais, entre outros.

Todos esses problemas devem ser levados em conta para a criação de uma aplicação ou ferramenta que será executada em uma grade computacional.

## 2.2 Ferramentas de programação para grades computacionais

As ferramentas de programação para grades computacionais têm como objetivo abstrair do programador a complexidade de uma grade computacional. As ferramentas devem ser capazes de prover desempenho, tolerância a falhas, segurança e independência de plataforma (KIELMANN et al., 2006).

O alto desempenho é um dos grande motivos de se usar grades computacionais. Dessa forma espera-se que uma ferramenta de programação para grades computacionais tenha um bom desempenho em todos os aspectos.

Muitas das operações da API de uma grade computacional envolve comunicação com nós, serviços e recursos remotos. Por causa dessas comunicações ocorrerem numa rede instável (Internet) e a autonomia administrativa da instituição pertencente a grade computacional, é comum que ocorram erros. Esses erros devem ser repassados para o progra-

mador em forma de exceção.

Como os dados são transmitidos através de uma rede pública (Internet), a ferramentas devem ter suporte a segurança dos dados transmitidos, caso seja de interesse do usuário que seus dados sejam protegidos.

É importante que uma ferramenta de programação abstraia do programador detalhes da grade computacional, como nome de máquinas ou estrutura do sistema de arquivos. Isso deve ser provido pelo ambiente de execução da ferramenta.

Entre as ferramentas citadas, este trabalho realiza a comparação de ProActive e POP-C++, que são descritas nas seções a seguir. Essas ferramentas foram escolhidas pois provém um ambiente de execução fácil de instalar e configurar e possuem uma API para programação e comunicação. Essa escolha também foi influênciada pela cooperação no *Grid Crunching Day*.

### 2.2.1 ProActive

*ProActive* (CAROMEL; KLAUSER; VAYSSIÈRE, 1998) foi desenvolvido pelo OASIS Team, INRIA Sophia-Antipolis, na França. *ProActive* é uma biblioteca implementada utilizando a linguagem de programação Java, que busca oferecer um modelo de programação concorrente e distribuído com transparência. Como essa biblioteca é construída inteiramente a partir da API padrão Java, ela não requer qualquer modificação no ambiente de execução, uso de compiladores especiais, pré-processadores ou máquina virtual modificada.

Esta biblioteca é implementada através de um modelo de programação chamado de objetos ativos. Nesse modelo, cada objeto ativo tem sua própria *thread* de controle, que tem habilidade de gerenciar a execução de chamadas de métodos remotos, armazenado em um sistema de fila.

Objetos Ativos são remotamente acessíveis através da invocação de métodos. A chamada desses métodos ocorre de forma assíncrona através de um mecanismo de sincronização automático oferecido pela biblioteca. Esse mecanismo baseia-se em objetos futuros, que são retornados imediatamente após a chamada de métodos remotos e substituídos automaticamente quando o método retorna realmente. A utilização do retorno antes de sua disponibilidade bloqueia o fluxo que chamou método, em um mecanismo de espera por necessidade.

Além de comunicação ponto-a-ponto, outro mecanismo de comunicação oferecido é baseado em um modelo de comunicação em grupo (BADUEL; BAUDE; CAROMEL, 2002). A comunicação em grupo permite a invocação de métodos em um grupo de objetos ativos de tipo compatível, e a geração de resultados de grupo.

O ambiente de execução de *ProActive* é configurado através de um arquivo de descrição de máquinas. Esse arquivo possui a lista de máquinas, o protocolo de lançamento do programa para cada máquina (ssh, rsh, etc), o caminho para a biblioteca *ProActive*, entre outras configurações. Esse arquivo é no formato XML e é usado pelo programa para obter os nós que fazem parte da grade computacional.

### 2.2.2 POP-C++

*POP-C++ (Parallel Object Programing C++)* (NGUYEN; KUONEN, 2003) é uma extensão da linguagem C++ que permite a criação de objetos C++ de forma distribuída (objetos paralelos). Para tanto faz uso de um *wrapper* para compilação de seus programas e de um ambiente de execução.

*POP-C++* é capaz de instanciar objetos paralelos de forma transparente ao programador. Cada objeto paralelo pode ser instanciado em uma máquina diferente, da mesmo forma que se instância um objeto C++ padrão, o programador pode também realizar instanciamento de objetos em máquinas específicas, usando diretivas que podem ser adicionadas ao construtor do objeto.

A chamada de métodos a objetos paralelos pode seguir duas semânticas do ponto de vista de quem invoca o método, invocação síncrona ou assíncrona, e três semânticas do ponto de vista do objeto, invocação seqüencial, concorrente e *mutex*. Essas semânticas são descritas abaixo (ver Figura 2.1):

- Invocação síncrona: O objeto invocador do método espera até que o método invocado retorne para continuar sua execução.

- Invocação assíncrona: O objeto invocador do método continua sua execução imediatamente após a invocação do método, sem esperar pelo retorno. A versão atual do POP-C++ não possui retorno para métodos assíncronos, dessa forma o retorno para esse tipo de método se dá através de uma chamada de retorno (*callback*) ao objeto invocador pelo objeto invocado.

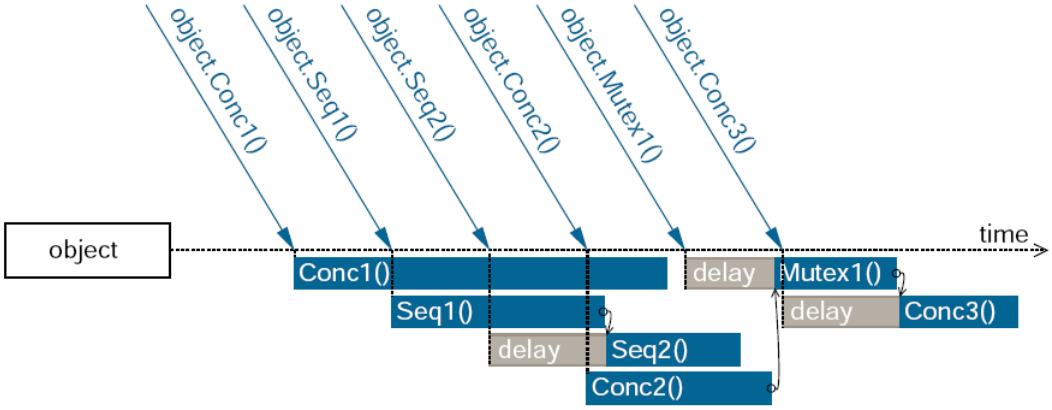


Figura 2.1: Diretivas de invocação de um objeto POP-C++. (NGUYEN; PASIN; KUONEN, 2006)

- Invocação seqüencial: Invocação a métodos seqüenciais são executados em exclusão mútua, seguindo a ordem de chegada. Se várias invocações forem feitas sobre o mesmo objeto, cada chamada será atendida de forma seqüencial. No entanto, métodos concorrentes podem ser executados normalmente.

- Invocação *mutex*: Invocação para métodos *mutex* ocorrem em total exclusão com qualquer outro método do mesmo objeto. A requisição é executada somente se não tiver nenhum outro método sendo executado.

- Invocação concorrente: Na invocação concorrente, todos os métodos são executados de forma concorrente, a menos que haja uma invocação *mutex* em execução ou aguardando. Todas as invocações compartilham os atributos do objeto.

Para compilar um programa POP-C++ utiliza-se um pré-compilador que gera um código C++ para as classes POP-C++ e na seqüência invoca o compilador C++ padrão. É possível escolher o compilador C++ que irá gerar programa executável.

O ambiente de execução é composto por um servidor de gerenciamento de recursos. Esse servidor aloca e gerencia os recursos utilizados pela aplicação. Ele também escolhe o recurso que melhor se adapte as exigências do usuário.

## 2.3 Problema utilizado para análise

Para analisar as ferramentas foi utilizado como *benchmark* uma implementação do problema das N-Rainhas. Esse problema é muito utilizado como *benchmark* para sistemas

paralelos pois é fácil aumentar a sua escala e gera um imenso número de operações. Esse problema cresce de forma exponencial com o valor de $N$, exigindo um grande tempo de processamento para valores grande de $N$ (maior ou igual a 19).

O problema das N-Rainhas consiste em encontrar as várias possibilidades de posicionar $N$ rainhas em um tabuleiro $N \times N$, de maneira que não possam ser capturadas pelas demais rainhas, segundo as regras do xadrez. Uma forma de resolver esse problema é posicionar as rainhas uma a uma, sendo que as rainhas já posicionadas impedem que as rainhas sejam posicionadas na mesma coluna e na mesma diagonal (esquerda e direita).

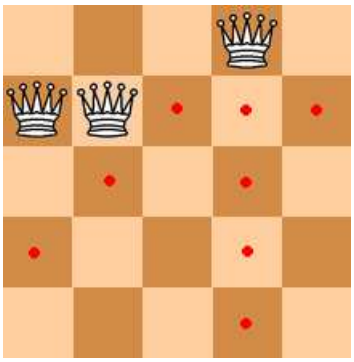

Figura 2.2: Tabuleiro com uma rainha posicionada.

Podemos ver na figura 2.2 um tabuleiro $5 \times 5$ com a primeira rainha posicionada na quarta casa, como a rainha pode capturar outras peças na linha horizontal, vertical e nas diagonais, as posições possíveis para a segunda rainha são na primeira e segunda casa.

O algoritmo de resolução do problema das N-Rainhas (ver figura 2.3) mostra como é realizado o posicionamento das rainhas nas linhas do tabuleiro, de forma recursiva. Caso tenha conseguido chegar na última linha, encontrou-se uma solução.

```
posiciona (linha)
    se linha > N
            incrementa número soluções
    senão
          para cada posição válida
                coloca a rainha nessa posição
                posiciona (linha + 1)
```

Figura 2.3: Algoritmo para solução do problema das N-Rainhas.

Esse algoritmo gera uma árvore de resolução como podemos ver na figura 2.4, na qual a primeira rainha está na primeira posição, então para cada posição válida da segunda

linha, é posicionada uma rainha, e assim por diante. Há sub-árvores que serão abortadas por não haver alguma posição válida. Todas as folhas de altura $N$ contêm soluções, ou seja, ao posicionar uma rainha na última linha, tem-se uma solução.

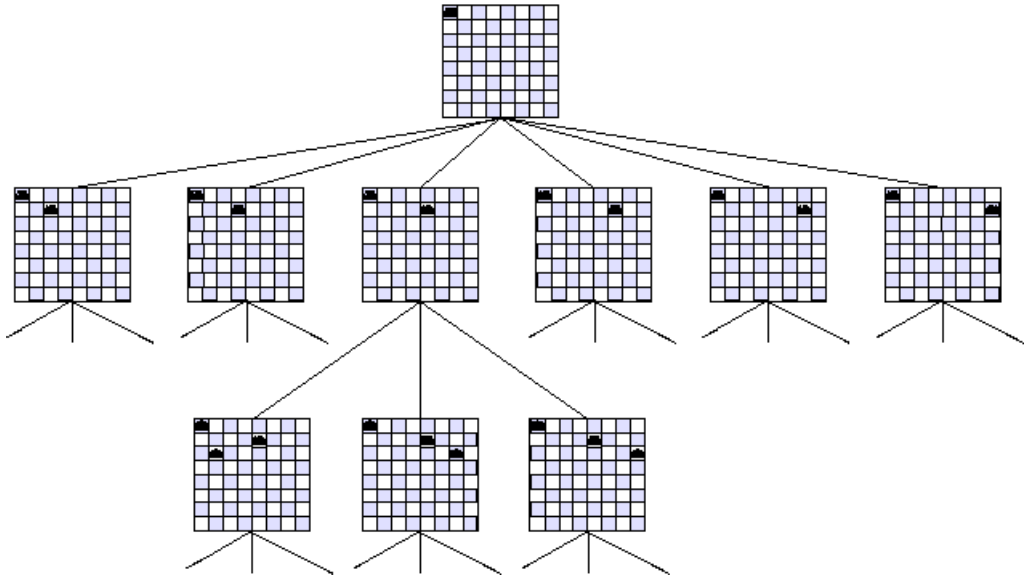

Figura 2.4: Árvore de resolução do problema das N-Rainhas.

Dos vários algoritmos existentes para solucionar o problema das N-Rainhas, o algoritmo seqüencial mais rápido, de que se tem conhecimento, é o algoritmo de Takaken (TAKAHASHI, 2003). Esse algoritmo utiliza mapas de bits para representar as rainhas no tabuleiro e para descobrir as posições válidas, fazendo com que cada posicionamento seja uma operação extremamente rápida. Também há uma otimização na identificação de simetrias do tabuleiro, dessa forma, só ocorre o posicionamento de metade das rainhas da primeira linha, há uma eliminação de posições a qual se sabe que há simetria e há uma verificação de simetria de cada solução (exceto para quando a primeira rainha estiver no canto, pois sabe-se que essa solução tem multiplicidade 8).

# 3 PROJETO

Para poder comparar as ferramentas de forma significativa, foi implementada uma aplicação capaz de resolver o problema das N-Rainhas de forma paralela. Este capítulo visa descrever o projeto dessa aplicação.

## 3.1 Arquitetura da aplicação

Como as grades computacionais podem ser compostas por uma grande quantidade de nós (pode alcançar milhares de nós), optou-se por desenvolver uma arquitetura hierárquica, diminuindo os gargalos gerados por uma abordagem centralizada (mestre-escravo).

Pode-se considerar que os nós de uma grade computacional se organizam de maneira hierárquica (ver figura 3.1), onde no topo da hierarquia está a máquina do usuário da grade, que, normalmente, não possui acesso direto a todos os nós da grade, mas sim acesso indireto através de um *frontend* do *cluster*, que por sua vez tem acesso aos nós de seu *cluster*.

Baseado nessa arquitetura hierárquica, a aplicação foi mapeada de forma a obter vantagem dessa arquitetura. Na base da hierarquia encontram-se os nós dos *clusters*, os quais hospedam a classe *Worker*. No meio da hierarquia encontram-se os *frontends* dos *clusters*, os quais hospedam a classe *Boss*. E no topo da hierarquia, encontra-se a máquina do usuário da aplicação, responsável por hospedar a classe *BigBoss*. No projeto dessa arquitetura foram previstas 3 classes principais (ver figura 3.2):

- *BigBoss*: Responsável pela geração de tarefas, recebidas da Interface (ver seção 3.2.1), pelo gerenciamento do nível inferior da hierarquia e por reportar resultados à Interface. Esta classe constitui o topo da hierarquia.

- *Boss*: Responsável por gerenciar o nível inferior e por reportar resultados ao nível superior. Esta classe constitui a região intermediária da hierarquia.

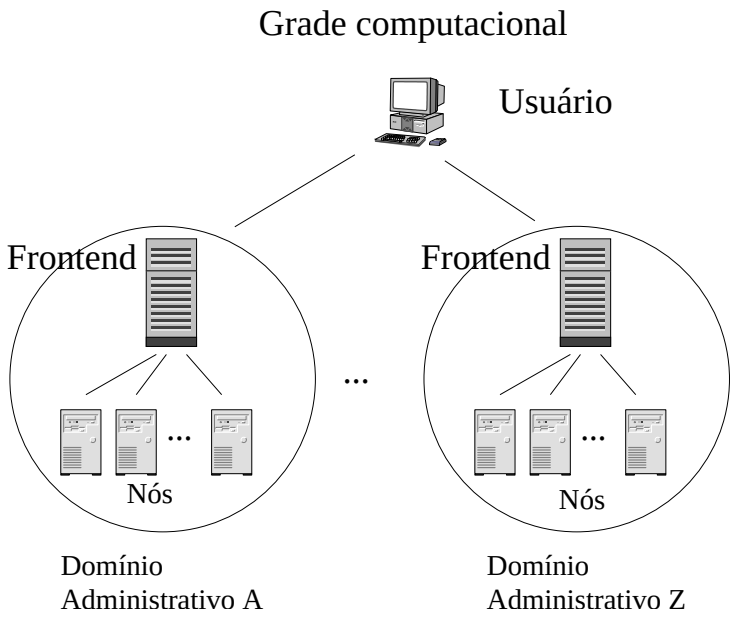


Figura 3.1: Hierarquia de uma grade computacional.

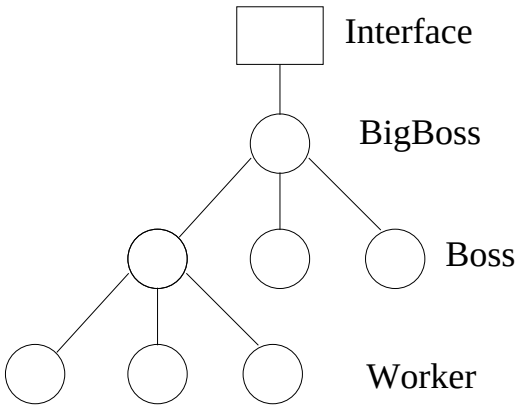


Figura 3.2: Hierarquia das classes da aplicação.

- *Worker*: Responsável pela resolução das tarefas e por reportar os seus resultados ao nível superior. Essa classe constitui a base da hierarquia.

O cálculo das tarefas ocorre de forma paralela na classe *Worker*, mas para isso deve-se paralelizar o algoritmo de resolução (ver figura 2.3). A paralelização desse algoritmo pode ser feita através da quebra do trabalho, baseando-se na árvore de resolução (ver figura 2.4). Essa quebra consiste em interromper a recursão de resolução em um determinado nível, e enviar essa subárvore para os trabalhadores calcularem. Como foi visto na seção 2.3, os nós da árvore de resolução não dependem de seus irmãos, dessa forma as tarefas geradas não possuem dependências entre si.

Na figura 2.4 podemos ver a árvore de resolução para um tabuleiro $8 \times 8$, supondo que a recursão fosse interrompida no segundo nível, todas as tarefas da segunda linha seriam tarefas a serem calculadas pelos trabalhadores. Dessa forma, as suas sub-árvores (como por exemplo, a sub-árvore da terceira tarefa) seriam resolvidas em paralelo nos nós da grade.

A seção a seguir descreve o funcionamento de cada classe.

## 3.2 Projeto das Classes

Esta seção tem como objetivo mostrar o funcionamento de cada classe bem como o relacionamento entre elas. De forma geral, do ponto de vista externo às classes, há uma série de interações entre as classes conforme demonstrado na figura 3.3. Nessa figura pode-se ver que a classe *Interface* envia para a classe *BigBoss* a especificação de tabuleiros a serem calculados. A classe *BigBoss* gera as tarefas recebidas via a *Inteface*, e as distribui entre seus subordinados. Cada *Boss* recebe a sua tarefa e a quebra e envia para seus subordinados. Cada *Worker* recebe uma tarefa e calcula a resposta para tarefa, reportando-a para seu superior, que por sua vez aglutina todas respostas até que tenha terminado sua tarefa e envia sua resposta ao seu superior. Quando *BigBoss* percebe que terminou um tabuleiro, reporta para a interface o número de soluções encontradas para aquele tabuleiro.

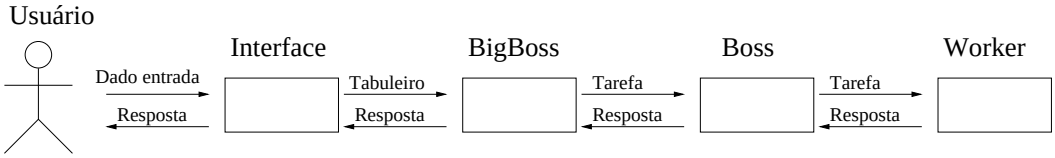


Figura 3.3: Visão externa das classes

A seguir a descrição de cada classe.

### 3.2.1 Classe *Interface*

A classe *Interface* é responsável em se relacionar com o usuário da aplicação. Ela permite que o usuário insira informações na aplicação, como por exemplo, calcular um tabuleiro de tamanho $20 \times 20$.

Essa classe também é responsável por manter o usuário informado sobre o andamento da computação. Para isso busca em *BigBoss* as informações, que podem ser tempo decorrido, tempo total de processamento, tarefas terminadas e soluções já encontradas.

A classe *Interface* é a primeira a ser executada, por isso ela é responsável por criar um objeto da classe *BigBoss*.

### 3.2.2 Classe *BigBoss*

A classe *BigBoss* é responsável por criar objetos da classe *Boss*, receber um tabuleiro para ser calculado, enviar tarefas para *Boss*, finalizar tarefas respondidas por *Boss*, esperar por pedidos de tarefas do *Boss* e por fim destruir os objetos da classe *Boss*.

A criação de objetos da classe *Boss* é feita assim que *BigBoss* inicia a sua execução. Tendo em vista que cada *Boss* é criado em uma máquina diferente, é passado para *BigBoss* as máquinas que irão hospedar os objetos da classe *Boss*.

*BigBoss* é capaz de receber um tabuleiro como trabalho, isso se dá através da interface que cria um *job* de um tabuleiro inteiro e o envia para *BigBoss*. Para cada tabuleiro é mantido o seu estado atual, como soluções já encontradas, tarefas terminadas, tempo total de processamento (tempo de CPU acumulado) e tempo decorrido. Esses dados ajudam o usuário a ver o andamento do trabalho.

Assim que *BigBoss* recebe trabalho, ele quebra o trabalho (conforme algoritmo descrito na seção 3.3) e envia para os seus subordinados. O trabalho enviado é estimado de modo que se possa ser quebrado posteriormente em dois *jobs* para cada *Worker*. Inicialmente cada *Boss* recebe dois *jobs*. Essa sobra de trabalho inicial é pois a princípio não se sabe o poder computacional de cada *Boss*.

Após que um *Boss* tenha terminado um *job*, *BigBoss* finaliza esse *job* somando as respostas no nó da árvore de resolução a qual esse *job* pertence. Caso tenha sido calculado todos os filhos desse nó, ele repassa sua resposta para seu pai. Isso é feito até que todas as respostas chegam até a raiz, a qual guarda as respostas de todo tabuleiro. Quando todos os

filhos da raiz tiverem sido calculados, esse tabuleiro foi finalizado, e então é enviada para a Interface a resposta para esse tabuleiro. Para cada resposta de *job* recebida é atualizado o poder computacional de *Boss*.

*BigBoss* deve estar sempre pronto para atender pedidos de novos *jobs* por parte de seus subordinados. Para cada requisição de trabalho *BigBoss* quebra um *job* baseando-se no poder computacional de que quem fez a requisição.

Ao final da computação *BigBoss* se encarrega de enviar um sinal de final de computação para seus subordinados e após eles terem terminado tudo, ele se encarrega de destruir os objetos *Boss*.

### 3.2.3 Classe *Boss*

A classe *Boss* é responsável por criar objetos da classe *Worker*, receber tarefas, enviar tarefas para *Worker*, finalizar tarefas respondidas por *Worker* e destruir objetos da classe *Worker*.

A criação de objetos da classe *Worker* é feita assim que *Boss* inicia a sua execução. O número de objetos a serem criados varia de acordo com a quantidade de nós do *cluster*.

Inicialmente *Boss* recebe trabalhos de seu superior. Posteriormente, há uma verificação nos trabalhos que ainda restam e conseqüente requisição a *BigBoss*, caso tenha pouco trabalho. Essa requisição ocorre de forma a prevenir a falta de trabalho para seus subordinados.

No início da computação *Boss* envia dois *jobs* para cada *Worker*, o qual executará um por vez. Dessa forma *Worker* terá trabalho para continuar sua execução mesmo que não tenha recebido um novo trabalho, e assim, evita períodos ociosos de espera de trabalho.

Ao receber uma resposta de algum *Worker*, *Boss* finaliza o *job* correspondente da mesma maneira que *BigBoss* o faz, porém ao invés de agregar as respostas até que cheguem a raiz, ele agrega as respostas até que seja finaliza do *job* recebido, ou seja, até chegar em um nó da árvore de resolução o qual o nó pai se encontra em *BigBoss*. Ao chegar nesse ponto, sabe-se que foi finalizada essa tarefa, então ocorre o envio da resposta para *BigBoss*. Para cada resposta de *job* recebida é atualizado o poder computacional *Worker* e é quebrado um novo *job* (segundo o algoritmo descrito na seção 3.3) baseado no poder computacional de *Worker*.

Ao final da computação *Boss* se encarrega de enviar um sinal de final de computação

para seus subordinados e após eles terem terminado tudo, ele se encarrega de destruir os objetos *Worker*.

### 3.2.4 Classe *Worker*

A classe *Worker* é responsável por receber uma tarefa, calcular a quantidade de posições para essa tarefa e enviar a resposta para seu superior.

Inicialmente um objeto da classe *Worker* recebe dois *jobs* para serem resolvidos. Cada *job* é resolvido seqüencialmente, sendo que nunca haverá dois *jobs* sendo calculados por um mesmo *Worker*.

Como será visto na seção 3.3, um *job* pode ser composto por uma ou várias folhas da árvore de resolução, por isso cabe ao trabalhador quebrar de modo que uma folha seja calculada por vez. Após ter resolvido todas as folhas do *job*, soma suas respostas e envia uma resposta para seu superior. Imediatamente inicia a computação de um novo *job*.

### 3.2.5 Classe *Job*

A classe *Job* é responsável por guardar todas as informações sobre um trabalho bem como prover métodos que manipulam essas informações. As informações guardadas por *Job* são informações sobre o contexto de uma tarefa, informações sobre a árvore de resolução, tempo de execução, respostas encontradas e outras variáveis de apoio. O métodos principais são para prover suporte a quebra, resolução da tarefa e manipulação da árvore de soluções.

### 3.2.6 Relacionamento inter-classe

Essa seção visa mostrar o protocolo de comunicação e as dependências entre as três principais classes da hierarquia.

A figura 3.4 mostra o diagrama de seqüência das 3 classes. A primeira classe a iniciar usa execução é *BigBoss*, que cria objetos da classe *Boss* e envia trabalho. Esse por sua vez faz o mesmo criando objetos da classe *Worker* e enviando trabalho para eles. *Worker* ao receber um trabalho, resolve e envia a resposta ao seu superior, o qual lhe envia mais trabalho. Após vários trabalhos serem resolvidos por *Worker*, finaliza-se um trabalho de *Boss*, o qual responde para seu superior. O pedido de mais trabalho por parte da classe *Boss* pode ser realizada no momento que *Boss* detecta que possui pouco trabalho restante.

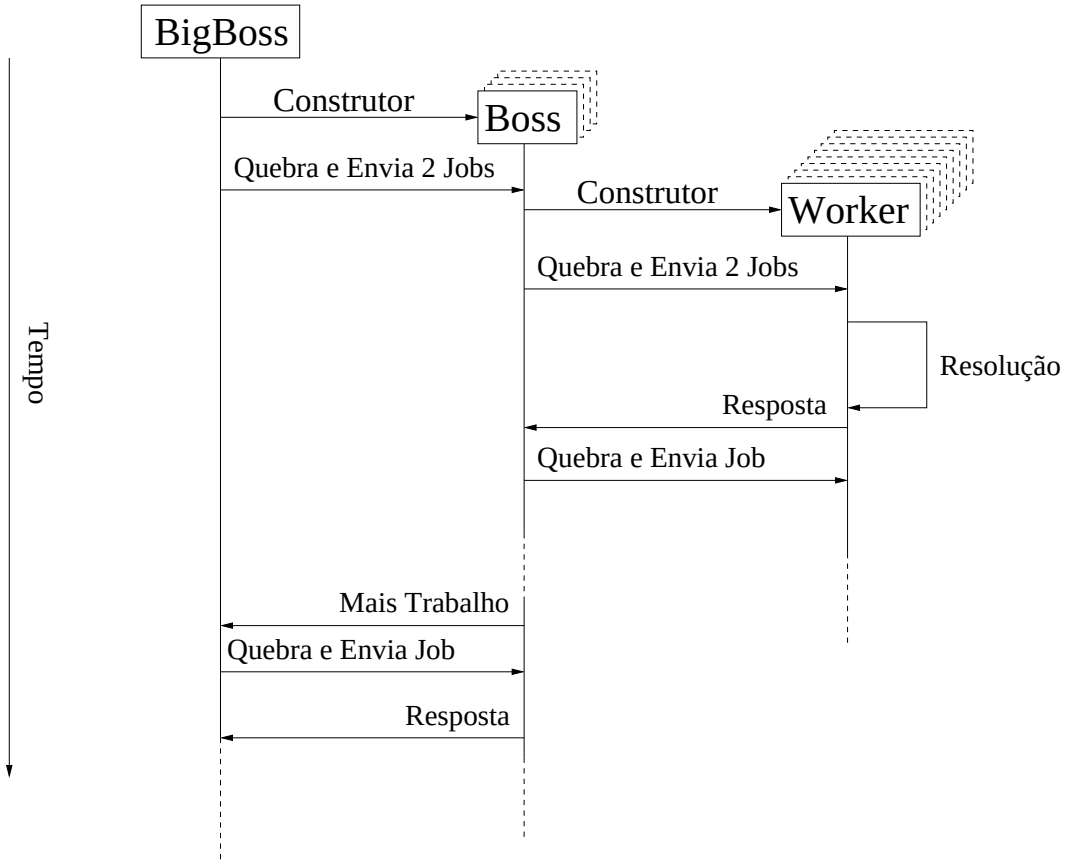


Figura 3.4: Relacionamento inter-classe

## 3.3 Algoritmo de quebra de tarefas

Um problema da quebra em níveis, descrita anteriormente, é que em geral as diversas sub-tarefas de um nível têm tamanhos bastante diferentes. Utilizando o algoritmo de Takaken, essa diferença entre os tamanhos se acentua, por causa do uso de simetrias do tabuleiro, gerando tarefas com diferença de tempo de processamento de até 100 vezes. A distribuição dessas tarefas entre os nós pode gerar um grande desequilíbrio de carga entre eles, podendo gerar um grande período de ociosidade nos nós da grade, e conseqüentemente, desperdício de poder computacional.

Para tentar reduzir esse problema, foi desenvolvido um algoritmo de quebra dinâmica (figura 3.5), que é capaz de quebrar tarefas em diferentes níveis. A escolha do nível em que a tarefa é quebrada depende de 3 fatores:

- Tamanho da tarefa (*workload*): O tamanho da tarefa é aproximado através de uma fórmula empírica, que estima a carga de trabalho gerada por uma tarefa de tal coordenada (ex: tarefa 1-3-5, tarefa com a primeira rainha posicionada na primeira casa, segunda rainha na posicionada terceira casa e terceira rainha posicionada na

quinta casa, gera uma tarefa de tamanho $x$).

- Poder computacional do nó de destino: O poder computacional do nó de destino é considerado para adequar o tamanho da tarefa às suas capacidades. Esse poder computacional inicia-se com um valor inicial comum a todos os nós, e é atualizado a cada resposta vinda desse nó. O poder computacional de um nó é a carga de trabalho de uma tarefa divida pelo tempo levado pelo nó para calcular tal tarefa.

- Tempo esperado de uma tarefa: O tempo esperado de uma tarefa serve para controlar a quantidade de trabalho enviada para um determinado nó. Esse tempo é o máximo de tempo que uma tarefa deve levar em um nó.

```
01: breakJob(job)
02:     jobGroup = vazio
03:     foreach subjob in job
04:         if toobig(subjob.workload)
05:             outputJob(jobGroup)
06:             breakJob(subjob)
07:         else
08:             if toobig(subjob.workload + jobGroup.workload)
09:                 outputJob(jobGroup)
10:             jobGroup.addJob(subjob)
11:     end
12:     outputJob(jobGroup)
```

Figura 3.5: Algoritmo de quebra de tarefas.

Baseando-se nesses fatores é possível adequar uma tarefa a um determinado nó, e dessa forma, reduzir a enorme diferença de tempo entre as tarefas.

Pode-se observar o algoritmo na figura 3.5 o qual recebe um *job* (objeto da classe *Job*), contendo um tabuleiro com as rainhas das primeiras linhas posicionadas e seu contexto (variáveis auxiliares). O primeiro *job* é um tabuleiro com as rainhas da primeira linha posicionadas.

Primeiramente, inicia-se *jobGroup* em vazio (linha 02). Então para cada cada folha (*subJob*) da árvore de soluções de *job* (linha 03) testa-se a adequação da carga de trabalho (linha 04). Esse teste consiste em confrontar os três fatores, ou seja, *toobig* retorna verdadeiro caso a carga de trabalho de *subjob* dividido pelo poder computacional do nó de destino for maior que o tempo esperado de um tarefa. Caso seja verdadeira a expressão,

envia-se para o *buffer* de saída (*outputjob*) o que há em *jobGroup* (linha 05) (esse *buffer* ignora *jobGroup* caso esteja vazio e é responsável por enviar o *job* para o nível inferior da hierarquia) e chama recursivamente a função, passando como argumento o *subJob* a ser quebrado (linha06). Caso essa tarefa esteja abaixo do tempo esperado, tenta-se aglutinar com outras tarefas de mesmo nível (irmão da árvore de resolução). Caso a soma da carga de trabalho de *jobGroup* com a carga de trabalho de *subJob* ultrapasse o tempo esperado (linha08), o conteúdo de *jobGroup* é enviado para o *buffer* de saída (linha 09). Então, adiciona-se *subJob* em *jobGroup*, o qual será aglutinado com outras tarefas de seu nível, caso a expressão da linha 08 seja falsa. Por fim, o que sobrou em *jobGroup* é enviado para o *buffer* de saída.

Esse algoritmo é capaz de enviar uma ou mais folhas da árvore de resolução em um único *job*, o que evita tarefas demasiadamente pequenas. Ele também permite que vários tabuleiros (podendo ser de tamanhos diferentes) sejam calculados ao mesmo tempo, pois cada *job* possui uma referência à quem o criou. Dessa forma poderá haver tarefas com as mesmas rainhas posicionadas, mas pertencentes a árvores diferentes. Essa qualidade do algoritmo permite a utilização dos nós da grade o tempo todo para o cálculo de vários tabuleiros, pois não há interferência entre as suas tarefas e não necessita a finalização de um tabuleiro para o início do cálculo de outro.

# 4 IMPLEMENTAÇÃO

Este capítulo tem como objetivo demonstrar as especificidades de implementação da aplicação em cada ferramenta. As duas aplicações implementadas seguiram o projeto descrito no capítulo 3. Segue abaixo algumas mudanças quanto ao projeto devido à particularidades das ferramentas. Essas mudanças são necessárias tendo em vista que as ferramentas tem comportamentos diferentes quanto as invocações de métodos remotos. É extremamente difícil fazer um projeto único que, sem mudança alguma, tire o máximo proveito das qualidades das ferramentas. Apenas serão abordados neste capítulo detalhes que se diferem do projeto.

## 4.1 POP-C++

A aplicação foi desenvolvida utilizando a versão 1.1.1 da biblioteca e pré-compilador POP-C++. As principais mudanças realizadas foram:

**Classe *BigBoss***

A criação de um objeto paralelo em POP-C++ pode ser feita passando como parâmetro a máquina onde o objeto será criado, ou caso não seja especificado, a escolha da máquina é feita por POP-C++. Como foi dito anteriormente, deseja-se que se tenha um *Boss* por *cluster*, por isso deve ser passado para *BigBoss* uma lista com as máquinas que hospedarão os objetos da classe *Boss*. Então o objeto *BigBoss* cria um objeto *Boss* em cada máquina da lista.

O envio de tarefas para *Boss* é feito através de uma chamada assíncrona seqüencial (diretivas async seq) ao método *newJob* da classe *Boss*. Esse método recebe como parâmetro um *job*, o qual é posto em uma lista de *jobs* pendentes, onde aguarda até que seja quebrado e enviado. As invocações a esse método ocorrem no início (duas vezes) e

sempre que é requisitado mais trabalho por parte do *Boss*.

A finalização de um *job* ocorre através da invocação do método assíncrono concorrente (async conc) *endJob* da classe *BigBoss* por *Boss*. Como esse método por ser executado de forma concorrente com outros métodos do objeto, teve-se o cuidado de garantir exclusão mútua em regiões críticas. Isso foi possível utilizando a classe POPSynchronizer(NGUYEN; PASIN; KUONEN, 2006).

Pedidos de novos trabalhos para *BigBoss* ocorrem através do método assíncrono concorrente *moreWork*. Esse método recebe como argumento quem está pedindo trabalho e envia um novo *job* (invocando *newJob*).

**Classe *Boss***

A criação de objetos da classe *Worker* em POP-C++ ocorre diferente da criação de um *Boss*, pois não se conhece os nós do *cluster*. Por isso não é especificado onde os objetos serão hospedados, deixando a cargo de POP-C++ disponibilizar recursos para a criação de objetos *Worker*.

O envio de tarefas para *Worker* é feito da mesma forma que *BigBoss* envia para *Boss*. Porém o envio de novos *jobs* ocorre assim que um *job* é respondido.

A finalização de tarefas em *Boss* ocorre da mesma forma que em *BigBoss*. Porém ao finalizar um *job*, é enviado para *Worker* um novo *job*.

**Classe *Worker***

A classe *Worker* é responsável por gerenciar *jobs* a serem executados. Como foi dito do capítulo 3, há apenas um *job* sendo executado em um dado momento em cada *Worker*, por isso o *Worker* mantém uma fila com os *jobs* pendentes. Como a finalização de um *job* é assíncrona, ele inicia a computação de um novo *job* imediatamente após ter finalizado o *job* anterior.

O recebimento de tarefas ocorre de forma concorrente a computação de um *job*, para que *Worker* não tenha períodos ociosos entre uma computação e outra.

**Classe *Job***

Essa classe é responsável pela computação do problema, é ela que implementa o algoritmo de Takaken. Essa classe sofreu algumas otimizações utilizando desvios incondicionais, o que permitiu a eliminação da recursão, e uso de macros, o que permitiu o

desenrolamento de laços.

## 4.2 ProActive

A aplicação foi desenvolvida utilizando a versão 3.1 da biblioteca ProActive. As principais mudanças foram:

**Classe *BigBoss***

Em ProActive a obtenção de nós disponíveis para computação se dá através de um conjunto de métodos de sua API que extraem informações do arquivo descritor de nós. A classe *BigBoss* recebe uma lista de arquivos descritores (um por *cluster*) que descrevem os recursos da grade. Com essa informação, *BigBoss* cria um objeto da classe *Boss* em um nó de cada cada descritor (um objeto em cada *cluster*), passando como parâmetro o descritor ao qual o nó pertence.

O envio de tarefas para *Boss* ocorre de maneira muito semelhante a POP-C++, pois as chamadas simultâneas a um método remoto são serializadas.

*BigBoss* está sempre disponível para responder a novos pedidos. Diferente do POP-C++, o atendimento a esses pedidos são serializados, sendo que é atendido um pedido por vez. Em ProActive, o método *moreWork* retorna um novo *job* para *Boss*, e isso é possível por causa do retorno assíncrono de ProActive, que permite que *Boss* continue sua execução mesmo quando invocar esse método no *BigBoss* e obtenha o retorno desse método posteriormente.

**Classe *Boss***

A criação dos objetos *Worker* em ProActive é feita através do uso do arquivo descritor de nós. Cria-se um *Worker* para cada nós descrito no arquivo. Essa operação é realizada em paralelo, pois cada *Boss* tem o seu arquivo descritor.

O envio de tarefas para *Worker* é feito através da invocação do método que calcula, chamado *work*. Esse método recebe como argumento um *job* e retorna o número de soluções encontradas. No início da computação esse método é invocado duas vezes, onde o primeiro é executado e o segundo fica bloqueado esperando. Após invocar esse método para todos os seus subordinados, *Boss* espera até que algum dos objetos futuros tenha a resposta, então ele finaliza essa resposta e envia novo *job* para o *Worker* que respondeu.

O pedido de novos *jobs* por *Boss* é realizado invocando o método *moreWork*, o qual

possui um retorno assíncrono. Então *Boss* deve alternar a espera por um novo trabalho e por uma resposta vinda de seus subordinados. Para tanto, a espera de um *job* novo possui um *timeout* bem pequeno para evitar que *Boss* fique bloqueado por muito tempo e demore a responder a uma finalização por parte de seus subordinados.

**Classe *Worker***

Em ProActive a classe *Worker* possui apenas o método *work* o qual recebe um *job* como argumento, resolve e retorna a resposta. Como as chamadas ao método *work* são serializadas, nunca há duas resoluções simultâneas.

# 5 AVALIAÇÃO

Este capítulo tem como objetivo avaliar as ferramentas de programação para grades computacionais *POP-C++* e *ProActive*. A fim de avaliar a sobrecarga imposta pelas ferramentas de programação, mediu-se os tempos de processamento das seguintes operações: criação de um objeto remoto, comunicação entre 2 objetos e execução da aplicação. Também há a avaliação do algoritmo de quebra, o qual influi diretamente no comportamento da aplicação.

O ambiente utilizado para os testes foi um *cluster* composto por 4 máquinas Pentium III 1.0GHz bi-processadas e por 2 máquinas AthlonMP 2400 bi-processadas. Todas as máquinas possuem 1GB de memória RAM e estão interligadas por uma rede Gigabit Ethernet. Para a execução da aplicação que usa ProActive foi utilizada a JVM da Sun versão 1.5.0.

Este capítulo está organizado em seções. Primeiramente é exposto a avaliação do algoritmo quebra, seguida da avaliação de desempenho da aplicação e por fim há a avaliação da ferramenta.

## 5.1 Avaliação do algoritmo de quebra

Esta seção tem como objetivo avaliar o algoritmo de quebra do trabalho. Para realizar essa avaliação foi executado um tabuleiro $20 \times 20$ em 4 máquinas Pentium III 1.0GHz bi-processadas. Essa configuração foi escolhida pois um tabuleiro de lado 20 representa uma grande carga de trabalho e essas máquinas formam um sistema homogêneo. O tempo estimado de trabalho pelo algoritmo de quebra para cada tarefa foi fixado em 60 segundos.

Na figura 5.1 podemos ver o tempo das tarefas quebradas de maneira estática no terceiro nível. Dessa forma, pode-se ver a grande diferença entre o tamanho das tarefas, pois há tarefas na casa de 40 segundos, enquanto há tarefas que levam menos de 1 segundo.

Outro problema desse algoritmo é a geração de muitas tarefas pequenas, como podemos ver na figura 5.1 as últimas 200 tarefas levam menos de 1 segundo para serem resolvidas.

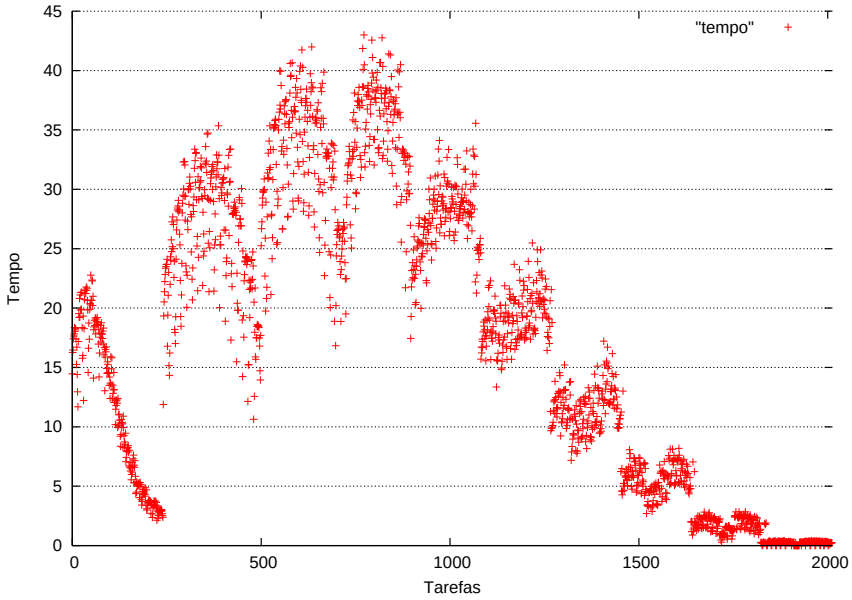


Figura 5.1: Tempo de tarefas quebradas no terceiro nível

Já na figura 5.2 tem-se o tempo de todas das tarefas quebradas com o algoritmo de quebra dinâmica. Nela pode-se observar que a grande maioria das tarefas ficaram abaixo de 60 segundos. Das tarefas acima de 60 segundos nenhuma levou mais que o 120 segundos, ou seja, o erro da estimativa nunca é maior que 100%, o que é um erro aceitável. Também não há criação de tarefas muito pequenas, sendo que algumas poucas tarefas ficam abaixo da casa de 10 segundos.

Dessa forma pode-se observar que o algoritmo de quebra dinâmica consegue diminuir a enorme diferença entre tamanho das tarefas quebradas em um nível fixo e também evita a criação de tarefas muito pequenas.

Também buscou-se comprovar a qualidade da previsão da carga de trabalho da tarefa. A figura 5.3 mostra a divisão da carga de trabalho da tarefa pelo tempo de resolução da tarefa. O esperado dessa razão é um mesmo valor para todas as tarefas. Como pode-se ver na figura, a razão ficou um pouco abaixo de 0.3 para a maioria os casos. Como há um grande erro para quando a rainha está no canto (por causa das otimizações do algoritmo de Takaken) há um grande aumento na razão, pois a previsão da carga de trabalho é super-estimada. Também pode-se perceber que para o final da execução também há uma perturbação na razão, o que pode-se atribuir ao fato de que as últimas tarefas são muito pequenas e o cálculo da carga de trabalho não é muito preciso para esse tipo de tarefas.

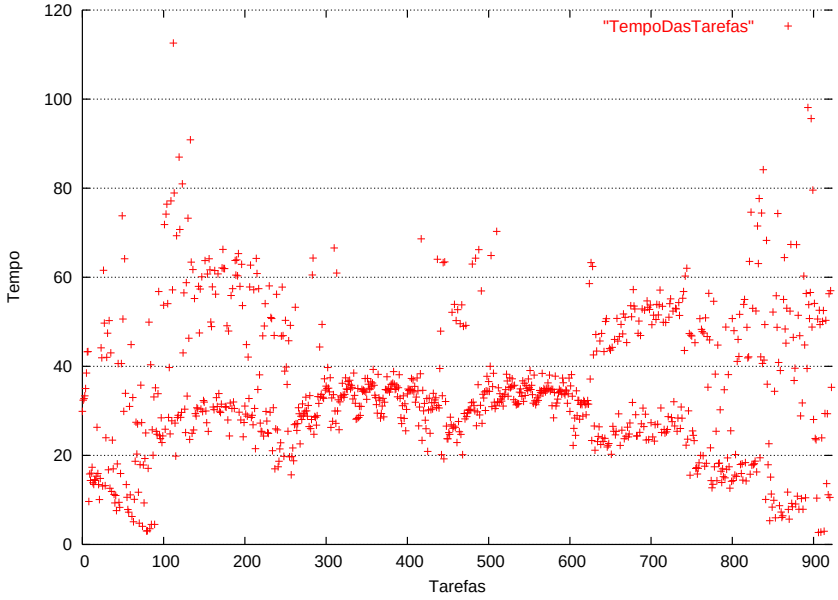


Figura 5.2: Tempo das Tarefas

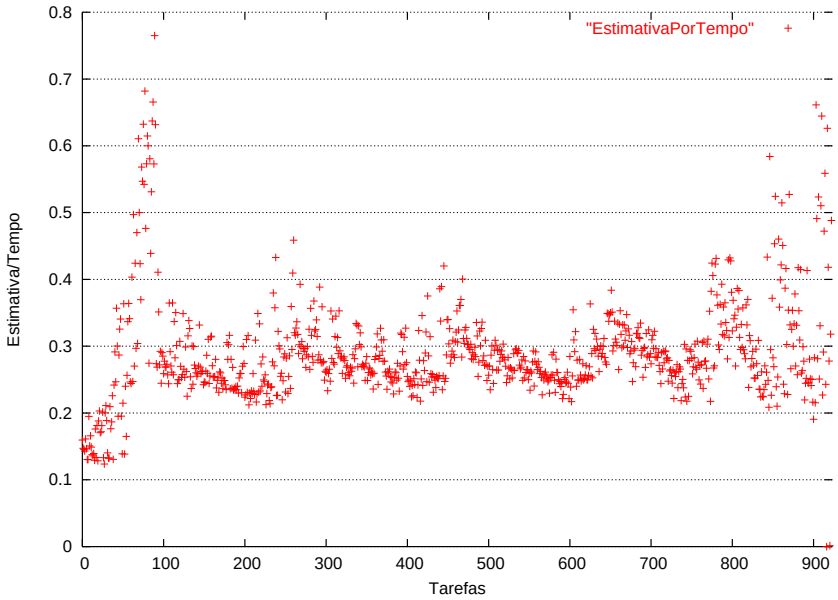


Figura 5.3: Estimativa por tempo

## 5.2 Avaliação de desempenho da aplicação

O primeiro teste realizado foi computar um tabuleiro $19 \times 19$ (tabela 5.1), pois sabe-se que este tabuleiro possui uma quantidade de cálculo não muito grande. Para resolver o problema foram utilizados 12 objetos *Worker*, sendo executado um *Worker* por processador, e 2 objetos *Boss*, que gerenciavam 6 objetos *Worker* cada (4 processadores Pentium III e 2 processadores AthlonMP). No teste em questão foi primeiro executada a aplicação criando e resolvendo um tabuleiro e outra execução criando e resolvendo 3 tabuleiros. Lembrando que nesse último caso, a computação do segundo tabuleiro pode iniciar antes mesmo do primeiro tabuleiro ter sido completamente computado. Nos mesmos moldes foi executada a resolução de um tabuleiro $20 \times 20$. A tabela mostra a média de 20 execuções para cada caso com o tabuleiro de lado 19 e 10 execuções para o tabuleiro de lado 20. O número de execuções baseou-se no tempo que leva cada execução, de modo a diluir possíveis picos e obter uma média mais realista. Na tabela 5.1 estão os resultados das execuções das versões em POP-C++, normal e com as otimizações da classe *Job*4.1, e ProActive.

Tabela 5.1: Tempos para cálculos de tabuleiros (em segundos).

| | POP-C++ Otimizado | POP-C++ | ProActive |
|---|---|---|---|
| Um tabuleiro $19 \times 19$ | 300 | 377 | 487 |
| Três tabuleiros $19 \times 19$ | 771 | 1.065 | 1.143 |
| Um tabuleiro $20 \times 20$ | 1.936 | 2.627 | 2.977 |
| Três tabuleiros $20 \times 20$ | 5.743 | 7.691 | 8.828 |

Pode-se ver na tabela 5.1 que a versão utilizando POP-C++ levou cerca de 89 % do tempo da versão em ProActive. Essa diferença não é muito grande, tendo em vista que POP-C++ é compilado e ProActive é interpretado pela JVM, embora sabe-se que a JVM utiliza técnicas de compilação em tempo de execução (*Just-in-time compiler* (SILVA; COSTA, 2005)).

Para poder avaliar a eficiência (KRONSJÖ, 1996) das aplicações, foram tomados os tempos de cálculo das tarefas durante a sua execução. Somando esses tempos tem-se uma aproximação para o tempo seqüencial da aplicação. A tabela 5.2 contém os tempos acumulados de computação.

Com base nos dados das tabelas 5.1 e 5.2, foi possível obter a eficiência das aplica-

Tabela 5.2: Tempos acumulados de computação (em segundos).

| | POP-C++ Otimizado | POP-C++ | ProActive |
|---|---|---|---|
| Um tabuleiro $19 \times 19$ | 2.868 | 3.689 | 4.359 |
| Três tabuleiros $19 \times 19$ | 8.425 | 11.028 | 12.752 |
| Um tabuleiro $20 \times 20$ | 22.328 | 30.319 | 34.840 |
| Três tabuleiros $20 \times 20$ | 66.922 | 90.941 | 104.686 |

ções. Nesse caso a eficiência trata-se da porcentagem de uso de processador pela parte de cálculo da aplicação. A eficiência é importante para poder analisar a sobrecarga da ferramenta sobre a aplicação. Na tabela 5.3 se pode observar a eficiência do programa para as execuções.

No caso da execução do tabuleiro $19 \times 19$ a eficiência é baixa pois as tarefas não têm o mesmo tempo de execução, e ao final da execução há um período de ociosidade nas máquinas até que finalize a tarefa mais demorada. Na figura 5.4 temos a visualização de um rastro de execução utilizando Pajé (STEIN; KERGOMMEAUX; BERNARD, 2000) para um tabuleiro $20 \times 20$. As barras em cinza escuro mostram o estado calculando para cada objeto *Worker*, e pode-se observar que alguns objetos terminam seu cálculo antes que outros. No caso dessa figura a diferença entre o primeiro e o último a terminar chega a 60 segundos, num tempo total de execução de 1950 segundos. Um erro desse tamanho, para o tabuleiro $19 \times 19$, é bastante significativo.

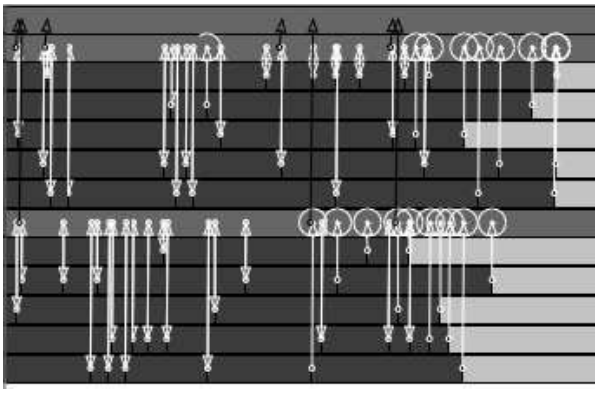

Figura 5.4: Visualização de um rastro de execução utilizando Pajé

Com o aumento do volume de computação, esse tempo de espera da última tarefa se torna menos significativo, também deve-se levar em conta que, segundo o algoritmo descrito na seção 3.3, a estimativa do poder computacional dos nós melhora em função do tempo, com isso há uma diminuição no erro do tamanho das tarefas.

Observando a tabela 5.3 podemos perceber que ProAtive teve uma menor sobrecarga na aplicação (cerca de 1,2 %) enquanto POP-C++ teve uma sobrecarga um pouco maior. Esse fato ocorre por causa que a aplicação em ProActive consome mais tempo calculando (tabela 5.2). Em ambos os casos as duas ferramentas possuem uma boa eficiência.

Tabela 5.3: Eficiência (em porcentagem).

| | POP-C++ Otimizado | POP-C++ | ProActive |
|---|---|---|---|
| Um tabuleiro $19 \times 19$ | 79,55 | 81,54 | 74,50 |
| Três tabuleiros $19 \times 19$ | 91,02 | 86,29 | 92,93 |
| Um tabuleiro $20 \times 20$ | 96,07 | 96,17 | 97,51 |
| Três tabuleiros $20 \times 20$ | 97,09 | 98,53 | 98,81 |

## 5.3 Avaliação das ferramentas

Para avaliar os custo de criação de nós e o custo de invocação de método remoto nas ferramentas, foi criada uma aplicação especializada em obter essas métricas. Na tabela 5.4 o tempo de criação dos objetos e o tempo de comunicação. O tempo de criação de objetos foi tomado para a criação e lançamento de um objeto e para 6 objetos. Optou-se por fazer essa medida por causa que as duas ferramentas são capazes de criar objetos em paralelo. Já o tempo de comunicação foi considerado o custo para a invocação de um método vazio pertencente a um objeto remoto.

Como pode-se ver na tabela, o tempo para a criação de um objeto ativo de ProActive é menos custoso que a criação de um objeto paralelo em POP-C++. Porém POP-C++ demonstra que possui um boa escalabilidade para a criação de vários objetos simultaneamente. Já o tempo de invocação de um objeto remoto, em POP-C++ leva menos tempo que em ProActive. Talvez uma explicação para a grande diferença entre os tempos é o fato de ProActive gerar um objeto futuro (ver seção 2.2.1) a cada invocação. Dessa forma POP-C++ tem uma tendência a ser mais escalável, embora as duas ferramentas tem uma boa escalabilidade.

Tabela 5.4: Tempos para criação de objetos e comunicação (em segundos).

| | POP-C++ | ProActive |
|---|---|---|
| Lançamento de um objeto | 2.0101 | 0,8183 |
| Lançamento de 6 objetos | 2.0178 | 1,0892 |
| Comunicação | 0.00015 | 0,01055 |

# 6 CONCLUSÃO

Este trabalho apresentou a comparação entre as ferramentas de programação para grades computacionais POP-C++ e ProActive. Esta comparação poderá ser usada como base para que os programadores possam escolher a ferramenta que melhor atenda as suas necessidades.

Primeiramente, foi realizado o estudo e análise das ferramentas de programação para grades, sendo aprofundado o estudo sobre as ferramentas POP-C++ e ProActive. O passo seguinte foi a criação de uma aplicação que pudesse analisar o desempenho das ferramentas. Para tanto foram implementadas duas aplicações, uma usando POP-C++ e outra usando ProActive.

Com as duas aplicações foi possível fazer a análise dos tempos, a fim de determinar a eficiência do programa usando as ferramentas. Com base nesses dados, alcançou-se o objetivo de prover informações complementares aos manuais.

As duas ferramentas são uma boa escolha para fazer programas para grades computacionais, sendo que POP-C++ permite a criação de uma aplicação com maior desempenho. Porém ProActive possui uma melhor documentação o que facilita ao programador obter maior informação sobre as capacidades da ferramenta.

Este trabalho pode ser melhorado utilizando outra aplicação, com comportamento diferente da aplicação utilizada, para comparação entre as ferramentas. Além disso, pode-se incluir outras ferramentas no teste, como por exemplo, as ferramentas citadas na Introdução.

Outro trabalho que pode ser realizado a partir deste é a criação de um *framework*, usando POP-C++ e ProActive, para aplicações que resolvem problemas semelhantes ao problema das N-Rainhas. Dessa forma pode-se aproveitar o algoritmo de quebra dinâmica e a hierarquia para resolver problemas semelhantes em grades computacionais. Isso é

possível pois toda a parte dependente da aplicação se encontra na classe *Job*.

# REFERÊNCIAS

AL-JAROODI, J.; MOHAMED, N.; JIANG, H.; SWANSON, D. A Comparative Study of Parallel and Distributed Java Projects for Heterogeneous Systems. In: INTERNATIONAL PARALLEL DISTRIBUTING PROCESSING SYMPOSIUM, 16., 2002. **Proceedings...** [S.l.: s.n.], 2002. p.115–115.

ANDERSON, D. P. BOINC: A system for public-resource computing and storage. In: GRID, 2004. **Anais...** IEEE Computer Society, 2004. p.4–10.

ANDRADE, N.; CIRNE, W.; BRASILEIRO, F.; ROISENBERG, P. OurGrid: an approach to easily assemble grids with equitable resource sharing. In: FEITELSON, D. G.; RUDOLPH, L.; SCHWIEGELSHOHN, U. (Ed.). **Job Scheduling Strategies for Parallel Processing**. [S.l.]: Springer Verlag, 2003. p.61–86. Lect. Notes Comput. Sci. vol. 2862.

BADUEL, L.; BAUDE, F.; CAROMEL, D. Efficient, Flexible, and Typed Group Communications in Java. In: JOINT ACM JAVA GRANDE - ISCOPE 2002 CONFERENCE, 2002, Seattle. **Anais...** ACM Press, 2002. p.28–36. ISBN 1-58113-559-8.

BUYYA, R. **Grid Computing Info Centre**: frequently asked questions. Acesso em Fevereiro de 2007, http://www.gridcomputing.com/gridfaq.html.

CAROMEL, D.; KLAUSER, W.; VAYSSIÈRE, J. Towards Seamless Computing and Metacomputing in Java. **Concurrency: Practice and Experience**, [S.l.], v.10, n.11–13, p.1043–1061, 1998.

CIRNE, W.; BRASILEIRO, F. V.; ANDRADE, N.; COSTA, L.; ANDRADE, A.; NOVAES, R.; MOWBRAY, M. Labs of the World, Unite!!! **J. Grid Comput**, [S.l.], v.4, n.3, p.225–246, 2006.

FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid**: Blueprint for a New Computing Infrastructure. [S.l.]: MORGAN-KAUFMANN, 1999. 259–278p.

I. Foster; C. Kesselman. Globus: A metacomputing infrastructure toolkit. **International Journal of Supercomputer Applications and High Performance Computing**, [S.l.], v.11, n.2, p.115–128, 1997. ftp://ftp.globus.org/pub/globus/papers/globus.pdf.

KIELMANN, T.; MERZKY, A.; BAL, H.; BAUDE, F.; CAROMEL, D.; HUET, F. **Grid Application Programming Environments**. [S.l.]: CoreGRID Technical Report, 2006.

KRONSJÖ, L. PRAM Models. In: **Parallel and Distributed Computing Handbook**. New York: McGraw-Hill, 1996.

KUONEM, P. **Grid Crunching Day**. Acesso em MarÃ§o de 2007, http://www.gridinitiative.ch/gridcrunchday.html.

NGUYEN, T.-A.; KUONEN, P. ParoC++: a requirement-driven parallel object-oriented programming language. **Hips**, Los Alamitos, CA, USA, v.00, p.25, 2003.

NGUYEN, T. A.; PASIN, M.; KUONEN, P. **Parallel Object Programming C++ User and Installation Manual**. 2006.

NIEUWPOORT, R. V. van; MAASSEN, J.; HOFMAN, R.; KIELMANN, T.; BAL, H. E. Ibis: an efficient java-based grid programming environment. In: ACM-ISCOPE CONFERENCE ON JAVA GRANDE (JGI-02), 2002., 2002, New York. **Proceedings...** ACM Press, 2002. p.18–27.

SILVA, A. F. da; COSTA, V. S. An Experimental Evaluation of JAVA JIT Technology. **J. UCS**, [S.l.], v.11, n.7, p.1291–1309, 2005.

STEIN, B. O.; KERGOMMEAUX, J. C. de; BERNARD, P.-E. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. **Parallel Comput.**, [S.l.], v.26, p.1253–1274, 2000.

TAKAHASHI, K. **N-Queens Problem**. Acesso em Fevereiro de 2007, http://www.ic-net.or.jp/home/takaken/e/queen/index.html.