UNIVERSIDADE FEDERAL FLUMINENSE

FELIPE DOS SANTOS RIBEIRO

SIMULAÇÕES CIENTÍFICAS INTELIGENTES - TORNANDO APLICAÇÕES MPI AUTÔNOMAS E MALEÁVEIS

NITERÓI

2012

UNIVERSIDADE FEDERAL FLUMINENSE

FELIPE DOS SANTOS RIBEIRO

SIMULAÇÕES CIENTÍFICAS INTELIGENTES - TORNANDO APLICAÇÕES MPI AUTÔNOMAS E MALEÁVEIS

Trabalho submetido ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. Área de concentração: Processamento paralelo e distribuído.

Orientador:
Prof. PhD. EUGENE FRANCIS VINOD REBELLO

Co-orientador:
Prof. D.Sc. ALEXANDRE DA COSTA SENA

NITERÓI

2012

Simulações Científicas Inteligentes - Tornando Aplicações MPI Autônomas e Maleáveis

Felipe dos Santos Ribeiro

Trabalho submetido ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Aprovada por:

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

Prof. D.Sc. Alexandre da Costa Sena / IC-UFF
(Co-orientador)

Profa. D.Sc. Aline de Paula Nascimento / IC-UF

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

Niterói, 24 de outubro de 2012.

"Mera mudança não é crescimento. Crescimento é a síntese de mudança e continuidade, e onde não há continuidade não há crescimento."

-C. S. Lewis

"A humildade é o primeiro degrau para a sabedoria."

-Tomás de Aquino

Agradecimentos

Agradeço primeiramente a Deus, por estar sempre presente comigo e permitir que eu chegasse até aqui, me ajudando a superar as dificuldades e me ensinando valores para toda a vida.

À minha mãe, por me apoiar sempre que eu precisei, pelo seu amor, compreensão, companheirismo, ensinamentos e por todas as vezes que me motivou a seguir em frente.

Aos orientadores Vinod Rebello e Alexandre Sena, pela paciência e dedicação, pelos conselhos que vão além desse projeto, e por me ajudarem sempre que eu precisei.

A Jacques Alves e Fernanda Oliveira, pela paciência ao me ajudarem em C, Linux e EasyGrid. Quando entrei para a iniciação científica eu não sabia nada dessas coisas, posso dizer que só aprendi o que eu sei hoje por causa da ajuda deles.

A Aline Nascimento pela ajuda com escalonamento dinâmico e EasyGrid, e também, incluindo Cristina Boeres, por participarem da minha banca, me ajudando a melhorar a forma de apresentar meu trabalho.

À turma do SGCLab por terem me ajudado sempre que precisei e pelos momentos de descontração, com conversas divertidas, boliche, cinema, videogames, conselhos e tudo mais.

A Jefferson Mello, Matheus Chung Nin e Vinicius Marmontelle, por se tornarem grandes amigos com quem sempre posso contar. E a todos os amigos da UFF, que me incentivaram e me ajudaram.

Aos professores que me deram todo o conhecimento necessário e alguns foram muito além da área acadêmica, influenciando toda a minha vida.

Resumo

Muitas aplicações em diversas áreas como engenharia, física e biologia requerem poder computacional e capacidade de armazenamento em larga escala. Para atender a essas necessidades a um custo razoável, as plataformas disponíveis para os cientistas, principalmente em países em desenvolvimento, são frequentemente compostas de diversos recursos interligados por redes de capacidades variadas, formando *clusters*, *grades computacionais* e/ou *nuvens computacionais*. Os recursos de uma grade ou nuvem geralmente são compartilhados e, portanto, de natureza dinâmica, tornando-os difíceis de serem aproveitados eficientemente. Devido à essa dificuldade, por causa da natureza dinâmica dos recursos e do pouco conhecimento do usuário sobre o ambiente, foi criado o gerenciador EasyGrid AMS, que têm como objetivo facilitar o trabalho do programador, permitindo à aplicação adaptar-se às mudanças ocorridas nos recursos disponíveis, tornando-a mais eficiente. O gerenciador EasyGrid AMS gerencia aplicações paralelas que utilizam a biblioteca MPI. Neste trabalho será acrescentado ao gerenciador EasyGrid AMS a propriedade de auto-configurar uma aplicação executada por meio dele, ou seja, habilidade de modificar a estrutura da aplicação paralela dinamicamente para um melhor aproveitamento dos recursos disponíveis. Essa auto-configuração será feita modificando-se o grau de paralelismo da aplicação dinamicamente (maleabilidade), e sem a necessidade de intervenção do usuário (autonomia). Como estudo de caso será utilizada uma aplicação de simulação astrofísica chamado N-Corpos, que calcula iterações entre corpos celestes e se assemelha com diversos programas de simulação. A aplicação N-Corpos foi escolhida como base, pois, além de ser uma importante aplicação científica utilizada atualmente, suas tarefas possuem relações de dependência que tornam necessário um gerenciamento extremamente eficaz.

Palavras-chave: Grades Computacionais, Maleabilidade, MPI, Aplicações Autônomas, EasyGrid AMS, Paralelismo

Abstract

Many applications in different areas such as engineering, physics and biology require large quantities of computational power and storage capacity. To meet these requirements at a reasonable cost, platforms available to scientists, especially in developing countries, are often composed of various resources connected by networks of varying capacities, forming clusters, grids or clouds. The resources of a grid or cloud generally are shared and therefore dynamic in nature. This makes developing applications that are capable of executing efficiently in these environments extremely difficult. The EasyGrid Application Management System (EasyGrid AMS) can be used to transform cluster-based MPI applications into autonomic ones capable executing robustly and efficiently in these distributed environments. Each application becomes responsible for managing their execution and seeks to adapt itself to the changes that occur in the environment. The work presented here aims to enhance the EasyGrid AMS with the property to self-configure an application, i.e., provide ability to modify the structure of the parallel application during its execution to make better use of available resources. This self-configuration will be achieved by modifying the degree of parallelism of the application dynamically (known as *malleability*), and without the intervention of the user (autonomy). As a case study, an astrophysics simulation, called N-body, which calculates interactions between celestial bodies, has been integrated with the new version of the EasyGrid AMS. The N-body application was chosen because, besides being an important and extensively used scientific application, its tasks have dependency relationships which require a highly effective management strategy and present characteristics similar to many other scientific simulation applications.

Keywords: Grids, Malleability, MPI, Autonomic Applications, EasyGrid AMS, Parallelism

Palavras-chave

1. Grades Computacionais
2. Maleabilidade
3. MPI
4. Aplicações Autônomas
5. EasyGrid AMS
6. Paralelismo

Glossário

GM	:	Global Manager;
HM	:	Host Manager;
SM	:	Site Manager;
MPI	:	Message Passing Interface;
AMS	:	Application Management System;
RMS	:	Resource Management System;

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Diversos problemas que estão sendo estudados atualmente necessitam de grande quantidade de poder computacional e capacidade de armazenamento para serem solucionados. Entre estes podemos citar muitas simulações e projetos científicos, como por exemplo, o processamento de dados gerados por aceleradores de partículas, como o *Large Hadron Collider* (LHC), o maior acelerador de partículas do mundo [2]. Para analisar os dados gerados pelo LHC, mais de 140 centros computacionais, em 35 países, estão colaborando através da Worldwide LHC Computing Grid [3]. Há também o projeto Genoma, cujo objetivo é mapear o código genético de vários organismos, sendo útil na identificação e tratamento de diversas doenças [4]. Além desses exemplos, há problemas de previsão do tempo, simulações de interações entre moléculas, exploração de petróleo [5] e muitos outros.

Como a quantidade de trabalho para resolver esses problemas é muito maior que o poder computacional de um único computador atual, utiliza-se, principalmente, a computação paralela para atender a essa demanda por recursos computacionais. Desta forma, divide-se o trabalho a ser realizado em várias tarefas, que são executadas paralelamente pelos processadores.

As plataformas paralelas mais comumente utilizadas são os *clusters*, que são agregados de computadores compostos de vários computadores homogêneos interconectados por uma rede [6]. O maior problema dos *clusters* é a falta de escalabilidade, pois, caso o pesquisador necessite de mais recursos, este deve comprar mais equipamentos para seu cluster, ou outro cluster mais moderno. Para contornar esse problema, atualmente são utilizadas estruturas heterogêneas e dinâmicas, como grades computacionais (*grids*) [7] e

nuvens computacionais (*cloud computing*) [8]. Tanto grades quanto nuvens, por exemplo, compõe-se de vários *clusters* espacialmente distribuídos, onde os recursos são compartilhados entre seus usuários. Dessa forma, o usuário pode pagar apenas pelo tempo e quantidade de recursos utilizados e diminui-se a quantidade de recursos ociosos. Assim, países em desenvolvimento tendem a utilizar cada vez mais essas plataformas, devido ao poder computacional disponibilizado a baixo custo.

O maior desafio ao se utilizar plataformas heterogêneas e compartilhadas está na dificuldade de gerenciar a execução da aplicação. Isso porque, além de haver a dificuldade de escolher onde as tarefas da aplicação irão executar, o escalonamento dessas tarefas, com o tempo, pode ser prejudicado significativamente devido às constantes mudanças que podem ocorrer no poder computacional oferecido a esta aplicação pelos recursos disponíveis.

Para executar eficientemente aplicações onde as tarefas devem se comunicar entre si, o cientista deve ter bastante conhecimento sobre as características da aplicação e dos recursos disponíveis para sua execução. Essa é uma tarefa muito complexa para os cientistas de uma maneira geral. Além disso, o administrador do sistema também não é capaz de resolver esse problema, pois, apesar dele saber sobre o estado do sistema a todo o momento, não conhece as características da aplicação, lidando apenas com processos ou *jobs* (uma solicitação de um usuário para a execução de uma aplicação).

Uma forma de solucionar esse problema é encontrar métodos para tornar as aplicações autonômicas [9]. Uma aplicação autonômica consegue tomar decisões por conta própria sobre seu gerenciamento sem a interferência do usuário. Essas decisões podem envolver auto-otimização, auto-recuperação, auto-proteção e mudanças na estrutura da aplicação, definida como auto-configuração.

Para implementar a autonomia, deve-se considerar que aplicações paralelas podem ser classificadas do seguinte modo: *rígidas*, *moldáveis*, *evolutivas* e *maleáveis* [10]. As *rígidas* só podem executar em um número pré-programado de recursos; as *moldáveis* podem executar em um número variado de recursos, mas é necessário especificar uma quantidade de recursos antes da execução. Por outro lado, aplicações *evolutivas* e *maleáveis* conseguem modificar a quantidade de recursos utilizados durante a execução. Uma aplicação maleável pode modificar a granularidade dos processos dinamicamente [11]. A granularidade pode ser medida através da relação entre tempo de computação e tempo de comunicação de uma tarefa [12], e ela é dependente do número de tarefas em que um problema pode ser decomposto, o tamanho dessas tarefas e a comunicação entre elas. Desta forma, uma aplicação *maleável* é capaz se auto-configurar para adaptar-se às mudanças ocorridas no

ambiente, sendo esse um fator importante para a autonomia.

O objetivo desse trabalho é mostrar a viabilidade de transformar aplicações científicas originalmente desenvolvidas para *clusters* (aplicações moldáveis) em maleáveis (autoconfiguráveis) e, dessa forma, aumentar a eficiência das aplicações paralelas nas plataformas de mais baixo custo e escaláveis, como grades e nuvens, cada vez mais utilizadas atualmente.

Como um estudo de caso, uma avaliação foi realizada através da implementação da maleabilidade em uma aplicação real da física. Este trabalho também tem como objetivo mostrar os benefícios da maleabilidade. Para atingir esses objetivos, é utilizado o apoio do middleware EasyGrid AMS, que gerencia aplicações paralelas que foram originalmente desenvolvidas para ambientes homogêneos e utilizam a biblioteca MPI [13, 14], que é uma interface para a troca de mensagens entre processos muito utilizada em programas científicos.

As principais contribuições desse trabalho são:

1. Implementação da maleabilidade em uma aplicação real utilizando o middleware EasyGrid AMS;
2. Comparação do desempenho da versão moldável, evolutiva e maleável dessa aplicação em ambientes dinâmicos;
3. Análise da sobrecarga da maleabilidade;
4. Levantamento, implementação e comparação de heurísticas para encontrar o grau de paralelismo próximo do ótimo dinamicamente;
5. Criação de uma ferramenta para avaliar o desempenho e comportamento de aplicações paralelas graficamente;
6. Mudança na forma como o poder computacional era calculado no AMS, para obter melhor desempenho em computadores *multicore*;
7. Projeto e implementação de uma heurística de escalonamento dinâmico para ambientes *multicore*;

Essa monografia está organizada em 6 capítulos. O capítulo 2 identifica os trabalhos relacionados, com foco no EasyGrid AMS, do qual esse trabalho deriva. Já o capítulo 3 apresenta um estudo de caso e o por quê dessa escolha. No capítulo 4, será mostrado como

implementar a maleabilidade utilizando o EasyGrid AMS, e todos os ajustes necessários para uma execução eficiente. No capítulo 5 veremos as análises experimentais. Por último, no capítulo 6, apresenta as conclusões e trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Neste capítulo será explicado melhor os conceitos de computação autônoma e maleabilidade e os trabalhos relacionados a eles. A área de pesquisa a respeito da maleabilidade é relativamente nova e, por isso, ainda tem poucos trabalhos relacionados. Também serão apresentados o *middleware* EasyGrid AMS e o modelo de execução *1PTASK*, que serão utilizado nesse trabalho.

2.1 Computação Autônoma

O conceito de computação autônoma é baseado no Sistema Nervoso Autônomo do corpo humano. O Sistema Nervoso Autônomo é a parte do sistema nervoso que controla funções involuntárias do corpo, como a circulação do sangue, atividade intestinal e a produção de hormônios. Esse auto-gerenciamento biológico inspirou um novo paradigma na computação, para a criação de aplicações e sistemas computacionais capazes de se gerenciarem [15]. Em [16], a computação autônoma é definida como o conjunto de sistemas computacionais capazes de se gerenciarem, dados os objetivos de alto nível dos administradores. Esses sitemas devem possuir as seguintes propriedades: auto-configuração *(self-configuration)*, auto-recuperação *(self-healing)*, auto-otimização *(self-optimization)* e auto-proteção *(self-protection)*. Essas propriedades vem acompanhadas de quatro atributos: auto-monitoramento *(self-monitoring)*, auto-conhecimento *(self-awareness)*, conhecimento do ambiente *(environment-awareness)* e auto-ajuste *(self-adjusting)* [15].

Para facilitar a criação de aplicações paralelas autonômicas, vários projetos vêm desenvolvendo *middlewares*, isto é, camadas de software de interface entre a aplicação e a

infra-estrutura [17]. Atualmente há duas abordagens para o projeto de sistemas autônomos: Uma abordagem centrada nos recursos, e outra centrada na aplicação.

Na primeira abordagem, centrada nos recursos disponíveis do ambiente distribuído, o gerenciamento é feito por meio de um Sistema Gerenciador de Recursos (RMS - *Resource Management System*). Os RMSs têm o objetivo de maximizar a utilização dos recursos, independentemente dos requisitos e características internas da aplicação [17]. Como exemplos de RMSs temos o Condor-G [18] e o Cactus [19], utilizados em ambientes distribuídos de larga escala. O RMS Condor-G, projetado para grades computacionais, integra ferramentas do Globus *toolkit* [20] e o sistema de gerência Condor [21], sendo responsável pela descoberta e aquisição de recursos, inicialização, monitoramento, gerenciamento da execução, notificação de término, detecção e tratamento de falhas dos processos definidos pelo usuário. Seu objetivo é maximizar a utilização dos recursos disponíveis, combinando pedidos de recursos dos usuários com as ofertas de recursos do sistema. O RMS Cactus tem como objetivo esconder atrás de um único ponto as complexidades do ambiente distribuído. Ele possui recursos de tolerância a falhas e balanceamento de carga, e utiliza a ferramenta Globus para autenticação e descoberta de recursos.

A abordagem centrada nos recursos pode não ser suficiente para executar eficientemente todo tipo de aplicação distribuída, pois é necessário considerar as características de cada aplicação para que possam ser feitos ajustes adequados a sua execução [22]. A segunda abordagem, centrada na aplicação, leva em consideração essas características. O gerenciamento da aplicação é feito através de um *middleware* chamado Sistema Gerenciador de Aplicação (AMS - *Application Management System*). Através de um AMS, as aplicações podem conhecer o ambiente de execução e se auto-ajustar às suas mudanças para obter maior eficiência na execução. Um exemplo de AMS é o GrADS [23], que possui o objetivo de minimizar o tempo de execução da aplicação. Para atingir esse objetivo esse sistema fornece ferramentas e bibliotecas que permitem ao usuário criar aplicações que possam ser encapsuladas como programas objetos configuráveis (COPs) que inclui um modelo que estima o desempenho da aplicação em um conjunto de recursos. Possui um escalonamento centralizado e mecanismos de tolerância a falhas e monitoramento de recursos. Outro exemplo de AMS é o EasyGrid, utilizado nesse trabalho, que possui funcionalidades de tolerância a falhas, controle de mensagens, monitoramento, criação dinâmica e escalonamento dinâmico de processos [24]. O objetivo e funcionalidades do EasyGrid AMS serão explicados com mais detalhes na seção 2.3.

2.2 Maleabilidade

A maleabilidade de uma aplicação é a capacidade de se modificar a granularidade dos seus processos dinamicamente [11], ou seja, o grau de paralelismo da aplicação, relacionado a quantidade de tarefas que a aplicação será dividida, poderá mudar ao longo da execução. A maleabilidade é uma forma de auto-configuração, uma das propriedades necessárias para a autonomia. A necessidade de tornar uma aplicação maleável, para executar em um ambiente distribuído e dinâmico, ocorre por, em muitos casos, não ser possível estabelecer a priori um grau de paralelismo ótimo para que a aplicação se adapte às possíveis variações do ambiente. Por exemplo, suponha um ambiente com 10 processadores homogêneos e uma aplicação formada por tarefas independentes. Nesse caso, podemos supor que dividir a aplicação igualmente em 10 processos, um para cada processador, seria o ideal para esse ambiente (na maioria das situações). Mas se, durante a execução, mais 10 processadores tornarem-se disponíveis, não será possível aproveitar todo o poder computacional. Por outro lado, se a aplicação fosse inicialmente divida em 20 processos, haveria um custo desnecessário de comunicação e gerenciamento dos processos, principalmente se o número de processadores disponíveis diminuisse. Se a aplicação não for formada por tarefas independentes essa diferença se torna ainda mais evidente, pois os custos de comunicação e gerenciamento podem ser ainda maiores, por causa dos relacionamentos entre as tarefas. Com a maleabilidade esse problema pode ser solucionado, pois a própria aplicação irá mudar o seu grau de paralelismo, quando necessário.

Para implementar a maleabilidade é necessário, de forma geral, realizar as seguintes tarefas:

1. Monitorar o estado dos recursos disponíveis
2. Avaliar o desempenho da aplicação no ambiente atual
3. Reconfigurar a aplicação, quando for necessário, para melhorar seu desempenho

Métodos baseados no *checkpoint* de processos são comumente utilizados para realizar a reconfiguração da aplicação. Em [25], foi examinado um *framework* para desenvolver aplicações paralelas MPI maleáveis e migráveis através do uso de *checkpoints*. Nesse caso, o programador precisa inserir chamadas a bibliotecas no seu código para identificar os dados que serão guardados ao realizar os *checkpoints* e os pontos de restauração. Além disso, nesse framework, toda a aplicação deve ser interrompida para que possa ser feita a reconfiguração, quando considerada necessária.

Uma outra forma de maleabilidade foi implementada em [26] através da extensão de uma biblioteca para *checkpointing* e migração de processos que utilizam a interface MPI. A estratégia utilizada foi dividir ou agrupar processos quando necessário, modificando assim a granularidade. Ao contrário do trabalho em [25], para modificar um pequeno grupo de processos, a aplicação inteira não necessita ser interrompida. Porém, é necessário instalar um sistema operacional específico (*Internet Operating System*) em todos os recursos utilizados. Isso pode não ser prático em vários casos.

A necessidade da realização de *checkpoints* e migração de processos decorre do modelo de execução tradicionalmente utilizado em aplicações MPI, o modelo *1PPROC*, que cria um processo para cada processador disponível. O modelo *1PPROC* é baseado na suposição de que o ambiente é composto de recursos homogêneos, dedicados a aplicação, livre de falhas e interconectados por uma rede rápida. As aplicações são tipicamente projetadas para executar processos idênticos e de longa duração, onde cada processador executa apenas um processo durante toda a execução do programa [27, 28]. Em [1], foi proposto um modelo alternativo, *1PTASK*, no qual, ao contrário do anterior, é criado um processo por tarefa da aplicação. Entende-se por tarefa uma unidade pequena de trabalho, que faz parte do programa. Já um processo é um programa em execução, que realiza uma ou várias tarefas. Ao executar um processo por tarefa, isso a torna independente da arquitetura e do ambiente de execução, ou seja, não associa o número de processos da aplicação ao número de recursos disponíveis, permitindo o desenvolvimento de aplicações paralelas que explorem o máximo do paralelismo sem se importar com o ambiente disponível. As figuras 2.1 e 2.2 representam, um exemplo de aplicação no modelo *1PPROC* e no modelo *1PTASK*, respectivamente. Nota-se que no modelo *1PTASK* é possível que haja mais processos do que processadores. Portanto, uma desvantagem desse modelo é a necessidade de lidar e gerenciar uma quantidade maior de processos.

O sistema gerenciador de aplicação EasyGrid AMS [29], que gerencia aplicações paralelas que utilizam a biblioteca MPI, consegue implementar o modelo *1PTASK* através da criação dinâmica de processos. Nele, cada processo é criado apenas quando os dados necessários para realizar sua execução já foram obtidos e um processador está disponível para a execução desse processo. O número máximo de processos que serão executados por processador ao mesmo tempo é definido pelo usuário ou AMS. Em [1] foi apresentado o modo para transformar aplicações MPI moldáveis em evolutivas através do EasyGrid AMS e do modelo *1PTASK*. A aplicação evolutiva se adapta às mudanças no poder computacional oferecido, através da realocação de tarefas que ainda não foram executadas (assim evitando migração e checkpointing de processos). Esta estratégia é benéfica em termos

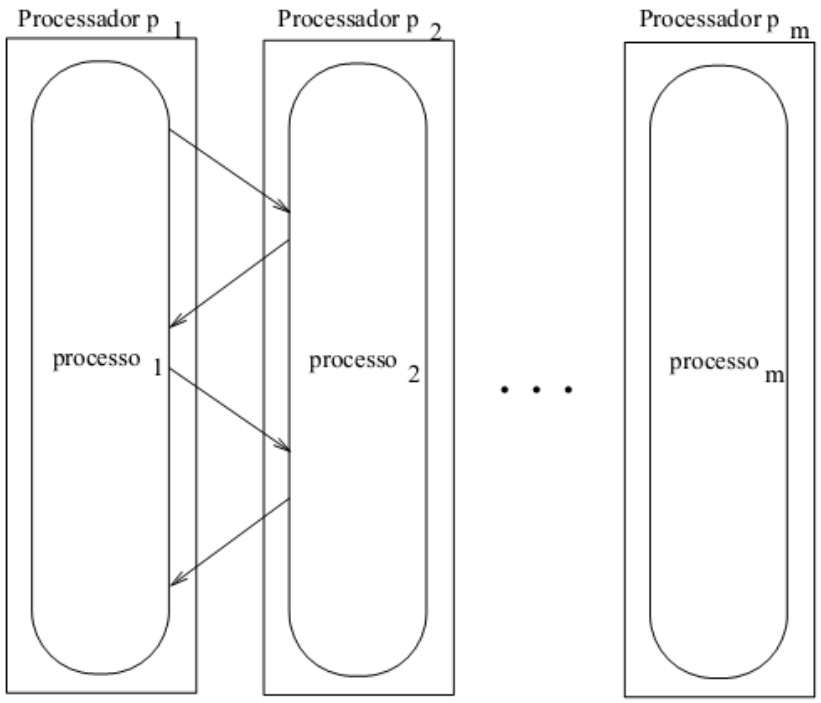


Figura 2.1: Exemplo de aplicação no modelo de execução *1PPROC* [1]

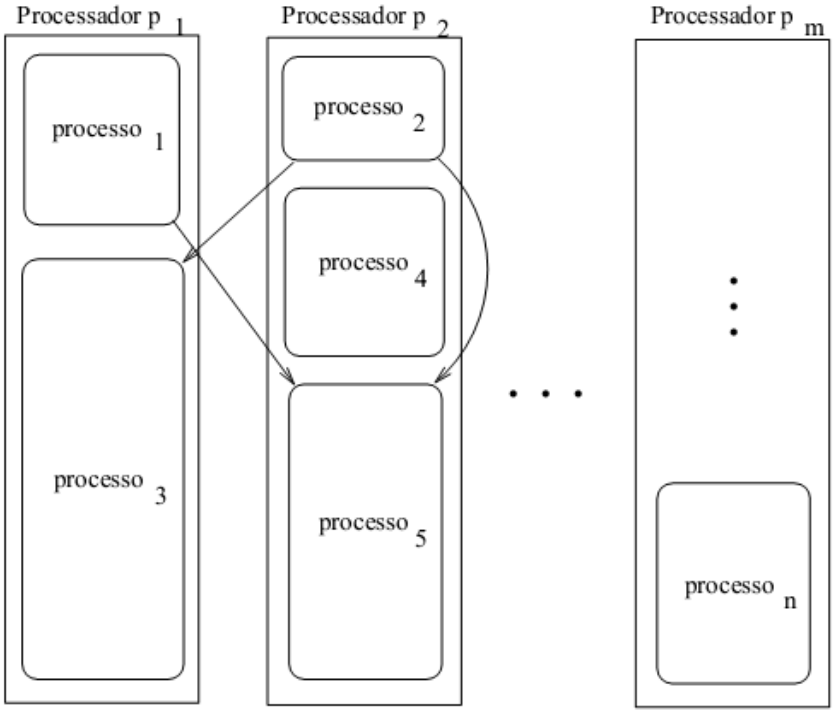


Figura 2.2: Exmeplo de aplicação no modelo de execução *1PTASK* [1]

de balanceamento de carga, mas, devido ao fato da granularidade ser definida somente no início da execução pode haver perda de desempenho por causa de mudanças que podem ocorrer no ambiente. Por exemplo, se houver uma redução na quantidade de recursos disponíveis, o desempenho pode ser prejudicado pela granularidade ser muito fina, por causa da sobrecarga de comunicação e gerenciamento desnecessários num mesmo recurso. E se houver um aumento do número de recursos disponíveis, talvez não haja tarefas suficientes para aproveitar os novos recursos.

Este trabalho apresenta a estratégia para permitir ao EasyGrid AMS prover a maleabilidade em uma aplicação anteriormente moldável. Isso será feito sem a necessidade de instalar um sistema especializado, nem de conhecimento adicional do programador sobre características do ambiente.

2.3 O Middleware EasyGrid AMS

A principal finalidade do Sistema Gerenciador de Aplicações EasyGrid AMS é executar e gerenciar eficientemente aplicações MPI numa grade que ofereça serviços básicos, como o Globus [20] e a biblioteca de comunicação MPI [30, 14]. Sua arquitetura é composta de três níveis, conforme mostra a Figura 2.3:

1. O gerenciador global (Global Manager-GM), responsável por supervisionar os sites onde a aplicação está executando (ou poderia executar);
2. Em cada site, um processo gerenciador do site (Site Manager-SM), responsável pela alocação dos processos da aplicação nos seus recursos;
3. E em cada máquina, um gerenciador da máquina (Host Manager-HM), responsável por escalonar, criar e executar os processos da aplicação associados a sua máquina.

Cada gerenciador está estruturado em uma arquitetura de camadas, conforme a Figura 2.4. Cada camada é responsável por um serviço essencial na execução eficiente de aplicações MPI. Primeiramente temos uma camada de abstração, que substitui as chamadas a procedimentos MPI do programa do usuário por funções adequadas à estrutura do AMS. As informações sobre o ambiente de execução e o estado das tarefas são obtidas através da camada de monitoramento. Através das informações de monitoramento uma camada pode modificar seu estado ou o de uma camada de nível mais baixo.

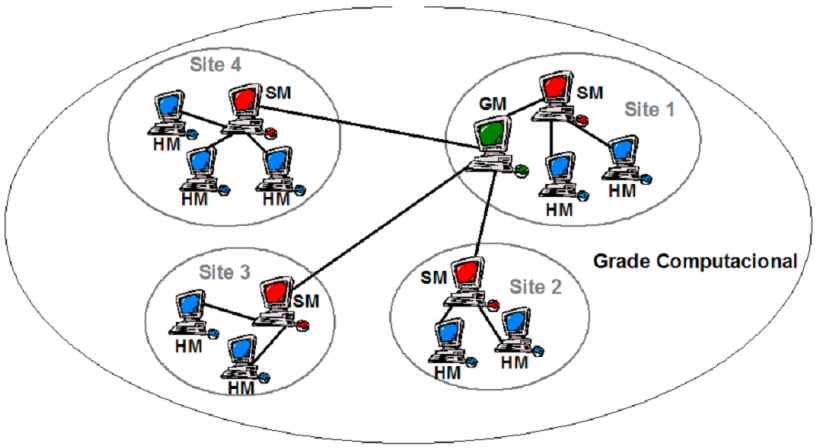


Figura 2.3: Hierarquia de gerenciadores do EasyGrid AMS [1]

As principais funções disponíveis no EasyGrid AMS são: monitoramento, gerenciamento de processos e a comunicação entre eles, escalonamento dinâmico e tolerância a falhas. Uma visão geral da arquitetura do EasyGrid pode ser encontrada em [24]. A implementação inicial do *middleware* EasyGrid, a descrição básica dos serviços de monitoramento, gerenciamento e comunicação de processos, escalonamento dinâmico e tolerância a falhas podem ser encontrados em [22, 29, 31]. Uma explicação mais detalhada sobre tolerância a falhas no EasyGrid pode ser encontrada em [32]. O estudo do escalonamento dinâmico e suas implementações podem ser encontrados em [33, 34, 35]. Uma explicação completa sobre o escalonamento dinâmico no projeto EasyGrid pode ser vista em [36].

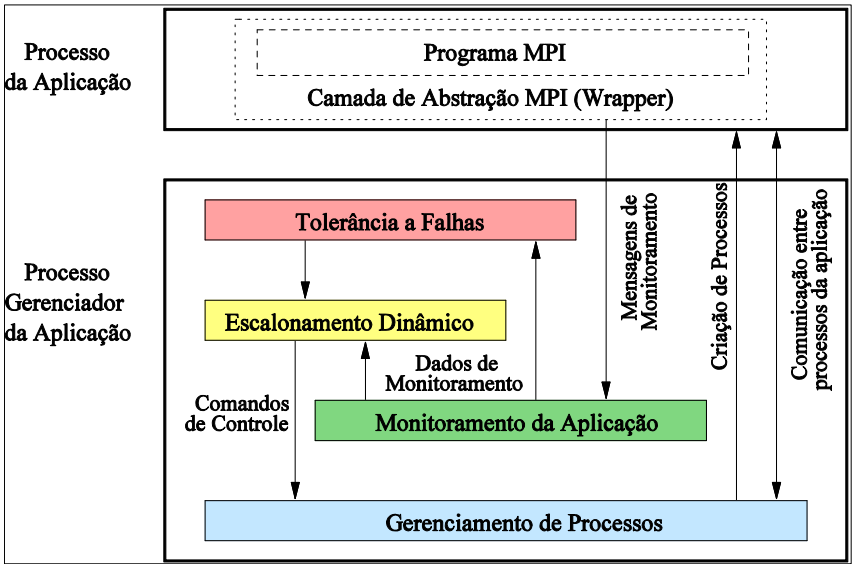


Figura 2.4: Arquitetura de camadas de cada processo gerenciador AMS [1]

Durante esse trabalho foi acrescentada uma nova camada ao EasyGrid AMS, a camada de maleabilidade, que será explicada com mais detalhes no capítulo 4. A figura 2.5 mostra

a mudança na arquitetura de camadas do EasyGrid AMS ocorrida nesse trabalho.

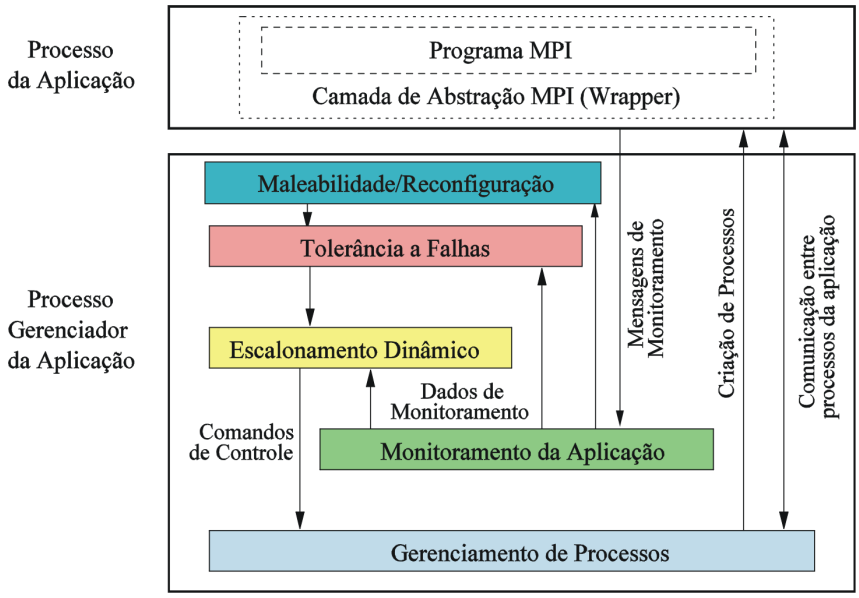


Figura 2.5: Arquitetura de camadas de cada processo gerenciador AMS após a inclusão da camada de maleabilidade

Neste capítulo foram vistos com mais detalhes os conceitos de computação autônoma e maleabilidade e seus trabalhos relacionados. Foi apresentado o modelo de execução *1PTASK*, e suas vantagens em relação ao modelo de execução normalmente utilizado, *1PPROC*. Também foi apresentado um resumo do funcionamento do gerenciador EasyGrid AMS, por meio do qual é possível executar aplicações eficientemente utilizando o modelo *1PTASK*, e que foi utilizado durante este trabalho para prover a maleabilidade. No próximo capítulo veremos qual foi a aplicação escolhida para se tornar maleável, sua utilidade e o porquê dessa escolha.

Capítulo 3

Estudo de Caso

Como um estudo de caso foi utilizada uma aplicação de simulação astrofísica chamada N-Corpos, que calcula interações entre N corpos celestes. Esta simulação considera cada corpo celeste como um ponto singular com suas características associadas, como velocidade, massa e aceleração. Como a maioria de simulações científicas, ela é uma aplicação paralela e iterativa, com cada iteração sendo executada numa forma distribuída com a troca de dados durante e entre iterações. Aplicações desse tipo são difíceis de se projetar para serem executadas de forma eficiente em sistemas computacionais heterogêneos, ainda mais quando necessário gerenciá-las em ambientes dinâmicos. Logo, a maleabilidade seria fundamental para tirar proveito das novas tendências nos sistemas computacionais de larga escala.

3.1 Aplicação *N*-Corpos

Os algoritmos numéricos para solucionar o problema N-Corpos podem ser divididos em duas categorias: uma abordagem direta ou um esquema aproximado. Na abordagem direta [37], são calculadas as N^2 forças que são exercidas entre os corpos para cada passo da evolução (*time step*), tendo uma complexidade $O(N^2)$. Na outra abordagem são realizadas aproximações para as interações entre partículas muito distantes, obtendo uma complexidade inferior, porém com perda de precisão em relação à abordagem direta. Como exemplos de algoritmos aproximados temos o de Barnes-Hut [10] e Greengard [78], com complexidade *O(NlogN)* e *O(N)*, respectivamente.

A abordagem direta é utilizada para simular tanto regiões de pouca densidade como

de alta densidade [38], *globular star clusters* [39] ou *galatic nuclei* [40], pois nessas regiões é necessário haver alta precisão nos cálculos. Mas, devido a sua complexidade $O(N^2)$, simulações com milhares, e principalmente milhões de estrelas ainda são um desafio.

A aplicação N-Corpos foi escolhida como base por que, além de ser um importante problema físico, suas tarefas estão fortemente acopladas, havendo muita comunicação entre elas, tornando necessário um gerenciamento extremamente eficaz. Também foi escolhida a abordagem direta, em razão da sua maior precisão e, consequentemente, resultados mais confiáveis. Em [1] foram feitos estudos com essa aplicação que geraram a sua versão *evolutiva*, utilizando o modelo de execução *1PTASK* no EasyGrid AMS.

Existem duas implementações para o cálculo da força entre as partículas na aplicação N-Corpos distribuída: o algoritmo de cópia e o algoritmo *ring* [41]. No algoritmo de cópia, a cada *time step*, todas as partículas são copiadas para cada máquina utilizada. Dessa forma, esse algoritmo requer grande quantidade de memória em cada máquina, para um número grande de partículas. No algoritmo *ring* cada processador precisa armazenar apenas $N/|P|$ partículas (onde $|P|$ é o número de processadores utilizados) e, após computar as suas $N/|P|$ partículas, cada processador as envia para seu vizinho. Esses algoritmos foram projetados para ambientes homogêneos e dedicados, como *clusters*, por serem os mais utilizados atualmente e pela simplicidade de implementar algoritmos para esses ambientes.

3.2 Algoritmo ring

Neste trabalho foi utilizado o algoritmo *ring*. Seu funcionamento na versão tradicional do N-Corpos ocorre de acordo com o Algoritmo 1, considerando um conjunto de processadores[1] $P = \{p_1, p_2, ..., p_m\}$. Em cada processador p_k, é alocado apenas um processo v_k correspondente (modelo *1PPROC*). De acordo com esse algoritmo, cada processo calcula a interação de suas $N/|P|$ partículas com elas mesmas e com todas as demais, que são enviadas a ele. Esse algoritmo é, então, repetido para cada *time step*.

A Figura 3.1 representa o algoritmo *ring* executando em $|P|$=4 processadores. Pode-se perceber que o bom desempenho desse algoritmo só é possível se os processadores forem homogêneos. Em um ambiente heterogêneo o desempenho fica limitado pela velocidade do processador mais lento.

[1] Usamos o termo processador no sentido de um elemento de processamento, podendo ser um monoprocessador ou core/núcleo de um processador multicore

Algoritmo 1: *MPI_ ring(v_k)*

```
1 Calcula interação entre suas N/|P| partículas;
2 Envia suas N/|P| partículas para v_{(k+1)mod|P|};
3 para todo i ← 1 até |P| − 1 faça
4 |   Recebe N/|P| partículas envidadas por v_{(|P|+k−1)mod|P|};
5 |   Calcula força das partículas sobre suas próprias partículas;
6 |   Envia partículas recebidas para v_{(k+1)mod|P|};
7 fim
```

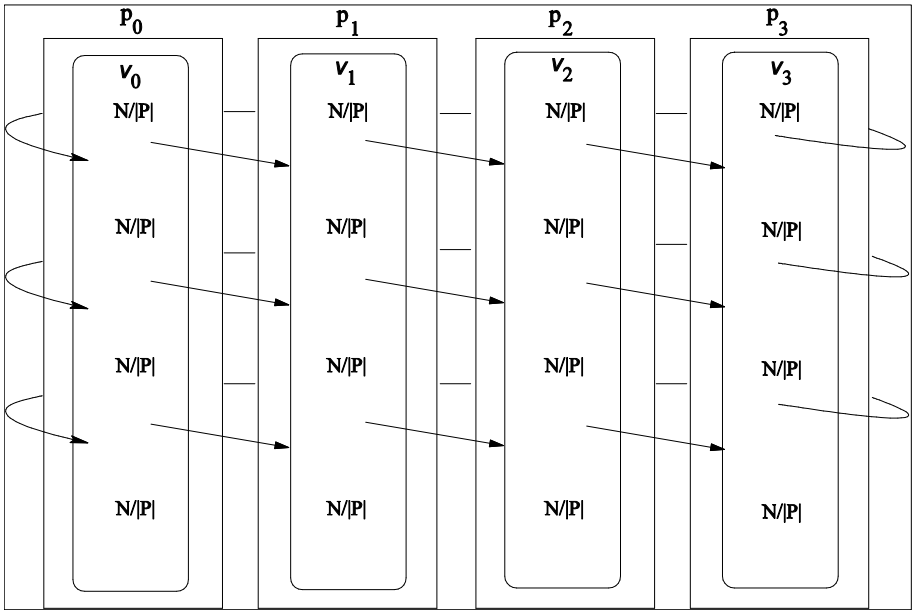


Figura 3.1: Um *time step* do algoritmo ring em 4 processadores

A versão evolutiva do N-Corpos, que implementa o modelo *1PTASK* [1], ou seja, um processo por tarefa, é mostrada no Algoritmo 2. Ao invés do grupo de N partículas ser dividido pelo número de processadores, este é dividido por uma determinada largura (W) formando W grupos de N/W partículas. Cada processo fará a interação entre dois grupos de partículas, ou seja, serão criados W^2 processos. Cada grupo de partículas inicialmente está em um dos W primeiros processos, que calculam a interação do grupo com ele mesmo e então enviam as partículas do seu grupo para dois processos distintos do nível abaixo, que irão calcular a interação entre os dois grupos recebidos e repetir o procedimento. Assim, todas as tarefas, exceto as W primeiras, terão que receber os dados dos seus dois predecessores antes de calcular a força entre suas respectivas partículas.

A Figura 3.2 representa um *time step* do algoritmo *ring* que utiliza o modelo *1PTASK* num ambiente inicialmente homogêneo. Se houverem mudanças no ambiente de execução, as tarefas que ainda não foram executadas podem ser reescalonadas para outros processadores, através do escalonamento dinâmico do EasyGrid AMS, melhorando o desempenho.

Um exemplo desse algoritmo em um ambiente heterogêneo pode ser visto na Figura 3.3, onde as tarefas foram adaptadas para executar eficientemente de acordo com a heterogeneidade do ambiente. Essa execução eficiente só foi possível porque a aplicação estava com o grau de paralelismo ideal para aqueles recursos.

Algoritmo 2: *AMS_ring_evolutivo*(v_k)

```
1  se i ≥ W então
2  |  Recebe N/W partículas enviadas por v_{i−W};
3  |  se i mod W = 0 então
4  |  |  Recebe N/W partículas enviadas por v_{i−1};
5  |  senão
6  |  |  Recebe N/W partículas enviadas por v_{i−W−1};
7  |  fim
8  fim
9  Calcula a força das partículas recebidas sobre suas próprias partículas;
10 se i < (W² − W) então
11 |  se i mod W = W − 1 então
12 |  |  Envia as N/W partículas para v_{i+1};
13 |  senão
14 |  |  Envia as N/W partículas para v_{i+W+1};
15 |  fim
16 |  Envia suas N/W partículas para v_{i+W};
17 fim
```

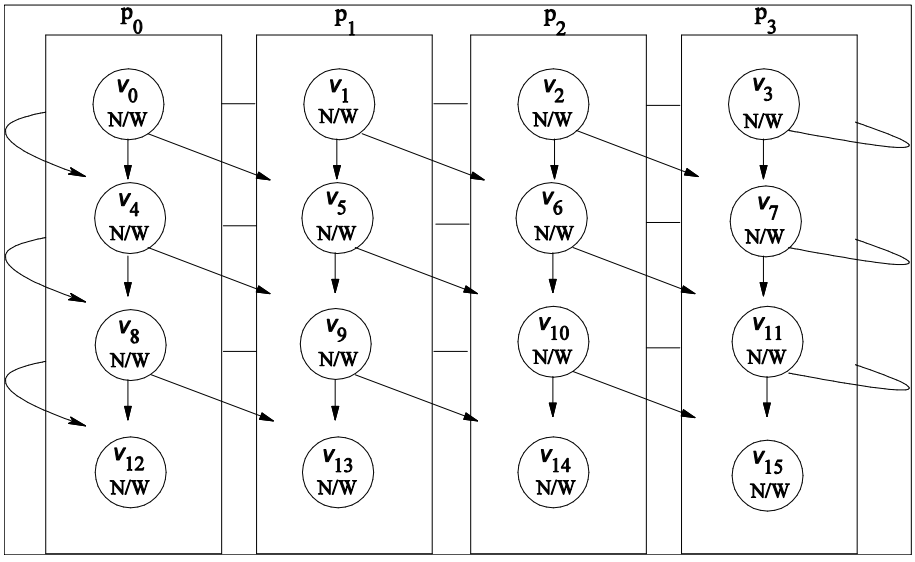


Figura 3.2: Um *time step* do algoritmo *ring* 1PTASK em 4 processadores homogêneos.

Ao final do *time step*, os últimos W processos irão enviar a informação resultante para um processo centralizador, que realizará a soma total dos resultados obtidos. A Figura 3.4 mostra como a versão evolutiva do N-corpos se comporta com dois *time steps* e W=4. É importante notar que o valor de W é fixo, definido apenas no início da execução.

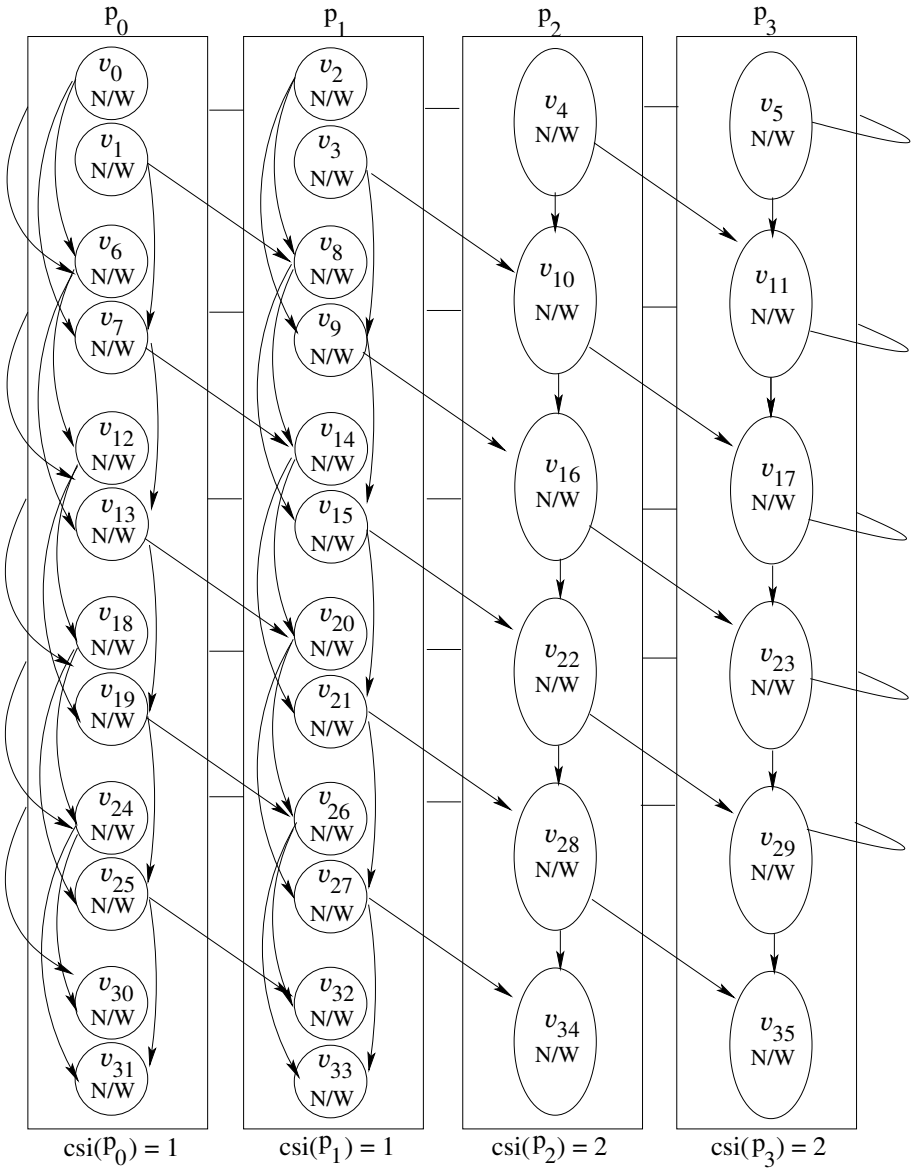


Figura 3.3: Um *time step* do algoritmo *ring* 1PTASK em 4 processadores heterogêneos.

A escolha da largura W interfere na granularidade das tarefas geradas, pois cada tarefa irá realizar $(N/W)^2$ iterações entre partículas. Com isso, a maleabilidade, para a aplicação N-Corpos, modificará o valor de W, durante a execução, de forma a obter melhor aproveitamento dos recursos disponíveis.

Neste capítulo foi apresentada a aplicação científica N-Corpos, que será o estudo de caso deste trabalho. Foi visto que o problema N-Corpos é dividido em duas abordagens, a direta e a aproximada. A abordagem direta foi escolhida por ter maior precisão e necessitar de um gerenciamento extremamente eficaz. A implementação da aplicação N-Corpos pode ser feita através de um modelo de execução de um processo por processador ($1PPROC$) ou um processo por tarefa ($1PTASK$). Essa última tem a capacidade de se adaptar a

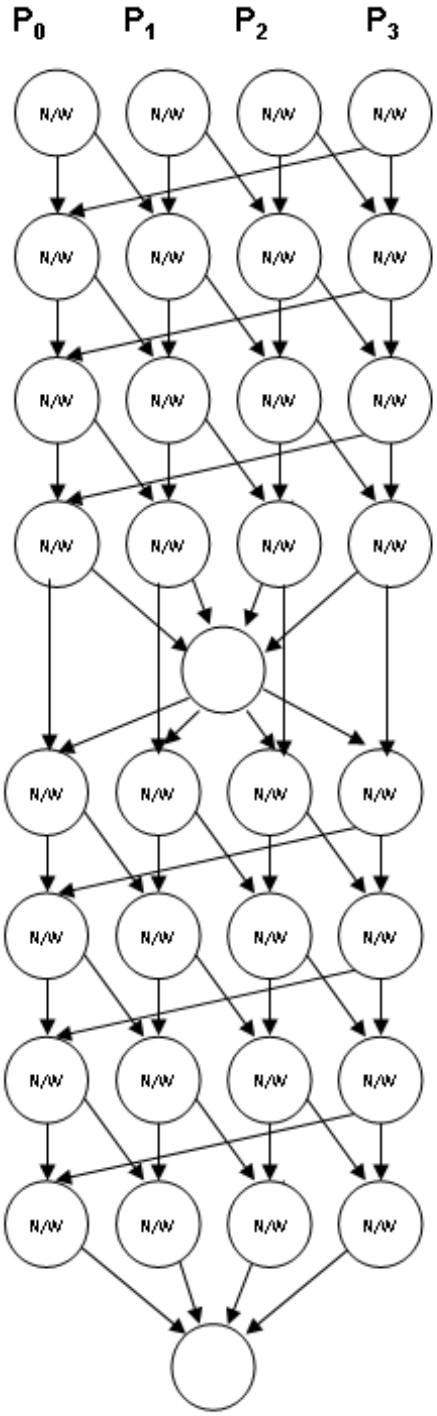


Figura 3.4: Tarefas da aplicação N-Corpos evolutiva, com W=4 e 2 *time-steps*.

mudanças no ambiente ao mover tarefas que ainda não foram executadas para máquinas que otimizem o tempo de execução. Mas só é possível ter uma execução eficiente se o grau de paralelismo for o ideal para o ambiente utilizado. A maleabilidade tem o objetivo de atender a aplicação mesmo quando o grau de paralelismo não for o ideal, ao reconfigurar a aplicação e, assim, tornar esse grau de paralelismo adequado. No próximo capítulo será

mostrado como implementar a maleabilidade utilizando o *middleware* EasyGrid AMS, e todos os ajustes necessários para uma execução eficiente.

Capítulo 4

Maleabilidade na Aplicação *N*-Corpos através do EasyGrid AMS

Nete capítulo são apresentados os passos necessários para implementar a maleabilidade através de um sistema gerenciador de aplicações, tomando como exemplo o *middleware* EasyGrid AMS. O primeiro passo foi definir o mecanismo de reconfiguração, que será explicado mais adiante. Foram estudadas heurísticas para tornar a reconfiguração mais eficiente, já que esta precisa ser executada dinamicamente e pode comprometer o desempenho da aplicação caso não ocorra rapidamente. Foi necessário ajustar o monitoramento das informações sobre o ambiente de execução, definir como será a criação dinâmica das novas tarefas e notificar a aplicação sobre a mudança na sua estrutura. Por fim, foi feito um estudo sobre melhorias no escalonamento dinâmico do EasyGrid AMS para este tipo de aplicação.

Para tornar uma aplicação maleável através do EasyGrid AMS, é necessário definir e implementar um *mecanismo de reconfiguração* eficiente, que consiste no cálculo do novo grau de paralelismo baseado nas informações sobre o ambiente e na criação da nova estrutura. Além disso deveriam ser definidos *pontos de reconfiguração*, que identificam os momentos durante a execução onde a aplicação irá avaliar a necessidade dessa mudança e ativar o mecanismo de reconfiguração, caso necessário. Numa aplicação iterativa, uma escolha natural é definir os *pontos de reconfiguração* após a sincronização que ocorre no fim de cada *time step*. Porém, se o *mecanismo de reconfiguração* for executado nesse instante, ele poderá causar um atraso na execução do próximo *time step*, aumentando então o tempo total de execução. Por outro lado, se o momento em que essa decisão for tomada ocorre muito antes, o estado dos recursos pode estar muito diferente de quando

a reconfiguração for efetivamente implementada.

Além disso, para realizar a reconfiguração da aplicação, é necessário definir como será o cálculo do novo grau de paralelismo, no mecanismo de reconfiguração. Esse cálculo deve ser feito de forma eficiente, para evitar atrasos na aplicação.

4.1 O mecanismo de reconfiguração

O mecanismo de reconfiguração calcula um novo grau de paralelismo W para o próximo *time step*. A escolha da largura W interfere na granularidade das tarefas geradas, pois cada tarefa irá realizar $(N/W)^2$ iterações entre partículas. Logo, em um ambiente dinâmico, devido às mudanças nos recursos, o W ideal nem sempre será o mesmo a cada *time step*. Estudos feitos com a aplicação puderam estimar um intervalo $[wMin, wMax]$ onde o melhor W se encontraria, dados os fatores de heterogeneidade dinâmicos das máquinas disponíveis, o número N de partículas, o custo da criação dinâmica e o custo de comunicação [1]. Nesse intervalo, $wMin$ é o número de processadores do cluster homogêneo mais rápido e $wMax = \sum_{i=1}^{N} MMC_{\forall P_k \in P}(csi(P_k))/csi(P_i)$, onde P é o conjunto de N processadores do ambiente de execução e $csi(P_i)$ é o índice de retardo computacional do processador P_i (quanto menor, mais rápida será a execução). Para encontrar o melhor W bastaria testar todos os candidatos desse intervalo, escalonando as possíveis W^2 tarefas e verificando em qual deles o escalonador retornou o menor *makespan* (tempo previsto pelo escalonamento para a execução da aplicação).

Para escalonar as tarefas foi utilizada a heurística HEFT [42], que define prioridades para cada tarefa de uma aplicação que será escalonada e, então, seleciona o processador que mais reduz o instante de conclusão de cada tarefa. Apesar dessa escolha, outros algoritmos de escalonamento de tarefas também poderiam ser utilizados [43]. Primeiramente, o HEFT tem como uma das entradas o grafo com as relações de precedência entre as tarefas, este teria que ser criado para cada valor de W testado. Esse arquivo não é necessário para essa aplicação, pois as tarefas seguem uma relação de precedência tão simples que se pode substituir a criação e leitura do arquivo por funções matemáticas que definem o sucessor e o predecessor de cada tarefa para a aplicação N-Corpos, segundo a sua lei de formação. Além disso, testar todos os possíveis valores de W pode ser inviável, pois se o ambiente for muito heterogêneo o intervalo poderá ser muito grande, com centenas ou até milhares de candidatos a melhor W. Lembrando que a cada possível W deverão ser escalonadas W^2 tarefas, tornando o cálculo muito mais custoso à medida que W aumenta.

Outro problema é que inicialmente o cálculo do escalonamento gerado por apenas um valor de W já é custoso. É necessário ainda que o tempo para o cálculo de W seja muito inferior que o tempo de execução de cada *time step*, pois senão, as informações sobre o ambiente poderão estar defasadas quando forem criadas as novas tarefas, inutilizando o escalonamento feito. Com base nisso, para tornar viável a maleabilidade da aplicação, foi necessário:

1. Substituir a leitura do grafo por funções matemáticas que indicam o sucessor e o predecessor das tarefas para a aplicação N-Corpos segundo sua lei de formação.
2. Simplificar a heurística de escalonamento, retirando cálculos que pouco influenciam na escolha do melhor W para este tipo de aplicação.
3. Criar um algoritmo que, com o intervalo de W possíveis, encontre um W próximo do ótimo testando a menor quantidade de valores possível.

Para cada uma dessas necessidades, é apresentada a solução adotada abaixo, respectivamente:

1. Após analisar o comportamento do grafo, conseguiu-se funções bem simples para uma tarefa calcular sua lista de predecessores, a lista de sucessores e o número de sucessores e predecessores. Por exemplo, para tarefas do primeiro nível, a tarefa v_0 é a única predecessora. Para qualquer outra tarefa v_i, os predecessores são $v_i - W$ e $(v_i - inic + 1 + W) \mod W + (nivel - 1) \times W + inic$, sendo $inic$ a primeira tarefa do *time step* onde se localiza v_i.
2. Para simplificar o HEFT, que possui complexidade $O(n^2 \times p)$, para n tarefas e p processadores, foram retiradas ou simplificadas algumas funções, como a função que verifica se existe um espaço no processador onde possa ser inserida uma tarefa. A retirada dessa função não afetou o escalonamento obtido devido à proporção do tempo de computação em relação ao de comunicação, pois o tempo de computação é muito maior que o de comunicação (dezenas de vezes maior), o que evita que os períodos ociosos no processador, durante a comunicação, possam ser ocupados por alguma outra tarefa. Com essa retirada foi possível acelerar o desempenho, tornando a complexidade para esse tipo de aplicação igual a $O(n \times p)$, de forma que o tempo de escalonamento quando W=150, por exemplo, caiu de 30 segundos para 0,4 segundos.

3. Para criar um algoritmo que testasse o menor número possível de valores foi necessário analisar o comportamento do *makespan* em relação ao W avaliado em determinados ambientes e realizar os procedimentos que serão mostrados a seguir.

4.1.1 Estratégias para otimizar a busca do melhor W

Nesta subseção serão mostradas e avaliadas as estratégias propostas para tornar o algoritmo de busca do melhor W mais eficiente.

4.1.1.1 Diminuir o intervalo de busca:

Baseado em experimentos foi percebido que não é eficiente executar processos muito pequenos (menores que 1 segundo, por exemplo) através do EasyGrid AMS, devido à *sobrecarga* do gerenciamento dos processos e da dificuldade de medir e prever seu comportamento [29]. Na aplicação N-Corpos, essa imprevisibilidade começou a se tornar aparente quando cada processo trabalha com menos que 4000 partículas no ambiente testado.

Então é desnecessário testar valores de W, do intervalo de busca $[wMin, wMax]$, cujo número de partículas calculado seja menor que 4000. Com isso o valor máximo (limite superior) do intervalo de busca, $wMax$, foi alterado para max($wMax$ (valor antes dessa mudança), N/k), onde N é o número de partículas e k é uma constante que depende de valores arquiteturais do ambiente, no ambiente testado fica em torno de k=4000.

4.1.1.2 Considerar heurísticas para otimização da escolha do melhor W

Na Figura 4.1 é apresentada a relação entre o valor de W(no eixo horizontal) e o *makespan* gerado (no eixo vertical) no intervalo proposto em [1], mas com o $wMax$ alterado de acordo com a política anteriormente apresentada na subseção 4.1.1.1, na simulação de um ambiente com 300 processadores heterogêneos (esses processadores não são reais, é apenas uma lista de entrada da heurística de escalonamento). O melhor W corresponde então ao valor que retornou o menor *makespan* do gráfico. Como se pode observar, o gráfico tem mínimos locais, e possui irregularidades em pontos próximos.

Pensando em minimizar as chances da busca pelo melhor W parar em algum mínimo local, mas ainda com um custo baixo, foi feito um levantamento de heurísticas de otimização não linear que poderiam ser utilizadas:

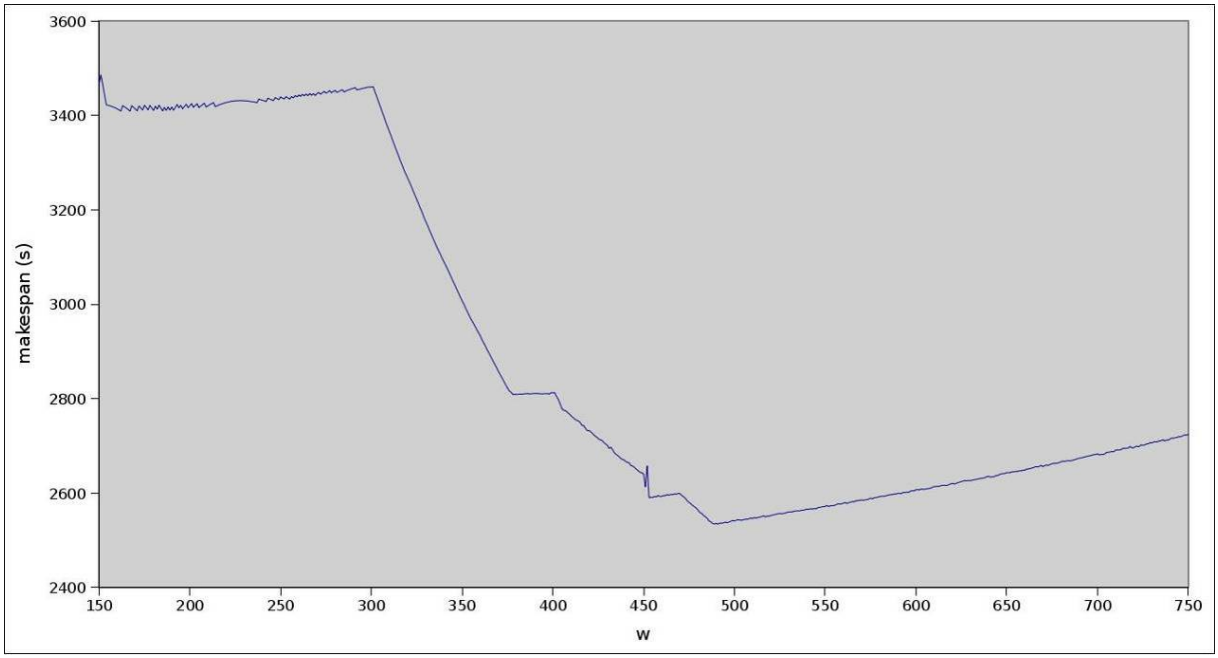


Figura 4.1: Exemplo da relação entre W e *makespan* num ambiente com 300 processadores.

1. **Métodos analíticos:** Usam técnicas de cálculo diferencial para encontrar os extremos da função. Para nossa aplicação de interesse, estes métodos são inviáveis, pois não temos a função na forma analítica, e uma tentativa de se interpolar uma função analítica iria requerer o cálculo de muitos pontos, invalidando o objetivo de minimizar o custo.

2. **Descida de encosta**[45]**:** De um modo geral, essa heurística consiste em investigar os pontos adjacentes do espaço de busca e mover-se na direção que diminua o valor da função objetivo. Ou seja, a partir de um ponto inicial (por exemplo, o menor valor do intervalo, $wMin$), calcular os próximos pontos um por um até que seja detectado um vale, que pode ser um mínimo local ou global. O cálculo de cada ponto até haver um vale, além de ser muito custoso por causa da grande quantidade de pontos avaliados, é muito susceptível a mínimos locais, devido a oscilações/irregularidades na evolução da função makespan=f(W), que serão interpretadas como vales. Uma forma de minimizar esse custo e diminuir a chance de se ficar preso em mínimos locais é, ao invés de calcular os pontos um por um, realizar saltos entre cada ponto.

   Pode-se utilizar uma variação com reinícios aleatórios para diminuir a probabilidade de a heurística parar em um mínimo local, mas o desempenho seria difícil de avaliar, pois, por reiniciar em pontos aleatórios, seriam necessários muitos experimentos para se avaliar a eficiência do algoritmo na prática.

3. **Algoritmos não determinísticos:**

(a) **Simulated Annealing**[46]**:** A diferença entre esse algoritmo e o de descida de encosta, é que este, ao encontrar um mínimo local, retrocede para escapar desse mínimo local, continuando o cálculo de mais alguns pontos. Esses retrocessos são chamados de passos indiretos. Cada passo indireto ocorre com uma probabilidade $e^{-\Delta E/T}$, onde ΔE = Valor[próximo] - Valor[atual], T é o *makespan* obtido e e é uma constante tal que $0 < e < 1$.

Por utilizar probabilidades, também seria difícil avaliar esse algoritmo, mas a idéia de calcular mais pontos após encontrar um mínimo local pode ser considerada.

Não foram considerados outros algoritmos não-determinísticos pela dificuldade de se avaliar seus desempenhos.

4. **Métodos de busca por eliminação** [47, 48]**:** São algoritmos onde o intervalo de busca é reduzido progressivamente até que o extremo (ponto de máximo ou mínimo) seja delimitado dentro da precisão desejada. Nesses métodos é feita a suposição que a função objetivo é unimodal, ou seja, que há apenas um único vale, em caso de minimização, ou um único pico, em caso de maximização. Se a função não for unimodal, o algoritmo pode convergir para um extremo local. Alguns métodos por eliminação são explicados abaixo.

   (a) **Dicotomia:** Na busca por dicotomia [48], são realizadas amostragens da função em dois pontos tão próximos quanto possível do centro do intervalo de incerteza. Esse algoritmo obtém o resultado ótimo em curvas (um único vale no intervalo em caso de minimização). A idéia é eliminar a cada passo praticamente metade do intervalo de incerteza, baseado nos valores relativos da função objetivo nos pontos amostrados e em sua unimodalidade. A desvantagem desse método, é que é necessário calcular dois pontos muito próximos, e como podem haver irregularidades nos pontos próximos, ele tenderá a retornar mínimos locais como solução.

   (b) **Método de Fibonacci** [49]: Assim como o método de dicotomia, este método propõe uma redução contínua do intervalo de incerteza a cada iteração da função, até que este intervalo de incerteza seja suficientemente pequeno. Para avaliar cada ponto, é utilizada a sequência de Fibonacci. Além de não calcular pontos tão próximos quanto no método da Dicotomia, este é mais eficiente por necessitar calcular apenas um ponto a cada iteração (Apenas na primeira iteração são efetuados dois cálculos).

(c) **Método da Seção áurea** [49]**:** Simplificação do método de Fibonacci, onde não é necessário gerar a sequência de Fibonacci, reduz sensivelmente o custo computacional do método. Baseia-se no princípio que o valor do i-ésimo elemento da sequência de fibonacci tende a ser α vezes maior que o valor do elemento anterior, conforme se aumenta o valor de i, sendo α a razão áurea=$\frac{\sqrt{5}-1}{2}$. Dessa forma, quanto maior o intervalo de incerteza, mais a eficiência do método da Seção Áurea se aproxima à do método de Fibonacci. Pela simplicidade, baixo custo e eficiência, esse algoritmo foi considerado na análise. Ele é explicado com mais detalhes no Algoritmo 3, onde os nomes das variáveis e funções estão adaptadas para o problema de busca do melhor W:

Algoritmo 3: *Seção Áurea*

```
1  A ← wMin; B ← wMax;// intervalo de incerteza inicial
2  α ← (√5−1)/2 ;
3  λ ← A + (1 − α) ∗ (B − A) ;
4  μ ← A + α ∗ (B − A) ;
5  calculaMakespan(λ) ;
6  calculaMakespan(μ) ;
7  enquanto B − A > 1 faça
8  |  se makespan(λ) > makespan(μ) então
9  |  |  A ← λ;
10 |  |  λ ← μ;
11 |  |  μ ← A + α ∗ (B − A);
12 |  |  calculaMakespan(μ);
13 |  senão
14 |  |  B ← μ;
15 |  |  μ ← λ;
16 |  |  λ ← A + (1 − α) ∗ (B − A);
17 |  |  calculaMakespan(λ);
18 |  fim
19 fim
```

No Algoritmo 3, inicialmente o intervalo de busca $[A, B]$, que representa onde o melhor W provavelmente se encontra, é equivalente ao intervalo $[wMin, wMax]$. Esse intervalo é reduzido na proporção α a cada iteraçao. Inicialmente são avaliados dois pontos dentro desse intervalo (linhas 5 e 6). Após isso, cada iteração avalia apenas um ponto, de forma que, se a função só tiver um mínimo, o algoritmo irá convergir para ele.

O problema do método da seção áurea é a possibilidade deste ficar preso em mínimos locais. Para que haja pouca probabilidade disso ocorrer foi proposto um algoritmo diferente do algoritmo da seção áurea para realizar a busca:

1. Primeiro divide-se o intervalo de busca em partições de tamanho fixo.

2. Calcula-se o makespan para o W no início de cada partição e para o W do fim do intervalo.

3. Com o melhor W calculado, refina-se o cálculo tomando como intervalo, a partição onde se encontra o melhor W e a partição da esquerda, ou seja, as duas partições em torno do melhor W.

Essa estratégia diminui a probabilidade da busca retornar um mínimo local, pelo fato de fazer uma análise global da curva antes de realizar uma busca local. A Figura 4.2 mostra quais valores de W teriam o *makespan* calculado na etapa inicial do algoritmo para a curva apresentada na Figura 4.1. Neste exemplo, o melhor W encontrado é 540, logo intervalo [540-L, 540+L] será refinado, onde L é o tamanho da partição. O valor de L está em função do intervalo de busca [wMin, wMax], de forma que o número de testes necessários seja equivalente a $log_2(wMax - wMin)$, ou seja, $\frac{(wMax-wMin)}{L} = log_2(wMax - wMin)$, logo $L = (wMax - wMin)/log_2(wMax - wMin)$. Essa medida visa aumentar a quantidade de pontos testados caso o intervalo de busca seja muito grande, mas mantém uma complexidade O(log(n)) de pontos testados (onde n é o tamanho do intervalo de busca).

O refinamento, no passo 3 do algoritmo, pode ser feito utilizando o algoritmo da seção áurea, já que a probabilidade de haver um mínimo local com *makespan* muito maior que o mínimo global torna-se bem menor com a diminuição do intervalo de busca.

Uma outra estratégia para o refinamento, que foi proposta visando diminuir a probabilidade de o algoritmo parar em mínimos locais é, ao invés de realizar a seção áurea, fazer o seguinte:

1. Considerando o intervalo [*melhorW* - L, *melhorW* + L] obtido anteriormente pela divisão inicial do intervalo de busca, divide-se o intervalo à esquerda e à direita do *melhorW* ao meio, calculando-se o *makespan* dos pontos gerados.

2. Atualiza-se *melhorW* baseado no ponto em que foi gerado o menor *makespan* e repete-se o procedimento, tomando como intervalo os pontos calculados à esquerda e à direita mais próximos do *melhorW*, até que não se possa reduzir o intervalo.

Esse algoritmo é visto com mais detalhes no Algoritmo 4. O intervalo onde se econtra o *melhorW* se torna duas vezes menor a cada iteração. Para isso é calculado o *makespan*

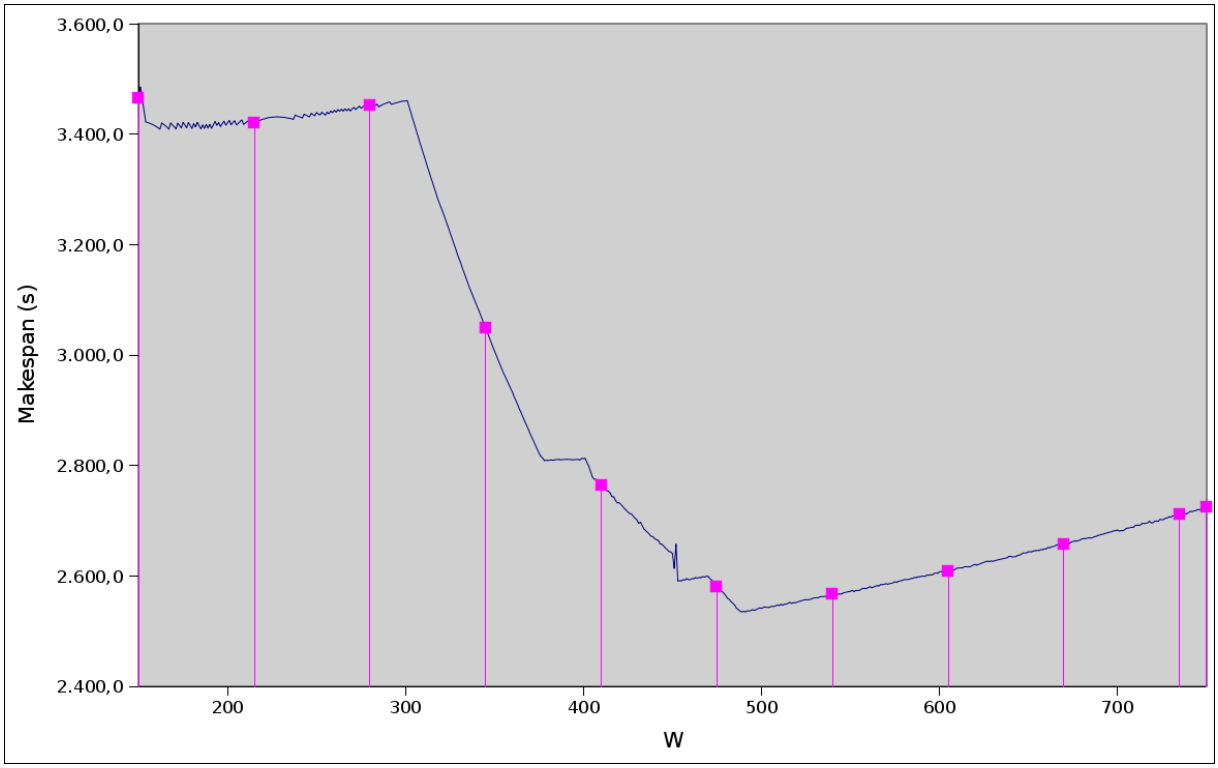


Figura 4.2: Valores de W que serão avaliados após dividir o intervalo de busca em $\log_2$(wMax-wMin) partes

de dois valores de W que estão entre o meio e os estremos desse intervalo (passos 6 e 7). Sempre que for encontrado um W melhor do que o até então $melhorW$ (passos 9 e 17), o valor de $melhorW$ será atualizado.

Após avaliar quais heurísticas são adequadas ao problema de encontrar o melhor W, foram propostos experimentos com 3 delas para analisar qual é a mais eficiente:

1. Algoritmo da seção áurea, denominado **GOLD**;
2. Algoritmo baseado na divisão do intervalo em $log_2(wMax - wMin)$ partes, e após isso, refinar o intervalo da vizinhança do melhor W utilizando o algoritmo da seção áurea, denominado **LOG_GOLD**;
3. Algoritmo semelhante ao LOG_GOLD, mas com o refinamento baseado na divisão do intervalo à esquerda e à direita do $melhorW$, denominado **LOG**;

A partir destes 3 algoritmos foram realizados experimentos envolvendo vários ambientes. Em alguns deles, percebeu-se a necessidade de limitar a quantidade de refinamentos, principalmente se o melhor W está muito afastado de $wMin$, devido ao alto custo do

Algoritmo 4: *LOG*

```
1  wInic ← melhorW − L;
2  wFim ← melhorW + L;
3  enquanto wFim − wInic > 1 faça
4  |  wMeioInic ← (wInic + melhorW)/2;
5  |  wMeioFim ← (wFim + melhorW)/2;
6  |  calculaMakespan(wMeioInic);
7  |  calculaMakespan(wMeioFim);
8  |  se makespan(wMeioInic) < makespan(wMeioFim) então
9  |  |  se makespan(wMeioInic) < makespan(melhorW) então
10 |  |  |  wFim ← melhorW;
11 |  |  |  melhorW ← wMeioInic;
12 |  |  senão
13 |  |  |  wInic ← wMeioInic;
14 |  |  |  wFim ← wMeioFim;
15 |  |  fim
16 |  senão
17 |  |  se makespan(wMeioFim) < makespan(melhorW) então
18 |  |  |  wInic ← melhorW;
19 |  |  |  melhorW ← wMeioFim;
20 |  |  senão
21 |  |  |  wInic ← wMeioInic;
22 |  |  |  wFim ← wMeioFim;
23 |  |  fim
24 |  fim
25 fim
```

cálculo e à pouca diferença entre os *makespans* de pontos próximos. Para limitar esta quantidade de refinamentos foram utilizadas duas estratégias:

1. No algoritmo LOG, ao invés de fazer o refinamento até que o intervalo tenha tamanho 1, o tamanho mínimo do intervalo foi colocado em função do *melhorW* encontrado anteriormente da seguinte forma:

   tamanhoMinimo = (melhorW- wMin) / log_2(melhorW-wMin).

   Assim, o tamanho do intervalo irá variar de *(wMax - wMin)/ log2(wMax - wMin)* até *(melhorW- wMin)/ log_2(melhorW-wMin)*. Dessa forma quanto mais próximo o *melhorW* for de *wMax* menos testes serão feitos. Esse algoritmo foi denominado **LOG2**.

2. A segunda estratégia leva em consideração a relação entre o tempo levado para calcular o *makespan* e o ganho obtido. Considerando que a diferença entre o melhor *makespan* e o *makespan* dos últimos pontos calculados tende a diminuir conforme

a distância entre os pontos diminui, antes de avaliar os próximos pontos, é verificado se essa diferença é maior que o tempo levado para avaliar. Se não for maior, significa que provavelmente não valerá a pena continuar a busca, considerando que o escalonamento calculado só será utilizado uma vez. Nos trabalhos futuros, a possibilidade de reaproveitar o escalonamento feito para mais de um *time step* poderá ser avaliada. Nesse caso, deverá haver um estudo de como essa estratégia deverá ser alterada para considerar essa possibilidade.

Seja h1 e h2 as diferenças entre o melhor makespan e os dois últimos calculados, conforme mostra a Figura 4.3, supõe-se que os próximos cálculos provavelmente não obterão um *makespan* inferior a *melhorMakespan - max(h1, h2)*. O custo de avaliar os próximos dois pontos pode ser estimado como a soma dos custos para avaliar *w1* e *w2* pelo fato de serem pontos próximos aos que serão avaliados. Dessa forma, se max(*h1, h2*) < custo(*w1*)+custo(*w2*) a busca é interrompida. No exemplo da Figura 4.3, esse critério de parada foi avaliado da seguinte forma:

$makespan1 = 2580$, $makespan2 = 2608$, $melhorMakespan = 2566$
$h1 = 2580 - 2566 = 14$
$h2 = 2608 - 2566 = 42$
$\text{custo}(w1) = 26$, $\text{custo}(w2) = 40$
$\max(h1, h2) = 42$
$\text{custo}(w1) + \text{custo}(w2) = 66$
$\max(h1, h2) < \text{custo}(w1) + \text{custo}(w2)$, logo a busca deve ser interrompida.

Essa estratégia também pode ser utilizada no algoritmo GOLD e LOG_GOLD. Os algoritmos LOG, GOLD e LOG_GOLD alterados por essa estratégia serão denominados respectivamente **LOG3**, **GOLD3** e **LOG_GOLD3**.

Os algoritmos selecionados (GOLD, LOG, LOG_GOLD, LOG2, LOG3, GOLD3, LOG_GOLD3) foram comparados em três cenários diferentes. Em cada cenário, é variado o número de processadores, mas mantendo a proporção do fator de heterogeneidade em cada processador. O fator de heterogeneidade de um processador é diretamente proporcional ao tempo de execução de um processo nele. Por exemplo, se um processo gasta t segundos para executar em um processador com fator de heterogeneidade=1, esse mesmo processo levaria $2t$ segundos para executar no processador com fator=2 (desconsiderando os custos de comunicação).

Esses três cenários são simulações, ou seja, não são processadores reais, apenas uma

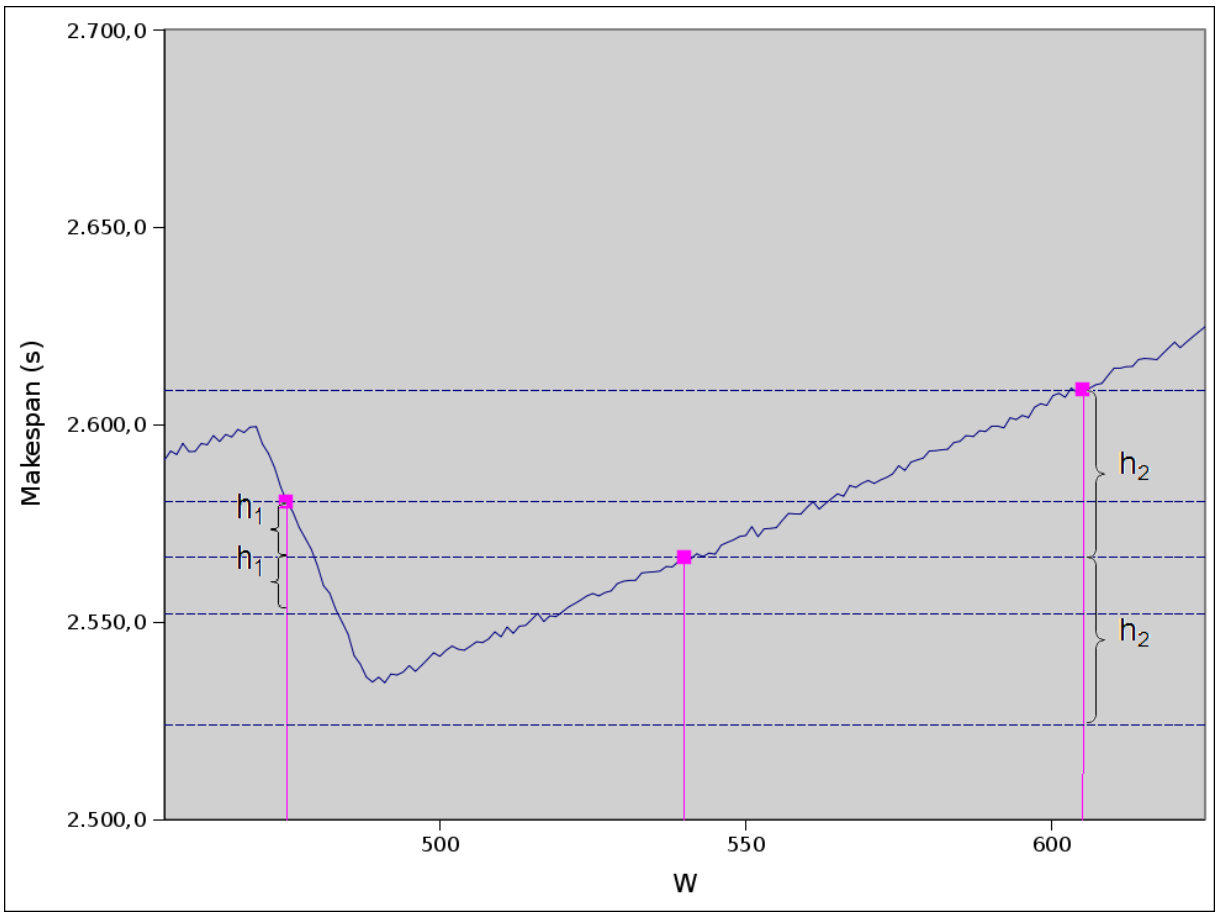


Figura 4.3: Diferença entre o melhor *makespan* e os últimos calculados (h1 e h2) para estimar um limite inferior para o menor *makespan* possível

lista de entrada para o algoritmo de escalonamento. A Sessão 5.3 mostra resultados que indicam que o algoritmo de escalonamento é preciso, isto é, o *makespan* estimado corresponde aproximadamente ao tempo de execução num ambiente real.

Sendo k o número de processadores:

1. Cenário A: k/2 processadores tem fator de heterogeneidade 1, k/4 tem fator 2, e k/4 tem fator 3
2. Cenário B: k/2 processadores tem fator 1 e k/2 tem fator 2
3. Cenário C: 2*k/3 processadores tem fator 1, e k/3 tem fator 2

Além disso, o resultado também foi comparado com a busca sequencial (denominada SEQ), que testa todos os valores do intervalo. As Figuras 4.4 a 4.6 comparam o custo de cada algoritmo e o *makespan* gerado por eles nos três cenários testados. Nesse experimento foi colocado o número de partículas igual a 10.000 vezes o número de processadores, para manter constante a relação entre o tempo de computação e de comunicação.

Para comparar as heurísticas foram analisados dois critérios:

- A soma do custo com o *makespan*: Apesar de ser possível esconder o custo da busca do melhor W, executando-o antes do fim do *time step*, o custo deve ser baixo, pois quanto mais for antecipada a busca, maior a chance de haver mudanças no ambiente antes do resultado ser utilizado. Além disso, se o custo for maior que o tempo do *time step*, não será possível ocultá-lo completamente, causando atrasos na aplicação.

  Por outro lado, se o *makespan* de uma heurística for muito maior que o de outra, mesmo que essa tenha um custo um pouco inferior, então a execução poderá ter resultados piores.

  Não é simples avaliar a relação custo $\times$ *makespan* ideal, nem o tempo ideal que o escalonamento pode ser antecipado. Isso depende das características do ambiente executado. Então, uma forma mais simples de comparar as heurísticas, considerando tanto o custo como o *makespan*, é supor que o custo do cálculo não poderá ser ocultado (ou o tempo ocultado é o mesmo para todas as heurísticas), ou seja, para cada *time step*, o tempo total de execução será estimado como a soma do custo de calcular o melhor W com o *makespan* obtido.

- O *makespan*: Se a diferença entre a soma do custo com *makespan* de duas heurísticas for pequena, o critério de desempate será o *makespan*, já que o custo poderá ser ocultado.

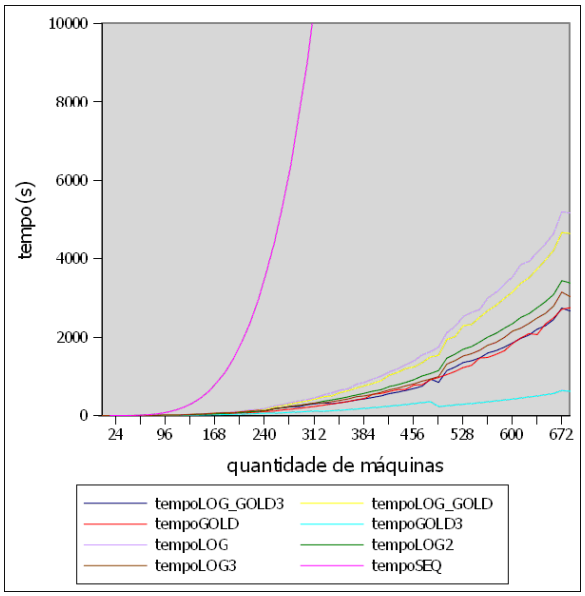
Pode-se perceber que todas as heurísticas obtiveram uma melhora bastante significativa em relação à busca sequencial. Por exemplo, na Figura 4.4(a), com 312 processadores, o algoritmo sequencial chega a ser mais do que 26 vezes mais custoso que os outros. E, mesmo assim, as outras heurísticas obtiveram o *makespan* próximo ao obtido pela busca sequencial, com uma diferença menor que 0,4 % pior.

Também nota-se que os algoritmos GOLD3, LOG_GOLD3, LOG2 e LOG3 obtiveram desempenho superior aos algoritmos GOLD, LOG_GOLD e LOG, respectivamente, e com uma diferença de *makespan* bem menor que a diferença de custo. Isso significa que as estratégias para limitar a quantidade de refinamentos obtiveram resultados satisfatórios.

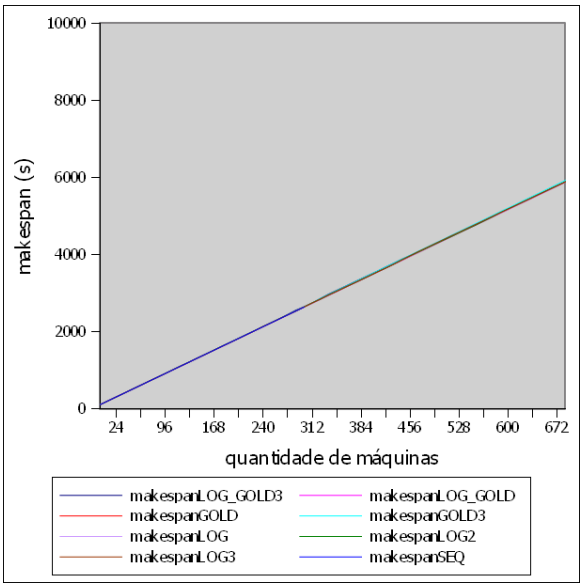
Comparando os algoritmos GOLD3, LOG_GOLD3, LOG2 e LOG3, pode-se perceber que o algoritmo GOLD3 tem um custo inferior aos outros. Por outro lado, em alguns casos ele pode obter um *makespan* maior que o gerado pelos outros devido a sua maior probabilidade de parar em mínimos locais. Por exemplo, nas Figuras 4.6(a) e 4.6(b),

pode-se perceber que a diferença entre os *makespans* do algoritmo GOLD3 em relação aos outros é maior que a diferença de custo quando o número de processadores é menor que 600. Com isso, a soma do *makespan* com o custo foi maior que a dos outros, sendo até 18% maior em alguns casos, logo ele foi menos eficiente. Também deve ser considerado que o custo da busca do melhor W pode ser atenuado se ela for feita um pouco antes do resultado ser necessário. Assim, é preferível ter um *makespan* mais preciso se a diferença de custo for baixa.

Comparando-se os algoritmos LOG_GOLD3, LOG2 e LOG3, vemos que a relação de desempenho entre eles varia muito conforme cada caso. Mas percebe-se que quando há um grande número de processadores (maior que 490), o algoritmo LOG_GOLD3 apresentou desempenho melhor ou igual aos outros dois, e sem diferença no *makespan*. Com essas considerações, o algoritmo LOG_GOLD3 foi escolhido para realizar a busca do melhor W.



(a) tempo levado para cada algoritmo calcular o melhor W

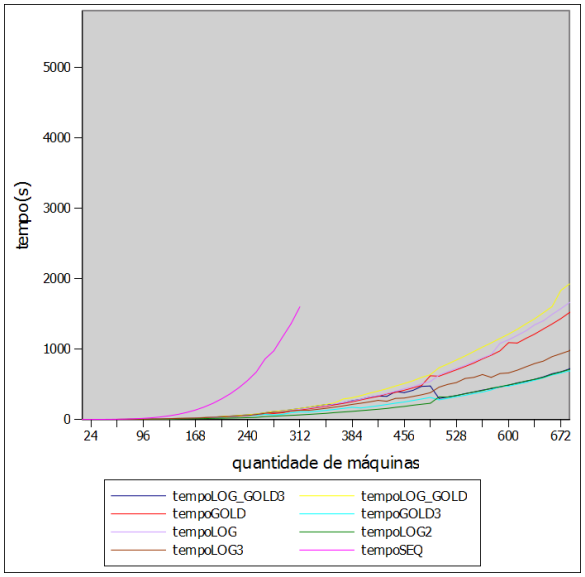


(b) *makespan* obtido por cada algoritmo ao calcular o melhor W

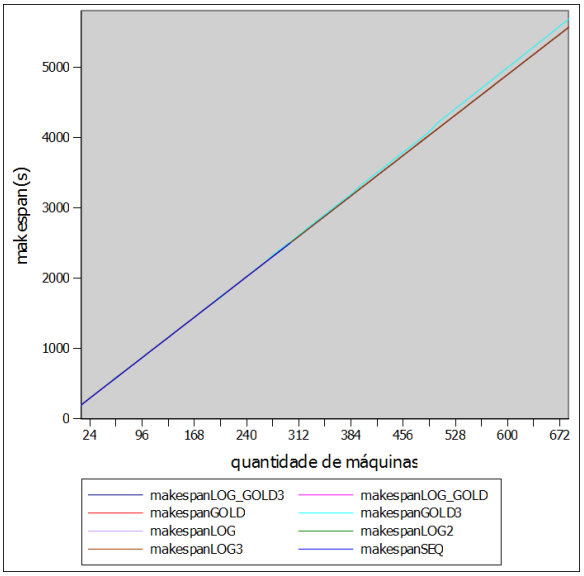
Figura 4.4: tempo e *makespan* de cada algoritmo no cenário A

Embora não seja possível garantir que o algoritmo retornará o melhor W, por não testar todos os valores e também pelo escalonamento ser uma heurística, o algoritmo retornará um valor de W com *makespan* próximo do melhor W, o que já é suficiente para aumentar a eficiência da aplicação N-Corpos. Com esse algoritmo, o tempo para calcular o melhor W depende da heterogeneidade do ambiente (quanto mais homogêneo for o ambiente, mais rápida será a execução do algoritmo).

Após obter resultados satisfatórios em relação ao tempo para o cálculo de W, o pró-

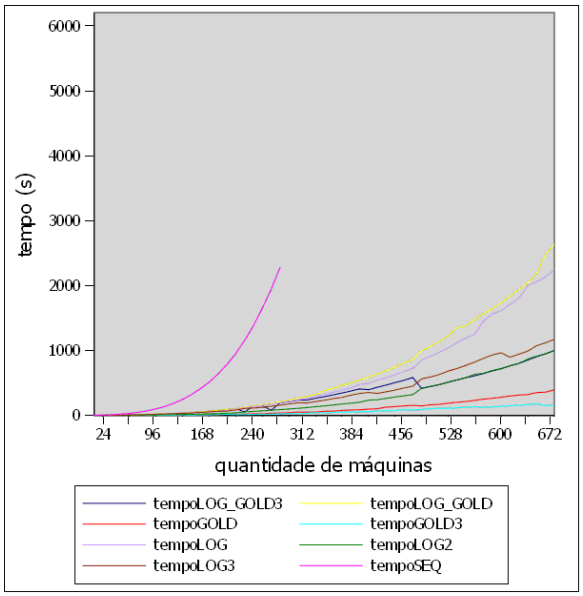


(a) tempo levado para cada algoritmo calcular o melhor W

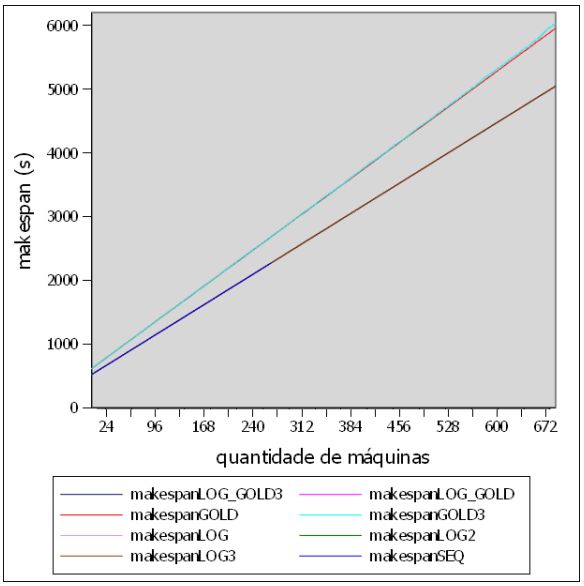


(b) *makespan* obtido por cada algoritmo ao calcular o melhor W

Figura 4.5: tempo e *makespan* de cada algoritmo no cenário B



(a) tempo levado para cada algoritmo calcular o melhor W



(b) *makespan* obtido por cada algoritmo ao calcular o melhor W

Figura 4.6: tempo e *makespan* de cada algoritmo no cenário C

ximo passo foi adaptar o N-Corpos para suportar a variação do W. Para isso, as partículas do final de um *time step* não poderiam mais ser repassadas diretamente para processos do próximo *time step* seguindo uma relação de um para um, pois o próximo *time step* poderia ter uma largura diferente do anterior. Uma forma de contornar esse problema é centralizar as partículas obtidas pelos processos do ultimo *time step* e redistribuí-las para os processos do *time step* seguinte. Para isso é necessária uma tarefa para coordenar,

agrupar e distribuir os dados a serem processados, coletando também os resultados. Foi escolhido o processo 0 para realizar essa função pois, ao contrário da versão evolutiva apresentado na Figura 3.4, onde há um processo diferente para redistribuir os dados no final de cada *time step*, utilizar apenas o processo 0 para realizar essa função simplificou a forma de implementar a maleabilidade no *EasyGrid AMS*, devido ao fato deste processo ter uma comunicação direta com o gerenciador global. Dessa forma o novo grafo do N-Corpos seria representado conforme a Figura 4.7.

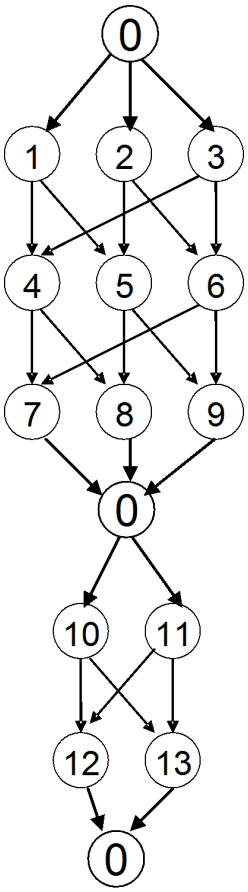


Figura 4.7: Representação de 2 *time steps* onde no primeiro *time step* W=3, no segundo W=2. As informações sobre as partículas são enviadas para o processo 0 no final de cada *time step* e que as redistribuirá de acordo com o novo W.

4.2 Implementação da maleabilidade no middleware EasyGrid AMS

Foram feitos alguns ajustes na estrutura da aplicação N-Corpos para suportar um valor de W variável. O primeiro foi a mudança do modelo de execução para *1PTASK*. Depois foi colocada uma função para o processo 0, no final do *time step*, receber o novo

W do gerenciador global e redistribuir as partículas em novos W conjuntos. Após essas modificações foi necessário modificar o gerenciador *EasyGrid AMS* para atender os seguintes requisitos:

1. Utilizar as informações obtidas sobre o estado atual das máquinas para calcular um novo W;
2. Criar dinamicamente as novas W^2 tarefas;
3. Possibilitar a comunicação do processo 0 com essas novas tarefas a fim deste enviar e receber os dados necessários;
4. Comunicar-se com o processo 0, para informá-lo do novo W, para que este possa redistribuir as partículas.

4.2.1 Monitoramento das informações sobre o ambiente de execução

O poder computacional de cada máquina é obtido pelo seu respectivo gerenciador de máquina (HMs, Host Manager) utilizando a camada de monitoramento e enviado ao Gerenciador de Site (SMs, Site Manager) (pois ele os utiliza para realizar o escalonamento dinâmico). Como é o gerenciador global (GM, Global Manager) que realiza o mecanismo de reconfiguração, estas informações devem ser enviadas de todos os SMs para ele. Para evitar sobrecarga de mensagens, esse envio é feito apenas quando necessário. Para isso, quando for mandada uma mensagem para a primeira tarefa do último nível de cada *time step*, o HM informa ao GM, por meio do SM intermediário. Então o GM faz o pedido do poder computacional para os SMs por meio de uma mensagem MPI, e estes o retornam a lista com o poder computacional de cada máquina do site. Então, O GM irá juntar essa informação e utilizar como entrada do algoritmo de cálculo da granularidade ideal.

Um detalhe relevante, é que a medição do poder computacional obtido pelo HM não funcionava adequadamente em máquinas *multicore* quando há menos tarefas executando do que o número de processadores (núcleos). O poder computacional é a porcentagem de processamento da máquina que está disponível para a a aplicação. O cálculo antigo do poder computacional era feito baseado na relação entre o tempo que o processo do AMS esteve em execução na CPU (tempo de CPU), o tempo total da duração deste (tempo de parede), o número de processos do AMS executando concorrentemente e o número de núcleos da máquina onde esse processo executou, retornando a porcentagem de CPU dessa

máquina que os processos do AMS estão utilizando. Então, por exemplo, se um processo do AMS executando numa máquina de 2 núcleos possui tempo de CPU igual ao tempo de parede, o poder computacional é medido como $\frac{numero\_de\_processos\_AMS \times tempo\_de\_CPU}{tempo\_de\_parede \times numero\_de\_nucleos} = 1/2 = 50\%$. Mas, nessa máquina, esse resultado só é verdade se houver uma carga externa no outro núcleo, caso o contrário o poder computacional seria 100%. Esse erro não causava muito impacto na maioria das aplicações que utilizam o AMS, formada por muitas tarefas independentes, pois a maior parte do tempo havia tarefas disponíveis para ocupar todos os núcleos da máquina. Em aplicações onde há relações de precedência entre as tarefas, como o N-Corpos, é mais comum que alguns núcleos fiquem ociosos em determinados instantes, devido às tarefas alocadas àquelas máquinas ainda estarem dependendo dos resultados de seus predecessores. Por isso, a medição do poder computacional nesse caso pode influenciar na escolha do grau de paralelismo apropriado.

Além disso, o escalonamento de processos nos *kernels* mais recentes de Linux dá mais prioridade para processos com menor tempo de execução [50]. Como a maleabilidade permite alterar o tamanho dos processos, o poder computacional disponibilizado a eles pode mudar, mesmo que não haja mudança no ambiente, por causa da mudança na prioridade, de forma que a medição feita antes se tornava insuficiente.

O novo método de medir o poder computacional disponível é baseado na contagem do número de cargas externas que estão executando. Para isso, foi considerado que a maioria dos processos que concorrem com o AMS ou são processos que quase não consomem CPU (tarefas do sistema operacional, por exemplo), ou processos *CPU bound* (realizam poucas operações de entrada e saída e muito processamento de dados). Não foram considerados processos que variam muito a utilização do CPU durante a execução, pois seria uma tarefa custosa e pouco praticável, exigindo conhecimento sobre o comportamento das aplicações concorrentes.

A contagem das cargas baseia-se nas informações armazenadas pelo sistema operacional sobre os processos executados recentemente. Essas informações são lidas de arquivos que o sistema operacional cria no diretório `/proc`, e os processos do AMS são diferenciados de cargas externas através do identificador do processo (PID). São ignorados os processos que utilizaram pouco poder computacional, por serem, provavelmente, processos de sistema, que executam com pouca frequência e não teriam grande influencia na execução da aplicação N-Corpos.

Ao contar quantas cargas (que não são do AMS e consomem muito a CPU) estão executando, é feita uma estimativa do poder computacional que estará disponível ao AMS.

Foi feita uma abordagem pessimista, supondo que os processos do AMS terão a mesma prioridade dos processos concorrentes, embora, na realidade, eles tenham maior prioridade na maioria dos casos, por serem processos de curta duração. Essa consideração ameniza os efeitos indesejáveis que ocorreriam caso o poder computacional real fosse menor do que o estimado, pois atrasos na execução de um processo podem afetar todo o escalonamento feito e o tempo total para executar o *time step*.

4.2.2 Criação dinâmica das novas tarefas

Após obter as informações sobre o estado do ambiente e encontrar o melhor W, os HMs serão responsáveis por criar as novas tarefas, utilizando a informação a respeito do escalonamento das novas W^2 tarefas. Para isso, primeiro o GM terá que acrescentar as novas tarefas à sua lista de tarefas (contendo informações a respeito de todas as tarefas). Depois essa lista será enviada aos SMs, que acrescentarão nas suas listas somente as tarefas que foram escalonadas em máquinas de seu respectivo site. Então os SMs enviarão sua lista para seus HMs, que irão acrescentar à suas listas de tarefas as tarefas correspondentes à máquina na qual cada HM está associado. Além disso, na lista de parâmetros de cada tarefa nova, foi colocada uma indicação do valor de W, o *time step* corrente e o número identificador da primeira tarefa desse *time step*. Pois, senão, elas não saberiam onde estariam situadas no grafo de dependências, não sabendo de quem devem receber a informação e para quem enviar.

4.2.3 Notificação sobre a mudança na estrutura

Para o processo 0 poder se comunicar com as novas tarefas criadas para redistribuir as partículas entre elas, foi necessário informá-lo quais os identificadores dessas tarefas, para que ele saiba que elas existem. Fora isso, a comunicação da tarefa 0 com qualquer outra tarefa da aplicação é feita por meio dos gerenciadores.

Para informar o novo W para o processo 0, o GM enviou uma mensagem MPI para ele. Esta mensagem será lida pela aplicação, para isso foi necessário reservar uma *tag* de comunicação MPI (utilizada para envio e recebimento de mensagens), apenas para a comunicação entre o GM e o processo 0. Essa *tag* foi necessária para que o AMS não confunda mensagens comuns da aplicação, com mensagens entre a aplicação e o GM.

4.2.4 Mudanças no Escalonamento dinâmico

Ao analisar a execução da aplicação N-Corpos maleável, percebeu-se que era necessário alterar a heurística do escalonamento dinâmico do EasyGrid AMS, pois o escalonamento dinâmico original, que é mais apropriado para aplicações formadas por tarefas independentes, conhecidas como aplicações *bag of tasks (BoT)*, tornou-se ineficiente em alguns casos. Existe uma versão do EasyGrid AMS voltada para aplicações com relações de precedência [36, 51], como a aplicação N-Corpos. Essa versão representa as relações de precedência entre as tarefas através de um grafo acíclico direcionado (*GAD*) e utiliza esse modelo de representação para realizar o escalonamento dinâmico. No entando, ela não foi utilizada por estar desatualizada, não tendo ainda os recursos mais novos do EasyGrid AMS. O escalonamento dinâmico implementado nessas duas versões do EasyGrid AMS pode ser visto com mais detalhes em [36].

Em ambas as versões do AMS as tarefas que ainda não estão sendo executadas são divididas em duas categorias: tarefas prontas e tarefas pendentes. Uma tarefa é considerada pendente quando ela ainda não obteve as informações necessárias para sua execução das suas tarefas predecessoras. Por isso, uma tarefa pendente não pode ser executada, mesmo que a máquina onde ela se encontra esteja ociosa, até que as suas tarefas predecessoras enviem os dados necessários à ela. As tarefas prontas são as que já receberam toda a informação necessária, para iniciar a execução, podendo ser executadas assim que um processador estiver disponível para a alocação delas.

Na versão *BoT* do EasyGrid AMS é feito o escalonamento dinâmico apenas de tarefas prontas. O escalonamento de tarefas prontas costuma ser bem mais simples e eficiente do que o escalonamento de tarefas pendentes, pois, como não há relações de precedência entre as tarefas, não é importante saber exatamente quais delas precisam ser reescalonadas de uma máquina para outra para reduzir o tempo de espera para dados, sendo necessário saber apenas a quantidade de tarefas que precisa ser reescalonada. Para realizar a redistribuição de tarefas, considera-se que cada tarefa tem um peso associado, correspondente ao tempo de CPU necessário para que ela termine sua execução. Daí, a redistribuição é feita calculando-se a porcentagem do peso total que a máquina mais sobrecarregada deve ceder para as menos sobrecarregadas, no caso do escalonamento dinâmico do site. O mesmo princípio se aplica na redistribuição entre os sites, onde é calculada a porcentagem de peso que o site mais sobrecarregado deve ceder para os menos sobrecarregados.

A versão *GAD* do EasyGrid AMS realiza tanto o escalonamento de tarefas prontas como o de pendentes. O escalonamento das tarefas prontas possui características similares

à heurística proposta para aplicações *BoT*. Já o de pendentes deve considerar a alocação das tarefas em cada recurso e suas precedências. O escalonador dinâmico do site armazena a informação do recurso onde está localizada cada tarefa daquele site, e assim, verifica qual o recurso que determina o *makespan* do site e tenta reduzir esse *makespan*, buscando tarefas predecessoras críticas que pertençam a outros recursos ou cedendo tarefas para recursos menos sobrecarregados. O escalonador dinâmico global apenas repassa o pedido de tarefas de um site para outro. Como o escalonamento de tarefas pendentes é mais custoso que o de tarefas prontas, ele deve ser aplicado com menor frequência.

Apesar de a aplicação N-Corpos não ser formada por tarefas independentes, a versão *BoT* do escalonamento dinâmico do EasyGrid AMS obteve resultados satisfatórios em muitos casos. Isso se deve ao fato de que a própria auto-configuração da aplicação é feita de forma a reescalonar as novas tarefas aos recursos utilizados, ou seja, a cada *time step* é realizado um escalonamento que considera as precedências entre as tarefas para calcular o novo W. E, durante o tempo entre um *time step* e o próximo, o escalonamento de tarefas prontas realiza pequenos ajustes, quando há mudanças no poder computacional. Apesar disso, existem casos em que o escalonamento de tarefas prontas não é suficiente (ou suficientemente rápido) para garantir uma execução eficiente.

A Figura 4.8 representa um dos casos em que o escalonamento dinâmico de *BoT* não é suficiente para a execução eficiente do *time step*. Considerando um ambiente de três máquinas inicialmente homogêneas (Figura 4.8.a), as tarefas do *time step* foram escalonadas durante a auto-configuração da forma mais eficiente possível. Supondo que imediatamente após começar a execução o processador p_2 passasse a disponibilizar apenas 1/3 do seu poder computacional, o escalonamento de tarefas prontas se comportaria conforme a Figura 4.8.b, não reescalonando nenhuma tarefa. Mas, como pode ser percebido na Figura 4.8.c, existe um escalonamento mais eficiente, onde, apesar de a tarefa 3 não poder ser reescalonada, pois já estava executando ao ocorrer a mudança, as tarefas 6 e 9 foram reescalonadas para uma das outras máquinas (mais rápidas), aumentando a eficiência da execução.

O resultado ineficiente do escalonamento de tarefas prontas pode ser compreendido da seguinte forma: No momento em que a tarefa 6 obtém as informações de suas tarefas predecessoras, elas terminam a execução, deixando o processador ocioso. Então, a tarefa 6 inicia a sua execução, sem permanecer na lista de tarefas prontas. Dessa forma o escalonamento dinâmico de tarefas prontas não tem tarefas para escalonar. O mesmo processo ocorre para a tarefa 9. Esse problema pode ocorrer sempre que o número de

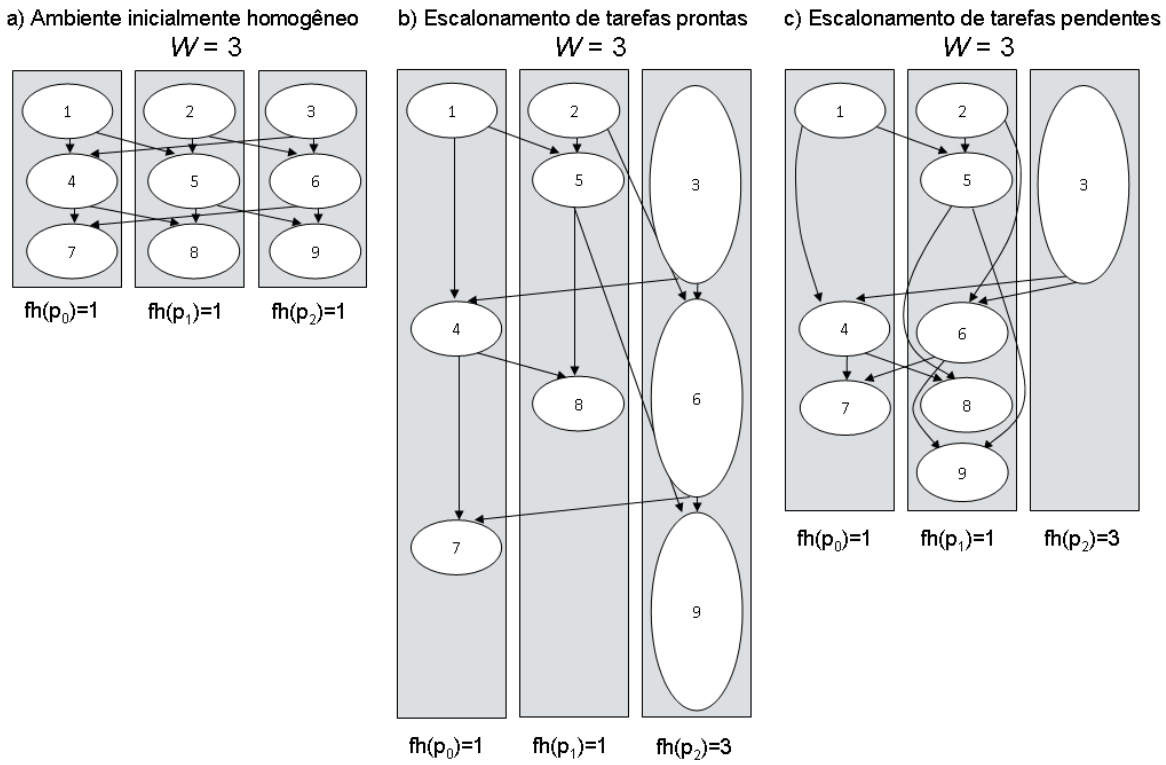


Figura 4.8: Exemplo da execução de um *time step* da aplicação N-Corpos em que o escalonamento de tarefas prontas não é suficiente para uma execução eficiente

tarefas do mesmo nível de precedência for igual ou menor ao número de núcleos na máquina mais lenta, pois sempre haverá núcleos disponíveis para as tarefas que se tornarem prontas executarem sem ser avaliadas pelo escalonamento dinâmico.

As possíveis soluções para esse problema são:

- Utilizar a versão *GAD* do EasyGrid AMS, realizando as atualizações necessárias;
- Integrar as heurísticas de escalonamento dinâmico de tarefas pendentes da versão *GAD* na versão *BoT* do AMS;
- Propor uma heurística que forneça um escalonamento dinâmico de tarefas pendentes apropriado;
- Permitir a preempção de tarefas já em execução, ou seja, interromper a execução de algumas tarefas para realocá-las em outra máquina, afim de otimizar a execução da aplicação.

Ao considerar a complexidade da heurística de escalonamento na versão *GAD* do EasyGrid e da implementação da preempção de tarefas, foi proposta uma heurística mais

simples, que visa aproveitar as características de aplicações que utilizam o algoritmo *ring*, sem que para isso seja necessário realizar grandes alterações no EasyGrid AMS. A estratégia adotada consiste em considerar que o escalonamento ótimo para um dos níveis de precedência do *time step* que está sendo executado tende a se repetir para os próximos níveis. Exemplos onde isso ocorre podem ser vistos na Figura 4.9, onde as tarefas com o mesmo nível de precedência foram representadas pela mesma cor. Assim, busca-se otimizar um único nível, e, ao reescalonar as tarefas desse nível de uma máquina para outra, reescalona-se também as tarefas dos outros níveis de forma a repetir o escalonamento feito. Para isso, essa heurística considera que as tarefas da aplicação podem ser agrupadas em *colunas*. Uma $coluna_i$ é definida como um subconjunto S_i do conjunto T das W^2 tarefas de um *time step*, onde $1 \leq i \leq W$ de tal forma que $\forall p \in S_i, \forall q \in T, p \equiv q$ se $q \mod W = i$. Dessa forma, o conjunto de *colunas* possui as seguintes propriedades:

1. Duas tarefas da mesma $coluna_i$ nunca executarão ao mesmo tempo (por causa da relação de precedência entre elas).

2. Cada $coluna_i$ tem exatamente uma tarefa de cada nível, e cada tarefa pertence a apenas uma *coluna*.

Logo, se ao reescalonar uma tarefa de uma máquina para outra também for movido todo o restante da $coluna_i$ pertencente a esta tarefa, garante-se que todos os níveis terão o mesmo escalonamento. Isso simplifica o escalonamento, pois, como as tarefas do mesmo nível são independentes entre si, pode-se escalonar as *colunas* sem precisar considerar relações de precedência.

Por enquanto esse escalonamento baseado em *colunas* foi desenvolvido apenas no escalonador associado ao gerenciador de site. O ideal seria que houvesse também um escalonamento entre sites, no nível do gerenciador global. Mas, como a distância entre sites pode ser muito grande, aumentando o custo do envio de tarefas, o escalonamento global deverá ser realizado com menor frequência, de modo que a própria reconfiguração que ocorre a cada *time step* pode já ser suficiente, já que, ao ser efetuada a reconfiguração da aplicação, ocorre um escalonamento de tarefas a nível global. Como trabalho futuro poderá ser feito um estudo melhor da necessidade de um escalonamento dinâmico a nível global mais frequente.

O escalonamento dinâmico de site adotado baseia-se no número de *colunas* restantes em cada HM. O gerenciador de site recebe essa informação dos HMs através de mensagens de *heartbeats*, enviadas de tempos em tempos ou quando uma tarefa termina a execução.

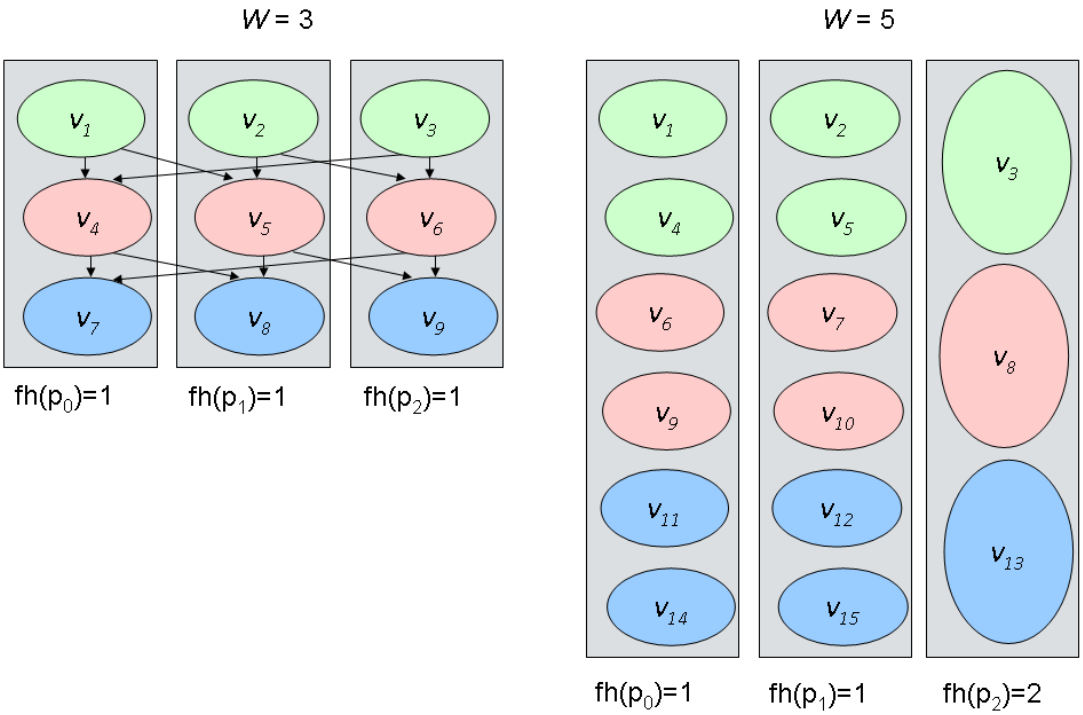


Figura 4.9: Exemplos de execuções da aplicação N-Corpos onde o escalonamento ótimo de um nível da aplicação se repete para os outros níveis

Após isso, ele verifica se será possível melhorar o *makespan* retirando algumas *colunas* da máquina em que a aplicação levará o maior tempo para executar todas as *colunas* associadas a ela, e colocando-as em outras máquinas menos carregadas. Se a diminuição do *makespan* for maior do que uma faixa de tolerância α, o algoritmo de balanceamento será ativado. Este retornará o número de *colunas* que devem ser cedidas e para quais máquinas elas deverão ser enviadas. Após isso, o SM irá pedir as *colunas* da HM mais sobrecarregada, receber as tarefas das *colunas* pedidas, e enviar para as máquinas determinadas a receber estas *colunas*. Apesar de nem sempre ser possível balancear todo o site em um único escalonamento, esse procedimento tende a melhorar o balanceamento iterativamente, até que não seja mais possível verificar uma melhoria no *makespan*.

O tempo que uma máquina, com a porcentagem de poder computacional *cPower* (calculado conforme a seção 4.2.1) e número de núcleos *nCores*, levará para executar o restante de suas colunas é obtido segundo o Algoritmo 5. O retorno do algoritmo não está em segundos, mas em uma unidade de tempo correspondente ao período de tempo que uma única *coluna* leva para executar em uma máquina com poder computacional 100%.

Caso haja *colunas* em quantidade menor que o número de núcleos, o tempo de exe-

cução será o mesmo que o necessário para executar uma única *coluna*, pois os processos da *coluna* serão executados cada um por um núcleo, sem concorrerem entre si. Caso contrário, os processos das *colunas* irão se intercalar de forma que, embora os processos não concorram entre si, devido ao limite de processos disparados ao mesmo tempo pelo EasyGrid AMS, processos do próximo nível poderão adiantar a execução, que tenderá a ter o tempo em função da média de *colunas* por núcleo.

Algoritmo 5: estima_tempo_restante($cPower, nCores, numColunas$)

```
1 se numColunas > nCores então
2 |  retorna (numColunas/nCores) * (100/cPower);
3 senão
4 |  se numColunas = 0 então
5 |  |  retorna 0;
6 |  senão
7 |  |  retorna 100/cPower;
8 |  fim
9 fim
```

Para verificar se é possível melhorar o *makespan*, o escalonador de site verifica quantas *colunas* a máquina mais sobrecarregada, *maiorProc* , deve ceder, no mínimo, para diminuir seu tempo restante, e quantas *colunas* as outras máquinas podem receber, no máximo, para que o *makespan* não se torne maior que o atual. Se a quantidade de *colunas* que devem ser cedidas for menor ou igual a soma da quantidade máxima de *colunas* que podem ser recebidas pelas outras máquinas, então o site é considerado desbalanceado.

O balanceamento se comporta conforme o Algoritmo 6, que verifica quantas *colunas* devem ser retiradas da máquina que determina o *makespan*, *maiorProc*, e calcula como deve ser a distribuição dessas colunas para as outras máquinas. Após a execução desse algoritmo, quando o SM receber as *colunas* pedidas, a distribuição de *colunas* será feita a partir da lista *maqsCandidatas*, que possuirá a quantidade de *colunas* que cada máquina deverá receber.

Ao comparar o novo escalonamento dinâmico com o *BoT*, em alguns casos o escalonamento *BoT* obteve melhor desempenho, e em outros pior. Na maioria das situações o escalonamento *BoT* é o mais recomendado, principalmente se não houve mudanças significantes no poder computacional desde a reconfiguração. Pois o reescalonamento das tarefas pendentes que ocorrem no escalonamento de *colunas* invalida o escalonamento ocorrido na reconfiguração, que tinha sido feito de forma precisa ao considerar todas as dependências, e substituindo-o por uma heurística mais simples e imprecisa. A figura 4.10 apresenta o tempo de execução de cada *time step* da aplicação N-Corpos maleável utili-

Algoritmo 6: algoritmo de balanceamento

```
1  maqsCandidatas ← lista com todas as máquinas candidatas a receber colunas,
   ordenada de forma decrescente pela quantidade máxima de colunas que cada
   máquina pode receber;
2  tempoMaq ← estima_tempo_restante(maqsCandidatas[0].cPower,
   maqsCandidatas[0].nCores, maqsCandidatas[0].numColunas + 1);
   // O tempo que a primeira máquina da lista levará para executar suas
      colunas, após uma coluna ser doada a ela
3  maxPodeCeder ← número máximo de colunas que maiorProc pode ceder, ou
   seja, somatório do número máximo de colunas que cada máquina em
   maqsCandidatas pode receber sem aumentar o makespan;
4  maxPodeCeder ← max(maxPodeCeder, maiorProc.numColunas);
5  numCedidas ← 0;
6  enquanto tempoMaq < makespan e numCedidas < maxPodeCeder faça
7  |  maqsCandidatas[0].maxRecebe ← maqsCandidatas[0].maxRecebe − 1;
   |  // Decrementa o número de colunas que maqsCandidatas[0] pode
   |     receber
8  |  maqsCandidatas[0].recebidas ← maqsCandidatas[0].recebidas + 1;
   |  // Incrementa o número de colunas que serão recebidas por
   |     maqsCandidatas[0]
9  |  maiorProc.numColunas ← maiorProc.numColunas − 1;
10 |  numCedidas ← numCedidas + 1;
11 |  Atualiza o makespan;
12 |  Reordena maqsCandidatas;
13 |  Recalcula tempoMaq;
14 fim
15 Faz um pedido de numCedidas colunas ao HM de maiorProc;
```

zando o escalonamento dinâmico de *colunas* e de *BoT* em um ambiente de 3 máquinas, com 12 núcleos cada uma. Foram realizadas duas execuções para cada opção de escalonamento. Nesse ambiente pode ser percebida a vantagem do escalonamento de *BoT*, que teve um desempenho 5% melhor que o de *colunas*.

Os casos onde o escalonamento de *colunas* apresenta melhor resultados são aqueles em que o poder computacional de pelo menos uma das máquinas com a quantidade de tarefas prontas menor ou igual ao número de núcleos, diminui consideravelmente (por exemplo, tornando-se mais do que 50% mais lenta). Nesse caso, retirar tarefas dessa máquina que se tornou lenta pode aumentar o desempenho. O escalonamento de *colunas* irá retirá-las, mas o escalonamento *BoT* não irá por não escalonar tarefas pendentes. A figura 4.11 apresenta novamente a comparação entre ambos os escalonamentos em um ambiente de 3 máquinas, com 12 núcleos cada, inicialmente homogêneo, mas que, durante a execução do sexto *time step*, uma das máquinas se torna três vezes mais lenta. Nesta figura pode-se

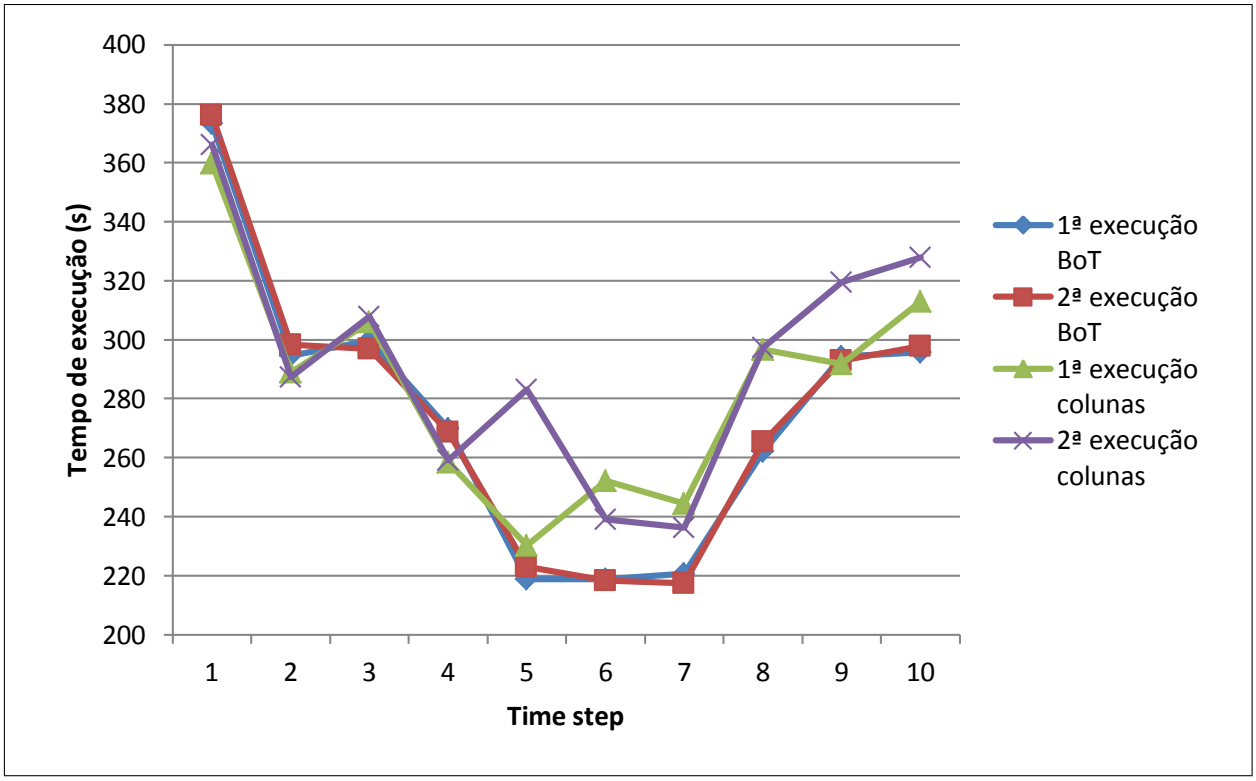


Figura 4.10: Comparação das heurísticas de escalonamento *BoT* e de *colunas* em um ambiente dinâmico

notar que o escalonamento *BoT* não conseguiu reescalonar nenhuma tarefa da máquina mais lenta, pelo fato de não haver tarefas prontas que não estão executando.

O ideal é uma solução híbrida, onde nos casos em que for detectado um desbalanceamento no sistema, e o escalonamento *BoT* não conseguir melhorá-lo, então o escalonamento de colunas poderá ser ativado temporariamente. Como trabalho futuro, essa solução híbrida pode ser melhor avaliada. Enquanto isso o escalonamento *BoT* foi utilizado na análise experimental, por obter resultados satisfatórios na maioria dos casos.

Nesse capítulo foram realizados todos os passos necessários para tornar a aplicação N-Corpos maleável através do gerenciador EasyGrid AMS: definir o mecanismo de reconfiguração e os pontos de reconfiguração; otimizar a busca do melhor grau de paralelismo, através da simplificação da heurística de escalonamento, da diminuição do intervalo de busca e do estudo de heurísticas de otimização da busca; utilizar o monitoramento das informações sobre o ambiente de execução como entrada do mecanismo de reconfiguração; configurar o AMS para criar dinamicamente as novas tarefas a cada *time step* e notificar a aplicação sobre as mudanças na sua estrutura. Após esses ajustes a aplicação ficou pronta para executar e realizar testes, o que será mostrado no próximo capítulo.

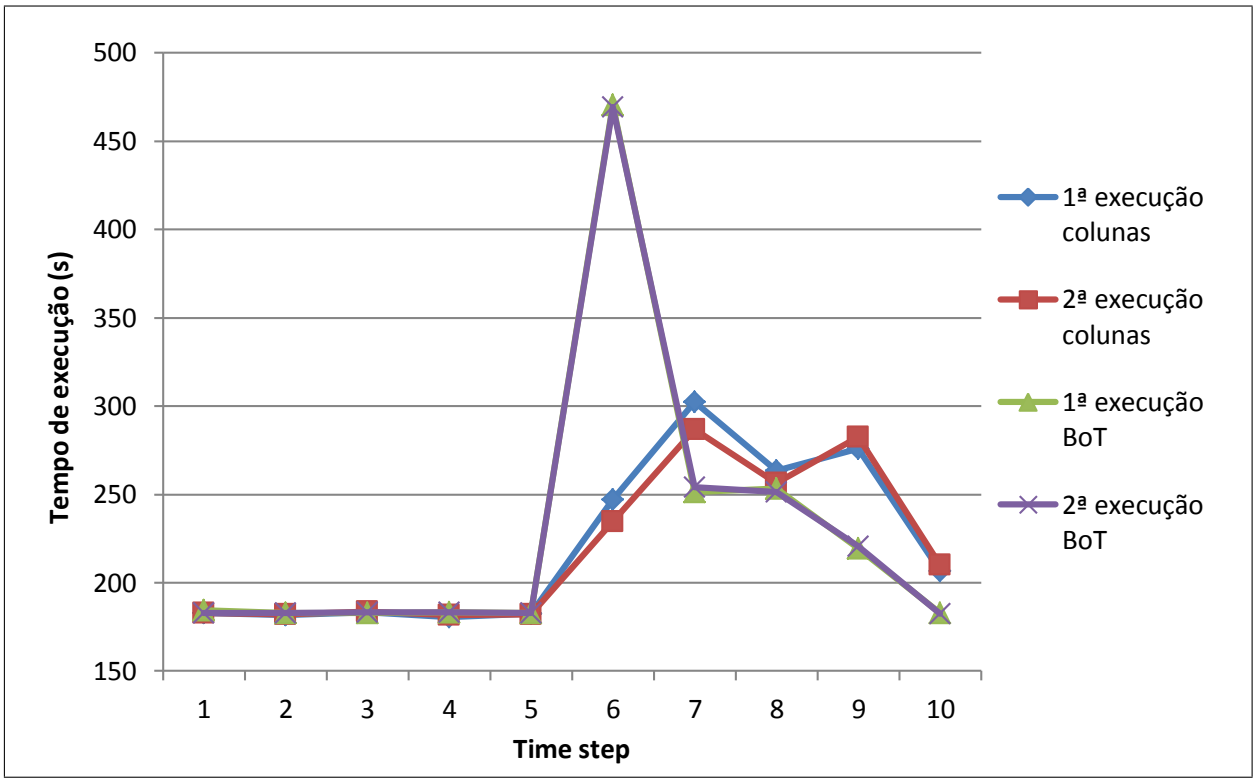


Figura 4.11: Comparação das heurísticas de escalonamento *BoT* e de *colunas* em um ambiente dinâmico que prevalece o escalonamento de *colunas*

Capítulo 5

Análise Experimental

Para analisar o tempo de execução do N-Corpos AMS maleável, utilizando o modelo *1PTASK*, em comparação com o N-Corpos original MPI sem maleabilidade, utilizando o modelo *1PPROC*, foram feitos experimentos em ambientes reais. Os resultados apresentados neste capítulo foram obtidos utilizando um site com 3 máquinas (*n1*, *n2* e *n3*), dual-processados com Intel(R) Xeon(R) CPU X5650 @ 2.67GHz, totalizando em cada máquina 12 núcleos, 24 GB de memória e o sistema operacional CentOS 6.0 64 bits, *kernel* 2.6.32. Para o AMS foi utlizada uma quarta máquina para os gerenciadores global e do site.

5.1 O Custo da solução proposta

Inicialmente foi realizado um teste com recursos homogêneos, ou seja, sem a adição de carga adicional em qualquer das máquinas, com o objetivo de avaliar a sobrecarga do uso do AMS maleável. Os resultados obtidos podem ser vistos na Tabela 5.1. A coluna MPI apresenta o tempo de execução em segundos da aplicação N-Corpos MPI moldável, enquanto que a coluna AMS apresenta o tempo de execução da aplicação N-Corpos maleável e a coluna Ovrhd. apresenta a porcentagem de sobrecarga do tempo de execução da versão maleável em relação ao da moldável. Os resultados foram obtidos através da média aritmética de duas execuções e a diferença entre os tempos das duas execuções foi menor que 1%.

A Figura 5.1 exibe a porcentagem de sobrecarga média em relação ao número de partículas. Por não haver heterogeneidade no sistema, já era esperado o desempenho

	150mil			200mil			250mil		
Nº de *time steps*	MPI	AMS	Ovrhd.	MPI	AMS	Ovrhd.	MPI	AMS	Ovrhd.
1	23,50	29,60	25,9%	41,44	47,20	13,9%	64,81	72,78	12,3%
2	46,86	57,67	23,1%	82,96	94,60	14,0%	129,38	145,51	12,5%
3	70,18	88,27	25,8%	124,48	141,85	13,9%	195,47	218,07	11,6%
4	93,62	116,27	24,2%	165,75	188,80	13,9%	259,83	290,77	11,9%
5	116,97	146,04	24,9%	207,95	235,83	13,4%	324,54	363,59	12,0%
6	140,34	174,32	24,2%	248,60	283,11	13,9%	389,04	436,31	12,1%
7	163,60	209,39	28,0%	291,14	330,50	13,5%	454,84	509,11	11,9%
8	187,43	234,43	25,1%	331,55	377,33	13,8%	520,92	581,53	11,6%
9	210,46	264,50	25,7%	373,28	424,43	13,7%	583,48	654,91	12,2%
10	233,72	290,16	24,2%	414,90	471,96	13,8%	649,64	725,89	11,7%

	300mil			350mil		
Nº de *time steps*	MPI	AMS	Ovrhd.	MPI	AMS	Ovrhd.
1	93,50	103,87	11,1%	129,61	140,70	8,6%
2	187,01	208,00	11,2%	259,01	282,14	8,9%
3	279,51	311,80	11,6%	387,32	422,44	9,1%
4	373,15	415,40	11,3%	515,88	562,86	9,1%
5	466,30	520,05	11,5%	644,31	703,71	9,2%
6	560,19	623,13	11,2%	773,55	844,05	9,1%
7	652,83	728,07	11,5%	902,13	985,12	9,2%
8	749,31	831,54	11,0%	1.029,69	1.125,48	9,3%
9	837,46	935,12	11,7%	1.162,14	1.268,01	9,1%
10	934,99	1.038,59	11,1%	1.287,97	1.407,23	9,3%

Tabela 5.1: Relação entre o tempo (segundos) do N-Corpos original (MPI) e do maleável (AMS) e a sobrecarga do maleável em relação ao original.

superior do N-Corpos MPI. [1] Contudo o custo da solução proposta diminui à medida que o tamanho do problema aumenta, e considerando que as simulações utilizarão centenas de

[1] Anteriormente os testes para descobrir o custo da solução tinham sido feitos em outro ambiente, com máquinas mais antigas. No ambiente antigo a sobrecarga da execução foi extremamente baixa (menor que 1% para 300.000 partículas). Após isso, foram feitas melhorias no desempenho do algoritmo que calcula o grau de paralelismo ideal e, por isso, os testes de sobrecarga precisaram ser refeitos. Contudo, devido a problemas de aquecimento no laboratório, as máquinas antigas apresentaram problemas de redução da velocidade dos processadores, atrapalhando a execução de novos testes. Como não era possível utilizar as mesmas máquinas, foram disponibilizadas novas máquinas, que apresentaram uma sobrecarga maior no EasyGrid AMS. Foi verificado que, com a mesma versão do EasyGrid, havia mais sobrecarga nas máquinas novas do que nas antigas. Por exemplo, em três execuções da aplicação N-Corpos, utilizando apenas uma máquina para as tarefas da aplicação, e número de partículas em torno de 100.000 (processos com duração de aproximadamente 8 segundos), a execução no ambiente novo apresentou sobrecarga de aproximadamente 7%, enquanto que no ambiente antigo a sobrecarga foi em torno de 0,1%. Por não ser o foco desse trabalho, deixa-se como trabalho futuro a análise dessa perda de desempenho no sistema mais recente. Mais detalhes sobre os resultados antigos, máquinas e sistema operacional utilizados podem ser vistos no Apêndice A.

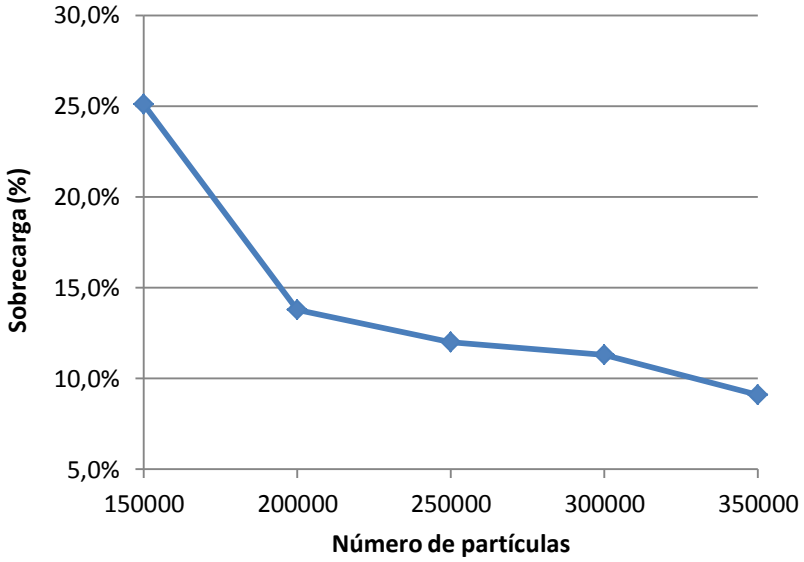


Figura 5.1: Porcentagem de sobrecarga de execução do N-Corpos maleável em relação ao N-Corpos estático (em um ambiente homogêneo) em função do número de partículas.

milhares de partículas ou mais, a perda na eficiência não será muito significativa. Também é importante observar que a sobrecarga se mantém praticamente constante em relação ao número de *time steps* executados, o que possibilita a execução de centenas ou milhares de *time steps* sem que a sobrecarga aumente.

5.2 Ambiente heterogêneo e dinâmico

Para simular um ambiente heterogêneo e dinâmico, foi utilizado um programa que utiliza CPU constantemente, de forma a alterar o poder computacional disponível à aplicação N-Corpos em função do tempo, e foram utilizadas as mesmas 3 máquinas com 12 núcleos do experimento anterior. Primeiramente, foi colocada uma carga em cada núcleo das máquinas *n1* e *n2* (24 núcleos no total) com o objetivo de cada núcleo de *n1* e *n2* disponibilizasse apenas 50% de processamento para a aplicação N-Corpos. Após o instante $t1$=1000 segundos retira-se a carga dos núcleos de *n1* e após o instante $t2$=2000 segundos (a partir do início da execução) coloca-se carga nos núcleos de *n3*. Na Figura 5.2 pode-se perceber a porcentagem de poder computacional das máquinas *n1*, *n2* e *n3* em função do tempo em segundos.

Foram realizados 8 experimentos (A, B, C, D, E, F, G e H), todos com 400 mil partículas e 10 *time steps*. A seguir são descritas as características de cada um dos experimentos:

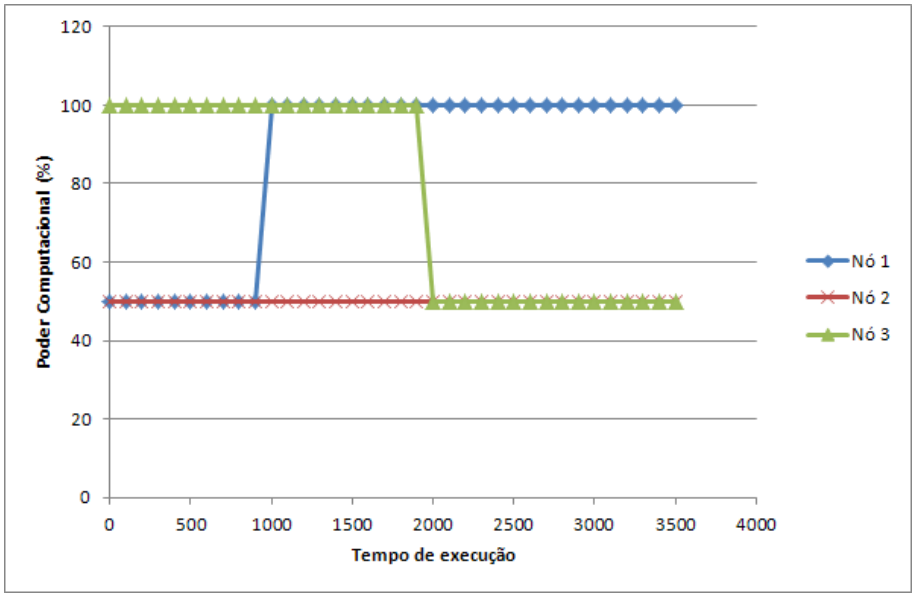


Figura 5.2: Poder computacional dos nós n1, n2 e n3 durante a execução da aplicação

1. Experimentos A e B - Aplicação moldável
2. Experimentos C e D - Aplicação evolutiva
3. Experimentos E e F - Aplicação maleável
4. Experimentos G e H - Aplicação evolutiva e maleável

O objetivo deste experimento é mostrar as vantagens do escalonamento dinâmico combinado à maleabilidade em uma aplicação fortemente acoplada em um ambiente dinâmico tanto em casos onde não há conhecimento da heterogeneidade inicial do ambiente como nos casos onde há esse conhecimento.

Os experimentos (A, C, E e G) foram realizados considerando-se que o usuário não tem conhecimento sobre a heterogeneidade inicial do ambiente, logo o primeiro *time step* possui a largura w=36 (como se o ambiente fosse homogêneo). Os experimentos (B, D, F e H) começam com a configuração mais adequada para a heterogeneidade inicial do sistema, ou seja, w=48 (com mais tarefas em *n3* por este possuir maior poder computacional). Para os experimentos C, D, G e H, o escalonamento dinâmico foi calibrado para a cada 1 segundo verificar se alguma tarefa precisa ser reescalonada. O tempo em segundos de cada *time step* e o tempo total de cada experimento é exibido na Tabela 5.2.

A aplicação maleável e a aplicação evolutiva e maleável apresentaram os melhores resultados. Para maior entendimento dos resultados, a Tabela 5.2 foi dividida em dois gráficos. O primeiro gráfico (Figura 5.3) representa os tempos obtidos nos experimentos

ts	A	B	C	D	E	F	G	H
1	364,5	280,8	362,9	294,6	367,1	280,9	367,4	290,6
2	366,8	282,8	357,1	292,7	280,4	282,0	280,5	291,7
3	358,9	281,1	318,1	295,1	281,6	282,8	280,2	292,7
4	357,1	286,1	253,8	287,1	285,2	287,5	277,6	290,3
5	357,7	284,2	253,0	247,0	219,5	219,4	216,2	219,3
6	356,8	283,4	254,4	247,5	218,6	218,6	216,2	217,6
7	366,0	283,4	284,8	245,2	219,7	219,9	216,8	215,8
8	366,6	514,4	352,6	302,9	307,6	221,3	241,3	231,8
9	365,5	537,5	354,6	293,6	285,6	360,5	284,2	294,1
10	366,7	533,9	355,1	291,4	283,4	284,7	284,1	292,6
TT	3626,8	3567,7	3146,4	2797,2	2748,8	2657,6	2664,3	2636,5

Tabela 5.2: Tempo de execução de cada experimento

A, C, E e G (não há conhecimento prévio sobre a heterogeneidade do ambiente), e o segundo (Figura 5.4) representa os tempos obtidos nos experimentos B, D, F e H (aplicação ajustada inicialmente à heterogeneidade do ambiente).

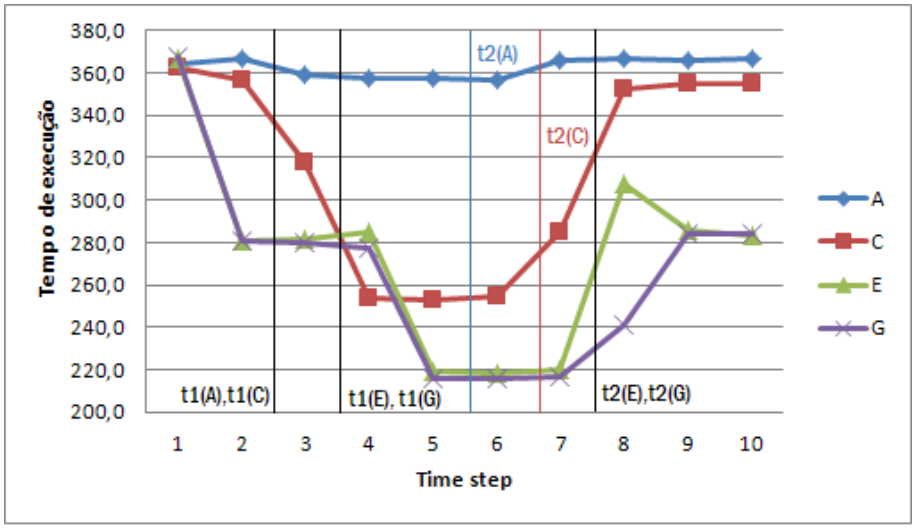


Figura 5.3: Tempo de execução de cada *time step* dos experimentos A,C,E e G - Usuário não tem conhecimento da heterogeneidade inicial.

Devido aos *time steps* terem tempos diferentes dependendo do experimento, as mudanças no ambiente são percebidas em tempos diferentes, ou até mesmo em *time steps* diferentes. A linha vertical no gráfico representa o momento em que ocorrem as mudanças no ambiente de acordo com o experimento feito. Por exemplo, o instante *t2* no experimento C ocorreu após a execução do *time step* 6, próximo ao término do *time step* 7, enquanto que no experimento G, ocorreu após a execução do *time step* 7.

Para melhor analisar o comportamento dos processos em função do tempo em um

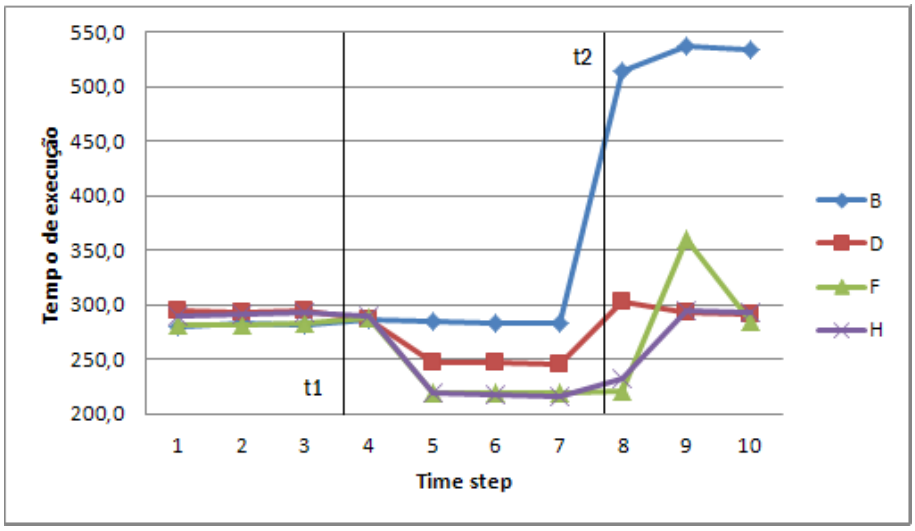


Figura 5.4: Tempo de execução de cada *time step* dos experimentos B, D, F e H.

ambiente heterogêneo, foi desenvolvida uma aplicação Java para gerar um diagrama de Gantt na qual, a partir do tempo de parede e do tempo de início de cada processo é possível visualizar cada processo em sua respectiva máquina. Os processos são representados por retângulos de forma que o tempo de início equivale à distância do retângulo ao topo do gráfico e o tempo de parede corresponde à altura do retângulo que representa a tarefa. Além disso, tarefas do mesmo nível de precedência foram representadas pela mesma cor.

5.2.1 Experimentos A e B:

Como era de se esperar, os experimentos A e B tiveram o pior desempenho por não se adaptarem às mudanças ocorridas no ambiente.

5.2.1.1 Experimento A:

Não mostrou mudanças no tempo de execução com as mudanças no ambiente, pois, como o número de tarefas é o mesmo em todas as máquinas, o tempo de execução tornou-se equivalente ao tempo que levaria se todas as máquinas estivessem com o menor poder computacional entre elas.

Utilizando diagramas de Gantt gerado pela ferramenta desenvolvida neste trabalho, pode-se perceber, nas figuras 5.5 e 5.6, que representam um dos *time steps* do experimento A no instante anterior a *t1* e entre *t1* e *t2* respectivamente, que o tempo de execução foi ditado pelas máquinas com o menor poder computacional ($n1$ e $n2$, na Figura 5.5 e $n2$, na Figura 5.6), e as máquinas com o melhor poder computacional ficaram boa parte do

tempo ociosas.

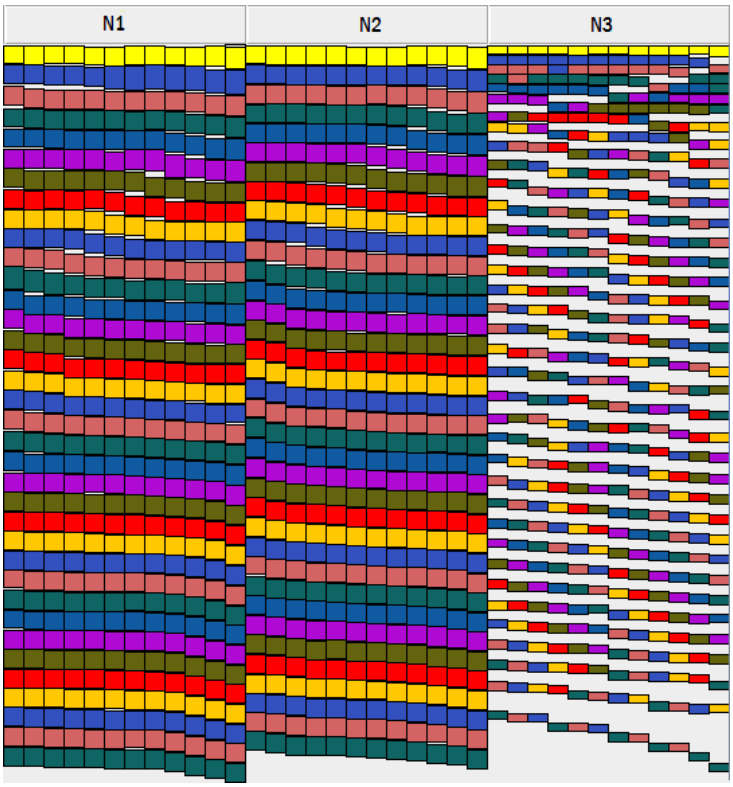


Figura 5.5: Execução de um dos *time steps* do experimento A no intervalo de tempo anterior a *t1*.

5.2.1.2 Experimento B:

Apesar de inicialmente a aplicação estar adaptada ao ambiente, por não haver escalonamento dinâmico, a aplicação não conseguiu se adaptar a mudança do ambiente ocorrida após *t1*, não aproveitando todo o poder computacional que ficou disponível em n1, como pode ser visto na Figura 5.7.

E após *t2*, como mostra a Figura 5.8, a aplicação perdeu muito desempenho por continuar executando muitas tarefas em n3, pois, como no início da execução a máquina n3 estava ociosa, o escalonamento estático foi definido de forma a colocar mais processos nela. Após *t2* n3 passou a disponibilizar apenas 50% de poder computacional, logo houve uma queda de desempenho que não ocorreria se houvesse escalonamento dinâmico. Isso mostra que, mesmo com um bom escalonamento inicial, em um ambiente dinâmico a aplicação pode perder bastante desempenho sem escalonamento dinâmico.

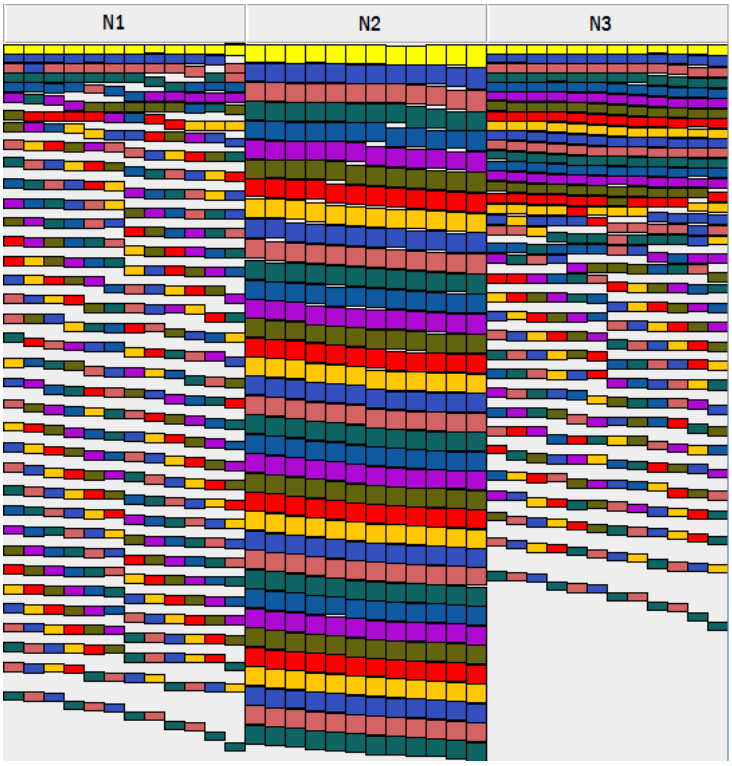


Figura 5.6: Execução de um dos *time steps* do experimento A no intervalo de tempo entre *t1* e *t2*.

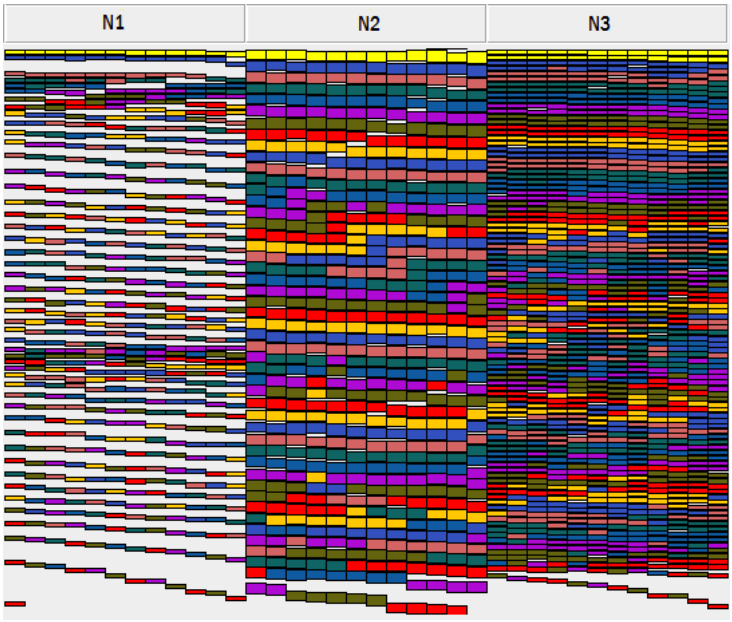


Figura 5.7: Figura 21: Execução de um dos *time steps* do experimento B no intervalo de tempo entre t1 e t2.

5.2.2 Experimentos C e D:

Os experimentos C e D adaptaram-se às mudanças do ambiente por causa do escalonamento dinâmico, aproveitando parte do poder computacional que foi disponibilizado

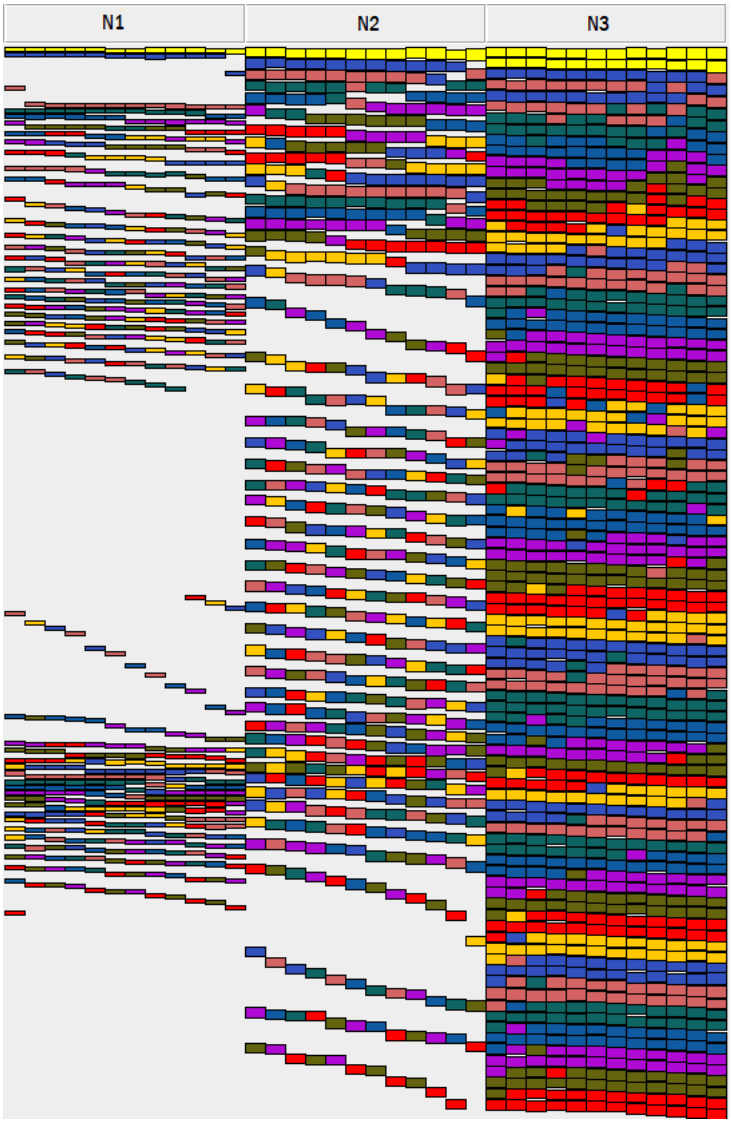


Figura 5.8: Execução de um dos *time steps* do experimento B no intervalo de tempo após *t2*.

em n1 no intervalo entre *t1* e *t2*. Mas, devido ao grau de paralelismo inapropriado, não obtiveram o máximo de desempenho.

5.2.2.1 Experimento C:

Mesmo com o escalonamento dinâmico, não foi possível melhorar significativamente o tempo de execução dos *time steps* executados antes de *t1* e após *t2*. A Figura 5.9 exibe um dos *time steps* do experimento C no intervalo de tempo anterior a *t1*, evidenciando que, com esse grau de paralelismo (w=36) não foi possível obter um escalonamento bom o suficiente para aproveitar o poder computacional de todas as máquinas, pois os núcleos de $n2$ ficaram a maior parte do tempo ociosos.

A Figura 5.10 exibe um dos *time steps* do experimento C no intervalo de tempo entre

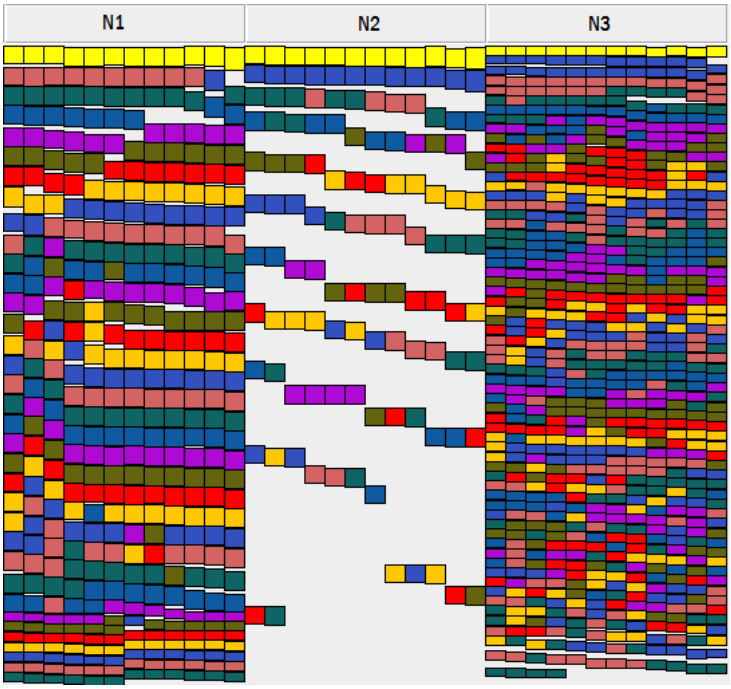


Figura 5.9: Execução de um dos *time steps* do experimento C no intervalo de tempo anterior a *t1*.

t1 e *t2*, mostrando que o ganho de desempenho obtido em relação ao experimento A (sem escalonamento dinâmico) ocorreu devido ao sistema distribuir a maior parte das tarefas da máquina mais lenta ($n2$) para as máquinas restantes.

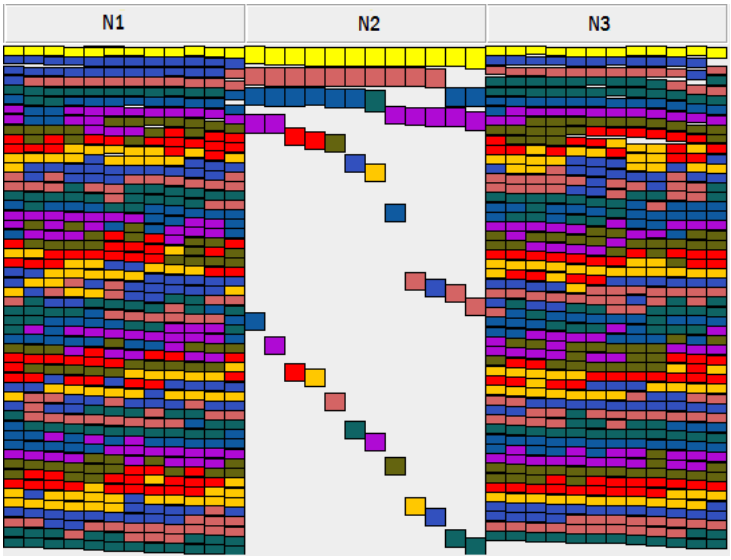


Figura 5.10: Execução de um dos *time steps* do experimento C no intervalo de tempo entre *t1* e *t2*.

5.2.2.2 Experimento D:

Similarmente ao experimento C, o escalonamento dinâmico conseguiu gerar ganho de desempenho entre *t1* e *t2* ao colocar poucos processos na máquina mais lenta ($n2$). E como o grau de paralelismo está mais apropriado que no experimento C, houve melhor aproveitamento da máquina $n2$. Como pode ser percebido na Figura 5.11, que exibe um dos *time steps* do experimento D nesse intervalo de tempo.

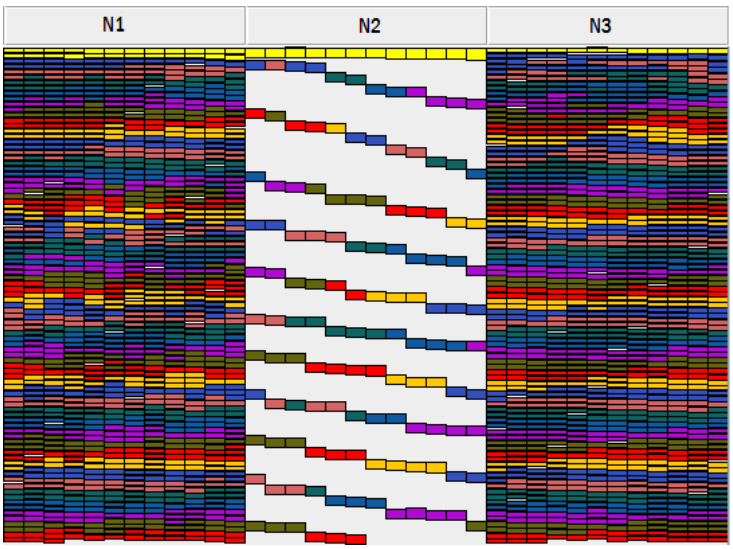


Figura 5.11: Execução de um dos *time steps* do experimento D no intervalo de tempo entre t1 e t2.

É importante notar, que ao contrário do experimento B, o escalonamento dinâmico ajustou os processos após *t2* de forma a não perder desempenho como ocorreu no outro. Isso pode ser percebido na figura 5.12, que exibe um dos *time steps* executados após o instante *t2*.

5.2.3 Experimentos E e F:

As aplicações apenas maleáveis só conseguiram se adaptar a mudança do ambiente no momento em que é calculado o próximo *time step* (onde ocorre a maleabilidade), ou seja, no último nível de cada *time step*, conforme foi definido anteriormente. Isso significa que se ocorrer uma mudança no ambiente no meio da execução de um *time step*, a aplicação só irá se ajustar à mudança no *time step* seguinte.

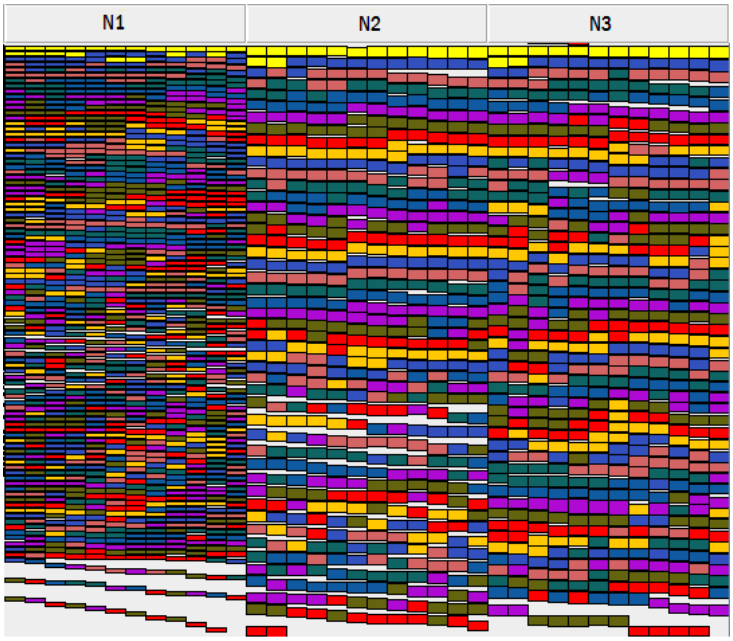


Figura 5.12: Execução de um dos *time steps* do experimento D no intervalo de tempo após t2.

5.2.3.1 Experimento E:

Como pode ser percebido na Figura 5.13, no meio da execução do *time step* 8 ocorreu a mudança do ambiente ($t2$). Com essa mudança, o poder computacional de $n3$ diminuiu 50%. Por não adaptar-se a essa mudança, o tempo de execução do *time step* 8 foi maior do que o esperado. A Figura 5.13 exibe o *time step* 8 do experimento E, indicando o instante em que ocorreu a mudança.

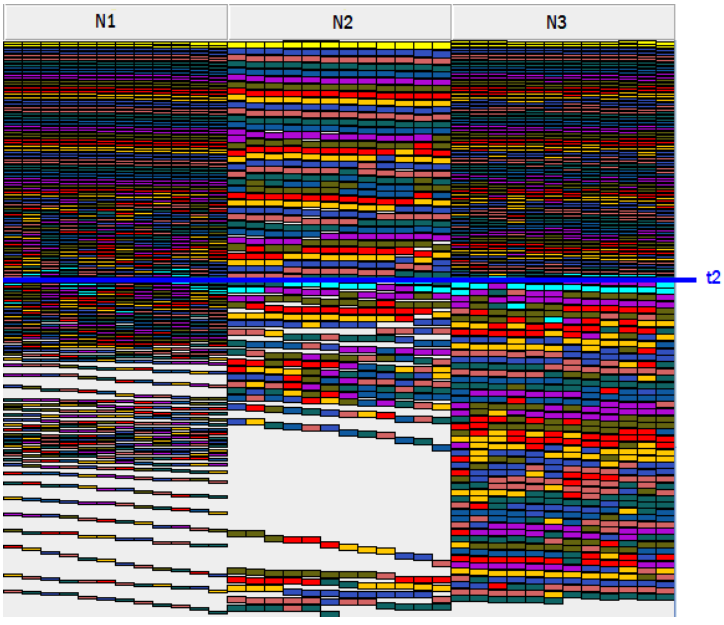


Figura 5.13: Execução do *time step* 8 do experimento E.

A Figura 5.14 exibe o *time step* 9 do experimento E, com a aplicação já adaptada à mudança do ambiente.

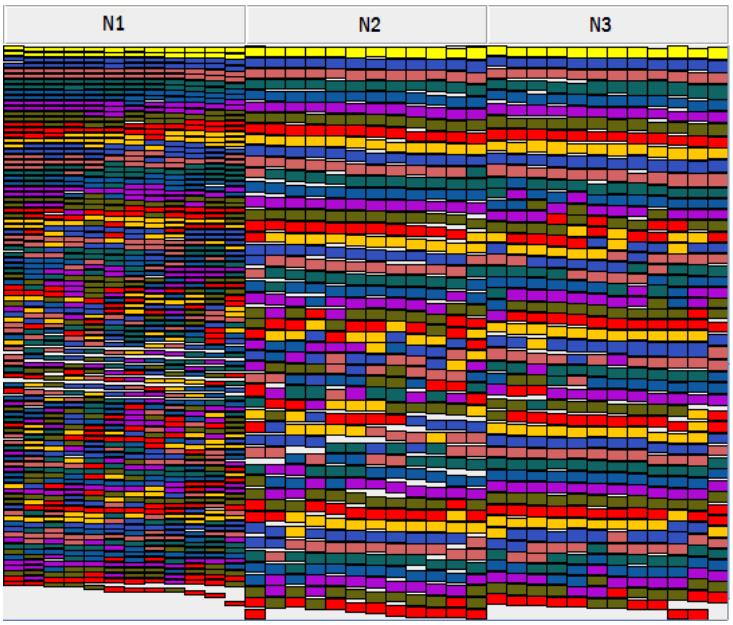


Figura 5.14: Execução do *time step* 9 do experimento E. A aplicação já foi adaptada à mudança ocorrida no ambiente.

5.2.3.2 Experimento F:

Assim como no experimento E, a maleabilidade tornou possível um alto ganho de desempenho, exceto nos *time steps* em que ocorreram a mudança no ambiente.

5.2.4 Experimentos G e H

Com o escalonamento dinâmico combinado a maleabilidade, as mudanças ocorridas foram identificadas antes do fim da execução do *time step* em que ocorreram (escalonamento dinâmico), e cada *time step* se inicia com o grau de paralelismo ideal para a heterogeneidade do ambiente (maleabilidade), gerando o melhor desempenho dentre estes experimentos.

5.2.4.1 Experimento G:

As Figuras 5.15 e 5.16 mostram a execução de um dos *time steps* do experimento G no intervalo de tempo anterior a *t1* e entre *t1* e *t2*, respectivamente. Além disso, é importante notar que a mudança no ambiente ocorrida no *time step* 8 não afetou significativamente

o desempenho da aplicação (ao contrário do experimento E), como pode ser notado na Figura 5.17.

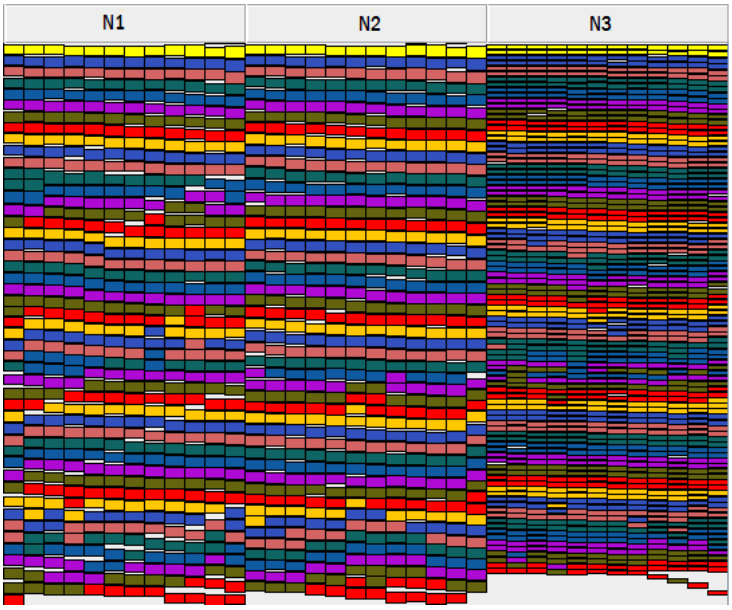


Figura 5.15: Execução de um dos *time steps* do experimento G no intervalo anterior a *t1*. (w=48)

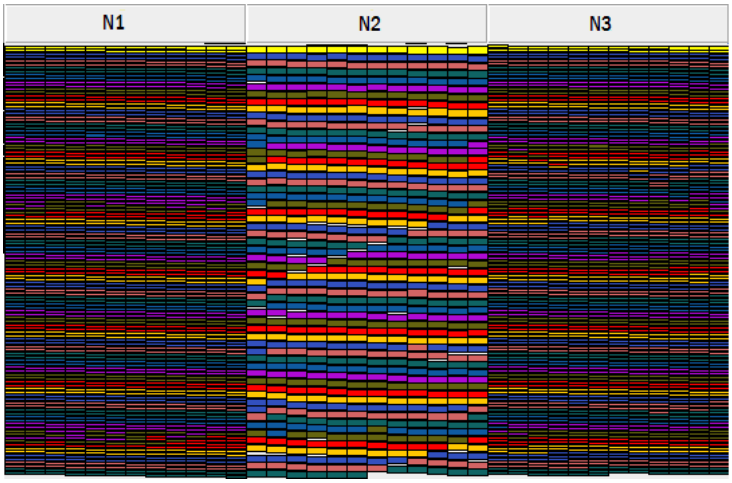


Figura 5.16: Execução de um dos *time steps* do experimento G no intervalo entre $t1$ e $t2$ (w=60)

5.2.4.2 Experimento H:

Apesar de haver um escalonamento inicial das tarefas no primeiro *time step*, o desempenho do experimento H foi semelhante ao do G, diferenciando-se apenas no tempo de execução do primeiro *time step*. Assim, esse foi o experimento que obteve os melhores resultados.

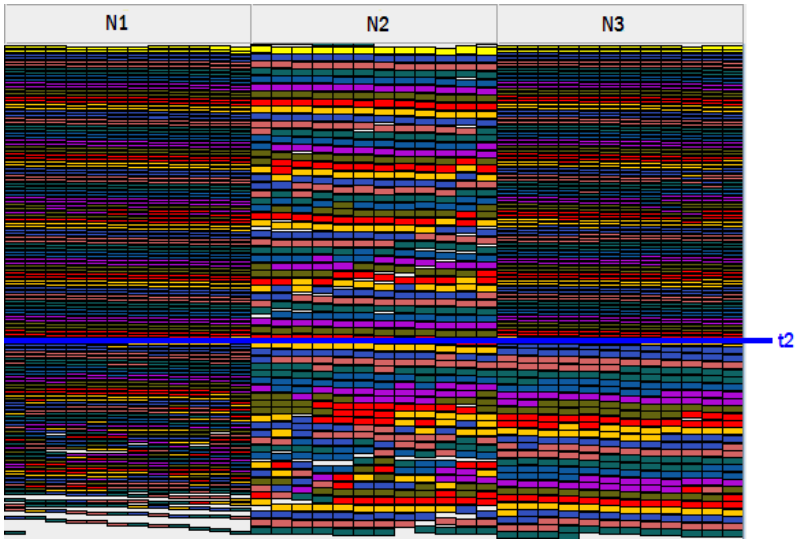


Figura 5.17: Execução do *time steps* 8 do experimento G.

5.3 Precisão e eficiência do algoritmo de escalonamento e de escolha da granularidade

Para analisar a precisão e a eficiência obtida pelo algoritmo de escalonamento e de escolha da granularidade, que é executado no último nível de cada *time step* da aplicação, foram realizados experimentos em diferentes ambientes. Cada ambiente utiliza de 1 a 6 máquinas semelhantes as utilizadas nos experimentos das seções 5.1 e 5.2. Na Tabela 5.3 é mostrada a porcentagem de poder computacional que cada máquina disponibiliza para cada ambiente. Como nem todos os ambientes contêm todas as máquinas, as que não foram utilizadas estão representadas por um traço.

Ambiente	Nó 1	Nó 2	Nó 3	Nó 4	Nó 5	Nó 6
A	100%	50%	50%	-	-	-
B	100%	100%	50%	50%	50%	50%
C	100%	100%	50%	33%	-	-
D	100%	100%	50%	-	-	-
E	100%	100%	100%	100%	50%	50%
F	100%	50%	-	-	-	-
G	100%	100%	50%	50%	-	-
H	100%	-	-	-	-	-
I	100%	100%	-	-	-	-
J	100%	100%	100%	-	-	-
K	100%	50%	25%	-	-	-

Tabela 5.3: Porcentagem de poder computacional disponibilizado para cada ambiente do experimento da seção 5.3

Em cada ambiente foi feita a execução do algoritmo para 100.000, 200.000, 300.000,

400.000, 500.000 e 1.000.000 de partículas. Para cada combinação (ambiente, número de partículas), é comparado o tempo previsto pelo escalonamento (*makespan*) com o tempo real de execução e é calculado um limite inferior (*lowerbound*) do tempo de execução da aplicação N-Corpos para verificar a precisão e eficiência do algoritmo, respectivamente.

Foram utilizadas duas abordagens para o cálculo do limite inferior. A primeira considera todas as máquinas do ambiente, com o objetivo de avaliar tanto a escolha da granularidade como o escalonamento das tarefas, nessa abordagem garante-se que nenhuma execução da aplicação N-Corpos no ambiente testado e com o mesmo número de partículas, seja qual for a configuração das tarefas, será feita em tempo inferior ao *lowerbound*. Na segunda abordagem, são consideradas apenas as máquinas que foram utilizadas pelo algoritmo e a mesma granularidade que o algoritmo retornou. Esta tem o objetivo de avaliar se, dada a granularidade, os recursos foram escolhidos de forma a se ter um bom aproveitamento deles. O maior motivo de também calcular esse segundo limitante, é porque o primeiro não considera que se a quantidade de processamento feita pela aplicação for muito pequena, nem sempre é conveniente utilizar todas as máquinas do ambiente, devido ao alto custo de comunicação embutido.

O primeiro *lowerbound* foi calculado por meio de uma simulação, desprezando-se o tempo de comunicação e as dependências entre as tarefas e distribuindo-as entre os processadores disponíveis de forma a obter o mesmo tempo de execução em cada processador. A aplicação é dividida em quantas tarefas forem necessárias para obter essa distribuição de carga. Dessa forma garante-se que não há nenhuma possível execução da mesma aplicação no mesmo ambiente que possa obter um tempo de execução inferior ao *lowerbound* calculado.

O segundo *lowerbound*, além de desprezar o tempo de comunicação e as dependências entre as tarefas, considera um número fixo de tarefas, o mesmo número de tarefas obtido pelo algoritmo de escolha da granularidade. Por isso não se pode garantir que haja um escalonamento em que todos os processadores terminem a execução ao mesmo tempo. Então o algoritmo para calcular esse *lowerbound* foi distribuir cada tarefa para o processador que minimize o tempo total de execução. Esse algoritmo foi comprovado ótimo em [52].

Os resultados desse experimento se encontram na Tabela 5.3, que também apresenta as porcentagens entre o tempo de execução real e o *makespan*, entre o tempo de execução real e o primeiro *lowerbound*, e entre o tempo real e o segundo *lowerbound*. As porcentagens negativas significam que a aplicação executou em menor tempo que o previsto.

Como se pode perceber, na maioria dos casos o algoritmo estimou o *makespan* próximo do tempo real e obteve-se uma boa aproximação do *lowerbound*, indicando a eficiência do algoritmo. Alguns casos tiveram pior desempenho devido à granularidade fina e à grande quantidade de cargas externas em alguma das máquinas em que foram executadas tarefas da aplicação. Estes dois fatores atrapalham o gerenciamento das tarefas (feitas pelo Gerenciador de Máquina) por causa da alta frequência de envio e recebimento de mensagens e da pouca permanência do Gerenciador no processador, já que ele precisará dividir o uso do processador com os processos da aplicação.

Para avaliar a eficiência da escolha da granularidade (através do cálculo do w), a aplicação foi executada com valores de w na vizinhança do melhor w para as seguintes configurações:

1. Ambiente B, 300.000 partículas (Melhor w=64): Valores de w escolhidos para execução = [48, 52, 56, 60, 68, 72, 76, 80];
2. Ambiente C, 300.000 partículas (Melhor W=53): Valores de w escolhidos para execução = [23, 33, 43, 63, 73, 83].

As figuras 5.18 e 5.19 mostram o resultado de execuções da primeira e segunda configuração respectivamente.

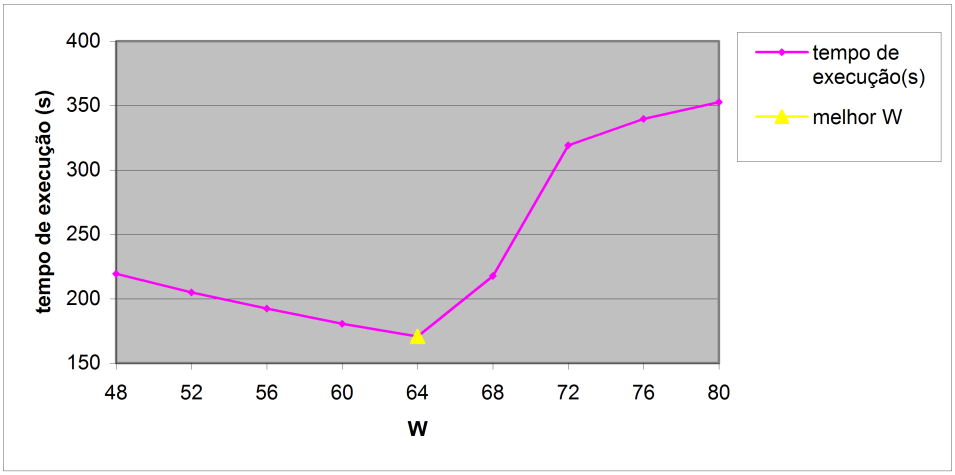


Figura 5.18: Comparação entre tempo de execução da aplicação N-Corpos com Ws próximos ao melhor W no ambiente B

Em ambos os ambientes, o valor de W escolhido pelo algoritmo foi o que proporcionou o melhor desempenho da aplicação, confirmando assim a eficiência do algoritmo na escolha do W.

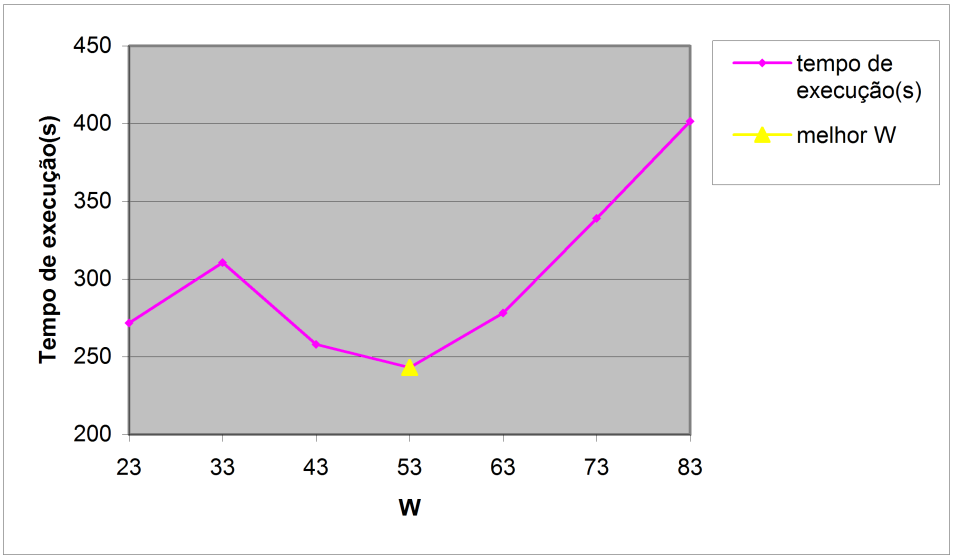


Figura 5.19: Comparação entre tempo de execução da aplicação N-Corpos com Ws próximos ao melhor W no ambiente C

Neste capítulo foram mostrados os resultados dos testes de desemepenho da aplicação N-Corpos maleável. Inicialmente foi calculado o custo da solução proposta, comparando-a com a versão N-Corpos original MPI sem maleabilidade em um ambiente homogêneo. Foi visto que a sobrecarga da solução diminui conforme aumenta-se o tamanho do problema, e é independente do número de *time steps*. A seguir foram feitos experimentos em ambiente dinâmico, onde a aplicação evolutiva e maleável obteve o melhor desempenho. Depois foi avaliada a precisão e eficiência do algoritmo de escalonamento e escolha da granularidade. Na maioria dos casos o algoritmo estimou o *makespan* próximo do tempo real, mostrando-se bastante preciso, e obteve-se uma boa aproximação do limite inferior da execução, mostrando-se bastante eficiente. Além disso, o valor de W escolhido pelo algoritmo foi o que proporcionou os melhores resultados, em comparação com valores próximos ao escolhido.

Ambiente	particulas	100.000	200.000	300.000	400.000	500.000	1.000.000
A	real	50,16	146,38	320,17	566,49	883,48	3.540,53
	makespan	54,09	153,67	330,80	578,96	898,10	3.555,07
	Lowerbound1	34,76	139,06	312,90	556,28	869,21	3.476,95
	Lowerbound2	35,59	139,06	312,90	556,28	869,21	3.476,95
	% real relativo makespan	-7%	-5%	-3%	-2%	-2%	0%
	% real relativo lowerbound1	44%	5%	2%	2%	2%	2%
	% real relativo lowerbound2	41%	5%	2%	2%	2%	2%
B	Real	34,65	88,35	170,82	294,73	461,38	1.786,92
	Makespan	38,01	91,94	182,65	307,34	466,86	1.796,20
	Lowerbound1	17,38	69,53	156,45	278,14	434,60	1.738,48
	Lowerbound2	28,73	69,53	156,45	278,14	434,60	1.738,48
	% real relativo makespan	-9%	-4%	-6%	-4%	-1%	-1%
	% real relativo lowerbound1	99%	27%	9%	6%	6%	3%
	% real relativo lowerbound2	21%	27%	9%	6%	6%	3%
C	real	33,72	147,56	243,15	407,23	631,41	2.509,68
	makespan	36,65	117,59	242,01	417,33	642,09	2.513,50
	Lowerbound1	24,54	98,16	220,87	392,67	613,56	2.454,32
	Lowerbound2	28,38	98,79	222,78	394,27	616,06	2.454,32
	% real relativo makespan	-8%	25%	0%	-2%	-2%	0%
	% real relativo lowerbound1	37%	50%	10%	4%	3%	2%
	% real relativo lowerbound2	19%	49%	9%	3%	2%	2%
D	Real	32,86	119,33	260,52	460,00	708,36	2.819,25
	Makespan	36,65	126,27	268,45	467,35	723,43	2.852,42
	Lowerbound1	27,81	111,25	250,32	445,03	695,37	2.781,56
	Lowerbound2	28,38	111,25	250,32	445,03	695,37	2.781,56
	% real relativo makespan	-10%	-5%	-3%	-2%	-2%	-1%
	% real relativo lowerbound1	18%	7%	4%	3%	2%	1%
	% real relativo lowerbound2	16%	7%	4%	3%	2%	1%
E	real	20,92	73,74	145,93	237,78	364,99	1.424,02
	Makespan	25,89	77,38	157,58	250,98	379,79	1.444,13
	Lowerbound1	13,90	55,62	125,16	222,51	347,68	1.390,78
	Lowerbound2	17,38	63,17	125,49	222,51	347,68	1.390,78
	% real relativo makespan	-19%	-5%	-7%	-5%	-4%	-1%
	% real relativo lowerbound1	50%	33%	17%	7%	5%	2%
	% real relativo lowerbound2	20%	17%	16%	7%	5%	2%
F	Real	51,38	190,55	424,90	753,72	1.175,53	4.697,48
	Makespan	56,04	197,87	434,13	765,12	1.190,86	4.741,16
	Lowerbound1	46,34	185,41	417,20	741,71	1.158,94	4.635,94
	Lowerbound2	46,34	185,41	417,20	741,71	1.158,94	4.635,94
	% real relativo makespan	-8%	-4%	-2%	-1%	-1%	-1%
	% real relativo lowerbound1	11%	3%	2%	2%	1%	1%
	% real relativo lowerbound2	11%	3%	2%	2%	1%	1%
G	Real	33,80	103,94	221,00	384,54	597,24	2.351,53
	Makespan	36,65	110,22	229,97	395,68	608,01	2.381,74
	Lowerbound1	23,17	92,71	208,60	370,85	579,47	2.317,97
	Lowerbound2	28,38	92,71	208,60	370,85	579,47	2.317,97
	% real relativo makespan	-8%	-6%	-4%	-3%	-2%	-1%
	% real relativo lowerbound1	46%	12%	6%	4%	3%	1%
	% real relativo lowerbound2	19%	12%	6%	4%	3%	1%

Ambiente	particulas	100.000	200.000	300.000	400.000	500.000	1.000.000
H	Real	71,87	284,20	636,69	1.133,93	1.774,03	7.295,70
	Makespan	73,15	287,87	646,34	1.148,59	1.794,61	7.181,17
	Lowerbound1	69,52	278,12	625,80	1.112,56	1.738,41	6.953,91
	Lowerbound2	69,52	278,12	625,80	1.112,56	1.738,41	6.953,91
	% real relativo makespan	-2%	-1%	-1%	-1%	-1%	2%
	% real relativo lowerbound1	3%	2%	2%	2%	2%	5%
	% real relativo lowerbound2	3%	2%	2%	2%	2%	5%
I	Real	36,67	142,63	318,74	566,03	882,80	3.537,12
	Makespan	39,52	146,49	325,17	575,07	895,84	3.567,64
	Lowerbound1	34,76	139,06	312,90	556,28	869,21	3.476,95
	Lowerbound2	34,76	139,06	312,90	556,28	869,21	3.476,95
	% real relativo makespan	-7%	-3%	-2%	-2%	-1%	-1%
	% real relativo lowerbound1	5%	3%	2%	2%	2%	2%
	% real relativo lowerbound2	5%	3%	2%	2%	2%	2%
J	Real	25,29	95,84	214,28	378,82	588,08	2.367,06
	Makespan	29,70	100,81	219,70	386,43	601,00	2.392,81
	Lowerbound1	23,17	92,71	208,60	370,85	579,47	2.317,97
	Lowerbound2	23,17	92,71	208,60	370,85	579,47	2.317,97
	% real relativo makespan	-15%	-5%	-2%	-2%	-2%	-1%
	% real relativo lowerbound1	9%	3%	3%	2%	1%	2%
	% real relativo lowerbound2	9%	3%	3%	2%	1%	2%
K	Real	49,92	233,96	369,12	654,80	1.023,44	4.066,96
	Makespan	54,85	178,00	381,05	665,83	1.030,04	4.054,76
	Lowerbound1	39,72	158,92	357,60	635,75	993,38	3.973,66
	Lowerbound2	46,89	160,15	360,36	639,04	998,51	3.973,66
	% real relativo makespan	-9%	31%	-3%	-2%	-1%	0%
	% real relativo lowerbound1	26%	47%	3%	3%	3%	2%
	% real relativo lowerbound2	6%	46%	2%	2%	2%	2%

Tabela 5.4: Relação entre o tempo real da aplicção N-Corpos e previsto pelo algoritmo de escolha de granularidade.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho tem por objetivo mostrar a viabilidade de transformar aplicações científicas originalmente desenvolvidas para *clusters* (aplicações moldáveis) em maleáveis (auto-configuráveis) e, dessa forma, aumentar a eficiência das aplicações paralelas nas plataformas de mais baixo custo e escaláveis, como grades e nuvens, cada vez mais utilizadas nos dias de hoje. Dessa forma, os cientistas poderão utilizar essas tecnologias de computação distribuída para resolver problemas complexos de forma eficiente e a um baixo custo.

Através dos resultados experimentais, este trabalho mostrou que aplicações que anteriormente foram desenvolvidas para ambientes homogêneos podem executar eficientemente em ambientes heterogêneos e dinâmicos, através da utilização do EasyGrid AMS e do modelo de execução *1PTASK*. Para isso, foi atribuída a propriedade de auto-configuração no EasyGrid AMS, tornando a aplicação do usuário capaz de ajustar seu grau de paralelismo para obter o melhor proveito dos recursos utilizados (maleabilidade). Foi comprovado que, além de não reduzir significativamente o desempenho em máquinas homogêneas, a aplicação se tornou altamente preparada para as possíveis variações de poder computacional e heterogeneidade dos recursos disponíveis a ela.

Com auto-configuração, combinada às outras características autonômicas que já estavam disponíveis através do EasyGrid AMS, é possível tornar as aplicações MPI mais autônomas, ou seja, capaz de se gerenciarem de forma inteligente, sem a necessidade da interação com o usuário para que isso ocorra. Isso possibilitará um melhor aproveitamento das tecnologias disponíveis. Com isso, podem ser reduzidos os custos de investimento em equipamentos dedicados, pois o compartilhamento oferecido nos ambientes dinâmicos

tende a evitar que os processadores disponíveis fiquem ociosos, aumentando a utilização dos mesmos. Além disso, haverá um aumento na escalabilidade das aplicações executadas, porque as máquinas utilizadas não precisam ser idênticas e podem estar geograficamente distribuídas. Isso permite que cientistas possam resolver problemas maiores e com maior precisão utilizando as infra-estruturas computacionais disponíveis atualmente.

Dessa forma, não será necessário ao programador o conhecimento específico do funcionamento da plataforma onde a aplicação será executada, já que é possível facilmente integrar uma aplicação desenvolvida para clusters ao middleware EasyGrid AMS, que se encarregará de gerenciar os recursos utilizados para que a aplicação tenha o máximo de proveito na plataforma de execução. Este trabalho beneficiará principalmente as aplicações fortemente acopladas e iterativas, como as que utilizam o algoritmo *ring*, comum em diversas simulações. Logo contribuirá para melhorias em diversas áreas de pesquisa em geral.

Como trabalho futuro, serão realizados testes em ambientes com mais máquinas e também será melhorado o escalonamento dinâmico das tarefas durante cada *time step* para que ele se torne mais eficiente em maquinas multicore para este tipo de aplicação, avaliando-se as outras opções de escalonamento propostas no capítulo 4. Também será importante estudar como será o acréscimo da maleabilidade em outras aplicações, de forma a tornar a maleabilidade no EasyGrid AMS mais genérica, para que o esse processo de transformação se torne o mais simples e automático possível.

Outro estudo que esse trabalho viabiliza, é utilizar a auto-configuração para alterar outras características da aplicação, além do grau de paralelismo. Um exemplo disso seria utilizar as características do ambiente para definir qual algoritmo a aplicação executará, podendo utilizar um algoritmo mais preciso caso haja recursos disponíveis para isso, e quando não houver mais, diminuir a precisão dos cálculos através de aproximações. Assim, podemos ensinar as aplicações a conseguirem o melhor custo-benefício dos seus objetivos em relação aos recursos disponibilizados a elas.

Referências

[1] SENA, A. C. *Um modelo Alternativo Para Execução Eficiente de Aplicações Paralelas MPI nas Grades Computacionais.* Tese (Doutorado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, 2008.

[2] LARGE Hadron Collider. Último acesso 20/11/2011. `http://lhc.web.cern.ch/lhc/`.

[3] WORLDWIDE LHC Computing Grid. Último acesso 20/11/2011. `http://lcg.web.cern.ch/lcg/`.

[4] PROJETO Genoma. Último acesso 13/01/2012. `http://www.genome.gov/`.

[5] SENA, A. C. et al. An approach to optimise the execution of rtm algorithm in multicore machines. *eScience, IEEE International Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 403–410, 2011.

[6] BUYYA, R. (Ed.). *High Performance Cluster Computing: Architectures and Systems.* [S.l.]: Prentice Hall, EUA, 1999. (Vol. 1).

[7] FOSTER, I.; KESSELMAN, C. (Ed.). *The GRID: Blueprint for a New Computing Infrastructure.* [S.l.]: Morgan Kaufmann, 1999.

[8] ARMBRUST, M. et al. *Above the Clouds: A Berkeley View of Cloud Computing.* [S.l.], Feb 2009. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html>.

[9] PARASHAR, M.; SALIM, H. Autonomic computing: An overview. In: *In Proceedings of the Unconventional Programming Paradigms: International Workshop (UPP 2004).* Le Mont St-Michel, França: Springer, 2005. p. 247–259.

[10] FEITELSON, D. G.; RUDOLPH, L. Towards convergence in job schedulers for parallel supercomputers. In: FEITELSON, D. G.; RUDOLPH, L. (Ed.). *Proceedings of the JSSPP.* [S.l.]: Springer, 1996. (Lecture Notes in Computer Science), p. 1–26.

[11] MAGHRAOUI, K. et al. Dynamic malleability in iterative MPI applications. In: *Proceedings of 7th International Symposium on Cluster Computing and the Grid (CCGrid 2007).* Rio de janeiro, Brasil: IEEE Computer Society, 2007. p. 591–598.

[12] SINNEN, O. *Task Scheduling for Parallel Systems.* [S.l.]: Wiley-Interscience, 2007. (Wiley Series on Parallel And Distributed Computing). ISBN 9780471735762.

[13] MESSAGE Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, Julho 1997. Último acesso 10/10/2008. `http://www.mpi-forum.org/`.

[14] Message Passing Forum. *MPI: A Message Passing Interface*. [S.l.], 1995.

[15] STERRITT, R. et al. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, Elsevier Science Publishers, v. 19, n. 3, p. 181–187, Julho 2005.

[16] KEPHART, J.; CHESS, D. The vision of autonomic computing. *IEEE Computer Magazine*, IEEE Computer Society, v. 36, n. 1, p. 41–50, Janeiro 2003.

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

[18] FREY, J. et al. Condor-G: A computational management agent for multi-institutional grids. *Cluster Computing*, Springer Netherlands, v. 3, n. 5, p. 237–246, 2002.

[19] THE Cactus Code Server. `http://www.cactuscode.org/`.

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

[21] THAIN, D.; TANNENBAUM, T.; LIVNY, M. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, v. 17, n. 2-4, p. 323–356, 2005.

[22] NASCIMENTO, A. P. et al. Autonomic application management for large scale MPI programs. *International Journal of High Performance Computing and Networking (IJHPCN)*, Inderscience publishers, (aceito para publicação em 2008), 2008.

[23] BERMAN, F. et al. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications*, v. 15, n. 4, p. 327–344, 2001.

[24] BOERES, C.; REBELLO, V. E. F. EasyGrid: Towards a framework for the automatic grid enabling of legacy MPI applications. *Concurrency and Computation: Practice and Experience*, John Wiley and Sons Ltd, v. 16, n. 5, p. 425–432, Abril 2004.

[25] VADHIYAR, S. S.; DONGARRA, J. J. Srs - a framework for developing malleable and migratable parallel applications for distributed systems. In: *In: Parallel Processing Letters*. [S.l.: s.n.], 2002. v. 2, p. 291–312.

[26] MAGHRAOUI, K. E. et al. Malleable iterative MPI applications. *Conc. Comput. : Pract. Exper.*, John Wiley and Sons Ltd., Chichester, UK, v. 21, n. 3, p. 393–413, 2009. ISSN 1532-0626.

[27] DONGARRA, J. et al. *Sourcebook of Parallel Computing*. [S.l.]: Morgan Kaufmann Publishers Inc., 2003.

[28] FOSTER, I. (Ed.). *Designing and Building Parallel Programs*. [S.l.]: An Online Publishing Project of Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation, 1995.

[29] NASCIMENTO, A. P. et al. Managing the execution of large scale MPI applications on computational grids. In: *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)*. Rio de Janeiro, Brasil: IEEE Computer Society, 2005.

[30] MESSAGE Passing Interface Forum. MPI: A Message Passing Interface Standard, junho 1995. Último acesso 10/10/2008. `http://www.mpi-forum.org/`.

[31] VIANNA, D. Q. C. *Um Sistema de Gerenciamento de Aplicações MPI para Ambientes Grid*. Dissertação (Mestrado) — Instituto de Computação, Universidade Federal Fluminense, 2005.

[32] SILVA, J. A. da; REBELLO, V. E. F. Low cost self-healing in MPI applications. In: *Proceedings of the 14th European PVM/MPI User's Group Meeting, Paris, France*. [S.l.]: Springer, 2007. p. 144–152.

[33] BOERES, C. et al. Efficient hierarchical self-scheduling for MPI applications executing in computational grids. In: *Proceedings of the 3rd International workshop on Middleware for Grid Computing*. Grenoble, France: ACM Press, 2005. p. 1–6.

[34] NASCIMENTO, A. P. et al. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience*, John Wiley and Sons Ltd, Chichester, Reino Unido, v. 19, n. 14, p. 1955–1974, 2007. ISSN 1532-0626.

[35] NASCIMENTO, A. P.; BOERES, C.; REBELLO, V. E. F. Dynamic self-scheduling for parallel applications with task dependencies. In: *Proceedings of the 6th international Workshop on Middleware for Grid Computing*. Leuven, Belgium: ACM Press, 2008.

[36] NASCIMENTO, A. P. *Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais*. Tese (Doutorado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, Brasil, Maio 2008.

[37] AARSETH, S. J. From NBODY1 to NBODY6: The growth of an industry. *Publications of the Astronomical Society of The Pacific*, v. 111, p. 1333–1346, Novembro 1999.

[38] GUALANDRIS, A.; ZWART, S. P.; TIRADO-RAMOS, A. Performance analysis of direct N-body algorithms for astrophysical simulations on distributed systems. *Parallel Computing*, Elsevier Science Publishers B. V., Amsterdam, Holanda, v. 33, n. 3, p. 159–173, 2007. ISSN 0167-8191.

[39] GIERSZ, M.; HEGGIE, D. C. Statistics of N-body simulations I. Equal masses before core collapse. *Royal Astronomical Society MONTHLY NOTICES*, v. 268, p. 257–275, 1994.

[40] MILOSAVLJEVIC, M.; MERRITT, D. Formation of galactic nuclei. *Astrophysical Journal*, Astrophysical Journal, v. 563, p. 34–62, 2001.

[41] MAKINO, J. An efficient parallel algorithm for $O(N^2)$ direct summation method and its variations on distributed-memory parallel machines. *New Astronomy*, Elsevier, v. 7, n. 7, p. 373–384, Outubro 2002.

[42] TOPCUOUGLU, H.; HARIRI, S.; WU, M.-y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 13, p. 260–274, March 2002. ISSN 1045-9219. Disponível em: <http://dl.acm.org/citation.cfm?id=566137.566142>.

[43] CASANOVA, H. et al. Heuristics for scheduling parameter sweep applications in grid environments. In: *Proceedings 9th Heterogeneous Computing Workshop (HCW '00)*. Cancun, Mexico: IEEE Computer Society, 2000. p. 349–363.

[44] SENA, A. C. et al. Easygrid enabling of iterative tightly-coupled parallel MPI applications. *International Symposium on Parallel and Distributed Processing with Applications (ISPA-08)*, IEEE Computer Society, Dezembro 2008.

[45] RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. [S.l.]: Prentice Hall, 2010. (Prentice Hall Series in Artificial Intelligence). ISBN 9780136042594.

[46] LAARHOVEN, P.; AARTS, E. *Simulated Annealing: Theory and Applications*. [S.l.]: D. Reidel, 1987. (Mathematics and Its Applications). ISBN 9789027725134.

[47] RAO, S. *Engineering optimization: theory and practice*. [S.l.]: Wiley, 1996. (Wiley-Interscience publication). ISBN 9780471550341.

[48] LUENBERGER, D. *Linear and nonlinear programming*. [S.l.]: Addison-Wesley, 1984. ISBN 9780201157949.

[49] BAZARAA, M.; SHERALI, H.; SHETTY, C. *Nonlinear Programming Theory and Algorithms*. New York: John Wiley, 1993.

[50] INSIDE the Linux 2.6 Completely Fair Scheduler. Último acesso 23/01/2012. `http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/`.

[51] NASCIMENTO, A. et al. On the feasibility of dynamically scheduling dag applications on shared heterogeneous systems. In: SIPS, H.; EPEMA, D.; LIN, H.-X. (Ed.). *Euro-Par 2009 Parallel Processing*. [S.l.]: Springer Berlin / Heidelberg, 2009, (Lecture Notes in Computer Science).

[52] GRAHAM, R. et al. Optimization and approximation in deterministic sequencing and scheduling: a survey. In: HAMMER, E. J. P.; KORTE, B. (Ed.). *Discrete Optimization II Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*. Elsevier, 1979, (Annals of Discrete Mathematics, v. 5). p. 287 – 326. Disponível em: <http://www.sciencedirect.com/science/article/pii/S016750600870356X>.

APÊNDICE A – Resultados Antigos

Este apêndice apresenta alguns dos resultados dos experimentos que foram realizados nas máquinas antigas, antes do problema de aquecimento no laboratório, que foi descrito na seção 5.1. Após esses experimentos foram feitas melhorias no desempenho do algoritmo que calcula o grau de paralelismo ideal e, por isso, esses testes precisaram ser refeitos.

Foram realizados experimentos semelhantes aos das seções 5.1 e 5.2, porém com máquinas dual-processados com Intel(R) Xeon(R) CPU E5410 @ 2.33GHz, totalizando 8 núcleos, 16 GB de memória e o sistema operacional CentOS 5.2 64 bits, *kernel* 2.6.18, em cada máquina. Esses resultados foram desconsiderados devido a melhorias no algoritmo para busca do W ideal.

Os resultados para os testes de custo da solução foram obtidos através da média aritmética de três execuções e a diferença entre os tempos de execução foi menor que 1% e podem ser vistos na Tabela A. A coluna MPI apresenta o tempo de execução em segundos da aplicação N-Corpos MPI moldável, enquanto que a coluna AMS apresenta o tempo de execução da aplicação N-Corpos maleável e a coluna Ovrhd. apresenta a porcentagem de sobrecarga obtida.

A Figura A.1 exibe a porcentagem de sobrecarga média em relação ao número de partículas.

Para simular um ambiente heterogêneo e dinâmico, primeiramente, foi colocada uma carga em cada núcleo das máquinas *n1* e *n2* (16 núcleos no total) com o objetivo de cada núcleo de *n1* e *n2* disponibilizar apenas 50% de processamento para o N-Corpos. Após o instante *t1* =1300 segundos retira-se a carga dos núcleos de *n1* e após o instante *t2*=2300 segundos (a partir do início da execução) coloca-se carga nos núcleos de *n3*. Na Figura A.2 pode-se perceber a porcentagem de poder computacional das máquinas *n1*, *n2* e *n3* em função do tempo em segundos.

	100mil			150mil			200mil		
Nº de *time steps*	MPI	AMS	Ovrhd.	MPI	AMS	Ovrhd.	MPI	AMS	Ovrhd.
1	23,63	24,94	6%	53,03	54,55	2,9%	94,36	94,36	1,1%
2	47,27	49,63	5%	106,11	109,01	2,7%	188,33	188,33	1,4%
3	70,89	74,88	6%	159,30	163,36	2,5%	282,45	282,45	1,3%
4	94,49	99,51	5%	212,18	217,72	2,6%	377,32	377,32	1,0%
5	118,09	124,14	5%	264,95	272,48	2,8%	470,97	470,97	1,2%
6	141,71	149,44	5%	318,32	329,00	3,4%	566,06	566,06	1,0%
7	165,32	174,50	6%	371,90	382,07	2,7%	660,51	660,51	1,1%
8	189,12	199,45	5%	424,28	436,36	2,8%	754,81	754,81	0,9%
9	212,54	224,45	6%	477,99	490,67	2,7%	847,68	847,68	1,2%
10	236,17	248,99	5%	530,10	545,85	3,0%	942,96	942,96	1,1%

	250mil			300mil		
Nº de *time steps*	MPI	AMS	Ovrhd.	MPI	AMS	Ovrhd.
1	147,07	148,32	0,8%	211,4	212,85	0,7%
2	294,98	296,40	0,5%	422,8	425,71	0,7%
3	442,88	444,24	0,3%	634,2	638,62	0,7%
4	589,07	592,51	0,6%	845,6	851,03	0,6%
5	735,47	740,77	0,7%	1057,0	1.064,10	0,7%
6	881,93	889,51	0,9%	1268,5	1.276,96	0,7%
7	1.028,56	1.037,12	0,8%	1479,9	1.489,81	0,7%
8	1.178,78	1.185,37	0,6%	1691,3	1.702,64	0,7%
9	1.328,32	1.333,54	0,4%	1902,7	1.916,26	0,7%
10	1.473,20	1.481,73	0,6%	2114,1	2.129,39	0,7%

Tabela A.1: Relação entre o tempo (segundos) do N-Corpos original (MPI) e do maleável (AMS) e a sobrecarga do maleável em relação ao original.

Foram realizados os mesmos 8 experimentos da seção 5.2,porém com 300 mil partículas. O tempo em segundos de cada *time step* e o tempo total de cada experimento é exibido na Tabela A.2.

ts	A	B	C	D	E	F	G	H
1	401,8	325,5	401,5	319,0	402,5	325,6	400,4	317,1
2	401,1	329,6	398,1	321,6	325,4	325,9	318,5	320,3
3	401,4	325,5	398,6	329,8	325,6	325,8	326,3	317,9
4	398,9	326,1	336,4	323,5	325,0	329,5	306,2	325,8
5	396,4	329,3	317,4	287,3	261,9	329,7	252,5	288,8
6	399,5	328,9	318,6	301,9	261,9	262,8	252,8	253,1
7	401,8	328,7	377,4	303,1	262,6	262,7	252,3	352,9
8	401,1	572,6	398,5	341,9	348,1	341,5	266,2	262,5
9	402,1	580,1	397,9	327,6	325,7	325,7	317,9	317,4
10	400,7	575,7	393,6	324,2	325,9	325,5	315,8	328,8
TT	4004,8	4022,2	3738,1	3179,9	3164,7	3154,8	3008,9	2984,5

Tabela A.2: Tempo de execução de cada experimento

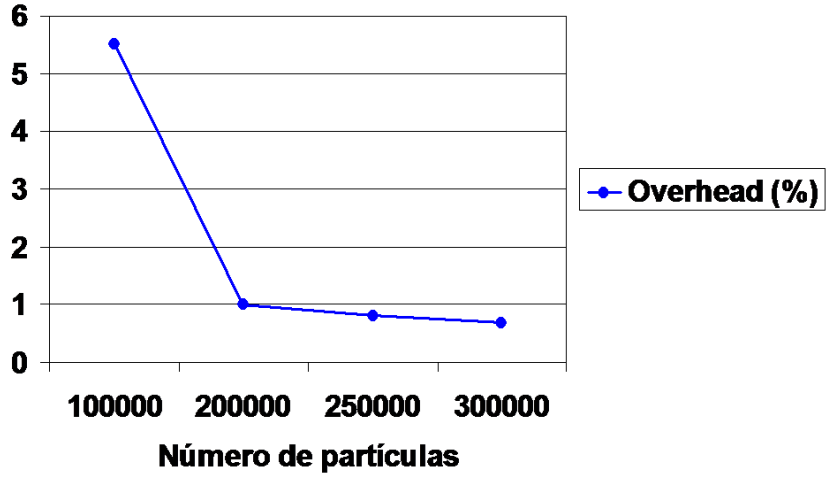


Figura A.1: Porcentagem de sobrecarga de execução do N-Corpos maleável em relação ao N-Corpos estático (em um ambiente homogêneo) em função do número de partículas.

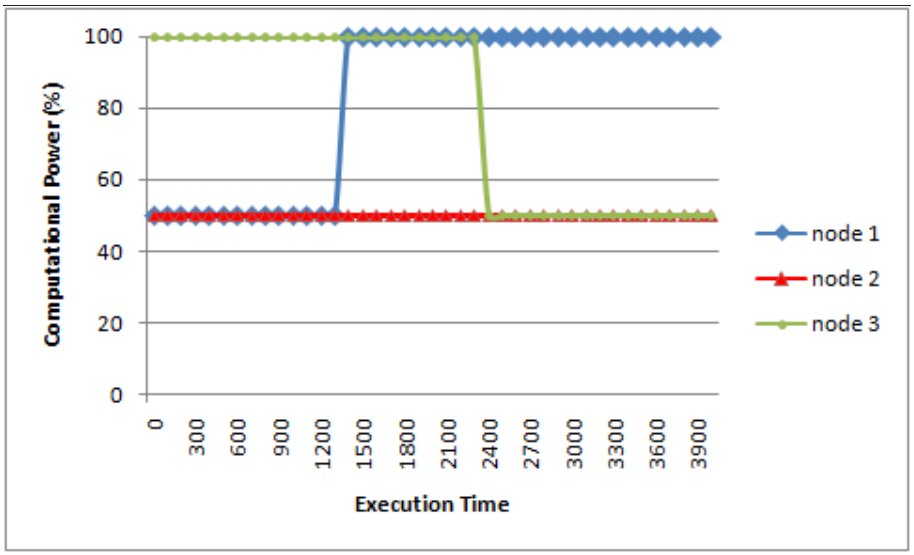


Figura A.2: Poder computacional dos nós n1, n2 e n3 durante a execução da aplicação

Assim como os experimentos da seção 5.2 A Tabela A.2 foi dividida em dois gráficos. O primeiro gráfico (Figura A.3) representa os tempos obtidos nos experimentos A, C, E e G (não há conhecimento prévio sobre a heterogeneidade do ambiente), e o segundo (Figura A.4) representa os tempos obtidos nos experimentos B, D, F e H (aplicação ajustada inicialmente à heterogeneidade do ambiente).

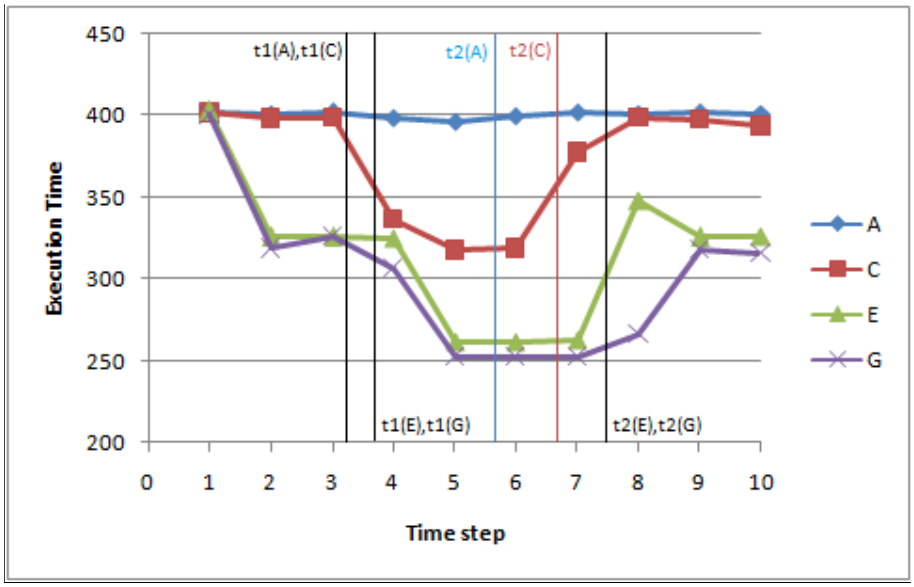


Figura A.3: tempo de execução de cada *time step* dos experimentos A,C,E e G.

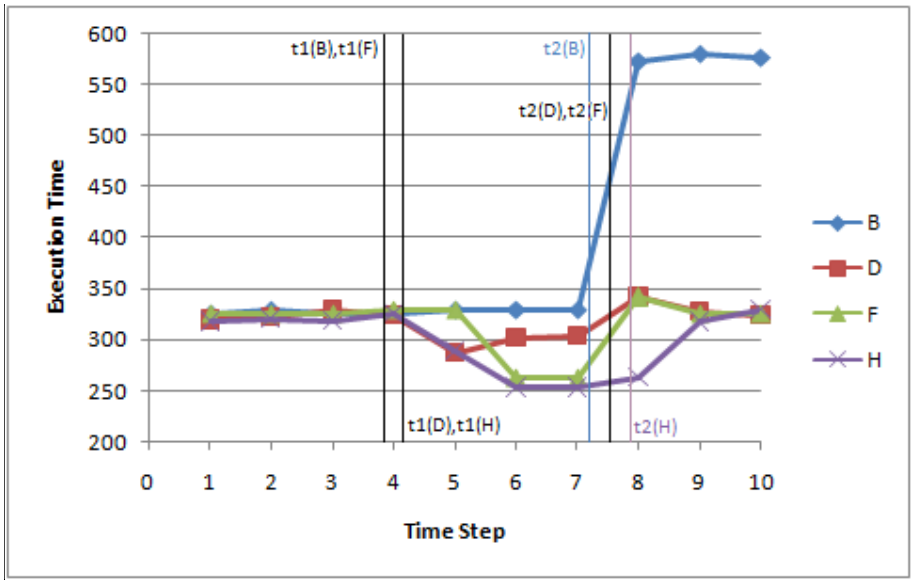


Figura A.4: Tempo de execução de cada *time step* dos experimentos B, D, F e H.

APÊNDICE B – Ferramenta para Monitoramento de Aplicações Paralelas no AMS

Este apêndice apresenta um resumo da ferramenta criada neste trabalho para monitorar a aplicação N-Corpos através do EasyGrid AMS. Apesar de ser desenvolvida para aplicações que utilizam o modelo 1PTASK, ela pode ser utilizada por outras aplicações paralelas, desde que o monitoramento siga um padrão para escrita dos arquivo de log.

Os arquivos de log devem conter as seguintes informações:

1. Lista de máquinas utilizadas: Deve haver um arquivo que contém um nome para cada máquina utilizada e o número de núcleos das máquinas;
2. O horário, de cada máquina, que corresponderá ao tempo zero (para sincronizar os relógios das máquinas);
3. O horário de início e fim da execução de cada tarefa que se deseja monitorar e em qual máquina ela foi executada. A tarefa deve ser identificada por um número natural;
4. As dependências entre as tarefas: Quando for informado o horário de início ou de fim da execução de uma tarefa, pode-se informar quais são os identificadores das tarefas predecessoras dela. Para a aplicação N-Corpos, a função de precedência já está implícita na ferramenta, sendo necessário apenas informar o valor de W de cada *time step*.

Para esse trabalho, as informações no arquivo de log, exceto a lista de máquinas e as dependências, foram escritas pela camada de monitoramento do EasyGrid AMS. A lista de máquinas foi escrita manualmente.

Ao iniciar a aplicação, o usuário seleciona a pasta onde estão os arquivos de log e informa o nome do arquivo com a lista de máquinas. Após isso, o sistema lê os arquivos e exibe a tela principal, conforme a Figura B.1. A ferramenta possui uma escala de tempo (em segundos) no canto superior. As tarefas são representadas por retângulos, nos quais o tempo de início e fim da execução de cada tarefa corresponde ao valor que está na escala de tempo, ou seja, quanto maior for o comprimento do retângulo, maior foi a duração da tarefa. Além disso, tarefas do mesmo nível de precedência foram representadas pela mesma cor. As máquinas são representadas pela sequência $N0$, $N1$,... Ao colocar o mouse sobre o identificador da máquina aparece um *tooltip* com o nome da máquina e quantos núcleos ela possui. Também é possível perceber quantos núcleos a máquina possui através da altura em que ela foi representada. Por exemplo, em $N0$ podem ser dispostas até 8 tarefas na vertical, logo ela possui 8 núcleos.

É possível filtrar quais tarefas serão exibida através do painel **Intervalo de Exibição**. Esse filtro pode ser feito pelo *time step*, ou pelo identificador númerico da tarefa, denominado *rank*. Também é possível mudar a escala de tempo, movendo o controle deslizante **Zoom** ou digitando a escala desejada e clicando no botão **Escala**. A escala está em pixels/s.

Através do painel **Exibir**, é possível mostrar as dependências, o identificador numérico, o tempo de CPU e o tempo de parede de cada tarefa. Esses tempos devem ser calculados durante a execução da aplicação e informados no arquivo de log. No caso desse projeto, isso foi feito pela camada de monitoramento do EasyGrid AMS, nos Gerenciadores de Máquina. A figura B.2 mostra um exemplo onde as dependências (representadas por seguimentos de reta azuis que ligam o início da tarefa ao final da sua predecessora), o identificador númerico da tarefa (*rank*) e os tempos de CPU e de parede foram habilitados.

Também é possível exibir as dependência de apenas algumas tarefas, clicando com o botão esquerdo do mouse sobre o retângulo que a representa. A tarefa ficará com a cor azul claro, e suas predecessoras ficarão com a cor verde claro, conforme mostra a Figura B.3. Nesse exemplo, a tarefa 3445 tem as predecessoras 3413 e 3414, e a tarefa 3421 tem as predecessoras 3389 e 3390.

Com essa ferramenta foi possível analisar melhor os resultados dos experimentos e identificar problemas e possíveis melhorias durante o projeto.

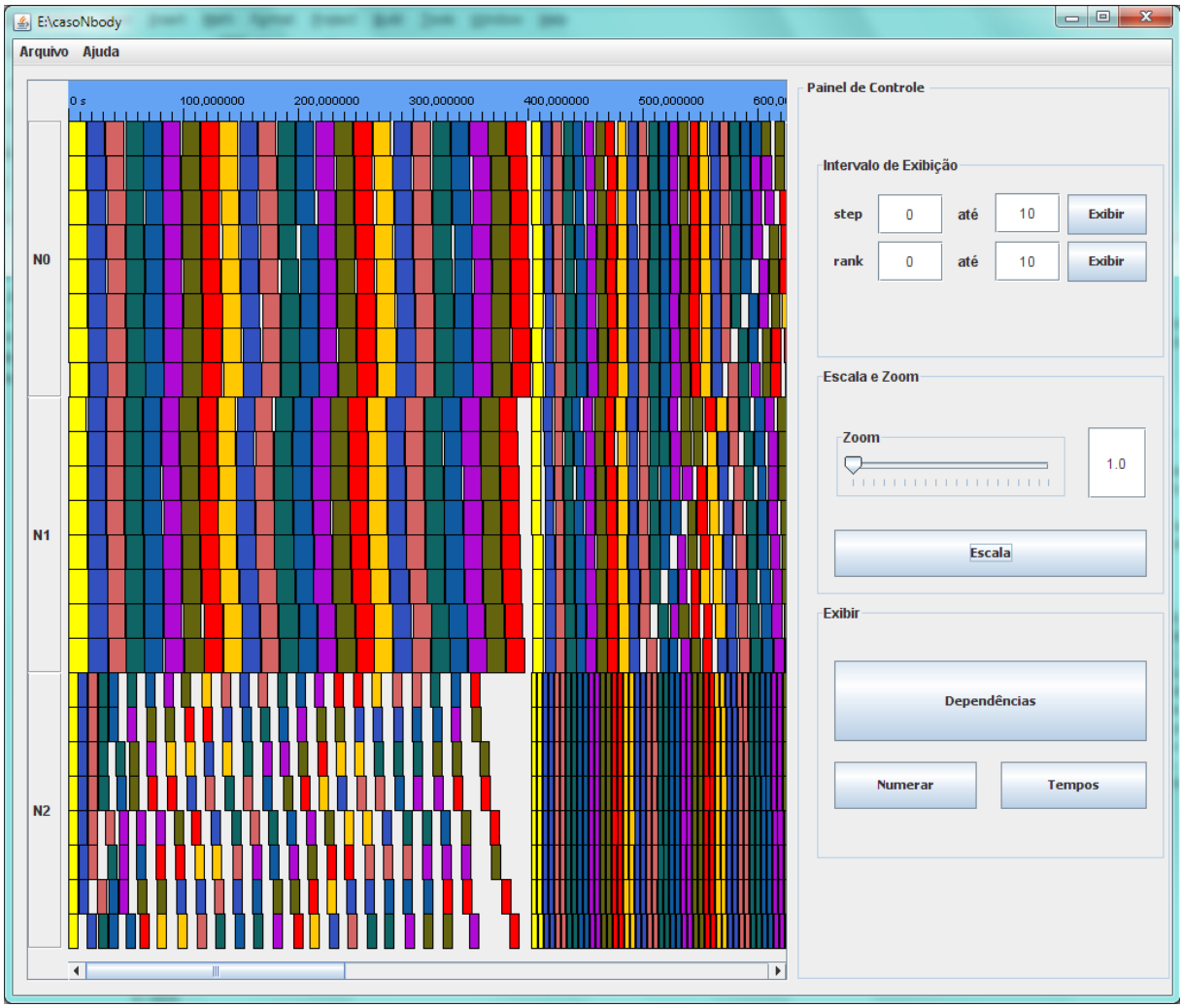


Figura B.1: Tela principal da ferramenta para monitoramento de aplicações paralelas

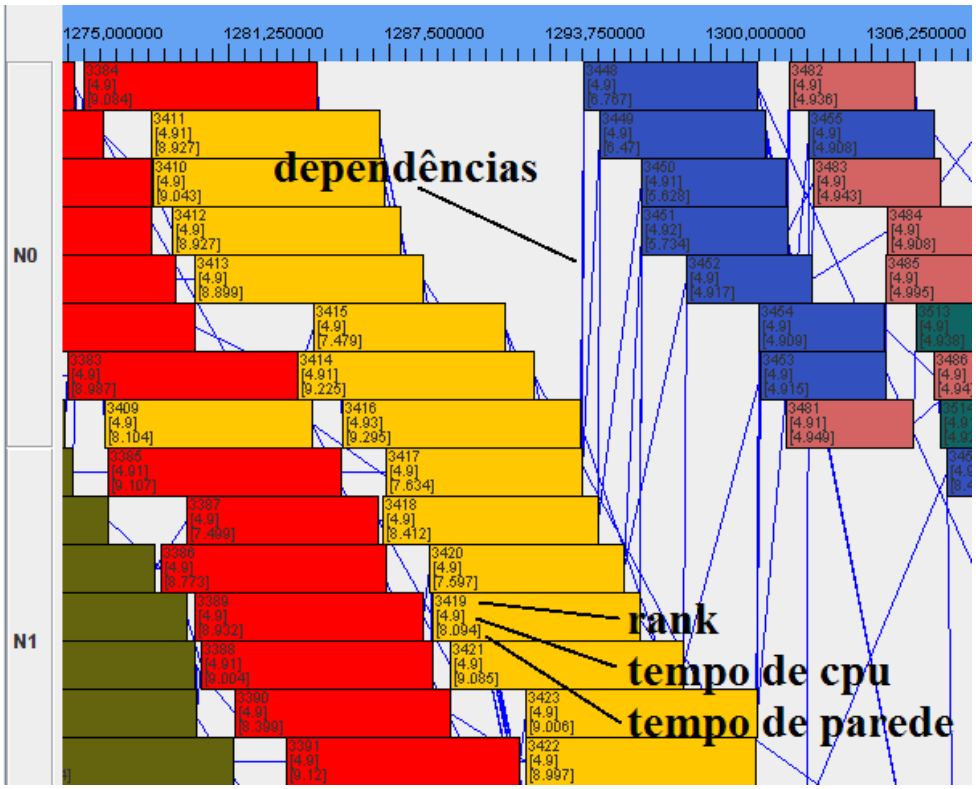


Figura B.2: Exemplo da ferramenta de monitoramento onde as dependências, o identificador número(*rank*), o tempo de CPU e o tempo de parede das tarefas estão sendo exibidos

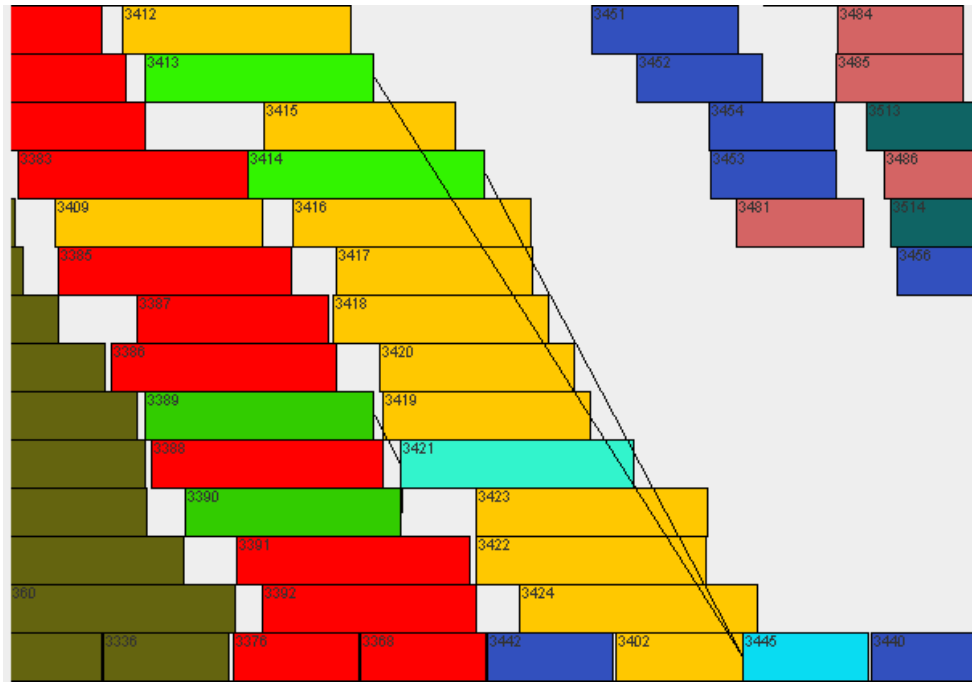


Figura B.3: Exemplo da ferramenta de monitoramento onde as predecessoras das tarefas 3445 e 3421 são indicadas