Universidade Federal Fluminense

THIAGO MAGALHÃES DE BRITO RODRIGUES

Escalonamento de Cargas Divisíveis em Clusters Computacionais

NITERÓI

2011

THIAGO MAGALHÃES DE BRITO RODRIGUES

Escalonamento de Cargas Divisíveis em Clusters Computacionais

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Redes de Computadores e Sistemas Distribuídos e Paralelos

Orientadora:

Maria Cristina Silva Boeres, Ph.D.

Universidade Federal Fluminense

NITERÓI

2011

Escalonamento de Cargas Divisíveis em Clusters Computacionais

Thiago Magalhães de Brito Rodrigues

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Redes de Computadores e Sistemas Distribuídos e Paralelos

Aprovada por:

Profª. Maria Cristina Silva Boeres, Ph.D. / IC-UFF
(Orientadora)

Prof. Eugene Francis Vinod Rebello, Ph.D. / IC-UFF

Profª. Silvana Rossetto, D.Sc. / DCC-UFRJ

Niterói, 16 de Março de 2011.

Para meus pais, amigos e ao meu grande amor.

Agradecimentos

Primeiramente à Deus, onde sempre depositei minhas esperanças e de onde sempre tirei forças para não desistir deste sonho.

Aos meus familiares, especialmente meus pais Elícia e Celso, que venceram inúmeros obstáculos para me proporcionarem uma bela educação.

À minha namorada Gardhênia, fonte de alegria, conforto e esperança, por estar sempre ao meu lado me dando forças e por me fazer entender que eu era capaz.

À professora Cristina Boeres, minha orientadora, que sempre me ajudou ao longo desta jornada e sem a qual este trabalho não existiria.

A toda equipe de professores da UFF, por toda sua competência e dedicação. Sinto orgulho em ter estudado aqui.

Agradeço também aos meus amigos, presentes e não presentes, pelos momentos de descontração, pelo incentivo e pelas ideias que contribuíram para esta dissertação.

Resumo

Diversas aplicações que processam uma massa significativa de volume de dados podem ser transformadas em aplicações paralelas compostas por tarefas independentes, chamadas de aplicações *Bag-of-Task* (BoT). Para isto, é necessário particionar os dados a serem processados em porções ou fatias de tamanhos menores e associá-las a tarefas ou processos independentes. Tais aplicações são classificadas como contendo uma carga de trabalho divisível e existem em diversos domínios da ciência, tais como análise de sequência de proteínas, simulação de fisiologia micro-celular, processamento de imagens e vídeos digitais, matemática computacional, entre outros. Aplicações de cargas divisíveis são tipicamente classificadas segundo as características de sua carga de trabalho, o que implica na necessidade de diferentes abordagens para que essas sejam processadas em um tempo computacional ótimo.

Diferentes heurísticas para divisão de carga presentes na literatura têm como foco particionar e distribuir sua carga de trabalho entre os recursos computacionais disponíveis de modo que essa seja executada no menor tempo possível. Todavia, tais heurísticas nem sempre levam em consideração características relevantes tanto da aplicação quanto dos recursos computacionais em suas metodologias de divisão de carga, o que muitas vezes resulta em uma má utilização do sistema alvo. Este trabalho de dissertação de mestrado tem como objetivo principal avaliar a execução de aplicações de cargas divisíveis considerando formas distintas de execução, variando aspectos como a granularidade dos processos, sob o gerenciamento do *middleware EasyGrid AMS*. Ainda, outros aspectos como uma utilização eficiente dos recursos de memória e seu compartilhamento em arquiteturas *multicore* são avaliados experimentalmente. Como resultados principais, os benefícios da utilização de um sistema gerenciador de aplicações inteligente são identificados e uma classificação das aplicações e metodologias de escalonamento de cargas divisíveis é traçada.

Abstract

Several applications that process large amounts of data can be transformed in parallel applications composed of independent tasks, the so called Bag-of-Task (BoT) applications. For doing so, it is necessary to partition the data to be processed into portions or chunks of smaller sizes and to assign them to tasks or independent processes. Such applications are classified as containing a divisible workload and they can be found various fields of science, such as protein sequence analysis, simulation of micro cellular physiology, imaging and digital video processing, computing mathematics, among others. Applications of divisible loads are typically classified according to the characteristics of their workload, which implies the need for different approaches to those processed in an optimal computational time.

Different load partitioning heuristics from the literature have focused on partitioning and distributing its workload among computing resources so that it runs in the shortest time possible. However, such heuristics do not always take into account characteristics of the application and computational resources in their load partition methods, which often results in a misuse of resources. This master dissertation aims to evaluate the performance of divisible load applications considering different ways of implementation, varying aspects like the granularity of processes, under the management of middleware EasyGrid AMS. Still, other aspects such as efficient use of memory resources, which may be shared, on multicore architectures, are evaluated experimentally. As the main results, the benefits of using an intelligent application management system are identified and a classification of applications and methodologies for scheduling divisible loads is drawn.

Palavras-chave

1. Arquiteturas *Multicore*
2. Aplicações de Cargas Divisíveis
3. Sistema Gerenciador de Aplicações *EasyGrid AMS*
4. Compartilhamento de Memória em *Clusters Multicore*

Glossário

AMS	:	Application Management System
DL	:	Divisible Loads
DLT	:	Divisible Load Theory
GM	:	Global Manager
HM	:	Host Manager
IC	:	Instituto de Computação
LAN	:	Local Area Network
MI	:	Multi Installment Algorithm
MRRS	:	Multi-Round Scheduling With Resource Selection
PTUMR	:	Parallel Transferable Uniform Multi Round
PWD	:	Performance-Based Workload Distribution
RAM	:	Random Access Memory
RUMR	:	Robust Uniform Multi Round
SM	:	Site Manager
SMP	:	Symmetric Multiprocessing
SPMD	:	Single Program-Multiple Data
UFF	:	Universidade Federal Fluminense
UMR	:	Uniform Multi-Round Algorithm
WAN	:	Wide Area Network

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Melhorias em infra-estrutura e tecnologia de redes de computadores tornaram possível a agregação de diferentes recursos computacionais em plataformas paralelas e distribuídas. Estas plataformas têm como fundamento prover capacidades de processamento que antes nunca existiram para aplicações computacionalmente intensivas. Porém, é necessário que seus recursos sejam explorados de maneira eficiente. A utilização destas plataformas para uma execução eficiente de aplicações computacionalmente intensivas depende da definição de um bom algoritmo de escalonamento, que especifique o montante de trabalho a ser executado por cada recurso e também a quantidade destes recursos que deverão ser utilizados.

Aplicações que requerem grande poder de processamento são encontradas em diversos domínios da ciência. Dentre tais aplicações, estão as de cargas divisíveis (DL) [31, 37, 38], as quais possuem uma entrada ou carga de trabalho que pode ser dividida em fatias que contenham um número arbitrário de unidades. Estas fatias podem então ser enviadas para recursos computacionais distintos e processadas independentemente, sem que seja necessário algum tipo de sincronização. Tal característica permite que esse tipo de aplicação seja perfeitamente adequável ao modelo de execução *mestre/trabalhador*, onde um processo mestre (o qual é o detentor da carga de trabalho a ser escalonada) envia fatias de carga para serem executadas por um conjunto de processos trabalhadores, coletando seus resultados ao final da execução. Aplicações de cargas divisíveis se assemelham à aplicações *Bag-of-Task* (BoT), diferenciado-se apenas pelo fato de uma carga divisível poder ser particionada em um número qualquer de fatias, o que nem sempre é válido para aplicações BoT.

Muitas aplicações recaem na categoria de cargas divisíveis, como por exemplo, aplicações de bioinformática, como HMMER [1], a qual recebe como entrada uma sequência DNA/proteína e busca em um arquivo de dicionário, contendo milhões de sequências, conjuntos de sequências ou subsequências similares. O arquivo de dicionário pode ser arbitrariamente dividido em diversas fatias e cada sequência DNA/proteína representaria uma unidade de carga. Um outro exemplo de aplicação é a de compressão de vídeo MPEG [35], onde um dado vídeo de entrada é composto por um grande número de quadros, sendo cada quadro uma unidade de carga, podendo ser processado independentemente. Outros exemplos incluem processamento digital de imagens [23], multiplicação de matrizes [12], análises de dados de radares [30] e *Datamining* [20].

Aplicações de cargas divisíveis são tipicamente classificadas segundo as características de sua carga de trabalho, a qual pode ser definida como sendo regular, onde todas as unidades de carga demandam o mesmo tempo de processamento, ou irregular, onde as unidades de carga possuem tempos computacionais distintos. Tanto as cargas de trabalho regulares quanto as irregulares são ainda agrupadas em duas sub-classes, as de natureza *data-intensive* e as de natureza *cpu-intensive*, sendo as aplicações *data-intensive* aquelas em que grandes volumes de dados são manipulados, gastando boa parte do seu tempo de processamento com acesso a estes. Já para aplicações *cpu-intensive*, a maior parte do tempo de processamento decorre da grande quantidade de operações executadas. As diferentes classes de cargas divisíveis são ilustradas na Figura 1.1.

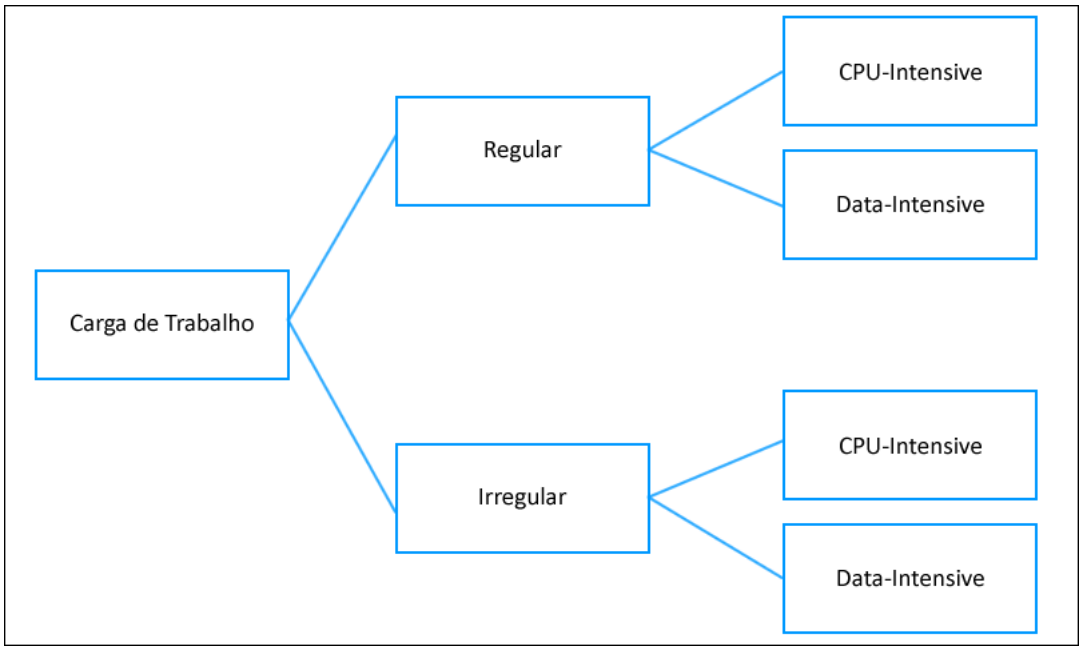


Figura 1.1: Classificação segundo a natureza da carga de trabalho.

Neste trabalho de dissertação de mestrado será feita uma análise de metodologias de escalonamento que consideram os princípios de divisão de carga. Serão também executadas aplicações de cargas divisíveis de diferentes classes e em diferentes cenários computacionais, visando coletar informações que permitam processá-las de maneira mais eficiente em *clusters multicore*. Por fim, os benefícios da utilização do sistema gerenciador de aplicações *EasyGrid AMS* [9, 10, 11, 13, 14, 15, 17, 18] no processamento de cargas divisíveis também serão abordados.

O gerenciador *EasyGrid AMS* é integrado ao código de aplicações MPI tradicionais, tornando-as *system-aware* ou autônomas. Aplicações autônomas são responsáveis pela sua própria gerência e buscam se auto-adaptar às mudanças ocorridas no ambiente. Dentre as funcionalidades disponibilizadas pelo *EasyGrid AMS* destacam-se sua capacidade de escalonamento dinâmico de tarefas [18] e de recuperação em caso de falhas de processos *e/ou* processadores [14]. Ainda, a especificação de um modelo de execução alternativo proposto em [13] para aplicações de diferentes classes a serem executadas pelo *EasyGrid AMS* tem levado a um melhor desempenho.

1.1 Objetivos

O objetivo principal deste trabalho é traçar aspectos que levem aplicações de cargas divisíveis executadas em *clusters* computacionais a atingirem um melhor desempenho. Tais aplicações serão executadas com o auxílio do gerenciador *EasyGrid AMS* [9, 10, 11, 13, 14, 15, 17, 18]. Para atingir esse objetivo, serão incorporadas ao *EasyGrid AMS* funcionalidades que visam disponibilizar para a aplicação de divisão de carga um conjunto de informações que possam auxiliar sua execução. Essas informações estão relacionadas à divisão da carga inicial a ser processada e à seleção dos recursos a serem utilizados em seu processamento.

Existe na literatura um conjunto de algoritmos que visam executar aplicações de cargas divisíveis em um tempo computacional ótimo. Especificamente, os algoritmos UMR [43, 42, 44] e MRRS [25, 26, 27, 24] serão incorporados ao *EasyGrid AMS* e suas informações serão disponibilizadas para a aplicação executada sob seu domínio. A principal característica desses algoritmos é que por meio de uma formulação matemática sólida, é possível encontrar uma maneira de dividir a carga de trabalho a ser processada entre os recursos disponíveis de tal modo que seu tempo de execução, ou *makespan*, seja próximo ao ótimo. Em relação aos recursos a serem utilizados, políticas de seleção terão como fi-

nalidade buscar dentre os recursos disponibilizados para o ambiente de execução, aqueles que possam finalizar a execução da aplicação no menor tempo possível. Para auxiliar tais políticas de seleção, a camada de monitoramento do *EasyGrid AMS* [9, 10] é acionada com o objetivo de prever o quão sobrecarregado cada um dos recursos se encontra.

Uma variante dos algoritmos de divisão de carga, encontrados na literatura e estudados neste trabalho, também é proposta. Uma análise experimental em todas as metodologias aqui construídas tem como objetivo avaliar o comportamento das aplicações de cargas divisíveis ao se utilizar tarefas contendo granularidades distintas, e seu impacto na utilização da plataforma computacional. A granularidade de uma tarefa representa a quantidade de trabalho que o recurso ao qual está alocada deverá executar. Tarefas de granularidades grossas tendem a consumir uma grande quantidade de recursos de memória, principalmente em aplicações *data-intensive*, gastando boa parte do tempo de processamento em acesso a sua massa de dados. Por outro lado, a utilização de tarefas de granularidades finas implica na existência de um número maior de tarefas e consequentemente, em um aumento do custo de gerenciamento da aplicação, devido a fatores como trocas de contexto e custos de criação de tais tarefas. Este trabalho irá avaliar o desempenho das aplicações de cargas divisíveis ao se utilizar tarefas tanto de granularidades finas, quanto de granularidades grossas em arquiteturas atuais como clusters de *multicores*.

O padrão MPI [2] ainda não oferece mecanismos para balanceamento automático de carga de trabalho, isto é, da quantidade de trabalho executada por cada recurso presente na plataforma computacional, e nem um controle eficiente que minimize a ociosidade de seus recursos. Estes mecanismos ficam a cargo dos desenvolvedores de aplicações. Neste trabalho, mecanismos que dividem a carga inicial de trabalho entre os processos trabalhadores foram adicionados ao sistema gerenciador de aplicações *EasyGrid AMS* para facilitar o desenvolvimento de aplicações de cargas divisíveis. Após a inserção destes mecanismos, serão verificados seus resultados por meio da execução de um grupo de aplicações de cargas divisíveis como:

- Algoritmos para processamento de imagens [45];
- Mandelbrot set computation fractal [28];
- Multiplicação de matrizes [12].

Por fim, os resultados mensurados serão apresentados a fim de verificar a validade da metodologia utilizada neste trabalho, considerando *clusters* de *multicores*.

1.2 Organização do Trabalho

O restante deste trabalho de dissertação de mestrado está organizado da seguinte forma: no Capítulo 2 é apresentada uma definição e classificação de aplicações de cargas divisíveis. No Capítulo 3 é feita uma descrição detalhada de um conjunto de trabalhos relacionados. O Capítulo 4 apresenta o sistema gerenciador de aplicações *EasyGrid AMS* e especifica o conjunto de novas funcionalidades relacionadas a aplicações de cargas divisíveis, desenvolvidas neste trabalho. No Capítulo 5 será verificada a validade da metodologia proposta além da apresentação dos resultados obtidos com a execução de aplicações de cargas divisíveis gerenciadas pelo *EasyGrid AMS*. Conclusões e propostas de trabalhos futuros serão discutidas no Capítulo 6.

Capítulo 2

Execução de Aplicações de Cargas Divisíveis

Este capítulo tem como objetivo descrever o conjunto de modelos, conceitos e notações utilizadas na definição e execução de aplicações de cargas divisíveis. Primeiramente, serão apresentadas as classes de algoritmos para processamento de aplicações de cargas divisíveis e seus respectivos modelos de divisão de carga. Na sequência, será definido o modelo da plataforma computacional utilizada para a execução de tais aplicações. Por fim, será descrito o modelo de execução alternativo *1PTask*, utilizado pelo *EasyGrid AMS*, o motivo de sua utilização e os resultados esperados.

2.1 Algoritmos de Escalonamento de Cargas Divisíveis

Em um problema de escalonamento de aplicações de cargas divisíveis, é dada uma carga de trabalho inicial de W_{total} unidades. Todas as unidades que compõem esta carga são independentes entre si e podem ser processadas por recursos computacionais distintos. Logo, W_{total} pode ser particionada em um número arbitrário de fatias, as quais serão processadas por estes recursos sem nenhuma restrição de precedência. Na literatura do problema de cargas divisíveis [7, 8, 19, 22, 24, 25, 26, 27, 32, 36, 39, 42, 43, 44] podemos encontrar duas classes de algoritmos heurísticos para este problema: A primeira, engloba os algoritmos de uma iteração ou *1-round* [8, 19, 32, 39]. Estes algoritmos são caracterizados por dividirem toda a carga de trabalho entre os processadores disponíveis em uma única iteração ou *round*, onde geralmente a carga de trabalho é dividida em tantas fatias quanto processadores disponíveis [8, 19, 32]. A principal limitação desta classe de heurísticas é que ao se particionar a carga de trabalho em uma única iteração, poderá resultar em fatias de grandes tamanhos e que demandam um tempo de envio potencialmente longo. Isto

faz com que principalmente os últimos recursos a receberem suas fatias de carga fiquem ociosos durante a etapa de transmissão para os primeiros recursos. Com o intuito de resolver essa ociosidade, foi proposta uma segunda classe de algoritmos para processamento de aplicações de cargas divisíveis, que é composta por algoritmos de múltiplas iterações ou *multi-rounds* [24, 25, 26, 27, 42, 43, 44]. A característica fundamental destes algoritmos é particionar a carga de trabalho em várias iterações, objetivando sobrepor o tempo de computação das fatias de carga com a transmissão destas para os recursos trabalhadores.

Algoritmos de múltiplas iterações assumem o pressuposto de que recursos computacionais podem processar uma fatia da carga de trabalho ao mesmo tempo em que recebem novas fatias para serem processadas em sequência. Esta característica pode ser observada em sistemas de computadores tradicionais, como por exemplo os computadores pessoais, onde interfaces de rede podem realizar um determinado conjunto de operações ao mesmo tempo em que o processador executa outro conjunto distinto. Desta forma, ocorre uma sobreposição entre os tempos de computação e comunicação, evitando que os recursos fiquem ociosos durante todo o intervalo de tempo necessário para a transmissão da carga de trabalho.

O primeiro algoritmo de múltiplas iterações com foco na minimização do tempo de execução, ou *makespan*, de uma aplicação foi o *Multi Instalment* (MI), proposto em [7], para plataformas de recursos computacionais homogêneos. O algoritmo MI inicia com o envio de pequenas fatias, e então incrementa seus tamanhos ao longo da execução da aplicação para alcançar uma efetiva sobreposição de computação com comunicação. Trabalhos sucessores, como os algoritmos UMR [43, 42, 44] e MRRS [24, 25, 26, 27], melhoraram os resultados obtidos em [7] ao considerarem as sobrecargas associadas a computação e comunicação. Estas sobrecargas estão relacionadas ao tempo de transferência dos dados de *entrada/saída* dos recursos computacionais (como por exemplo, o tempo gasto desde o momento em que uma dada informação esteja pronta para ser enviada pela rede, até o momento em que é recebida no recurso de destino) e aos atrasos ocorridos durante a inicialização da transmissão e da computação dos dados (como por exemplo, o tempo gasto para se estabelecer uma conexão de rede entre dois recursos ou *handshake*, o tempo decorrido para a leitura ou escrita de um arquivo em disco, ou mesmo o tempo gasto com algum pré-processamento da carga a ser distribuída para os recursos trabalhadores). A consideração de tais sobrecargas traz a tona a questão do número ótimo de iterações em que a carga de trabalho deverá ser fracionada. UMR e MRRS solucionam esta questão tanto para ambientes com recursos homogêneos quanto com recursos heterogêneos. Por fim, os algoritmos de múltiplas iterações UMR e MRRS enviam fatias da carga de trabalho

para os recursos computacionais de acordo com suas respectivas capacidades de processamento, enviando uma menor quantidade de trabalho para os recursos com menor poder computacional. Esta estratégia de divisão de carga permite que estes algoritmos sejam utilizados tanto em sistemas de recursos homogêneos, quanto em sistemas de recursos heterogêneos.

De acordo com o princípio de otimalidade do problema de divisão de carga, definido em [31, 37, 38], para que a utilização da plataforma de execução seja maximizada todos os recursos envolvidos no processamento da carga de trabalho devem finalizar suas execuções no mesmo instante de tempo. Tanto o algoritmo UMR quanto o MRRS abordam esta questão em suas respectivas metodologias de divisão de carga. A Figura 2.1 contém um exemplo que ilustra a diferença entre o comportamento de um algoritmo de múltiplas iterações e um de uma iteração, onde uma mesma carga de trabalho é particionada entre um conjunto de n recursos computacionais $\{p_1, p_2, ..., p_n\}$. A faixa em branco presente na Figura 2.1 representa o tempo ocioso no qual cada recurso aguarda pela chegada de trabalho, ou devido ao fato de finalizar sua execução enquanto os demais recursos ainda se encontram em processamento. O tempo de processamento da carga de trabalho total está diretamente relacionado à esta espera ociosa.

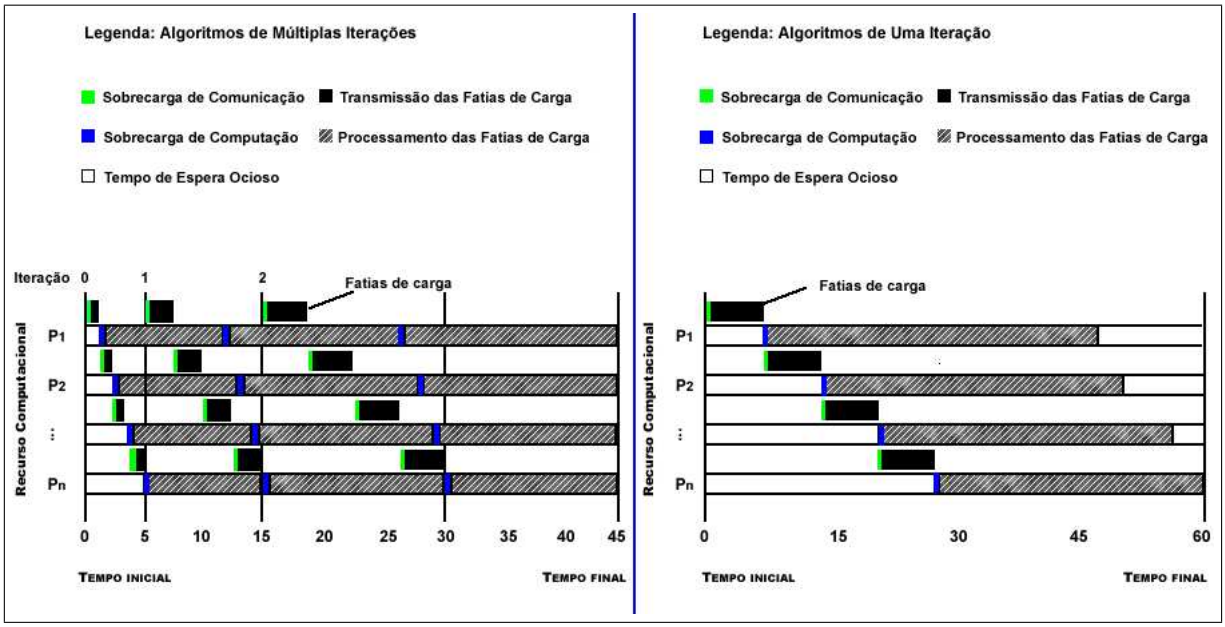


Figura 2.1: Comparativo entre algoritmos de uma iteração e de múltiplas iterações

2.2 Modelo da Arquitetura de Execução

O modelo da plataforma de execução é constituído de um conjunto de $n+1$ recursos computacionais $P = \{p_0, p_1, p_2, ..., p_n\}$ que estão conectados através de uma rede em *topologia estrela*, conforme ilustrado na Figura 2.2. Cada recurso possui um determinado poder computacional e os enlaces de rede que os conectam podem possuir diferentes larguras de banda.

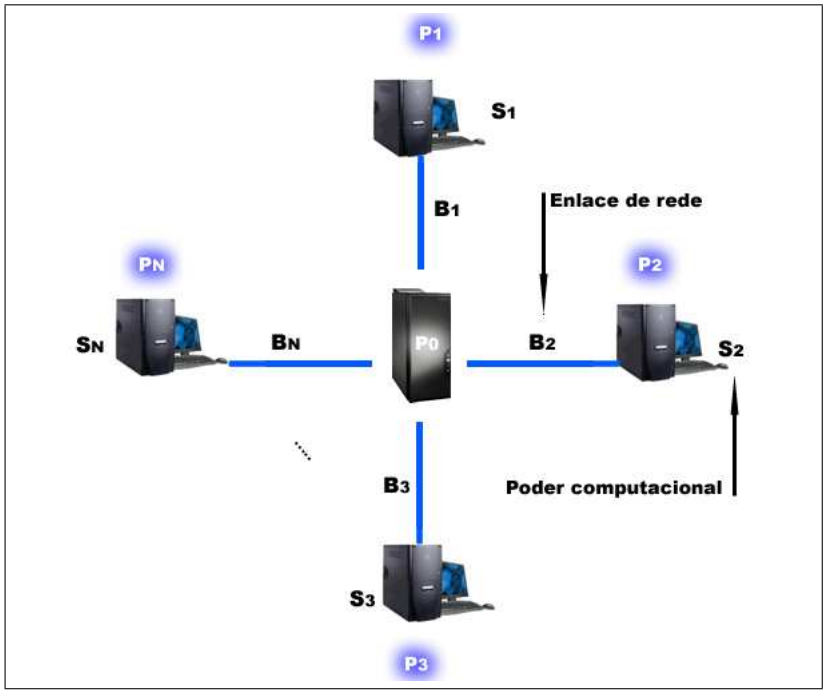


Figura 2.2: Plataforma de execução

Cada recurso computacional p_i tem associado um determinado poder computacional S_i, que representa a quantidade de unidades de carga que este recurso pode processar em uma unidade de tempo. Além disso, este modelo também identifica uma sobrecarga computacional fixa co_i (*computing overhead*) associada à inicialização da computação (como por exemplo, a inicialização de um processo remoto) no processador p_i.

Ainda, em uma *topologia estrela*, um enlace de comunicação une o recurso p_0 a cada um dos recursos p_i restantes. Estes enlaces possuem uma largura de banda B_i, que representa a quantidade de unidades de carga que podem ser transmitidas de p_0 para p_i em unidade de tempo. Também é identificada uma sobrecarga no_i (*network overhead*), em unidades de tempo, incorrida no início desta transferência de dados (como por exemplo, para pré-processar dados de entrada da aplicação e/ou iniciar uma conexão TCP).

O modelo de comunicação entre p_0 e os demais processadores segue as definições do modelo *one-port*, especificado em [5]. De acordo com o modelo *one-port*, um determinado recurso pode enviar ou receber uma mensagem em um determinado instante de tempo, mas nunca realizar os dois eventos de comunicação simultaneamente. Em outras palavras, o recurso p_0 pode somente enviar ou receber dados *de/para* qualquer recurso $\{p_1, p_2, ..., p_n\}$ em um dado instante de tempo.

2.3 Modelo de Divisão de Carga Para Algoritmos de Múltiplas Iterações

Aplicações de cargas divisíveis são tradicionalmente executadas conforme o modelo de execução *mestre/trabalhador*. A tarefa mestre, denotada por v_0, é executada no processador p_0 e envia fatias de carga para as tarefas trabalhadoras $v_i \in \{v_1, v_2, ..., v_n\}$, executadas em $p_i \in \{p_1, p_2, ..., p_n\}$. Os processos ou tarefas trabalhadoras v_i não trocam informações entre si, sendo ditos independentes. Além disso, estes processos são tipicamente de longa duração e são criados estaticamente no início da execução da aplicação. Conforme apresentado em [13, 34], neste modelo denominado por *1PProc* (um processo por processador), cada processador disponível executa apenas um processo ao longo de toda a execução da aplicação, sendo que em aplicações de cargas divisíveis este processo v_i é encarregado de receber e processar suas fatias em cada iteração da divisão de carga.

Seja W_{total} a carga de trabalho a ser processada, a qual reside em p_0. No problema de divisão de cargas, um algoritmo de escalonamento deve decidir o quão grande uma porção ou fatia da carga de trabalho será enviada para cada recurso computacional. Ainda, é importante especificar a quantidade de recursos que possa processar esta carga em um tempo ótimo.

De acordo com as definições dos algoritmos de múltiplas iterações UMR e MRRS, cada fatia de carga a ser processada por v_i (no processador p_i) durante cada iteração j é denotada por $chunk_{i,j}$. Assim, o tempo requerido para que o processador p_i realize a computação da fatia de carga $chunk_{i,j}$ é dado por

$$Tcomp_{i,j} = co_i + chunk_{i,j}/S_i,$$

onde co_i representa a sobrecarga de inicialização da computação em p_i. Similarmente, o tempo despendido para que sejam enviadas $chunk_{i,j}$ unidades de carga de v_0 em p_0 para

v_i em p_i é especificado por

$$Tcomm_{i,j} = no_i + chunk_{i,j}/B_i,$$

sendo que no_i representa a sobrecarga de comunicação entre v_0 e v_i. Este modelo é flexível o suficiente para que possa ser instanciado para representar diversos tipos de redes de interconexão, além disso, a consideração das sobrecargas de computação co_i e de comunicação no_i tornam a modelagem mais realística [43]. Por fim, conforme o modelo *one-port* de comunicação para o problema de escalonamento de aplicações de cargas divisíveis, uma tarefa trabalhadora v_i não pode inicializar o processamento da referida fatia $chunk_{i,j}$ antes de recebê-la completamente de v_0. Similarmente, v_i não pode enviar resultados de volta para v_0 antes de finalizar o processamento de toda a fatia de carga $chunk_{i,j}$ associada a ele.

O principal foco da estratégia de divisão de carga em múltiplas iterações é permitir às tarefas trabalhadoras inicializarem suas execuções o mais breve possível, evitando assim que fiquem ociosas durante a etapa de transmissão da carga de trabalho W_{total}. Para atingir este objetivo, a tarefa mestre envia inicialmente uma pequena fatia de carga $chunk_{i,0}$ para cada processo trabalhador v_i durante a primeira iteração da divisão de carga (iteração zero). Na sequência, ao longo das iterações seguintes, v_0 envia novas fatias de tal forma que o tempo necessário para que o último processo trabalhador v_n processe sua fatia de carga em uma determinada iteração j seja igual ao tempo necessário para que o processo mestre envie todas as novas fatias da iteração seguinte $j+1$ para os n processos trabalhadores. Isso pode ser formulado por

$$Tcomp_{j,n} = \sum_{i=1}^{n} Tcomm_{j+1,i}.$$

Desta forma, ocorre uma sobreposição entre os tempos de processamento e transmissão das fatias de carga. Este método é executado até que toda a carga de trabalho seja particionada entre os processos trabalhadores. Assim, sempre que um processo trabalhador finalizar a execução de uma determinada fatia de carga em uma iteração j, poderá imediatamente inicializar a execução da próxima fatia, relativa a iteração $j+1$, a qual já foi recebida, mantendo a utilização da plataforma computacional maximizada [42, 43, 44]. A Figura 2.3 ilustra a metodologia de divisão de carga dos algoritmos de múltiplas iterações, onde são introduzidas as sobrecargas de computação e de comunicação, além do tempo despendido para o envio e processamento de cada fatia de carga em cada iteração.

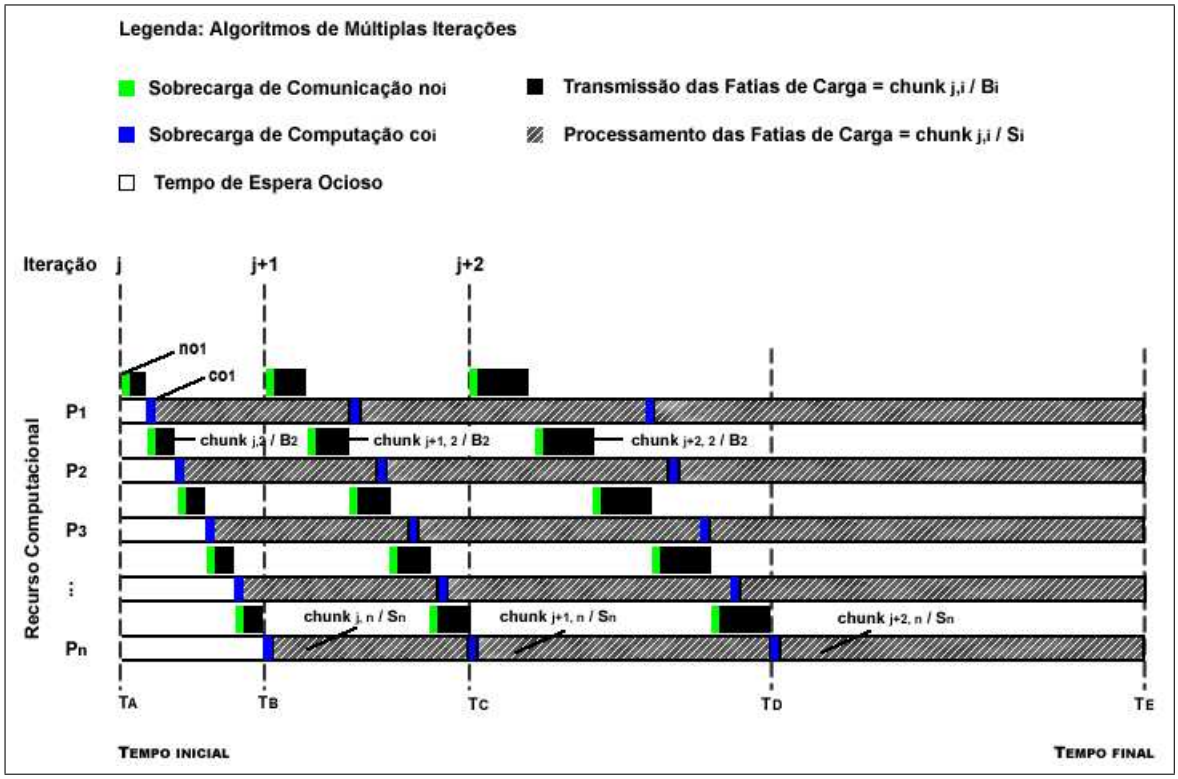


Figura 2.3: Estratégia de divisão de carga para algoritmos de múltiplas iterações

2.4 Modelo de Execução Alternativo 1PTask

Aplicações paralelas de alto desempenho, como as de cargas divisíveis, tradicionalmente adotam o modelo de execução tradicional *1PProc* (um processo por processador). A principal vantagem oferecida pelo modelo *1PProc* é a sua simplicidade em relação ao desenvolvimento de aplicações paralelas. Porém, este modelo tradicional também apresenta algumas limitações, entre elas estão a quantidade limitada de processos que podem ser criados estaticamente durante o início da execução da aplicação e a dificuldade de recuperação destes processos em caso de falha, uma vez que provavelmente todo o processo deverá ser re-executado do início [14].

Tendo em vista as limitações do modelo *1PProc*, o trabalho apresentado em [13] propõe um modelo alternativo para execução de aplicações paralelas, denominado *1PTask* (um processo por tarefa). No modelo *1PTask*, um número maior de *processos/tarefas* por processador, que consequentemente apresentam uma menor granularidade, é especificado. O número de processos definidos vem de encontro com a exploração do paralelismo da aplicação, sendo que fatalmente um número maior de processos com granularidade menor do que aqueles em *1PProc* são definidos. Entre as principais características do modelo

1PTask está a sua independência em relação a arquitetura do ambiente de execução, pois este não limita o número de processos da aplicação ao número de recursos disponíveis. Uma outra vantagem estabelecida neste modelo é a melhora de desempenho em ambientes compartilhados através do re-escalonamento de processos sem a necessidade de migração de tarefas [11, 18]. Em um ambiente compartilhado, os recursos computacionais são disputados por diferentes aplicações.

Para o problema de escalonamento de aplicações de cargas divisíveis estudado neste trabalho, será utilizado o modelo de execução alternativo *1PTask*. Cada porção de carga $chunk_{i,j}$ a ser entregue para cada processador p_i em uma iteração j será processada por uma tarefa independente $v_{i,j}$. Estas tarefas são criadas a partir do momento em que sua carga de trabalho se encontra disponível (a tarefa mestre, denotada por $v_{0,0}$, conclui o envio de sua fatia de carga) e tendem a apresentar granularidades menores do que as observadas com o modelo tradicional *1PProc*. Inicialmente, a tarefa mestre $v_{0,0}$, a qual especifica a carga de trabalho a ser processada, determina quais recursos serão selecionados e suas respectivas cargas a serem processadas. A diferença entre as tarefas especificadas nos recursos trabalhadores e aquelas definidas na literatura [5, 24, 25, 26, 27, 42, 43, 44], é que neste trabalho as tarefas são criadas dinamicamente no momento em que sua carga está disponível (foi completamente recebida) no recurso a ela alocada. Ao final do processamento da fatia de carga em cada iteração, a tarefa é finalizada. Na Figura 2.4 podemos observar as principais diferenças entre os modelos *1PProc* e *1PTask* para o processamento de aplicações de cargas divisíveis.

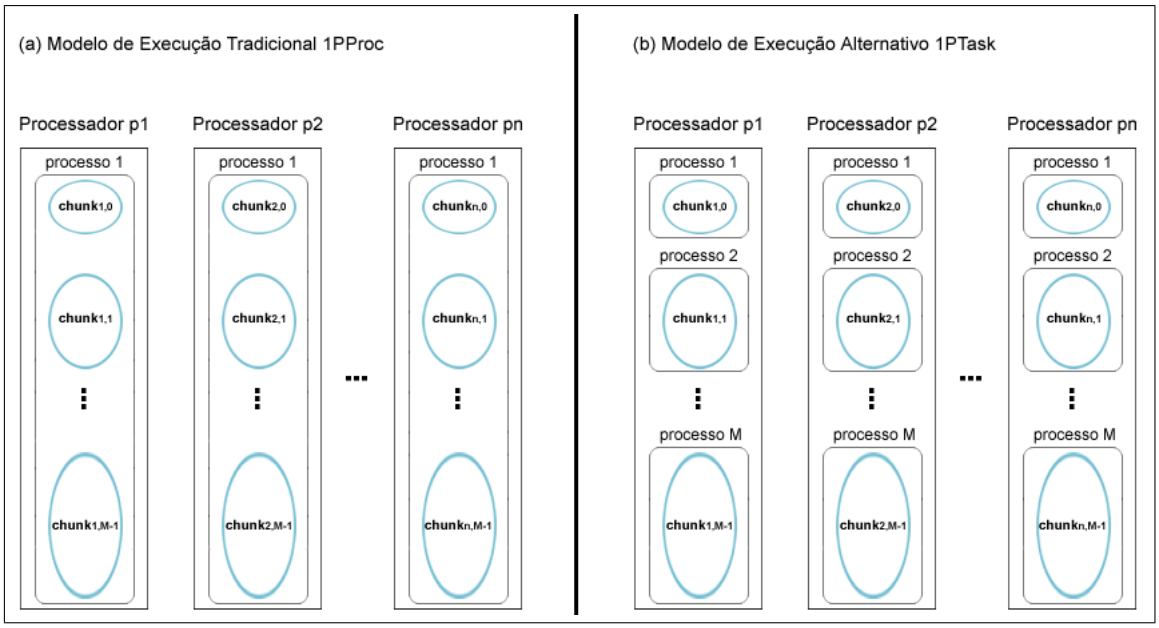


Figura 2.4: Comparativo entre os modelo de execução 1PProc e 1PTask

Conforme especificado na Figura 2.4, para o modelo de execução tradicional *1PProc*, em um ambiente composto por $n+1$ recursos computacionais (*topologia estrela*) e que contenha uma carga de trabalho de W_{total} unidades a ser particionada em M iterações, deverá possuir $n+1$ processos em execução, sendo n o número de processos ou tarefas trabalhadoras v_i, $1 \leq i \leq n$, e um processo mestre v_0. Cada processo v_i é criado estaticamente no início da execução da aplicação, sendo finalizado somente quando W_{total} for completamente processada. Já com o modelo de execução alternativo *1PTask*, para este mesmo ambiente, com esta mesma carga de trabalho e mesma quantidade de iterações, $(M \times n)+1$ processos serão criados ao longo da execução da aplicação ($M \times n$ tarefas trabalhadoras $v_{i,j}$, $1 \leq i \leq n$ e $0 \leq j \leq M-1$, e uma tarefa mestre $v_{0,0}$), sendo que em cada iteração n processos são criados dinamicamente, permanecendo ativos somente durante o processamento de suas devidas fatias de trabalho.

Este trabalho de dissertação de mestrado propõe ainda avaliar o impacto da definição do modelo *1PTask*, ou seja, a criação de processos de menor granularidade, na exploração do uso eficiente da hierarquia de memória em arquiteturas atuais, como os *clusters de multicores*. Nessas arquiteturas, os recursos de memória disponíveis, como por exemplo as memórias RAM e *cache*, são geralmente compartilhados entre um conjunto de núcleos e processadores, sendo que a utilização eficiente de tais recursos se torna um problema desafiador. Neste trabalho, a granularidade de uma tarefa está relacionada a quantidade de unidades carga que essa deverá processar, consequentemente, tarefas de granularidades menores processam fatias de carga menores e trabalham com faixas de endereços de memória menores. Resultados experimentais demostraram que para aplicações compostas por cargas de trabalho irregulares ou regulares e de natureza *data-intensive*, a utilização de tarefas de baixa granularidade executadas segundo o modelo *1PTask* produziram resultados interessantes, proporcionando uma utilização mais eficiente dos recursos de memória e reduzindo o tempo de processamento das aplicações de divisão de carga.

2.5 Estrutura dos Algoritmos de Múltiplas Iterações

Os pseudocódigos apresentados a seguir especificam os passos executados pelos algoritmos de divisão de carga em múltiplas iterações UMR e MRRS. Esses algoritmos utilizam a mesma estrutura para divisão de sua carga de trabalho, diferenciando apenas na especificação das funções que as compõem. Primeiramente, será especificado o pseudocódigo para o modelo de execução tradicional *1PProc*. Em seguida, para o modelo de execução alternativo *1PTask*.

2.5.1 Pseudo-algoritmo para o Modelo de Execução 1PProc

Os algoritmos de múltiplas iterações iniciam por buscar o melhor conjunto de recursos computacionais para o processamento de sua carga de trabalho W_{total}. Em seguida, por meio de suas respectivas modelagens matemáticas, é calculado o número de iterações M em que W_{total} deverá ser escalonada. Na sequência, em cada iteração j uma determinada fatia de carga $chunk_{i,j}$ é entregue para ser executada por cada processo trabalhador v_i alocado ao processador p_i. No modelo de execução tradicional *1PProc*, apresentado pelo Algoritmo 1, $n+1$ processos trabalhadores são criados estaticamente no início da aplicação e permanecem ativos durante todo o processamento da carga de trabalho.

Algoritmo 1 : Divisão de carga em múltiplas iterações 1PProc

```
1:  Início
2:    recursosSelecionados = Recursos_Selecionados();
3:    M = Numero_Iteracoes();
4:    // Processo mestre v_0
5:    se( processo == v_0 )
6:    {
7:      para(j = 0; j < M; j++)
8:      {
9:        para(i = 1; i ≤ recursosSelecionados; i++)
10:       {
11:         chunk_{i,j} = Tamanho_Fatia(i, j);
12:         Envia(chunk_{i,j}, v_i);
13:       }
14:     }
15:   }
16:   // Processo trabalhador v_i, 1 ≤ i ≤ n
17:   senão
18:   {
19:     para(j = 0; j < M; j++)
20:     {
21:       Recebe(chunk_{i,j}, v_0);
22:       Processa(chunk_{i,j});
23:     }
24:   }
25: Fim
```

2.5.2 Pseudo-algoritmo para o Modelo de Execução 1PTask

O Algoritmo 2 especifica os passos a serem seguidos por um algoritmo de múltiplas iterações sob o modelo de execução *1PTask*, onde $(M \times n) + 1$ processos são criados dinamicamente. Destes, $M \times n$ representam os processos trabalhadores $v_{i,j}$, que existem

somente durante o processamento de sua respectiva fatia da carga de trabalho $chunk_{i,j}$. Note que, para cada iteração $j, 0 \leq j \leq M-1$, apenas n processos trabalhadores são criados. Conforme estes processos vão finalizando suas execuções, novos processos são criados, mantendo a utilização dos recursos computacionais aos quais estão alocados maximizada.

Algoritmo 2 : Divisão de carga em múltiplas iterações 1PTask

```
1:  Início
2:    recursosSelecionados = Recursos_Selecionados();
3:    M = Numero_Iteracoes();
4:    // Processo mestre v_{0,0}
5:    se( processo == v_{0,0} )
6:    {
7:      para(j = 0; j < M; j++)
8:      {
9:        para(i = 1; i ≤ recursosSelecionados; i++)
10:       {
11:         chunk_{i,j} = Tamanho_Fatia(i, j);
12:         Envia(chunk_{i,j}, v_{i,j});
13:       }
14:     }
15:   }
16:   // Processo trabalhador v_{i,j}
17:   senão
18:   {
19:     Recebe(chunk_{i,j}, v_0);
20:     Processa(chunk_{i,j});
21:   }
22: Fim
```

2.6 Resumo

Neste capítulo, o conjunto de modelos utilizados para definição e execução de aplicações de cargas divisíveis foram formalizados. Em seguida, foram apresentadas as classes de algoritmos para processamento de cargas divisíveis e diversos de seus aspectos foram levantados. Dentre tais aspectos, encontram-se suas metodologias de divisão de carga e seus respectivos pseudocódigos. Foram também introduzidas as definições dos modelos de execução tradicional *1PProc* e alternativo *1PTask*.

No próximo capítulo serão formalizadas as estratégias de escalonamento de cargas divisíveis de trabalhos relacionados.

Capítulo 3

Trabalhos Relacionados

Este capítulo formaliza o conjunto de algoritmos para escalonamento de aplicações de cargas divisíveis que serão utilizados ao longo deste trabalho. Primeiramente, serão definidas as modelagens matemáticas adotadas pelos algoritmos UMR e MRRS, suas políticas de divisão de carga e de seleção de recursos. Estes algoritmos se destacam por produzirem um escalonamento próximo ao ótimo para aplicações de cargas divisíveis, visando obter um menor tempo de execução e uma melhor utilização dos recursos computacionais disponíveis. Em seguida, outros algoritmos de divisão de carga serão apresentados.

3.1 O Algoritmo UMR

O algoritmo *Uniform Multi-Round* (UMR) proposto em [43], utiliza o enfoque de divisão de carga em múltiplas iterações. Através de sua formulação matemática, um número próximo ao ótimo de iterações é derivado e em cada iteração, uma fatia da carga de trabalho é especificada e processada por cada *processo/tarefa* trabalhadora v_i executada em p_i, $1 \leq i \leq n$, de acordo o modelo de execução tradicional *1PProc*.

UMR despacha trabalho para as tarefas trabalhadoras em múltiplas iterações, com a restrição de que tais iterações tenham tamanhos uniformes. Tal restrição proporciona um grande benefício, pois torna o método propício a uma análise em que permite ser computado o número de iterações tanto para ambientes homogêneos quanto para ambientes heterogêneos. Duas abordagens são feitas para garantir esta uniformidade em cada iteração: para ambientes homogêneos, a tarefa mestre v_0 deverá despachar fatias de tamanhos idênticos para todos os trabalhadores dentro de cada iteração, enquanto que para ambientes heterogêneos, todas as tarefas trabalhadoras devem processar suas

devidas fatias de carga relativas a cada iteração na mesma quantidade de tempo [43]. UMR foi comparado com outros algoritmos da literatura [7, 33] e os resultados obtidos em simulações realizadas no *SimGrid* [3] demostraram sua eficiência para uma larga faixa de cenários, conforme reportado em [42, 43].

UMR está relacionado com o algoritmo *Multi-Installment* apresentado em [7], sendo que ambos os trabalhos assumem o pressuposto de que a utilização de pequenas fatias permite uma sobreposição entre comunicação e computação. Porém, em [7] não é definida uma formulação que permita computar o número ideal de iterações, o qual é dado estaticamente como parâmetro de entrada para a execução do referido algoritmo. Ainda, de acordo com [42, 43, 44], o tamanho das fatias de carga pode aumentar gradativamente no decorrer da execução da aplicação com o objetivo de reduzir as sobrecargas de comunicação e computação. Os principais focos do algoritmo UMR são a redução do tempo de execução da aplicação, isto é, seu *makespan* e uma efetiva utilização dos recursos computacionais. O trabalho proposto em [43] detalha o algoritmo UMR inicialmente para plataformas homogêneas, para em sequência, introduzir sua especificação para plataformas heterogêneas. Neste trabalho de dissertação de mestrado será descrito apenas o algoritmo UMR para plataformas heterogêneas.

3.1.1 UMR Para Plataformas Heterogêneas

O algoritmo UMR especifica o número de iterações em que a carga de trabalho W_{total} deverá ser particionada, denotado por M, e os respectivos valores das fatias de carga $chunk_{i,j}$ a serem executadas por cada processo trabalhador v_i executado em p_i, $1 \leq i \leq n$, em cada iteração j, $0 \leq j \leq M-1$. Para plataformas heterogêneas, o algoritmo UMR envia trabalho para os processos trabalhadores de acordo com a capacidade de processamento do recurso ao qual este processo está alocado.

De acordo com a restrição de uniformidade das iterações discutida anteriormente, o tempo de processamento de cada fatia de carga $chunk_{i,j}$ em uma iteração j deverá ser o mesmo para todas as tarefas trabalhadoras v_i, e é dado por:

$$co_i + chunk_{i,j}/S_i, \tag{3.1}$$

e o montante de carga processada em cada iteração j é definido como:

$$round_j = \sum_{i=1}^{n} chunk_{i,j}. \tag{3.2}$$

Conforme derivado em [43], combinando-se as Equações 3.1 e 3.2, obtém-se:

$$chunk_{i,j} = \alpha_i \times round_j + \beta_i, \tag{3.3}$$

onde

$$\alpha_i = \frac{S_i}{\sum_{k=1}^{n} S_k}, \tag{3.4}$$

$$\beta_i = \frac{S_i}{\sum_{k=1}^{n} S_k} \sum_{k=1}^{n} (S_k \times co_k) - S_i \times co_i. \tag{3.5}$$

Para que ocorra a sobreposição entre os tempos de processamento e envio das fatias de carga, a tarefa mestre v_0 deverá enviar os dados da iteração $j+1$ para todas as n tarefas trabalhadoras durante o mesmo intervalo de tempo em que v_n (última tarefa trabalhadora) realiza a computação de sua fatia relativa a iteração j, o que pode ser formulado por:

$$\sum_{i=1}^{n} (\frac{chunk_{i,j+1}}{B_i} + no_i) = co_n + chunk_{n,j}/S_n. \tag{3.6}$$

O lado esquerdo da Equação 3.6 representa o tempo necessário para que v_0 envie as fatias de carga para todas as n tarefas trabalhadoras durante a iteração $j+1$. O lado direito representa o tempo despendido para que o processo trabalhador v_n execute a computação de $chunk_{n,j}$ durante a iteração j. A sobreposição entre processamento e envio de fatias de carga é ilustrada na Figura 3.1 (do instante de tempo T_B à T_C por exemplo). Neste momento a utilização da plataforma de execução se encontra maximizada. Também é possível observar na Figura 3.1 que na última iteração do algoritmo UMR são enviadas fatias de tamanhos decrescentes, visando permitir a todos os recursos computacionais finalizarem suas execuções simultaneamente. Conforme especificado em [43], as Equações 3.3 e 3.6 podem ser combinadas e reduzidas à:

$$round_j = \theta^j (round_0 - \eta) + \eta, \tag{3.7}$$

onde

$$\theta = \left(\sum_{i=1}^{n} \frac{S_i}{B_i} \right)^{-1}, \tag{3.8}$$

$$\eta = \frac{\sum_{i=1}^{n}(S_i \times co_i) - \sum_{i=1}^{n} S_i \times \sum_{i=1}^{n}(\frac{\beta_i}{B_i} + no_i)}{\sum_{i=1}^{n} \frac{S_i}{B_i} - 1}. \tag{3.9}$$

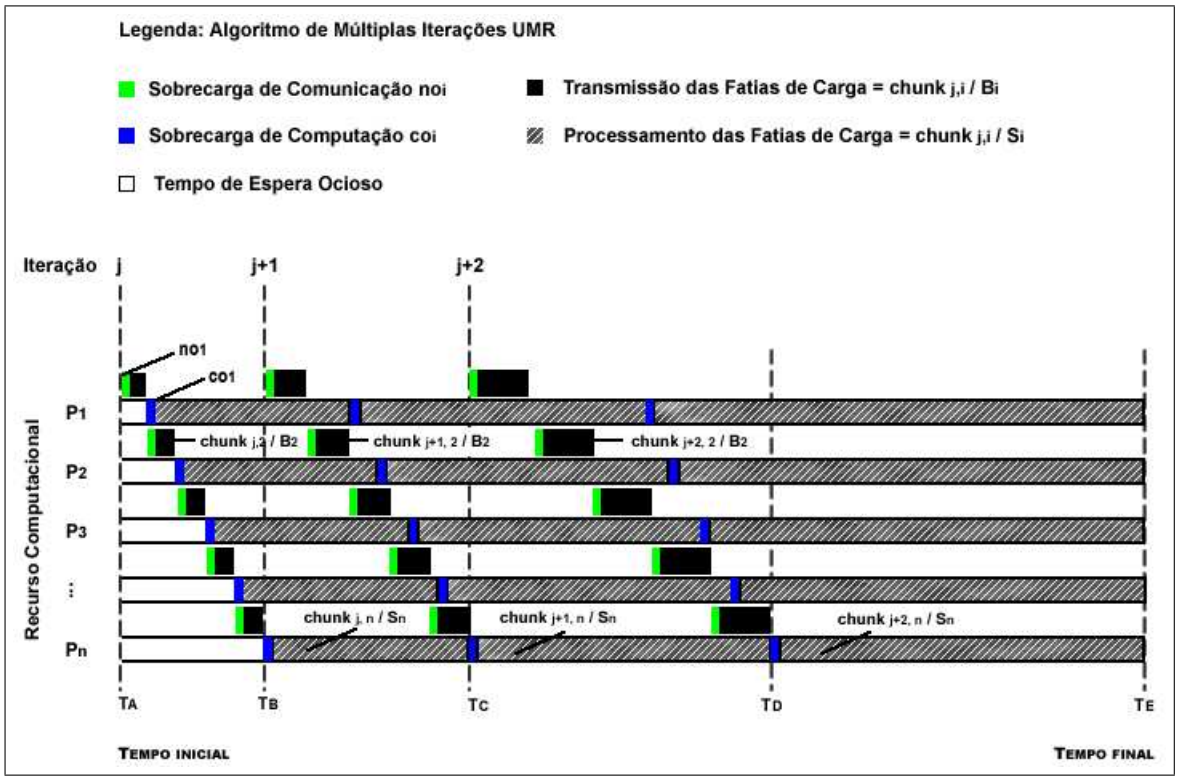


Figura 3.1: Execução do algoritmo UMR

A partir da Equação 3.6, uma série de transformações foram derivadas em [43] para definir o tempo de execução ou *makespan* da aplicação de divisão de carga, e que se resume em:

$$mspanUMR = \sum_{j=0}^{M-1} \left(co_n + chunk_{n,j}/S_n \right) + \frac{1}{2} \sum_{i=1}^{n} \left(\frac{chunk_{i,0}}{B_i} + no_i \right). \tag{3.10}$$

O objetivo do algoritmo UMR é minimizar $mspanUMR$, o *makespan* da aplicação. Este problema de minimização, onde M e $chunk_{i,0}$ são desconhecidos, pode ser resolvido utilizando-se o método *Lagrange Multiplier* [6], o qual deriva a equação para o cálculo do número de iterações M (Apêndice A Equação A.1). Uma vez que o valor de M é conhecido, pode-se então computar o valor de $round_0$ como:

$$round_0 = \frac{1-\theta}{1-\theta^M}(W_{total} - M \times \eta) + \eta. \tag{3.11}$$

Por fim, utilizando-se as Equações 3.3 e 3.7 , podem então ser computados todos os valores para $chunk_{i,j}$ e $round_j, \forall i = 1, 2, ..., n$ e $\forall j = 1, 2, ..., M-1$, respectivamente.

Em relação à seleção dos recursos que deverão fazer parte do processamento da carga de trabalho W_{total}, é especificado no algoritmo UMR uma política de seleção relativamente simples. Tal política consiste em ordenar todos os recursos computacionais de acordo com o fator $\frac{S_i}{B_i}$ em ordem crescente, e em seguida selecionar os n processadores do conjunto original de n_d processadores disponíveis, tal que:

$$\sum_{k=0}^{n} \frac{S_k}{B_k} < 1. \tag{3.12}$$

Essa política de seleção de recursos permite que os tamanhos das fatias de carga mantenham uma ordem crescente a cada iteração da divisão de carga. Caso contrário, ou seja, $\sum_{k=0}^{n} \frac{S_k}{B_k} > 1$, o comportamento do algoritmo seria invertido e os tamanhos das fatias de carga seguiriam sendo decrementados a cada iteração. Além disso, tal política garante que não ocorra uma divisão por zero e consequentemente um erro durante o cálculo da Equação 3.9. Após selecionados os n recursos computacionais p_i, nos quais residirão as tarefas trabalhadoras v_i, o algoritmo UMR procede por calcular o número de iterações M e os devidos valores das fatias de carga $chunk_{i,j}$.

Complementando a formalização do algoritmo UMR, é apresentada na Tabela 3.1 a relação entre as funcionalidades citadas nos Algoritmos 1 e 2 (definidos no Capítulo 2 para divisão de carga em múltiplas iterações com os modelos *1PProc* e *1PTask*) e as equações do algoritmo UMR as quais estas funcionalidades representam.

Funcionalidade	Equação do algoritmo UMR implementada
Recursos_Selecionados();	Equação 3.12
Numero_Iteracoes();	Equação A.1
Tamanho_Fatia(i, j);	Equação 3.3

Tabela 3.1: *Relação entre as funcionalidades contidas nos pseudo-algoritmos para divisão de carga em múltiplas iterações e as equações do algoritmo UMR as quais representam.*

3.2 O Algoritmo MRRS

Para se obter um bom algoritmo de escalonamento, além de solucionar a questão do particionamento da carga de trabalho, o algoritmo deve também atentar para o problema da seleção de recursos. Este problema está centrado em como selecionar o melhor conjunto de recursos que possam processar a carga de trabalho, tal que o *makespan* da aplicação seja mínimo.

No algoritmo UMR [43], o tamanho das fatias de carga enviadas para cada processo trabalhador é calculado somente com base no poder computacional dos recursos envolvidos na execução, conforme pode ser visto na Equação 3.3. O algoritmo *Multi-Round Scheduling With Resource Selection* (MRRS) [25, 26, 27] se propõe a aprimorar o algoritmo UMR em dois aspectos:

1. MRRS baseia seus cálculos para divisão da carga em parâmetros adicionais, tais como largura de banda e os diferentes tipos de latências;

2. MRRS implementa um mecanismo de seleção de recursos com um grau maior de detalhes que o UMR.

Similarmente ao modelo adotado pelo UMR, o algoritmo MRRS também considera um ambiente heterogêneo e distribuído (*topologia estrela*), no qual uma tarefa mestre v_0 envia trabalho para n processos trabalhadores v_i executados em p_i, $1 \leq i \leq n$, conforme o modelo *1PProc*. A tarefa mestre pode dividir a carga de trabalho W_{total} em fatias arbitrárias e entregá-las para os trabalhadores adequados. A seguir será formalizado o algoritmo MRRS para plataformas heterogêneas.

3.2.1 MRRS Para Plataformas Heterogêneas

Conforme observado na Equação 3.1, o algoritmo UMR define o tempo de execução de uma fatia de carga $chunk_{i,j}$ como sendo somente o tempo gasto em seu processamento. Por sua vez, o algoritmo MRRS adota uma abordagem diferente, definindo o tempo de execução de cada fatia de carga $chunk_{i,j}$ como sendo o tempo requerido para que a tarefa trabalhadora v_i a receba e a processe, em cada iteração j. Esta definição é formulada por:

$$no_i + \frac{chunk_{j,i}}{B_i} + co_i + \frac{chunk_{j,i}}{S_i}. \tag{3.13}$$

Conforme derivado em [25, 26], o tamanho de cada fatia de carga pode ser obtido por:

$$chunk_{j,i} = \alpha_i \times round_j + \beta_i, \tag{3.14}$$

onde

$$\alpha_i = \frac{A_i}{\sum_{i=1}^{n} A_i}, \tag{3.15}$$

$$\beta_i = A_i \frac{\sum_{k=1}^{n} A_k(o_k - o_i)}{\sum_{k=1}^{n} A_k}, \tag{3.16}$$

$$A_i = B_i S_i/(B_i + S_i), \tag{3.17}$$

$$o_i = co_i + no_i. \tag{3.18}$$

Para maximizar a utilização da largura de banda da rede, o envio de todas as fatias pelo mestre v_0 e a computação de v_n precisam finalizar ao mesmo tempo, o que pode ser formulado por:

$$\sum_{i=1}^{n}\left(no_i + \frac{chunk_{j+1,i}}{B_i}\right) = \frac{chunk_{j,n}}{S_n} + co_n. \tag{3.19}$$

Substituindo $chunk_{j+1,i}$ e $chunk_{j,n}$ por suas expressões na Equação 3.14, resulta em:

$$round_{j+1} = round_j \times \theta + \mu, \tag{3.20}$$

onde

$$\theta = \frac{B_n}{\left((B_n + S_n)\sum_{i=1}^{n} \frac{S_i}{B_i+S_i}\right)}, \tag{3.21}$$

$$\mu = \frac{\left(\frac{\beta_n}{S_n} + co_n - \sum_{i=1}^{n}(no_i + \frac{\beta_i}{B_i})\right)}{\sum_{i=1}^{n} \frac{\alpha_i}{B_i}}. \tag{3.22}$$

De acordo com os autores em [25, 26], por indução da Equação 3.20 é possível com-

putar o tamanho total de cada iteração j como:

$$round_j = \theta^j(round_0 - \eta) + \eta, \tag{3.23}$$

onde

$$\eta = \frac{\beta_n + co_n - \sum_{i=1}^{n}(no_i + \frac{\beta_i}{B_i})}{\sum_{i=1}^{n}\frac{\alpha_i}{B_i} - \frac{\alpha_n}{S_n}}. \tag{3.24}$$

3.2.2 Determinando os Parâmetros da Iteração Inicial

De forma similar a calculada em UMR, MRRS deriva o número de iterações M e o montante de carga processada na iteração zero ($round_0$), baseado no *makespan* da aplicação, que é definido por [25, 26] como sendo:

$$mspanMRRS(P) = \sum_{i=1}^{n}\left(\frac{chunk_{0,i}}{B_i} + no_i\right) + \sum_{j=0}^{M-1}\left(\frac{chunk_{j,n}}{S_n} + co_n\right). \tag{3.25}$$

Na Equação 3.25, detalhada no Apêndice A, o parâmetro P representa o conjunto de processadores disponíveis no ambiente de execução. O objetivo do algoritmo MRRS é minimizar $mspanMRRS(P)$. Assim como no caso do algoritmo UMR, é utilizado o método Lagrange Multiplier [6] para resolver tal problema de minimização e após sua solução é obtida a equação para o cálculo de M (número de iterações). É importante observar que a equação para o cálculo do número de iterações obtida para o algoritmo MRRS é exatamente **igual** à obtida para o algoritmo UMR, conforme apresentado em [27] e [43] respectivamente. Tal equação é apresentada no Apêndice A (Equação A.1) deste trabalho. Após M ser obtido, $round_0$ pode ser calculado por:

$$M\eta + (round_0 - \eta)\frac{1 - \theta^M}{1 - \theta} - W_{total} = 0. \tag{3.26}$$

Por fim, utilizando-se as Equações 3.14 e 3.23 obtém-se os valores de $round_j$ e de $chunk_{i,j}$, $1 \leq i \leq n$ e $1 \leq j \leq M - 1$, respectivamente.

3.2.3 Política de Seleção de Recursos

O algoritmo MRRS possui uma política de seleção de recursos que busca encontrar o melhor subconjunto de processadores disponíveis, que possa processar a carga de trabalho W_{total} no menor tempo possível. É importante lembrar que o MRRS segue o modelo de execução *1PProc*, e assim, cada processador executa um único processo v_i.

Seja P o conjunto de n processadores disponíveis ($|P| = n$). O objetivo da política de seleção de recursos é encontrar o melhor subconjunto P^* ($P^* \subseteq P$, $|P^*| \leq n$) que minimize o *makespan* da aplicação. O Algoritmo 3 descreve a política de seleção de recursos do MRRS conforme apresentada em [25, 26, 27]. Os termos p_1 e p_n denotam respectivamente o primeiro e o último processador, nos quais estão alocadas as tarefas trabalhadoras v_1 e v_n. Por sua vez, o termo $mspanMRRS(P^*)$ denota o valor de *makespan* calculado para o subconjunto de processadores P^*.

Algoritmo 3 : Seleciona_Recursos(P)

1: **Início**
2: $p_n = \{p_i \in P \mid \frac{B_i}{(B_i+S_i)} \leq \frac{B_j}{(B_j+S_j)} \quad \forall p_j \in P\}$;
3: $P_1^* = \{\forall p_i \in P \mid \sum \frac{S_i}{B_i+S_i} < \frac{B_n}{B_n+S_n}\}$;
4: $P_2^* = Greedy(P, \text{“}\theta < 1\text{”})$;
5: $P_3^* = Greedy(P, \text{“}\theta = 1\text{”})$;
6: $P_1^* = P_1^* \cup \{p_n\}$;
7: $P_2^* = P_2^* \cup \{p_n\}$;
8: $P_3^* = P_3^* \cup \{p_n\}$;
9: Selecione $P^* \in \{P_1^*, P_2^*, P_3^*\}$ tal que:
10: $mspanMRRS(P^*) = \min(mspanMRRS(P_1^*), mspanMRRS(P_2^*), mspanMRRS(P_3^*))$;
11: retorne (P^*);
12: **Fim**

O Algoritmo 3, que especifica a política de seleção de recursos do MRRS, inicia buscando o último processador p_n, o qual irá executar a tarefa trabalhadora v_n (linha 2). Em seguida, dependendo do valor de θ (definido na Equação 3.21) três casos são examinados usando diferentes algoritmos de busca para encontrar o melhor subconjunto P^*. No final, é selecionado o subconjunto que produzir o menor *makespan* (linhas 9 e 10).

Para se obter o melhor subconjunto de processadores, três políticas distintas são utilizadas para gerar P_1^*, P_2^* e P_3^*. Conforme definido em [25, 26], P_1^* é o subconjunto que maximize a soma:

$$\sum_{p_i \in P^*} \frac{B_i S_i}{B_i + S_i} \tag{3.27}$$

Condicionado a $\theta > 1$ ou

$$\sum_{p_i \in P^*} \frac{S_i}{B_i + S_i} < \frac{B_n}{B_n + S_n}. \tag{3.28}$$

Para gerar os subconjuntos P_2^* e P_3^* é utilizado o algoritmo *Greedy* (linhas 4 e 5). Nos dois casos, *Greedy* permanece progressivamente adicionando mais processadores p_i a P^*, enquanto P^* satisfazer suas respectivas condições de parada (valor de θ) para P_2^* e P_3^*. O pseudo código do algoritmo de busca *Greedy* é apresentado no Algoritmo 4 e possui complexidade $O(n)$.

Algoritmo 4 : Greedy($P, condicaoTheta$)

```
1:  Início
2:    Selecione p_1 ∈ P | B_1 ≥ B_i ∀p_i ∈ P;
3:    Selecione p_n ∈ P | B_n/(B_n+S_n) ≤ B_i/(B_i+S_i) ∀p_i ∈ P;
4:    P* = {p_1, p_n};    P = P − P*;
5:    mspan = mspanMRRS(P*);
6:    Repita
7:      ∀p_i ∈ P
8:      {
9:        se mspanMRRS(P* ∪ {p_i}) < mspan
10:       {
11:         P* = P* ∪ {p_i};
12:         P = P − {p_i};
13:         mspan = mspanMRRS(P*);
14:       }
15:     }
16:   Enquanto condicaoTheta
17:   retorne (P*);
18: Fim
```

Conforme definido no Algoritmo 4, inicialmente são selecionados o primeiro e o último recursos computacionais, denotados respectivamente por p_1 e p_n (linhas 2 e 3). Na sequência, o algoritmo *Greedy* segue adicionando mais processadores p_i à P^* de tal forma que o *makespan* da aplicação seja minimizado. Esta operação é executada enquanto sua condição de parada (valor de θ) não for atingida. Uma vez que o algoritmo de seleção de recursos determinou o melhor subconjunto que possa processar a carga de trabalho no menor tempo possível, a tarefa mestre inicializa a transmissão das fatias de carga para as tarefas trabalhadoras residentes nos recursos selecionados.

Similarmente ao realizado para o algoritmo UMR, é apresentada na Tabela 3.2 a relação entre as funcionalidades citadas nos Algoritmos 1 e 2 (para os modelos de execuções *1PProc* e *1PTask*, respectivamente) e as equações ou seções do algoritmo MRRS as quais estas funcionalidades representam.

Funcionalidade	Equação do algoritmo MRRS implementada
Recursos_Selecionados();	Seção 3.2.3
Numero_Iteracoes();	Equação A.1
Tamanho_Fatia(i, j);	Equação 3.14

Tabela 3.2: *Relação entre as funcionalidades contidas nos pseudo-algoritmos para divisão de carga em múltiplas iterações e as equações do algoritmo MRRS as quais representam.*

3.3 O Algoritmo PWD

O algoritmo *Performance-Based Workload Distribution* (PWD), proposto em [36], será utilizado neste trabalho para fins de avaliação de desempenho. PWD se caracteriza por apresentar uma abordagem *híbrida*, pois realiza escalonamento estático e dinâmico para aplicações de cargas divisíveis. O escalonamento da carga de trabalho no algoritmo PWD é realizado em duas etapas: na primeira, uma percentagem SWR (*Static-Workload Ratio*) da carga de trabalho é particionada estaticamente entre os processos trabalhadores, de acordo com suas capacidades de processamento. Na segunda etapa, a percentagem restante $(100 - SWR)$ da carga de trabalho é distribuída dinamicamente, enviando novas fatias para os recursos que primeiro finalizarem suas execuções, visando assim obter um balanceamento de carga e uma utilização mais eficiente da plataforma de execução. O esquema utilizado nesta etapa de escalonamento dinâmico é o *Guided Sef-Scheduling* (GSS), também apresentado em [36], e determina que o tamanho das novas fatias de carga a serem enviadas é obtido pela divisão entre o total de carga restante e a quantidade de processadores disponíveis.

Os autores em [36] apresentam o conceito de *função de performance* (PF) para estimar a capacidade de processamento de cada recurso computacional disponível. A utilização desta métrica permite uma distribuição justa da carga de trabalho disponível, enviando uma maior quantidade de trabalho para os recursos que apresentem maior capacidade de processamento. A *função de performance* para cada processador p_i, $1 \leq i \leq n$ é definida como:

$$PF_i = w_1 \times \frac{\frac{S_i}{CL_i}}{\sum_{j=1}^{n} \frac{S_j}{CL_j}} + w_2 \times \frac{B_i}{\sum_{j=1}^{n} B_j}, \tag{3.29}$$

onde

- n é o número de recursos computacionais p_i disponíveis no ambiente de execução;
- S_i representa o poder computacional de cada processador p_i, $1 \leq i \leq n$;
- CL_i indica a carga de CPU de cada processador p_i;
- B_i é a largura de banda entre p_0 e p_i;
- w_1 e w_2 representam os pesos do primeiro e segundo termos, respectivamente. A soma destes dois parâmetros é igual a 1.

Conforme apresentado em [36], os valores de S_i, CL_i e B_i são adquiridos por meio da ferramenta de monitoramento *TIGER tool* [4], a qual disponibiliza um conjunto de informações acerca de cada recurso computacional presente no ambiente de execução. Ainda, os valores dos parâmetros w_1 e w_2 são obtidos experimentalmente, isto é, um conjunto de execuções da aplicação é realizado com diferentes combinações de valores para w_1 e w_2 e aquela que apresentar o menor tempo de execução é selecionada para ser utilizada como padrão. Por fim, o algoritmo PWD segue as definições do modelo de execução tradicional *1PProc*, executando um único processo trabalhador v_i em cada processador p_i, e não especifica uma política de seleção de recursos.

Outro conceito definido em [36] foi o de carga de trabalho irregular. Uma carga de trabalho irregular é caracterizada por apresentar unidades que demandam diferentes tempos de processamento. A aplicação de *Mandelbrot*, a ser discutida na Seção 5.5, é um exemplo de aplicação composta por uma carga de trabalho irregular. A segunda etapa do algoritmo PWD (escalonamento dinâmico) ocorre somente para aplicações que apresentem uma carga de trabalho irregular, caso contrário, toda a carga é particionada estaticamente entre os recursos trabalhadores durante a primeira fase do algoritmo. O fator SWR que determina o percentual de carga a ser distribuído durante as etapas de escalonamento estático e dinâmico pode variar entre 0 e 1. Caso a carga de trabalho da aplicação seja regular, SWR é setado para 1, determinando que 100% da carga seja escalonada estaticamente, caso contrário, seu valor é determinado de acordo com a relação entre o menor e o maior tempo de execução coletados empiricamente para a aplicação. Esta relação pode ser formulada por:

$$SWR = \frac{min}{MAX}, \tag{3.30}$$

onde

- min representa o menor tempo de execução coletado para a aplicação;
- MAX é o maior tempo de execução também coletado para a mesma aplicação.

3.3.1 Pseudo-Algoritmo PWD

O pseudo código apresentado no Algoritmo 5 especifica o conjunto de operações executadas pelo algoritmo PWD em sua divisão de carga.

Algoritmo 5 : Pseudo-Algoritmo PWD

```
1:  Início
2:    // Processo mestre v_0 executado em p_0
3:    se( processo == v_0 )
4:    {
5:      Coletar S_i, CL_i e B_i de cada recurso p_i por meio da ferramenta TIGER tool.
6:      min = menor tempo de execução coletado para a aplicação.
7:      MAX = maior tempo de execução coletado para a aplicação.
8:      SWR = min/MAX;
9:      Calcular PF_i para cada recurso p_i, de acordo com a Equação 3.29.
10:     // Fase 1: Escalonamento Estático:
11:     Distribuir SWR% da carga de trabalho entre as tarefas v_i executadas em p_i,
        de acordo com PF_i.
12:     // Fase 2: Escalonamento Dinâmico:
13:     Distribuir os (100 − SWR)% restantes da carga de trabalho entre as tarefas
        que primeiro finalizarem sua execução de acordo com o algoritmo GSS
14:   }
15:   // Processo trabalhador v_i executado em p_i
16:   senao
17:   {
18:     Enquanto(chegar novas fatias de carga)
19:     {
20:       recebe e processa a fatia da carga de trabalho;
21:     }
22:   }
23: Fim
```

Conforme especificado no Algoritmo 5, a tarefa mestre v_0 inicialmente coleta os valores de S_i, CL_i e B_i para cada recurso computacional p_i (linha 5). Em seguida, v_0 calcula o valor de SWR (linha 8), o qual determina o montante da carga de trabalho a ser

distribuída em cada fase do escalonamento. Na sequência, v_0 determina o valor da PF_i (linha 9) de cada processador p_i, $1 \leq i \leq n$. Durante a fase de escalonamento estático (linha 11), v_0 envia trabalho para cada tarefa trabalhadora v_i de acordo com a função de performance PF_i do recurso ao qual v_i está alocada. Note que nesta fase apenas uma percentagem SWR da carga de trabalho total é escalonada. Por fim, v_0 distribui o restante da carga de trabalho entre as tarefas trabalhadoras v_i que primeiro finalizarem sua execução (linha 13).

3.4 Outras Propostas

Além dos algoritmos UMR, MRRS e PWD definidos acima, um conjunto de outros algoritmos para escalonamento de aplicações de cargas divisíveis também foram estudados a fim de coletar algumas de suas características que pudessem vir a fazer parte deste trabalho de dissertação de mestrado. Dentre os algoritmos analisados, podemos citar o RUMR (*Robust Uniform Multi Round*) proposto em [42] e que possui como foco ambientes com certo grau de incerteza, ou seja, ambientes em que não se pode esperar o mesmo comportamento em termos de disponibilidade dos recursos ao longo da execução da aplicação, característica comum em ambientes onde seus recursos são compartilhados entre diversos usuários. Análises feitas com o RUMR demonstraram que a partir do instante em que os recursos trabalhadores se tornam sobrecarregados é vantajoso aliviar a carga de trabalho a qual estão submetidos até que os recursos retornem ao seu estado normal. O algoritmo RUMR escalona sua carga de trabalho em duas fases distintas:

- Fase 1: Utiliza uma versão revisada do algoritmo UMR para pré-calcular o escalonamento inicial da aplicação, inicializando com o envio de pequenas fatias de carga e incrementando seus tamanhos gradualmente. Esta etapa tem como objetivo aumentar a performance de execução da aplicação por meio da sobreposição de computação com comunicação.
- Fase 2: Utiliza o enfoque de fatoramento proposto por [21] para decrementar o tamanho das fatias de carga enviadas aos recursos trabalhadores. Seu objetivo é aliviar os recursos alocados para o processamento da carga de trabalho.

O algoritmo RUMR utiliza o conceito de fator de previsão de performance definido em [42]. Este fator determina o atual estado de cada recurso computacional p_i, isto é, o quão sobrecarregado computacionalmente cada recurso se encontra. O RUMR segue

utilizando a Fase 1 até o momento em que o fator de previsão de performance ultrapassar um determinado valor tolerável. A partir deste instante, o algoritmo altera seu comportamento e inicializa a execução da Fase 2, enviando fatias de carga de tamanhos menores até o instante em que o recurso computacional retorne a seu estado normal. Porém, esta estratégia de escalonamento pode levar a um aumento inesperado no tempo de execução da aplicação, caso o recurso em questão permaneça sobrecarregado por uma grande faixa de tempo. Em virtude desta limitação, ao invés de utilizar esta característica do algoritmo RUMR este trabalho utiliza os serviços da camada de escalonamento dinâmico do *EasyGrid AMS*, a qual tem como finalidade redistribuir as fatias de carga inicialmente alocadas aos recursos sobrecarregados, entre aqueles que possam processá-las em um tempo menor. Todo o trabalho de escalonamento dinâmico é feito sem que haja a necessidade de intervenção do processo mestre, ou seja, o próprio *EasyGrid AMS* se encarrega de re-escalonar as tarefas trabalhadoras para outros recursos e repassar suas devidas fatias de carga (mensagens de dados) sem qualquer envolvimento do processo mestre.

Outro algoritmo analisado neste trabalho foi o *Parallel Transferable Uniform Multi Round* (PTUMR) proposto por [41] e que possui como foco uma adaptação do algoritmo UMR para permitir o envio de fatias de carga para os processos trabalhadores paralelamente em cada iteração do escalonamento. Esta abordagem do algoritmo PTUMR tem como finalidade aliviar o tempo de espera ocioso dos processos trabalhadores pela chegada de suas respectivas fatias de carga. Resultados obtidos através de simulações do algoritmo demonstraram uma redução do tempo de execução da aplicação se comparado ao UMR, devido principalmente à eliminação da espera ociosa sofrida pelos recursos trabalhadores. Duas limitações se fazem presente no algoritmo PTUMR, as quais inviabilizaram sua utilização neste trabalho: a primeira se deve ao fato de sua formulação matemática propor este novo modelo de escalonamento somente para ambientes homogêneos. Sua segunda limitação está relacionada a utilização do pressuposto de que um determinado recurso computacional pode simultaneamente enviar dados para diversos outros recursos, característica que ainda não se faz presente na maioria dos recursos computacionais.

3.5 Resumo

Neste capítulo, um conjunto de algoritmos para escalonamento de aplicações de cargas divisíveis foram apresentados. Dentre tais algoritmos estão o UMR e o MRRS, os quais se caracterizam por escalonarem suas cargas de trabalho em múltiplas iterações, visando atingir uma sobreposição entre o tempo de envio e o tempo de processamento desta carga.

Ainda, outros algoritmos de cargas divisíveis como o RUMR, PTUMR e PWD também foram detalhados, sendo que o algoritmo PWD será utilizado neste trabalho para fins de avaliação de desempenho.

No próximo, capítulo será apresentado o *framework EasyGrid AMS*, sua estrutura de escalonamento hierárquica e o conjunto de funcionalidades para escalonamento de aplicações de cargas divisíveis desenvolvidas neste trabalho.

Capítulo 4

Divisão de Cargas Autonômicas

O objetivo deste capítulo é descrever as principais características do sistema gerenciador de aplicações *EasyGrid AMS*, o qual será utilizado para automatizar a execução de aplicações de cargas divisíveis em *clusters* computacionais. Em seguida, serão abordadas as funcionalidades acopladas ao *EasyGrid AMS* para auxiliá-lo na execução de tais aplicações.

4.1 O Framework EasyGrid AMS

O *framework EasyGrid AMS* [9, 10, 11, 13, 14, 15, 17, 18] é um sistema gerenciador de aplicações MPI [2], que trabalha com criação dinâmica de processos e é responsável pela transformação de aplicações paralelas tradicionais em aplicações autônomas. Aplicações autônomas são programas adaptativos, capazes de reagirem às mudanças que possam ocorrer no ambiente de execução, tolerantes à falhas e auto-escalonáveis. A finalidade do *EasyGrid AMS* é executar e gerenciar eficientemente aplicações MPI em diferentes tipos de ambientes (distribuídos, dinâmicos, compartilhados), sejam eles compostos por recursos homogêneos ou heterogêneos. Seu sistema de escalonamento e gerenciamento é descentralizado por aplicação, visando assim prover uma maior eficiência e escalabilidade para a aplicação em questão. No modelo de execução adotado pelo *EasyGrid AMS*, cada aplicação possui uma hierarquia de processos gerenciadores e de agentes escalonadores, que possibilitam a definição de políticas distintas tanto entre as aplicações, como também para os diferentes níveis de uma hierarquia. A Figura 4.1 representa a arquitetura de camadas utilizada pelo *EasyGrid AMS*.

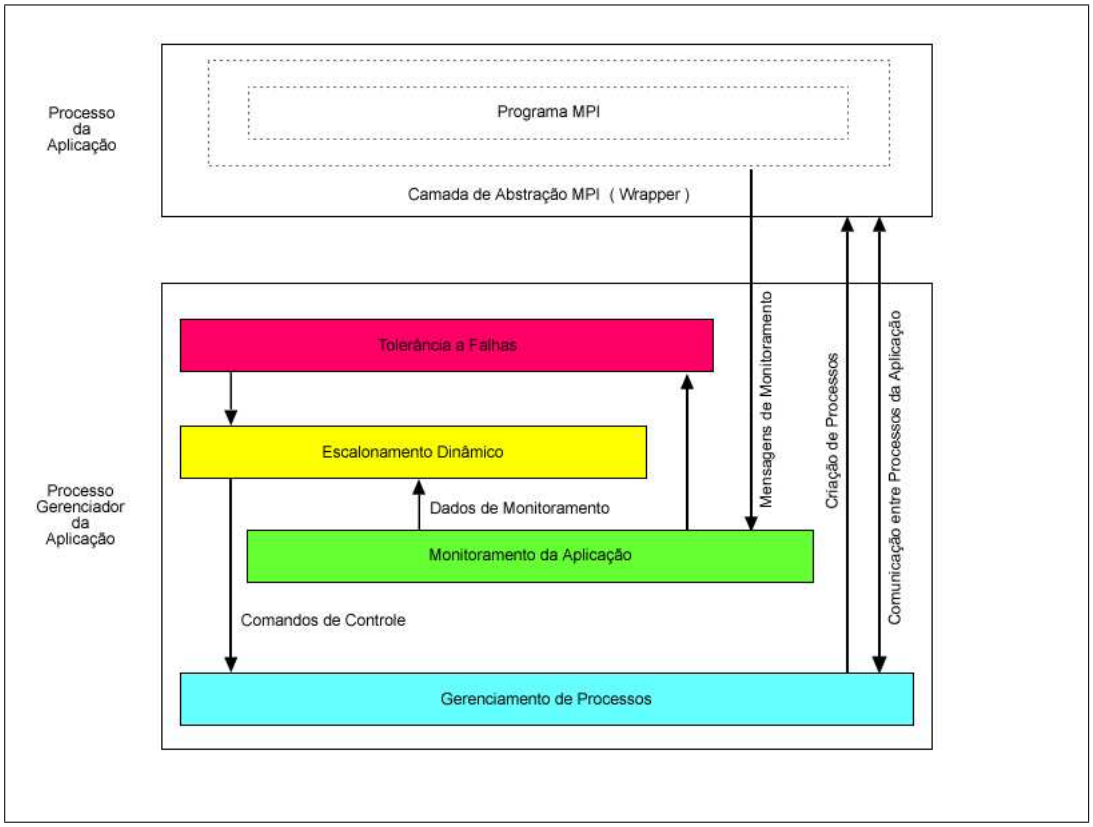


Figura 4.1: Arquitetura de camadas utilizada pelo *EasyGrid AMS*

Através do constante monitoramento do ambiente durante a execução da aplicação, o conjunto de escalonadores distribuídos do *EasyGrid AMS* executam suas ações de maneira coordenada. O monitoramento e a coleta de informações sobre o estado do sistema, do ambiente de execução e da aplicação são executados por funcionalidades embutidas no *EasyGrid AMS*. O objetivo principal do subsistema de escalonamento é minimizar o tempo de execução final (*makespan*) da aplicação MPI. A camada responsável pelo escalonamento dinâmico [18] permite que a aplicação se auto-ajuste (*self-optimising*), otimizando sua execução e se auto reconfigurando, através de uma redistribuição de tarefas, quando necessário (*self-configuring*). A camada de escalonamento dinâmico somente re-escalona tarefas que ainda não foram criadas. Esta estratégia diminui a sobrecarga do próprio processo de escalonamento dinâmico como um todo. A camada de tolerância a falhas provê a habilidade de auto-recuperação (*self-healing*) em caso de falhas de processos *e/ou* processadores [14]. A camada de gerenciamento de processos é responsável pela criação dinâmica de processos MPI da aplicação do usuário e também pelo redirecionamento de mensagens entre eles. Por fim, a camada de monitoramento tem como função fornecer informações que são utilizadas pelas camadas de escalonamento dinâmico e de tolerância a falhas. Desta forma, o *EasyGrid AMS* provê à aplicação executada sob seu domínio au-

tomonitoramento, autoconhecimento e conhecimento do ambiente. Esses aspectos fazem parte das características que classificam uma aplicação como autonômica [9, 10, 17]

4.1.1 Estrutura de Gerenciamento

A estrutura de gerenciamento do *middleware EasyGrid AMS* é formada por uma hierarquia de três níveis de processos gerenciadores: o gerenciador global (*Global Manager* - GM), o gerenciador do *site* (*Site Manager* - SM) e por fim, o gerenciador da máquina (*Host Manager* - HM). A Figura 4.2 ilustra um exemplo de ambiente de execução, onde podem ser vistos os três níveis da hierarquia de gerenciadores.

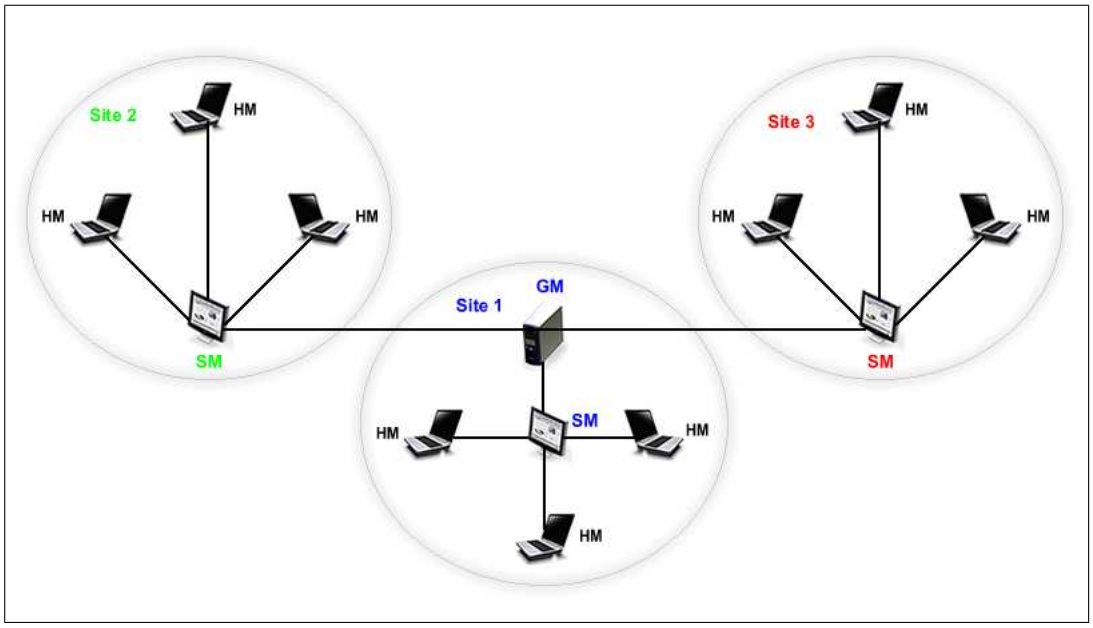


Figura 4.2: Hierarquia de processos gerenciadores do *EasyGrid AMS*

A seguir serão descritas as funcionalidades de cada um dos processos gerenciadores do *EasyGrid AMS*.

- GM (*Global Manager*): Situado no topo da hierarquia, tem como objetivo supervisionar a execução da aplicação como um todo, tendo uma visão geral de todos os *sites*.
- SM (*Site Manager*): Presente em cada um dos *sites* que compõem o ambiente de execução, é responsável por gerenciar os processos da aplicação alocados ao seu *site*.
- HM (*Host Manager*): O gerenciador da máquina existe em cada um dos recursos

computacionais disponíveis, sendo responsável pelo escalonamento, criação e execução dos processos do usuário no seu respectivo recurso.

Com exceção do processo gerenciador global (GM), todos os demais processos, sejam eles gerenciadores ou de aplicação, são criados dinamicamente pelo seu respectivo gerenciador de nível superior na hierarquia. Processos gerenciadores e da aplicação podem executar em um mesmo recurso computacional competindo pela CPU, em recursos com um único processador, ou paralelamente em recursos multiprocessados ou com processadores *multicore*. Porém, a intrusão dos processos gerenciadores do *EasyGrid* é pequena [15], já que são executados como *daemons* orientados a eventos, consumindo ciclos de CPU apenas para processar mensagens.

4.1.2 Características do EasyGrid AMS

O *EasyGrid AMS* possui suporte para a execução de um ou mais processos de aplicação por vez, de maneira concorrente ou paralela. Outra característica é sua escalabilidade, a princípio, não existe um número máximo de processos que uma aplicação pode ter, fato de grande relevância para aplicações distribuídas, as quais são frequentemente de grande escala. Além disso, sua estrutura de gerenciamento hierárquica permite uma maior escalabilidade com relação ao número de recursos do que ambientes que possuem uma estrutura centralizada. A implementação do *EasyGrid AMS* é capaz de tornar aplicações MPI específicas para ambientes homogêneos e dedicados (*clusters*), em versões cientes do ambiente (*system-aware*) para serem executadas em ambientes distribuídos, heterogêneos e dinâmicos. Toda troca de mensagens entre processos da aplicação é roteada pelos processos gerenciadores sem a percepção do usuário. Este mecanismo é necessário devido a possibilidade de re-escalonamento de tarefas, pois assim, os processos gerenciadores poderão diretamente encaminhar uma mensagem destinada a um processo que tenha sido re-escalonado para outro recurso sem que a aplicação do usuário tenha que fazer uma nova chamada de envio. Trabalhos apresentados em [15, 18] descrevem em detalhes o processo de roteamento das mensagens da aplicação entre a hierarquia de processos gerenciadores.

Outra característica do *EasyGrid AMS* é a utilização de mensagens de monitoramento para manter informações atualizadas tanto sobre a aplicação em execução, quanto sobre o desempenho dos recursos computacionais. Tais mensagens são disparadas sempre que um processo da aplicação é finalizado ou em virtude de eventos de tempo (*heartbeat*), ou seja, caso um processo gerenciador não receba nenhuma mensagem de monitoramento de algum de seus gerenciadores de nível inferior dentro de um intervalo de tempo limite. O

objetivo destas mensagens é manter a hierarquia de gerenciadores atualizada e também, auxiliar nos mecanismos de tolerância a falhas e escalonamento dinâmico. Para evitar uma maior interferência no ambiente de execução, tais mensagens são empacotadas juntamente com mensagens da aplicação, exceto aquelas enviadas em virtude de eventos de tempo (*heartbeat*).

4.2 Aplicação de Divisão de Carga Autonômica

As definições propostas pelos algoritmos UMR e MRRS para divisão de carga foram acopladas ao *framework EasyGrid AMS* tornando o capaz de transformar aplicações de cargas divisíveis tradicionais em autonômicas. Inicialmente, um conjunto de informações são repassadas ao *EasyGrid AMS* por meio de seu portal. Estas informações indicam aos escalonadores a quantidade de carga de trabalho (em unidades de carga) a ser processada, o algoritmo (UMR ou MRRS) que deverá ser utilizado para produzir o escalonamento inicial da aplicação, e por fim, informações referentes ao poder computacional, largura de banda e as respectivas latências envolvendo os recursos trabalhadores. O escalonador global de posse destas informações, incia por selecionar o conjunto de recursos computacionais que estarão presentes no processamento da carga de trabalho. Esta seleção é feita de acordo com a política de seleção do algoritmo selecionado no portal *EasyGrid AMS* (UMR ou MRRS) e que foram apresentadas no Capítulo 3.

Após determinar o conjunto de n processadores que serão utilizados, o escalonador global define a quantidade de iterações M em que a carga de trabalho deverá ser escalonada. O *framework EasyGrid AMS* é capaz de executar aplicações paralelas tanto no modelo tradicional *1PProc* quanto no modelo alternativo *1PTask*. Estudos realizados em trabalhos anteriores [13, 14] identificaram que para ambientes heterogêneos, compartilhados e mais propensos a falhas o modelo *1PTask* é mais eficiente que o modelo tradicional *1PProc*. Desta forma, será utilizado o modelo de execução alternativo *1PTask* para a execução de aplicações de cargas divisíveis neste trabalho. Note que a quantidade de iterações calculada representa a quantidade de processos trabalhadores que serão criados em cada recurso computacional pelo seu respectivo gerenciador de máquina, conforme definido pelo modelo de execução *1PTask*, onde cada fatia de carga é executada por um processo independente. Na sequência, o gerenciador global cria o conjunto de gerenciadores de *site*, os quais por sua vez são responsáveis pela criação de cada um dos gerenciadores de máquina sob seu domínio. Além disso, o gerenciador global também repassa para cada gerenciador de *site* o conjunto de tarefas que serão executadas em seus recursos e suas

respectivas quantidades de trabalho. Após criados, os gerenciadores de máquinas recebem de seu gerenciador de *site* o conjunto de tarefas e de quantidade de trabalho que deverão executar. Por fim, após toda hierarquia de processos gerenciadores ter sido criada, o gerenciador global dispara a tarefa mestre v_0 da aplicação. Em seguida, a cada iteração $j, 0 \leq j \leq M-1$, n processos trabalhadores são criados dinamicamente, sendo um em cada processador selecionado previamente.

Ainda, processos da aplicação poderão utilizar as funcionalidades para cargas divisíveis adicionadas ao *EasyGrid AMS*. Desta forma, poderão descobrir por exemplo a quantidade de processadores selecionados, o número de iterações, a quantidade de trabalho a ser despachada para cada processo trabalhador, entre outras. Estas funcionalidades são adicionadas à aplicação do usuário em tempo de compilação, liberando assim os desenvolvedores do trabalho de implementá-las para cada aplicação de carga divisível a ser executada.

4.3 Aplicações Autonômicas com Granularidades Finas

Apesar da utilização do modelo *1PTask* para execução de aplicações de cargas divisíveis segundo a alocação dos algoritmos UMR e MRRS, foi observado que as tarefas relativas principalmente às últimas iterações apresentaram uma granularidade grossa o suficiente para não aproveitar a disponibilidade de recursos e as funcionalidades do gerenciador *EasyGrid AMS*. Estudos experimentais realizados neste trabalho de dissertação levaram a uma adaptação desses algoritmos, a qual consiste em aumentar o número de iterações em que a carga de trabalho será escalonada, visando criar um número maior de tarefas de granularidades menores.

A estratégia utilizada para aumentar o número de iterações consiste em particionar a carga relativa à última iteração (a qual apresenta os maiores valores de fatias de carga) em diversas novas iterações contendo fatias de tamanhos menores. A quantidade de novas iterações é obtida ao se dividir a carga de trabalho relativa a última iteração pela carga relativa a penúltima iteração $\left\lfloor \frac{round_{M-1}}{round_{M-2}} \right\rfloor$. Desta forma, é gerado um conjunto extra de iterações cujas fatias de carga possuem o mesmo tamanho que as fatias da penúltima iteração. Note que os valores das iterações $round_{M-1}$ e $round_{M-2}$ são calculados inicialmente com base nos algoritmos UMR e MRRS (Equações 3.7 e 3.23 respectivamente). Esta estratégia de particionamento continua a sobrepor computação e comunicação, pois o tempo necessário para se processar uma destas novas fatias de carga é menor do que

o tempo para transmitir uma próxima fatia (de mesmo tamanho). Duas vertentes motivaram estas adaptações dos algoritmos UMR e MRRS para produzirem tarefas com menores granularidades:

- Explorar melhor a habilidade de escalonamento dinâmico de tarefas do *EasyGrid AMS*. Tarefas de granularidade grossa nem sempre podem ser re-escalonadas para outros recursos, pois tenderiam a provocar um alto grau de desbalanceamento na execução da aplicação. Por outro lado, tarefas de granularidades finas podem ser facilmente re-escalonadas sem que ocorra tal desbalanceamento. É importante observar que a camada de escalonamento dinâmico do *EasyGrid AMS* somente re-escalona tarefas que ainda não foram inicializadas.

- Utilizar de maneira mais eficiente os recursos de memória em máquinas *multicore*. Resultados experimentais obtidos neste trabalho verificaram que para aplicações compostas por uma carga de trabalho irregular ou então por uma carga regular e de natureza *data-intensive*, a utilização de tarefas de granularidades finas lhes proporcionou um melhor desempenho. Para aplicações com uma carga de trabalho irregular foi possível obter um bom balanceamento de carga entre os processos trabalhadores. Já para aplicações *data-intensive*, um número menor de atrasos de contenção de memória (eventos *cache-miss*) passou a ocorrer em virtude de tais processos manipularem faixas de endereços de memória menores.

Outras estratégias como gerar novas iterações com fatias de tamanhos menores ou maiores do que as da penúltima iteração também foram abordadas, porém, os melhores resultados foram obtidos com a estratégia que mantém as novas iterações com o mesmo tamanho da penúltima iteração.

O pseudocódigo apresentado no Algoritmo 6 especifica as modificações realizadas nos algoritmos UMR e MRRS para aumentar o número de iterações em que a carga de trabalho será escalonada e consequentemente, produzir tarefas de granularidades menores. Note que o Algoritmo 6 especifica somente o pseudocódigo para o modelo de execução alternativo *1PTask* (o qual será utilizado neste trabalho), onde cada fatia de carga $chunk_{i,j}$ é executada por um processo $v_{i,j}$ distinto.

Algoritmo 6 : Adaptação dos algoritmos UMR e MRRS

1: **Início**
2: // Processo mestre $v_{0,0}$
3: se($processo == v_{0,0}$) {
4: $n = Recursos\_Selecionados()$;
5: $M_{inicial} = Numero\_Iteracoes()$;
6: $round_0 = Calcula\_Round\_Zero()$;
7: $round_{M-2} = \theta^{M-2}(round_0 - \eta) + \eta$;
8: $round_{M-1} = \theta^{M-1}(round_0 - \eta) + \eta$;
9: $M_{extras} = \frac{round_{M-1}}{round_{M-2}}$;
10: $M = M_{inicial} + M_{extras}$;
11: para($j = 0$; $j < M$; $j++$) {
12: para($i = 1$; $i \leq n$; $i++$) {
13: se($j < M_{inicial}$)
14: $chunk_{i,j} = Tamanho\_Fatia(i, j)$;
15: senão
16: $chunk_{i,j} = \frac{Tamanho\_Fatia(i,M-1)}{M_{extras}}$;
17: Envia($chunk_{i,j}$, $v_{i,j}$);
18: }
19: }
20: }
21: // Processo trabalhador $v_{i,j}$
22: senão {
23: Recebe($chunk_{i,j}$, v_0);
24: Processa($chunk_{i,j}$);
25: }
26: **Fim**

Conforme apresentado no Algoritmo 6, a tarefa mestre $v_{0,0}$ inicialmente calcula a quantidade de processadores que estarão presentes na execução da carga de trabalho (linha 4). Em seguida, obtém o número de iterações em que esta carga deverá ser escalonada (linha 5) e os tamanhos da primeira (linha 6), penúltima (linha 7) e última (linha 8) iterações. O número extra de iterações é obtido ao se dividir o total de carga da última iteração pelo total de carga da penúltima iteração (linha 9). Esta quantidade extra de iterações é adicionada ao número inicial de iterações já calculado (linha 10). Por fim, em cada iteração j, $0 \leq j \leq M - 1$, uma fatia de carga é especificada para cada processo trabalhador $v_{i,j}$ executado no processador p_i. O tamanho das fatias relativas às iterações extras é obtido ao se dividir o tamanho da fatia de carga da última iteração pela quantidade de iterações extras (linha 16).

A Tabela 4.1 relaciona as funcionalidades citadas no Algoritmo 6 as suas respectivas equações/seções definidas para os algoritmos UMR e MRRS no Capítulo 3.

Funcionalidade	Equação algoritmo UMR	Equação algoritmo MRRS
Recursos_Selecionados();	Equação 3.12	Seção 3.2.3
Numero_Iteracoes();	Equação A.1	Equação A.1
Calcula_Round_Zero();	Equação 3.11	Equação 3.26
Tamanho_Fatia(i, j);	Equação 3.3	Equação 3.14

Tabela 4.1: *Equações dos algoritmos UMR e MRRS referentes as funcionalidades apresentadas no Algoritmo 6*

4.4 Resumo

Neste capítulo foi apresentado o *framework EasyGrid AMS*, sua hierarquia de processos gerenciadores bem como suas habilidades de escalonamento dinâmico de tarefas e de tolerância a falhas. Na sequência foi abordada a capacidade do *EasyGrid AMS* de executar tanto aplicações compostas por processos de longa duração (*1PProc*) quanto aquelas caracterizadas por um grande número de processos de curta duração (*1PTask*). Também foram apresentadas algumas variantes dos algoritmos UMR e MRRS, desenvolvidas neste trabalho para especificarem tarefas de granularidades finas.

O próximo capítulo irá descrever a avaliação de desempenho de um conjunto de aplicações de cargas divisíveis modeladas com diferentes algoritmos de divisão de carga.

Capítulo 5

Avaliação de Desempenho

Este capítulo apresenta a avaliação de desempenho do *EasyGrid AMS* acoplado com algoritmos de múltiplas iterações para escalonamento de aplicações de cargas divisíveis. Tais aplicações serão executadas conforme o modelo de execução *1PTask*, visto o ganho de desempenho oferecido por este modelo, conforme apresentado em [13] e também ao longo deste capítulo. Os resultados obtidos serão comparados com os apresentados pelas versões tradicionais dos algoritmos UMR, MRRS e PWD, conforme descritos no Capítulo 3. Esta avaliação será realizada através de experimentos em diferentes cenários computacionais.

5.1 Modelo de Avaliação

A avaliação de desempenho apresentada neste capítulo consiste em uma série de experimentos utilizando as seguintes aplicações paralelas:

- Multiplicação de matrizes;
- Mandelbrot set computation fractal;
- Filtro Gaussiano para processamento digital de imagens.

Cada uma destas aplicações apresenta um conjunto de características particulares. A primeira possui um alto custo computacional e realiza uma grande quantidade de acessos a memória, ou seja, é de natureza *data-intensive*. A segunda apresenta uma carga de trabalho irregular, o que dificulta sua adequação a alguns princípios da teoria de divisão de cargas. Já a terceira é uma aplicação *cpu-intensive* e possui um elevado custo de transmissão, pois sua carga de trabalho (uma imagem) reside em disco, sendo necessário

ler toda essa carga, preparar as fatias a serem enviadas e por fim, enviá-las. A modelagem de tais características, ou seja, a correta especificação do poder computacional S_i, largura de banda B_i e sobrecargas co_i e no_i de cada processador p_i, aliada com a utilização dos mecanismos oferecidos pelo *framework EasyGrid AMS* e pelos algoritmos de múltiplas iterações levaram a resultados interessantes.

5.2 Aplicações de Cargas Divisíveis Analisadas

Para cada aplicação de carga divisível a ser analisada, foram desenvolvidas versões utilizando-se os algoritmos *1-round*, PWD, UMR, MRRS, conforme vistos no Capítulo 3 e ainda, *EasyGrid UMR*, *EasyGrid MRRS*, *EasyGridMulti UMR* e *EasyGridMulti MRRS*. A seguir serão descritas as principais características de cada versão desenvolvida.

- *1-round*: A versão *1-round* consiste em dividir toda a carga de trabalho pelo número de recursos disponíveis em uma única iteração. As características da aplicação e dos recursos computacionais a serem utilizados não são consideradas.

- PWD: A versão PWD representa a implementação da metodologia de divisão de carga proposta pelo algoritmo PWD [36], apresentado na Seção 3.3, sendo este um algoritmo *híbrido*, pois conforme as características da aplicação, pode conter escalonamento estático e dinâmico para divisão de sua carga de trabalho. Além disso, PWD também se baseia nas capacidades computacionais dos recursos para sua divisão de carga.

- UMR e MRRS: As versões UMR e MRRS representam a implementação dos algoritmos UMR e MRRS, conforme [43] e [25], respectivamente. Utilizam a divisão de carga em múltiplas iterações para evitar que recursos computacionais fiquem ociosos enquanto aguardam pela chegada de trabalho.

- *EasyGrid UMR* e *EasyGrid MRRS*: Estas versões representam execuções utilizando-se o sistema gerenciador de aplicações *EasyGrid AMS*, onde o escalonamento inicial da aplicação e o número de recursos selecionados são obtidos segundo as definições dos algoritmos UMR e MRRS respectivamente. Além disso, nestas versões cada fatia de carga em cada iteração é processada por um processo independente, de acordo com o modelo de execução alternativo *1PTask*.

- *EasyGridMulti UMR* e *EasyGridMulti MRRS*: Estas versões são similares às duas citadas anteriormente, porém, utilizam as variantes dos algoritmos UMR e MRRS

desenvolvidas neste trabalho (Seção 4.3) para especificarem um número maior de iterações, visando criar um número maior de tarefas com granularidades menores.

5.3 Ambiente de Execução

Dois sistemas computacionais foram utilizados neste trabalho para a execução das aplicações de cargas divisíveis e serão descritos a seguir:

1. *Cluster Sinergia*: composto por oito nós com dois processadores Intel Xeon E5410 2.33GHz Core 2 Quad cada, totalizando 64 núcleos de processamento. Cada nó do sistema possui 16GB de memória RAM e 12MB de memória cache L2 para cada processador, sendo 6MB compartilhados por cada par de núcleos. Por fim, todos os nós executam o sistema operacional Linux CentOS 5.2 com o *kernel* 2.6.18-92, estão conectados através de uma rede *Gigabit Ethernet* e possuem a biblioteca de paralelização MPI LAM em sua versão 7.1.4.

2. *Cluster Oscar*: formado por 40 nós com dois processadores Intel Xeon X5355 2.66GHz Core 2 Quad cada, totalizando 320 núcleos de processamento. Cada nó possui 16GB de memória RAM e 8MB de memória cache L2 para cada processador, sendo 4MB compartilhados por cada par de núcleos. Todos os nós executam o sistema operacional Red Hat Enterprise Linux Server release 5.3 com o *kernel* 2.6.18-128.1.6, estão conectados através de uma rede *Gigabit Ethernet* e também possuem a biblioteca de paralelização MPI LAM versão 7.1.4.

Um dos nós de cada *cluster* é reservado para a execução do processo mestre v_0 da aplicação de divisão de carga e também dos gerenciadores global (GM) e local (SM). Note que tais processos gerenciadores só existem para as versões que utilizam o sistema gerenciador de aplicações *EasyGrid AMS*. Já os nós restantes (sete no *cluster Sinergia* e 39 no *cluster Oscar*) executam os processos trabalhadores da aplicação e os gerenciadores da máquina (HM). Com essa divisão de nós, as tarefas trabalhadoras permanecem isoladas da tarefa mestre e dos gerenciadores global e local, facilitando assim a análise de resultados no que tange a utilização dos recursos computacionais compartilhados (como por exemplo, a memória *cache*). Em cada um desses nós serão utilizados de um a oito núcleos para o processamento da carga de trabalho, sendo cada núcleo considerado um recurso computacional p_i independente. Logo, o *cluster Sinergia* poderá disponibilizar uma faixa entre sete (um núcleo disponível por nó) e 56 (oito núcleos disponíveis por nó) recursos

computacionais p_i para serem utilizados, enquanto o *cluster Oscar* poderá similarmente disponibilizar de 39 a 312 recursos. É importante lembrar que para todas as versões de algoritmos analisadas, somente um processo é executado por vez em cada núcleo utilizado.

Em todas as versões desenvolvidas, cada algoritmo de divisão de carga recebe como entrada a quantidade máxima de recursos computacionais disponíveis n_d, que na arquitetura alvo corresponde ao produto entre o número de nós e a quantidade de núcleos disponibilizados para a aplicação. A taxa de utilização dos núcleos de cada nó também será avaliada neste trabalho. Ainda, em todos os experimentos realizados ao longo deste trabalho, tanto a política de seleção de recursos do algoritmo UMR quanto a do MRRS determinaram que todos os n_d recursos computacionais fossem utilizados na execução de cada carga de trabalho. Isto ocorre porque o número de recursos computacionais disponíveis nas plataformas aqui utilizadas não é grande o suficiente para que os citados algoritmos selecionem um conjunto menor de processadores (conforme suas respectivas políticas de seleção apresentadas no Capítulo 3). Para o algoritmo UMR por exemplo, cuja política de seleção consiste em selecionar os n processadores tal que $\sum_{i=1}^{n} \frac{S_i}{B_i} < 1$, no experimento de multiplicação matrizes com 10.000×10.000 elementos, onde os valores de S_i e B_i (obtidos experimentalmente) são respectivamente 0.85 (unidades de carga processadas por segundo) e 11.500 (unidades de carga transmitidas de p_0 para p_i por segundo), poderiam ser selecionados até 13.529 processadores, caso estivessem disponíveis.

Os sistemas computacionais (*cluster Sinergia* e *cluster Oscar*) foram configurados de diferentes formas para representarem diferentes ambientes de execução, os quais serão descritos a seguir:

- Ambiente 1: *Cluster Sinergia* com recursos computacionais homogêneos totalmente dedicados à execução da aplicação de carga divisível.
- Ambiente 2: *Cluster Sinergia* heterogêneo controlado, onde um conjunto de tarefas externas (24 no total, sendo 8 em cada nó) são submetidas a três nós que executam os processos trabalhadores, visando reduzir seus respectivos poderes computacionais em até 50%. Tais tarefas são executadas durante todo o processamento da aplicação de carga divisível, são de natureza *cpu-intensive* e não realizam constantes acessos aos recursos de memória, visando somente reduzir o poder computacional dos recursos aos quais estão alocadas. Ainda, neste ambiente todos os experimentos foram realizados com o escalonador dinâmico do *EasyGrid AMS* desabilitado para as versões que o utilizam.

- Ambiente 3: *Cluster Sinergia* dinâmico controlado, com um conjunto de tarefas externas que são submetidas a três nós cinco segundos após o início da execução da aplicação de carga divisível, visando assim prover uma característica dinâmica ao ambiente.

- Ambiente 4: *Cluster Oscar* homogêneo e dedicado exclusivamente à aplicação de divisão de carga.

- Ambiente 5: *Cluster Oscar* dinâmico controlado, com um conjunto de tarefas externas submetidas a dez de seus nós. Novamente, tais tarefas externas são criadas cinco segundos após o início da execução da aplicação, permanecem ativas durante todo o processamento da carga de trabalho e também são de natureza *cpu-intensive*.

Os resultados dos experimentos a serem analisados neste capítulo representam a média de tempo (em segundos) de cinco execuções das aplicações de multiplicação de matrizes, *Mandelbrot set computation fractal* e de Filtro Gaussiano com as versões *1-round*, PWD, UMR, MRRS, *EasyGrid UMR*, *EasyGrid MRRS*, *EasyGridMulti UMR* e *EasyGridMulti MRRS*. Em algumas destas execuções foi observado um aumento na temperatura dos processadores de alguns dos nós que compõem o *cluster Sinergia* ao se utilizar a maioria de seus núcleos de processamento. Ao detectar este aumento de temperatura, o sistema operacional se encarrega de reduzir a velocidade de processamento destes processadores (*cpu clock*) até o momento em que suas temperaturas se normalizem. O resultado desta ação deteriora o desempenho das aplicações executadas, fazendo com que os nós finalizem suas execuções com tempos computacionais distintos. As execuções em que ocorreram tais eventos foram descartadas, e somente foram consideradas as cinco execuções com tempos computacionais similares, isto é, aquelas cuja diferença entre os tempos apresentados por cada nó não ultrapassaram a dez segundos.

As execuções das aplicações sob o gerenciamento do *EasyGrid AMS* podem ter sua camada de escalonamento dinâmico habilitada ou não. Neste caso, os seguintes cenários foram considerados: *cluster* homogêneo com a camada de escalonamento dinâmico habilitada e desabilitada; heterogêneo com a camada de escalonamento dinâmico desabilitada; e também ambientes dinâmicos controlados. Os cenários serão devidamente indicados.

5.4 Análise da Aplicação Multiplicação de matrizes

A primeira aplicação selecionada para fazer parte desta avaliação de desempenho foi a de multiplicação de matrizes. Por definição, dadas duas matrizes de entrada $A_{m,n}$ (contendo m linhas e n colunas) e $B_{n,p}$ (contendo n linhas e p colunas), é calculado o produto $C_{m,p} = A_{m,n} \times B_{m,p}$ (contendo m linhas e p colunas).

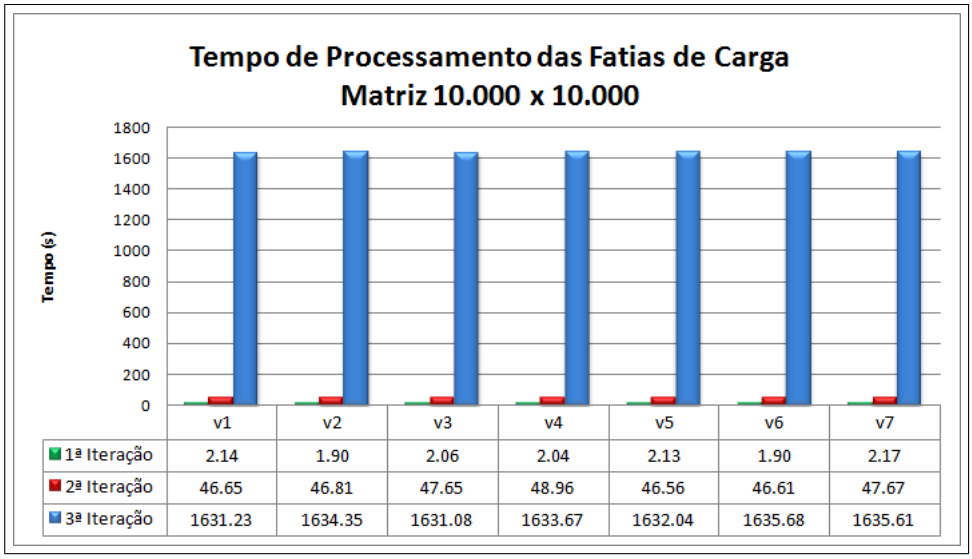
Na versão paralela do algoritmo de multiplicação de matrizes desenvolvida neste trabalho somente o processo mestre v_0 possui a matriz de entrada $A_{m,n}$, a qual representa a carga de trabalho a ser processada. Para a modelagem desta aplicação segundo a teoria de divisão de cargas, foi definido que cada linha da matriz de entrada $A_{m,n}$ representaria uma unidade da carga de trabalho a ser processada (unidade também adotada em [36]), logo, $W_{total} = m$. Cada processo trabalhador v_i possui uma cópia da segunda matriz de entrada $B_{n,p}$ e aguarda pela chegada das respectivas linhas (fatias) da matriz $A_{m,n}$, as quais será encarregado de efetuar a multiplicação. Ainda, cada processo ou tarefa v_i recebe um conjunto de informações de v_0, que indica a quantidade de linhas a serem multiplicadas e a devida fatia de carga, ou seja, as linhas da matriz $A_{m,n}$. Após efetuarem a multiplicação das suas fatias de $A_{m,n}$ por $B_{n,p}$, os processos trabalhadores retornam para o processo v_0 o resultado desta multiplicação, o qual deverá ser armazenado em $C_{m,p}$.

A Tabela 5.1 mostra a distribuição de uma matriz $A_{10.000,10.000}$ de acordo com os algoritmos UMR e MRRS, para um conjunto de sete processos trabalhadores v_i em um ambiente com sete recursos computacionais homogêneos (Ambiente 1 com um núcleo de cada nó sendo utilizado). De acordo tanto com o algoritmo UMR quanto com o MRRS (ambos produziram o mesmo particionamento) esta carga de trabalho ($W_{total} = 10.000$) deverá ser escalonada em três iterações e todos os sete recursos disponíveis deverão ser utilizados. Cada unidade de carga enviada para os processos trabalhadores representa uma linha da matriz $A_{10.000,10.000}$, logo, as colunas da Tabela 5.1 representam o número de linhas associadas a cada processo em cada iteração. Ainda, a Figura 5.1 mostra o tempo necessário para que cada processo v_i execute sua respectiva fatia de carga em cada uma das três iterações, utilizando a versão com o algoritmo UMR.

Iteração	v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	1	1	1	1	1	1	1
2	43	43	44	44	44	44	44
3	1383	1384	1384	1384	1384	1384	1384

Tabela 5.1: *Exemplo de distribuição de carga (em número de linhas) segundo os algoritmos UMR e MRRS.*

Conforme as definições dos algoritmos UMR e MRRS, uma pequena quantidade de carga é inicialmente enviada para os processos trabalhadores visando o início de execução desses processos o mais breve possível. Na sequência, uma quantidade maior de carga é enviada enquanto estes processos ainda estão executando a fatia inicial. Por fim, toda a carga restante é enviada em uma última iteração, pois o tempo necessário para se processar a quantidade de carga relativa a segunda iteração é grande o suficiente para que toda a carga restante possa ser enviada, conforme definido em [25, 43]. A explicação para o fato de os processos receberem diferentes quantidades de trabalho em uma mesma iteração, apesar do ambiente ser homogêneo, é que o cálculo do tamanho de cada fatia gera números fracionários. Portanto, somente a parte inteira do valor obtido é utilizada para referenciar a quantidade de linhas a serem enviadas para cada processo, enquanto a parte fracionária é acumulada até totalizar uma nova unidade de carga, que será adicionada a fatia destinada ao próximo processo.



	v1	v2	v3	v4	v5	v6	v7
1ª Iteração	2.14	1.90	2.06	2.04	2.13	1.90	2.17
2ª Iteração	46.65	46.81	47.65	48.96	46.56	46.61	47.67
3ª Iteração	1631.23	1634.35	1631.08	1633.67	1632.04	1635.68	1635.61

Figura 5.1: *Tempo de processamento das fatias de carga para uma matriz de 10.000×10.000 elementos.*

O particionamento de $A_{10.000,10.000}$ nas versões *EasyGrid UMR* e *EasyGrid MRRS* é semelhante ao apresentado na Tabela 5.1 e na Figura 5.1 para as versões UMR e MRRS, porém, segue o modelo de execução *1Ptask*, onde cada fatia de carga é executada por um processo trabalhador distinto, sendo que em cada iteração apenas um processo é executado em cada núcleo. Por sua vez, nas versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* a carga relativa a última iteração é particionada em um conjunto de novas iterações de tamanhos menores, conforme a adaptação dos algoritmos UMR e MRRS proposta neste trabalho e apresentada no Algoritmo 6 (Seção 4.3). Desta forma, são obtidas 32 novas

iterações (resultado da divisão entre a carga relativa a última iteração pela carga relativa a penúltima iteração) cujas fatias de carga possuem aproximadamente 43 unidades cada.

A Tabela 5.2 apresenta a distribuição da matriz $A_{10.000,10.000}$ para o mesmo conjunto de sete processos trabalhadores v_i no mesmo ambiente homogêneo (Ambiente 1) segundo as definições dos Algoritmos *1-round* e PWD. Para este ambiente homogêneo, os dois algoritmos produziram o mesmo particionamento. O algoritmo PWD é equipado com uma política de distribuição de carga de acordo com as características dos recursos disponíveis, porém, como o ambiente é homogêneo todos os processos irão receber a mesma quantidade de trabalho, resultando assim no mesmo particionamento produzido pelo algoritmo *1-round*. Outra característica acerca do algoritmo PWD é que para este exemplo de aplicação, a qual é composta por uma carga de trabalho regular (todas unidades de carga demandam o mesmo tempo de processamento), toda a carga é particionada em uma única iteração na primeira fase do algoritmo (fase estática), conforme definido na Seção 3.3.

Iteração	v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	1428	1428	1428	1429	1429	1429	1429

Tabela 5.2: *Exemplo de distribuição de carga segundo os algoritmos 1-round e PWD.*

A seguir, serão descritos os desempenhos obtidos com cada versão de divisão de carga, ao serem executadas em diferentes ambientes (homogêneos, heterogêneos e por fim, em ambientes dinâmicos).

5.4.1 Multiplicação de Matrizes de 10.000×10.000 Elementos no Ambiente 1 com o Escalonador Dinâmico Desabilitado

As informações contidas na Tabela 5.3 ilustram o comportamento das diferentes políticas de divisão de carga, quando executadas no *cluster Sinergia* (Ambiente 1) para uma carga de trabalho contendo 10.000 unidades ($A_{10.000,10.000}$). Note que para este ambiente a camada de escalonamento dinâmico do *EasyGrid AMS* permanece desabilitada, não havendo portanto re-escalonamento dinâmico de tarefas. Também são apresentados os resultados obtidos ao aumentar gradativamente o número de núcleos sendo utilizados em cada nó do ambiente, indicados na primeira coluna da Tabela 5.3. É importante lembrar que a quantidade de processos executados paralelamente em cada nó corresponde ao número de núcleos sendo utilizados, assim, cada nó poderá executar de um (apenas um núcleo sendo utilizado) a oito (oito núcleos sendo utilizados) processos paralelamente. Os rótulos EGUMR, EGMRRS, EGMUMR e EGMMRRS correspondem respectivamente às versões *EasyGrid UMR*, *EasyGrid MRRS*, *EasyGridMulti UMR* e *EasyGridMulti MRRS*.

Neste experimento, a carga de trabalho a ser processada necessita de um espaço de memória contendo 400 MB ($10.000 \times 10.000 \times 4$ *bytes*) livres para que a matriz $A_{10.000,10.000}$ seja armazenada (o mesmo também vale para as matrizes $B_{10.000,10.000}$ e $C_{10.000,10.000}$). Ainda, tempo necessário para que a matriz $A_{10.000,10.000}$ seja transmitida para os processos trabalhadores envolve os tempos gastos com os eventos de alocação de memória, inserção de valores nesta porção alocada e por fim, com seu envio para os respectivos processos destinatários.

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	1690.75	1691.21	1687.63	1687.21	1685.58	1685.51	1555.13	1556.01
2	855.50	855.92	852.49	852.14	849.75	849.73	786.58	786.58
3	573.95	573.03	570.15	570.02	569.17	568.17	526.74	527.06
4	419.02	419.54	416.45	416.41	415.86	415.85	396.83	397.80
5	372.42	373.70	369.56	370.55	368.73	368.77	338.69	338.67
6	320.43	319.37	316.47	316.90	315.66	314.67	303.62	304.61
7	278.07	279.62	276.94	275.78	274.61	274.61	272.56	272.55
8	**258.19**	**258.17**	**255.63**	**255.10**	**254.65**	**254.54**	**247.50**	**247.51**

Tabela 5.3: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 10.000×10.000 elementos no Ambiente 1 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

Analisando as informações contidas na Tabela 5.3, podemos observar uma pequena vantagem das versões que utilizam os algoritmos de múltiplas iterações, devido à sobreposição entre computação e comunicação. Ainda, o tempo de execução de todas as versões segue caindo na medida em que mais núcleos de processamento são utilizados. Isto ocorre porque para esta aplicação e para esta carga de trabalho é mais vantajoso utilizar uma quantidade maior de recursos e consequentemente, processar a carga de trabalho total com um número maior de processos, apesar do compartilhamento da memória *cache*. Em todas as versões, ao se utilizar um número maior de núcleos ocorre um aumento do número de processos sendo executados paralelamente em cada nó (um processo executando em cada núcleo). Para esta carga de trabalho, não ocorre uma piora nos tempos de execução ao aumentar o número de processos executados paralelamente porque a quantidade de memória *cache* disponível para cada núcleo é grande o suficiente para que mais de um processos a compartilhe eficientemente, desta forma, o número de eventos *cache-miss* ocorridos não degrada a execução da aplicação. Quando o processador necessita utilizar um determinado dado e este não está presente na memória *cache*, é feita uma busca diretamente na memória RAM e isto acaba reduzindo seu desempenho. Como provavelmente será requisitado novamente (localidade temporal) em instruções futuras, tanto o dado que foi buscado na memória RAM quanto um conjunto de seus vizinhos são copiados para

a memória *cache* para que possam ser acessados por instruções futuras de maneira mais rápida. Cenários analisados na sequência deste trabalho irão avaliar o comportamento desta aplicação quando a quantidade de memória *cache* não comportar a quantidade de dados necessários, implicando assim em um aumento do número de eventos *cache-miss*.

Ainda em relação à Tabela 5.3, os tempos de execuções das versões *1-round* e PWD são bastante próximos em virtude de serem algoritmos em que toda a carga de trabalho é dividida entre os recursos trabalhadores em uma única iteração, não havendo assim sobreposição entre computação e comunicação. Já as versões UMR e MRRS são mais eficientes que as versões *1-round* e PWD pois sua carga de trabalho, que para esta instância é de 400 MB, é enviada para os processos trabalhadores em múltiplas iterações. Por sua vez, os tempos obtidos com as versões *EasyGrid UMR* e *EasyGrid MRRS* para esta carga de trabalho apresentam uma pequena vantagem em relação as versões UMR e MRRS. Isto ocorre devido ao modelo de execução *1PTask* adotado pelo *EasyGrid AMS*, onde cada fatia de carga $chunk_{i,j}$ é processada por uma tarefa independente. Estas tarefas são criadas dinamicamente a partir do momento em que sua carga de trabalho se encontra disponível (quando a respectiva fatia de carga termina de ser recebida) e apresentam granularidades menores (processam fatias de carga de tamanhos menores) do que aquelas definidas no modelo de execução *1PProc*, o qual é utilizado nas versões *1-round*, PWD, UMR e MRRS. Conforme será discutido na sequência deste trabalho, ao processar fatias de cargas de tamanhos menores estes processos acessam faixas de endereços de memória menores, reduzindo o retardo sofrido devido a problemas de contenção de memória.

O trabalho apresentado em [13] demonstrou que a utilização de tarefas de granularidades finas em aplicações computacionalmente intensivas leva a uma utilização mais eficiente dos recursos computacionais em ambientes heterogêneos, dinâmicos, compartilhados e compostos por máquinas **monoprocessadas**. Similarmente, será analisado neste trabalho o impacto causado pela utilização de tarefas de granularidades menores em ambientes dedicados e compostos por máquinas **multiprocessadas**, onde os recursos de memória são geralmente compartilhados entre um conjunto de núcleos e processadores. Para realizar esta análise, foram desenvolvidas as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*, as quais possuem como característica fundamental a divisão da carga de trabalho entre um conjunto maior de tarefas que consequentemente, apresentam granularidades menores. A Tabela 5.3 mostra que para esse experimento as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* chegaram a apresentar um ganho de aproximadamente 8% em relação as versões *EasyGrid UMR* e *EasyGrid MRRS*. Com a utilização de tarefas de granularidades menores ocorrem menos atrasos relacionados a contenção de memória

(como por exemplo os eventos *cache-miss*), pois a faixa de endereços de memória buscados por estes processos são menores. Uma análise sobre a faixa de endereços que cada processo necessita acessar durante a execução de suas fatias de carga será feita na Seção 5.4.2, onde serão analisadas matrizes com dimensões maiores.

A Tabela 5.4 apresenta o número de iterações calculado para cada uma das versões dos algoritmos de múltiplas iterações, variando o número de núcleos utilizados, para a multiplicação das matrizes $A_{10.000,10.000}$ e $B_{10.000,10.000}$. Já a Tabela 5.5 traz o número total de processos trabalhadores criados em cada versão para esta mesma carga de trabalho, também variando o número de núcleos utilizados. Para as versões *EasyGrid UMR*, *EasyGrid MRRS*, *EasyGridMulti UMR* e *EasyGridMulti MRRS*, o total de processos trabalhadores a serem criados corresponde ao produto entre o número de iterações calculado, a quantidade de nós disponíveis e o número de núcleos utilizados (*iterações*×*nós*×*núcleos*), conforme o modelo de exucução *1PTask*. Para as demais versões, o número de processos trabalhadores criados corresponde ao produto entre a quantidade de nós disponíveis e o número de núcleos utilizados (*nós*×*núcleos*), conforme o modelo de execução *1PProc*.

Núcleos	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	3	3	3	3	34	34
2	3	3	3	3	28	28
3	4	4	4	4	20	20
4	4	4	4	4	16	16
5	4	4	4	4	14	14
6	4	4	4	4	13	13
7	4	4	4	4	12	12
8	5	5	5	5	12	12

Tabela 5.4: *Número de iterações calculadas ao variar o número de núcleos utilizados no Ambiente 1, para a multiplicação de matrizes de dimensões 10.000×10.000.*

Núcleos	1-round	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	7	7	7	7	21	21	238	238
2	14	14	14	14	42	42	392	392
3	21	21	21	21	84	84	420	420
4	28	28	28	28	112	112	448	448
5	35	35	35	35	140	140	490	490
6	42	42	42	42	168	168	546	546
7	49	49	49	49	196	196	588	588
8	56	56	56	56	280	280	672	672

Tabela 5.5: *Número total de processos criados ao variar o número de núcleos utilizados no Ambiente 1, para a multiplicação de matrizes de dimensões 10.000×10.000.*

Conforme apresentado na Tabela 5.5, a quantidade total de processos de aplicação que são criados segue aumentando na medida em que mais núcleos são utilizados. Este aumento do número de processos implica num custo maior para aplicação, principalmente para aquelas que utilizam o sistema gerenciador *EasyGrid AMS*, devido a sobrecarga incorrida com a criação e gerenciamento de tais processos. No entanto, mesmo com um número maior de processos as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* são vantajosas em relação as demais quando comparados os tempos de execuções. Todavia, conforme visto na Tabela 5.6 os percentuais de melhoria são menores para um número maior de núcleos (ou seja, um número maior de tarefas), devido a esta sobrecarga de criação e gerenciamento dos processos.

Núcleos	$\frac{UMR-EGUMR}{UMR}$	$\frac{MRRS-EGMRRS}{MRRS}$	$\frac{EGUMR-EGMUMR}{EGUMR}$	$\frac{EGMRRS-EGMMRRS}{EGMRRS}$
1	0,121	0,100	7,739	7,683
2	0,322	0,282	7,433	7,431
3	0,171	0,324	7,454	7,235
4	0,141	0,134	4,576	4,340
5	0,224	0,480	8,146	8,162
6	0,255	0,703	3,814	3,197
7	0,841	0,424	0,746	0,750
8	0,383	0,219	2,807	2,761

Tabela 5.6: *Percentual de ganho ao se utilizar o sistema gerenciador EayGrid AMS com a aplicação de multiplicação de matrizes de* 10.000×10.000 *elementos no Ambiente 1.*

5.4.2 Multiplicação de Matrizes de 12.000×12.000 Elementos no Ambiente 1 com o Escalonador Dinâmico Desabilitado

O cenário a seguir, apresentado na Tabela 5.7, exemplifica uma instância de carga de trabalho em que o simples fato de aumentar a quantidade de recursos computacionais disponíveis não implica numa imediata redução no tempo de execução da aplicação em um sistema de processadores *multicore*, contrariando o comportamento notado no cenário analisado anteriormente. A carga de trabalho a ser processada agora é representada pela matriz $A_{12.000,12.000}$ e o ambiente de execução utilizado foi novamente o *cluster Sinergia* homogêneo e com a camada de escalonamento dinâmico do *EasyGrid AMS* desabilitada (Ambiente 1). O objetivo destes experimentos é avaliar o impacto que o aumento da quantidade de trabalho causa na hierarquia de memória para a aplicação de multiplicação de matrizes.

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	2928.12	2929.45	2926.50	2927.71	2926.90	2926.13	2763.56	2763.55
2	1462.84	1463.76	1460.07	1459.55	1457.02	1457.01	1363.74	1361.74
3	991.31	991.90	988.68	987.11	987.13	987.00	909.83	910.83
4	**733.59**	**733.29**	**730.13**	**730.07**	**729.49**	**729.48**	701.42	700.38
5	923.44	922.52	919.20	919.26	873.69	875.89	684.38	686.74
6	819.71	820.34	817.99	817.16	741.54	740.50	**675.36**	**674.39**
7	821.00	820.91	815.22	815.04	762.77	763.57	684.52	686.41
8	803.37	803.49	798.11	799.35	733.63	733.81	701.43	702.62

Tabela 5.7: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 12.000×12.000 elementos no Ambiente 1 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

Para a instância cuja quantidade de dados alocada é maior, a Tabela 5.7 mostra que para a maioria das políticas, os melhores resultados foram alcançados ao utilizar apenas quatro núcleos de cada nó. A queda de desempenho das versões *1-round*, PWD, UMR, MRRS, *EasyGrid UMR* e *EasyGrid MRRS* ao se utilizar cinco, seis, sete e oito núcleos de cada nó ocorre devido ao aumento do número de eventos *cache-miss* nestes recursos. Para arquiteturas *multicore* ou *multiprocessadas*, enquanto houver unidades de processamento ociosas, o escalonador do sistema operacional Linux utiliza uma política *round-robin* para o escalonamento de processos entre os processadores livres [16]. Assim, de acordo com as características do ambiente de execução alvo, no cenário em que quatro núcleos são utilizados dois processos residem no primeiro processador, utilizando dois de seus núcleos, enquanto os outros dois processos ocupam o segundo processador da mesma maneira, conforme ilustrado na Figura 5.2 (a). Neste instante, cada processo dispõe de uma memória *cache* L2 com 6MB de espaço. Uma vez que esta *cache* não é compartilhada por processos distintos, a quantidade de eventos *cache-miss* que ocorrem não degrada a performance da aplicação.

A partir do momento em que cinco ou mais processos são lançados paralelamente, dois ou mais destes processos passam a ocupar núcleos vizinhos dentro de um mesmo processador, também ilustrado na Figura 5.2 (b). Como a quantidade de memória *cache* L2 disponível, compartilhada entre cada par de núcleos, não é grande o suficiente para comportar a quantidade de dados de dois ou mais processos concorrentemente, um número maior de eventos *cache-miss* passa a ocorrer. A maior parte destes eventos ocorre durante o processamento das últimas iterações, onde são encontradas as maiores fatias de carga, conforme especificado pelas versões *1-round*, PWD, UMR, MRRS, *EasyGrid UMR* e *EasyGrid MRRS*. Ao longo da execução destas fatias, cada processo trabalha com uma extensa faixa de endereços de memória, sendo que quanto maior a diferença entre os endereços

buscados pelos processos que compartilham a mesma área da memória *cache* maior será o número de ocorrência de eventos *cache-miss*, pois esta passa a ser frequentemente renovada para conter os dados buscados por cada processo. Como consequência, parte do tempo de execução da aplicação é gasta com acessos a hierarquia de memória.

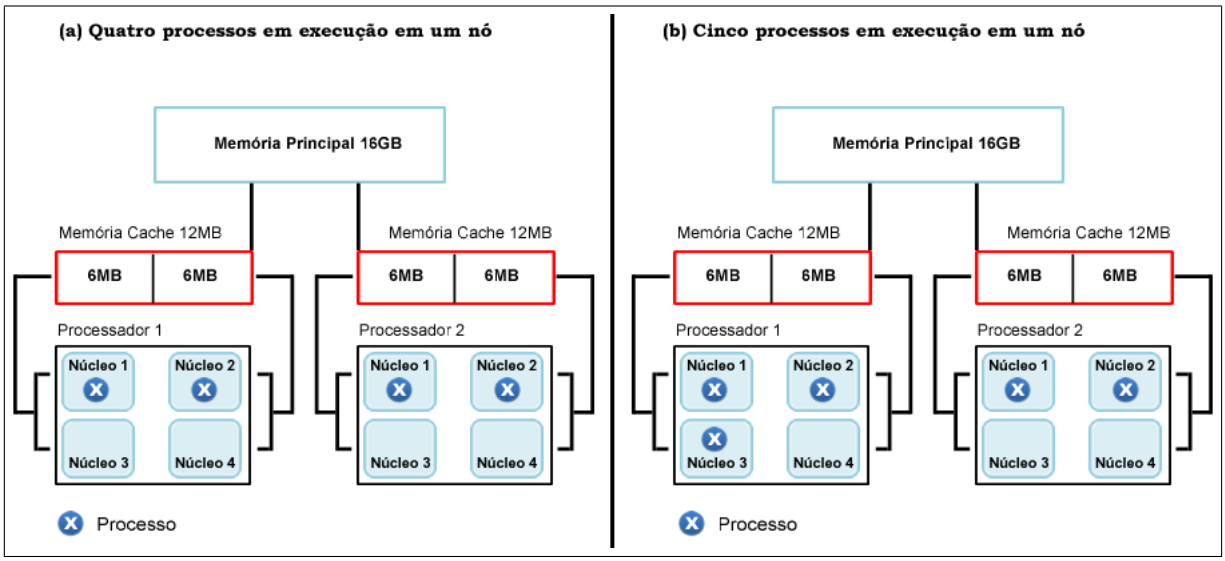


Figura 5.2: *Distribuição de processos entre os processadores de um nó do cluster Sinergia.*

Neste experimento, a quantidade de memória ocupada pela matriz $A_{12.000,12.000}$ é de 576 MB (12.000×12.000×4 *bytes*) e cada linha desta matriz ocupa 48 KB (12.000×4 *bytes*) de espaço. As matrizes $B_{12.000,12.000}$ e $C_{12.000,12.000}$ também ocupam esta mesma quantidade de espaço. As Tabelas 5.8 e 5.9 mostram as faixas de endereços virtuais acessados por cada processo da aplicação durante o processamento desta carga de trabalho ao se utilizar cinco núcleos em um dos nós do ambiente de execução. Note que estas tabelas descrevem somente as faixas de endereços acessados durante a manipulação da matriz $A_{12.000,12.000}$. Operações de leitura e escrita nas matrizes $B_{12.000,12.000}$ e $C_{12.000,12.000}$ também implicam em novos acessos a hierarquia de memória e consequentemente, novos atrasos durante a execução da aplicação ocorrem. Estes dados foram obtidos a partir de duas situações: uma utilizando a versão MRRS (Tabela 5.8), a qual definiu que esta carga de trabalho deveria ser escalonada em quatro iterações, e outra utilizando a versão *EasyGridMulti MRRS* (Tabela 5.9), que por sua vez definiu que esta mesma carga de trabalho deveria ser escalonada em 15 iterações. A primeira coluna contém o número de cada iteração, enquanto a segunda indica os números (ou *ranks*) dos processos trabalhadores executados no referido nó. Note que na versão *EasyGridMulti MRRS* cada fatia de carga é executada por um processo trabalhador distinto (*1PTask*), sendo que em cada núcleo, um único processo é executado em cada iteração. Já na versão MRRS, um processo é executado

por núcleo durante toda a execução da aplicação (*1PProc*). A terceira coluna mostra o tamanho da fatia de carga recebida por cada processo em cada iteração, enquanto as colunas quatro, cinco e seis representam as faixas de endereços virtuais acessados por estes processos. A diferença entre o endereço final e o inicial é dada pelo produto entre a quantidade de linhas recebidas em cada iteração e o espaço de memória que cada linha da matriz $A_{12.000,12.000}$ ocupa ($48.000 \times chunk_{i,j}$). A maior parte dos eventos *cache-miss* ocorrem durante a execução das iterações de maiores fatias, devido à grande quantidade de endereços de memória acessados por cada processo, implicando em constantes renovações das informações contidas na memória *cache*.

Iteração j	Processo v_i	Quant. Linhas	End. Inicial	End. Final	Qtde Endereços
0	1	1	0	47.999	48.000
0	2	1	48.000	95.999	48.000
0	3	1	96.000	143.999	48.000
0	4	1	144.000	191.999	48.000
0	5	1	192.000	239.999	48.000
1	1	2	240.000	335.999	96.000
1	2	2	336.000	431.999	96.000
1	3	3	432.000	575.999	144.000
1	4	2	576.000	671.999	96.000
1	5	3	672.000	815.999	144.000
2	1	26	816.000	2.063.999	1.248.000
2	2	27	2.064.000	3.359.999	1.296.000
2	3	27	3.360.000	4.655.999	1.296.000
2	4	27	4.656.000	5.951.999	1.296.000
2	5	27	5.952.000	7.247.999	1.296.000
3	1	312	7.248.000	22.223.999	14.976.000
3	2	313	22.224.000	37.247.999	15.024.000
3	3	312	37.248.000	52.223.999	14.976.000
3	4	313	52.224.000	67.247.999	15.024.000
3	5	313	67.248.000	82.271.999	15.024.000

Tabela 5.8: *Faixas de endereços virtuais acessados por cada processo trabalhador em um nó utilizando cinco núcleos com a versão MRRS.*

Iteração j	Processo v_i	Quant. Linhas	End. Inicial	End. Final	Qtde Endereços
0	1	1	0	47.999	48.000
0	2	1	48.000	95.999	48.000
0	3	1	96.000	143.999	48.000
0	4	1	144.000	191.999	48.000
0	5	1	192.000	239.999	48.000
1	6	2	240.000	335.999	96.000
1	7	2	336.000	431.999	96.000
1	8	2	432.000	527.999	96.000
1	9	2	528.000	623.999	96.000
1	10	3	624.000	767.999	144.000
2	11	26	768.000	2.015.999	1.248.000
2	12	26	2.016.000	3.263.999	1.248.000
2	13	26	3.264.000	4.511.999	1.248.000
2	14	26	4.512.000	5.759.999	1.248.000
2	15	26	5.760.000	7.007.999	1.248.000
3	16	26	7.008.000	8.255.999	1.248.000
3	17	26	8.256.000	9.503.999	1.248.000
3	18	26	9.504.000	10.751.999	1.248.000
3	19	26	10.752.000	11.999.999	1.248.000
3	20	26	12.000.000	13.247.999	1.248.000
	...	...	...	...	...
13	66	26	69.408.000	70.655.999	1.248.000
13	67	26	70.656.000	71.903.999	1.248.000
13	68	27	71.904.000	73.199.999	1.296.000
13	69	27	73.200.000	74.495.999	1.296.000
13	70	27	74.496.000	75.791.999	1.296.000
14	71	27	75.792.000	77.087.999	1.296.000
14	72	27	77.088.000	78.383.999	1.296.000
14	73	27	78.384.000	79.679.999	1.296.000
14	74	27	79.680.000	80.975.999	1.296.000
14	75	27	80.976.000	82.271.999	1.296.000

Tabela 5.9: *Faixas de endereços virtuais acessados por cada processo trabalhador em um nó utilizando cinco núcleos com a versão EasyGridMulti MRRS.*

Ainda em relação as informações contidas na Tabela 5.7, observa-se que a queda de desempenho da aplicação nas versões *EasyGrid UMR* e *EasyGrid MRRS* ao se utilizar mais do que quatro núcleos é menor do que a ocorrida nas versões *1-round*, PWD, UMR e MRRS. Sua explicação reside na utilização do modelo de execução alternativo *1PTask* adotado pelo *EasyGrid AMS* e que define processos de granularidades menores. Consequentemente, a quantidade de memória necessária e compartilhada por diferentes processos é menor. Processos ou tarefas de granularidades menores buscam endereços de memória residentes em intervalos de tamanhos menores, implicando assim numa redução

da quantidade de eventos *cache-miss* e consequentemente no retardo sofrido pela aplicação. Apesar do custo adicional para a criação e gerenciamento de uma quantidade maior de processos, o ganho alcançado com a utilização mais eficiente dos recursos de memória proporcionou uma redução do tempo de execução para a aplicação de multiplicação de matrizes. Por fim, ao executarem esta carga de trabalho, as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* conseguiram manter uma redução no tempo de execução da aplicação com a utilização de até seis núcleos, chegando a apresentar uma melhora de até 22% no tempo de execução se comparado com as versões *EasyGrid UMR* e *EasyGrid MRRS* e 26% se comparado com as versões *1-round* e PWD, conforme observado na Tabela 5.7.

O número maior de iterações nas versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* em que a carga de trabalho é escalonada acarreta em um número maior de processos a serem criados, o que aumenta a sobrecarga relacionada a criação destes processos. Mesmo assim, para a aplicação de multiplicação de matrizes esta estratégia apresenta melhores resultados do que aquelas que utilizam um número menor de processos. Esta vantagem é atingida pois o tempo ganho ao se utilizar a memória de cada recurso de maneira mais eficiente compensa o tempo gasto com a criação de novos processos. Ao longo deste trabalho será avaliado um exemplo de aplicação em que ao se aumentar o número de iterações do particionamento da carga de trabalho ocorre uma degradação no tempo de execução.

Conforme observado nestes últimos resultados, as políticas de seleção de recursos dos algoritmos UMR e MRRS não foram eficientes, pois permitiram que mais recursos do que o que minimizasse o tempo de execução fossem utilizados, conforme observado nos cenários utilizando-se cinco ou mais núcleos por nó. Isto ocorre porque tais políticas se baseiam principalmente na relação entre custo de processamento e custo de envio de suas cargas de trabalho, não levando em conta características da carga de trabalho a ser executada (como por exemplo, se possui uma carga de trabalho regular, irregular, *cpu-intensive* ou *data-intensive*) e nem do compartilhamento de recursos em arquiteturas *multicore* ou multiprocessadas.

5.4.3 Multiplicação de Matrizes de 10.000×10.000 Elementos no Ambiente 1 com o Escalonador Dinâmico Habilitado

Nesta etapa de avaliação da aplicação de multiplicação de matrizes serão verificados os desempenhos das versões *EasyGrid UMR*, *EasyGrid MRRS*, *EasyGridMulti UMR* e *EasyGridMulti MRRS* com a camada de escalonamento dinâmico habilitada. Desta forma,

recursos computacionais poderão ceder um ou mais de seus processos trabalhadores para serem executados em outros recursos, caso seja detectado algum desbalanceamento de carga entre os nós trabalhadores. Apesar de o algoritmo PWD especificar uma etapa em que parte de sua carga de trabalho é enviada dinamicamente para os processos da aplicação, esta somente ocorre para aplicações cuja carga de trabalho seja de natureza irregular, isto é, suas unidades de carga possuem diferentes tempos de processamento. Como a aplicação de multiplicação de matrizes é composta por uma carga de trabalho totalmente regular (todas as linhas das matrizes $A_{10.000,10.000}$ e $B_{10.000,10.000}$ demandam a mesma quantidade de tempo para serem multiplicadas), a fase de escalonamento dinâmico do algoritmo PWD nunca é realizada para esta aplicação, conforme descrito na Seção 3.3. Diante deste cenário, tanto a versão com o algoritmo PWD quanto as versões *1-round*, UMR e MRRS não estão incluídas neste experimento pois não implementam escalonamento dinâmico em suas metodologias de divisão de carga. Os resultados dessas políticas seriam os mesmos que os apresentados na Tabela 5.3, pois o conhecimento passado para tais políticas sobre o sistema alvo no início da execução da aplicação seria que todos os recursos computacionais teriam o mesmo poder computacional, ou seja, recursos homogêneos.

O ambiente de execução utilizado neste experimento foi o *cluster Sinergia* dedicado e com o escalonador dinâmico do *EasyGrid AMS* ativo (Ambiente 1). Os resultados apresentados a seguir mostram o tempo de execução obtido e a existência ou não de eventos de escalonamento dinâmico. Para as amostras onde ocorreram eventos de escalonamento dinâmico, será avaliada a quantidade de trabalho que foi redistribuída entre os recursos disponíveis.

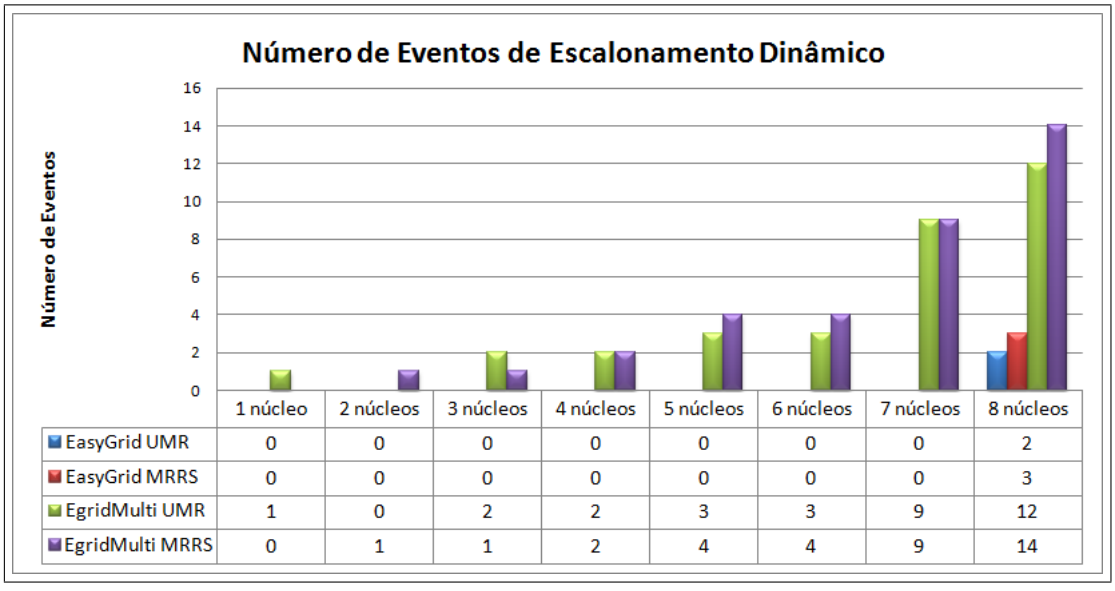
N	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	1694.54	1693.47	1560.16	1562.19
2	854.74	855.78	790.60	791.59
3	568.18	569.15	529.07	530.07
4	420.85	420.89	399.86	399.88
5	368.77	368.74	340.70	340.74
6	321.68	320.11	306.62	305.62
7	283.59	281.57	261.54	261.55
8	**252.54**	**252.55**	**236.52**	**237.48**

Tabela 5.10: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 10.000×10.000 elementos no Ambiente 1 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

De acordo com as informações apresentadas na Tabela 5.10, a habilidade de escalonamento dinâmico do *EasyGrid AMS* possibilitou um desempenho superior ao apresentado

no cenário com ambiente estático (Tabela 5.3) ao se utilizar sete e oito núcleos de cada nó do ambiente. Para as demais amostras, houve um acréscimo no tempo execução médio da aplicação devido ao custo computacional gerado pela camada escalonamento dinâmico do *EasyGrid AMS*.

A Figura 5.3 indica a quantidade de eventos de escalonamento dinâmico ocorridos durante a execução desta carga de trabalho ($A_{10.000,10.000}$). Note que a quantidade de eventos ocorridos não representa necessariamente a quantidade de processos que foram re-escalonados, pois em um único evento podem ser re-escalonados mais de um processo. É importante lembrar que o tamanho médio das fatias de carga segue caindo na medida em que mais unidades de processamento são disponibilizadas, pois a carga de trabalho passa a ser particionada entre um número maior de recursos. Além disso, o número de processos que cada nó deverá executar aumenta na medida em que mais de suas unidades de processamento são utilizadas, o que aumenta o grau de concorrência pelos recursos compartilhados e favorece a ocorrência dos eventos de escalonamento dinâmico. Este número de processos executados em cada nó também é proporcional ao número de iterações M em que a carga de trabalho é escalonada, devido a utilização do modelo de execução alternativo *1PTask*. Por fim, quanto maior o número de processos executados em cada nó e quanto menor forem suas granularidades, maior será a possibilidade de ocorrem eventos de escalonamento dinâmico, pois o re-escalonamento de tais processos não tenderia a provocar um alto grau de desbalanceamento nos tempos computacionais de cada recurso.



	1 núcleo	2 núcleos	3 núcleos	4 núcleos	5 núcleos	6 núcleos	7 núcleos	8 núcleos
EasyGrid UMR	0	0	0	0	0	0	0	2
EasyGrid MRRS	0	0	0	0	0	0	0	3
EgridMulti UMR	1	0	2	2	3	3	9	12
EgridMulti MRRS	0	1	1	2	4	4	9	14

Figura 5.3: *Número de eventos de escalonamento dinâmico ocorridos durante a multiplicação de matrizes 10.000×10.000 no Ambiente 1 ao variar o número de núcleos utilizados.*

O número de eventos de escalonamento dinâmico para as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* aumenta na medida em que mais unidades de processamento são disponibilizadas, conforme observado na Figura 5.3. Já para as versões *EasyGrid UMR* e *EasyGrid MRRS*, tais eventos somente ocorreram ao serem utilizados os oito núcleos de cada nó, e em nenhum momento os processos destinados à execução das fatias da última iteração foram re-escalonados. Isto ocorre porque a camada de escalonamento dinâmico identifica que estes processos possuem alta granularidade e que seu re-escalonamento levaria a um alto grau de desbalanceamento na execução da aplicação. Em outras palavras, para as versões *EasyGrid UMR* e *EasyGrid MRRS* somente processos destinados à execução das primeiras $M-1$ fatias em cada recurso foram em algum momento re-escalonados para outros recursos.

Analisando o conjunto de informações apresentadas na Tabela 5.10 e na Figura 5.3, podemos observar que a execução da camada de escalonamento dinâmico do *EasyGrid AMS* acrescentou uma leve sobrecarga aos tempos de execuções da aplicação de divisão de carga para $N = 1, 2, ..., 6$. Esta sobrecarga pode ser mensurada comparando os dados das Tabelas 5.3 e 5.10, as quais ilustram respectivamente o desempenho da aplicação de multiplicação de matrizes 10.000×10.000, quando executada com a camada de escalonamento dinâmico do *EasyGrid AMS* desabilitada (Tabela 5.3) e habilitada (Tabela 5.10). Todavia, quando todos os núcleos de cada nó foram disponibilizados, as execuções com a camada de escalonamento dinâmico habilitada apresentaram resultados melhores do que aquelas em que a referida camada esteve desabilitada. Conforme discutido no início deste capítulo, foi observado que em alguns dos nós que compõem o *cluster Sinergia*, utilizado em todos os experimentos até aqui analisados, ocorria um aumento da temperatura de seus processadores (principalmente ao se utilizar seus oito núcleos de processamento), forçando o sistema operacional a reduzir suas capacidades de processamento até o instante em que a temperatura normalizasse. Tal ação provocava um desbalanceamento entre os tempos computacionais dos nós do ambiente de execução, sendo que somente as execuções em que este desbalanceamento não ultrapassou a dez segundos foram consideradas. Nos cenários em que foram especificadas as maiores quantidades de processos (oito núcleos de cada nó sendo utilizados), que por sua vez apresentaram granularidades mais finas, a camada de escalonamento dinâmico do *EasyGrid AMS* conseguiu re-escalonar um conjunto de tarefas alocadas inicialmente aos nós que sofriam queda de performance para outros nós com total disponibilidade. Como tais tarefas necessitavam de uma pequena quantidade de tempo para serem executadas, não provocaram um desbalanceamento na execução da aplicação ao serem re-escalonadas para outros nós, pelo contrário, conseguiram aproximar

ainda mais os tempos computacionais de cada nó.

5.4.4 Multiplicação de Matrizes de 12.000×12.000 Elementos no Ambiente 1 com o Escalonador Dinâmico Habilitado

Neste experimento será avaliado o desempenho das diferentes versões que utilizam o sistema gerenciador *EasyGrid AMS*, ao trabalharem com uma carga de 12.000 unidades ($A_{12.000,12.000}$) em um ambiente dedicado e com os serviços da camada de escalonamento dinâmico habilitados.

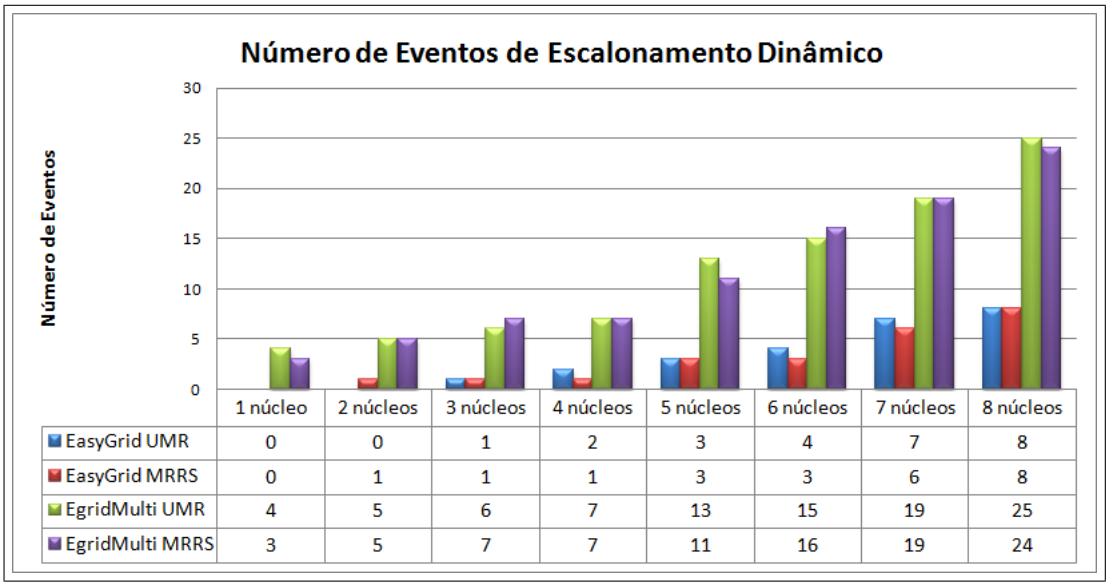
N	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	2935.05	2936.02	2772.6	2771.65
2	1467.26	1465.89	1375.83	1376.84
3	995.88	996.35	917.93	917.94
4	**738.94**	**739.44**	712.41	710.49
5	887.19	885.10	692.41	693.17
6	749.77	750.63	**685.43**	**682.49**
7	763.50	763.91	686.41	685.46
8	739.12	741.30	713.29	712.58

Tabela 5.11: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 12.000×12.000 elementos no Ambiente 1 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

Conforme especificado na Tabela 5.11, o desempenho das diferentes versões da aplicação de multiplicação de matrizes com 12.000 × 12.000 elementos foi similar ao apresentado para um ambiente homogêneo, dedicado e com a camada de escalonamento dinâmico do *EasyGrid AMS* desabilitada, conforme apresentado na Tabela 5.7. O desempenho das versões *EasyGrid UMR* e *EasyGrid MRRS* segue melhorando até o instante em que quatro núcleos de cada nó são utilizados e piora quando cinco ou mais núcleos são utilizados. Já para as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*, esta melhora acontece até o instante em que seis núcleos são utilizados. Houve também um custo extra devido a utilização da camada de escalonamento dinâmico, o qual pode ser medido ao comparar os resultados das Tabelas 5.7 e 5.11.

Ainda em relação à Tabela 5.11, a camada de escalonamento dinâmico não conseguiu prover um desempenho superior ao apresentado na Tabela 5.7, pois todos os recursos do ambiente de execução sofreram igualmente com os problemas de contenção de memória, não havendo portanto um desbalanceamento de carga entre tais nós. Similarmente ao realizado no experimento anterior, a quantidade de eventos de escalonamento dinâmicos

ocorridos durante a multiplicação de matrizes de 12.000 × 12.000 elementos pode ser observada na Figura 5.4.



	1 núcleo	2 núcleos	3 núcleos	4 núcleos	5 núcleos	6 núcleos	7 núcleos	8 núcleos
EasyGrid UMR	0	0	1	2	3	4	7	8
EasyGrid MRRS	0	1	1	1	3	3	6	8
EgridMulti UMR	4	5	6	7	13	15	19	25
EgridMulti MRRS	3	5	7	7	11	16	19	24

Figura 5.4: *Número de eventos de escalonamento dinâmico ocorridos durante a multiplicação de matrizes 12.000×12.000 no Ambiente 1 ao variar o número de núcleos utilizados.*

5.4.5 Multiplicação de Matrizes de 10.000×10.000 Elementos no Ambiente 2 com o Escalonador Dinâmico Desabilitado

Para analisar o comportamento da aplicação de multiplicação de matrizes com as diferentes versões de algoritmos quando executadas em um ambiente heterogêneo, foram submetidas tarefas externas com o objetivo de consumir parte do poder de processamento dos recursos disponíveis durante toda a execução da aplicação. Desta forma, um ambiente heterogêneo controlado pode ser emulado, facilitando assim a análise dos resultados. As informações a serem apresentadas para este experimento foram coletadas em execuções no Ambiente 2 (*cluster Sinergia* heterogêneo controlado), onde três dos sete nós trabalhadores foram selecionados para receberem tarefas externas, o que veio a reduzir seus poderes computacionais em até 50%. Será verificado também o impacto causado pelo compartilhamento de recursos computacionais entre aplicação de divisão de carga e estas novas tarefas externas, e ainda, a eficiência das políticas de divisão de carga que especificam trabalho para os processos trabalhadores de acordo com a capacidade de processamento dos recursos aos quais estes processos estão alocados. Os resultados serão apresentados inicialmente para matrizes de 10.000×10.000 elementos, em seguida, para matrizes de 12.000×12.000 elementos.

Algoritmos que analisam a disponibilidade dos recursos computacionais durante sua divisão de carga (isto é, enviam uma menor quantidade de trabalho para os recurso sobrecarregados ou com menor poder de processamento) produziram uma distribuição de trabalho mais eficiente do que a apresentada pela versão *1-round*. Para um melhor entendimento acerca da política de divisão de carga de cada algoritmo, serão apresentados nas Tabelas 5.12, 5.13, 5.14 e 5.15 os particionamentos produzidos respectivamente pelos algoritmos *1-round*, PWD, UMR e MRRS, ou seja, a quantidade de trabalho (número de linhas das matrizes) que cada processo trabalhador v_i deverá executar. Os dados coletados representam a execução em ambiente heterogêneo (Ambiente 2), utilizando sete processos trabalhadores v_i (um núcleo de cada nó) para cada versão de algoritmo. Observe que a política de divisão utilizada pelo algoritmo *1-round* (Tabela 5.12) particionou igualmente toda a carga de trabalho entre os recursos computacionais disponíveis, enviando a mesma quantidade de trabalho tanto para os recursos já sobrecarregados com a execução de tarefas externas, quanto para os recursos com total disponibilidade. Já para os demais algoritmos (PWD, UMR e MRRS), podemos observar que os recursos sobrecarregados receberam uma menor quantidade de trabalho (Tabelas 5.13, 5.14 e 5.15). O objetivo desta divisão mais justa da carga de trabalho é permitir que todos os recursos envolvidos em seu processamento finalizem suas execuções em tempos semelhantes, mantendo desta forma o princípio de maximização de utilização da plataforma de execução [31, 37, 38].

Iteração	v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	1714	1714	1714	1714	1714	1715	1715

Tabela 5.12: *Distribuição de carga segundo o Algoritmo 1-round no ambiente heterogêneo.*

Iteração	v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	**1153**	**1153**	**1153**	2135	2135	2135	2136

Tabela 5.13: *Distribuição de carga segundo o Algoritmo PWD no ambiente heterogêneo.*

Iteração	v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	**1**	**1**	**1**	1	1	1	1
2	**2**	**2**	**2**	3	4	3	4
3	**15**	**16**	**15**	26	27	26	27
4	**128**	**128**	**128**	218	217	217	218
5	**1080**	**1081**	**1081**	1831	1831	1831	1832

Tabela 5.14: *Distribuição de carga segundo o Algoritmo UMR no ambiente heterogêneo.*

Iteração	v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	**1**	**1**	**1**	1	1	1	1
2	**2**	**2**	**2**	3	4	3	4
3	**15**	**16**	**16**	26	27	26	27
4	**129**	**129**	**129**	217	217	217	218
5	**1086**	**1087**	**1087**	1826	1826	1826	1826

Tabela 5.15: *Distribuição de carga segundo o Algoritmo MRRS no ambiente heterogêneo.*

Conforme observado nas Tabelas 5.12, 5.13, 5.14 e 5.15, os algoritmos que levam em consideração as características de desempenho dos recursos computacionais produziram uma divisão de carga melhor e mais justa do que a definida pelo algoritmo *1-round*, ao despacharam uma menor quantidade de trabalho para os recursos sobrecarregados. A seguir, serão apresentados os desempenhos obtidos com cada versão de divisão de carga.

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	3193.70	2154.74	2149.79	2149.68	2148.55	2149.09	1988.58	1988.04
2	1612.59	1109.05	1104.27	1104.85	1104.71	1104.92	1024.59	1023.78
3	1090.93	735.82	729.35	728.89	727.29	728.53	678.44	678.39
4	823.87	553.86	548.96	548.13	547.88	547.93	525.88	524.96
5	690.67	473.73	468.39	468.02	466.05	466.03	430.01	429.74
6	610.17	433.10	427.04	427.12	425.68	426.17	412.30	412.37
7	505.98	371.20	365.82	366.19	365.07	364.80	352.14	352.56
8	**450.31**	**324.46**	**319.44**	**319.01**	**317.50**	**317.73**	**301.67**	**301.23**

Tabela 5.16: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 10.000×10.000 elementos no Ambiente 2 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

Conforme especificado na Tabela 5.16, o desempenho da versão *1-round* foi bastante prejudicado devido ao fato do algoritmo não considerar o poder computacional dos recursos disponíveis. Desta forma, os recursos com maior disponibilidade computacional finalizaram suas execuções enquanto os demais ainda permaneciam em execução, implicando numa má utilização da plataforma computacional.

O tempo de execução apresentado por cada versão segue caindo na medida em que mais unidades de processamento são disponibilizadas. Apesar da presença de outras tarefas externas à aplicação, a quantidade de memória disponível permitiu que tanto as tarefas da aplicação de divisão de carga quanto as tarefas externas a utilizassem de maneira eficiente (pois as tarefas externas não consomem uma grande quantidade de recursos de memória), não sofrendo portanto constantes atrasos devido a contenção de memória.

5.4.6 Multiplicação de Matrizes de 12.000×12.000 Elementos no Ambiente 2 com o Escalonador Dinâmico Desabilitado

Uma análise similar é realizada para a aplicação de multiplicação de matrizes de 12.000 × 12.000 elementos para o ambiente heterogêneo controlado (Ambiente 2). Note que as tarefas externas que são submetidas à alguns dos nós do ambiente de execução aqui utilizado têm características exclusivamente *cpu-intensive*. Conforme discutido anteriormente, tais tarefas não realizam constantes acessos à hierarquia de memória (consomem uma quantidade muito pequena de memória), visando somente reduzir o poder computacional dos recursos aos quais estão alocadas. A seguir, serão analisados os tempos de execuções atingidos por cada versão de algoritmo no referido ambiente de execução, tempos estes apresentados na Tabela 5.17.

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	5551.95	3747.12	3739.51	3739.11	3740.25	3740.84	3529.46	3528.33
2	2861.35	1926.92	1918.52	1918.55	1917.61	1918.05	1790.26	1791.34
3	2051.09	1289.03	1281.74	1282.2	1281.92	1282.77	1177.18	1178.15
4	**1396.92**	**964.76**	**957.7**	**956.96**	**955.07**	**954.88**	908.42	909.17
5	1543.43	1109.98	1101.69	1101.12	1073.21	1074.04	818.08	819.12
6	1499.1	983.43	975.25	974.98	968.14	967.05	**806.57**	**805.18**
7	1420.96	976.14	967.04	968.32	961.90	961.53	814.7	815.09
8	1434.96	969.09	961.58	961.37	958.19	957.11	844.04	844.75

Tabela 5.17: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 12.000×12.000 elementos no Ambiente 2 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

De acordo com a Tabela 5.17, podemos observar que o comportamento da aplicação de multiplicação de matrizes de dimensões 12.000 × 12.000 em um ambiente heterogêneo é semelhante ao apresentado para um ambiente homogêneo, onde o tempo de execução da maioria das versões é reduzido até o instante em que quatro núcleos de cada nó são utilizados. Note que, caso as tarefas externas presentes em alguns dos nós do ambiente de execução fossem de natureza *data-intensive*, tenderia a haver uma redução ainda maior no desempenho da aplicação, pois passariam a competir com os processos da aplicação de multiplicação de matrizes pelos recursos de memória, os quais são compartilhados.

Novamente, as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* foram uma exceção a regra, pois conseguiram utilizar eficientemente até seis dos oito núcleos disponíveis em cada nó. Conforme descrito nos cenários anteriores, este desempenho foi alcançado devido a utilização do modelo de execução *1PTask* (adotado pelo *EasyGrid AMS*) aliado com a especificação de um número maior de iterações para o escalonamento da carga de

trabalho, visando produzir tarefas de granularidades finas. Ainda, a presença de tarefas externas fez com que as políticas de divisão de carga mais sofisticadas (PWD, UMR e MRRS) enviassem uma menor quantidade de trabalho para os recursos sobrecarregados, proporcionando uma divisão de trabalho mais harmônica. Tais políticas possibilitaram que todos os recursos computacionais utilizados finalizassem suas execuções em momentos semelhantes, evitando assim a ociosidade de tais recursos. Ainda, a distinção entre o desempenho apresentado pelo algoritmo PWD e os demais algoritmos de múltiplas iterações é exatamente a sobreposição entre computação e comunicação. Por fim, os cenários analisados para ambientes heterogêneos apresentaram comportamentos semelhantes aos produzidos em um ambiente homogêneo, onde os tempos de execuções seguem caindo ao se adicionar mais núcleos para a carga de 10.000 unidades ($A_{10.000,10.000}$), enquanto sofrem uma degradação ao utilizar a carga de trabalho contendo 12.000 unidades ($A_{12.000,12.000}$), devido aos problemas de contenção de memória.

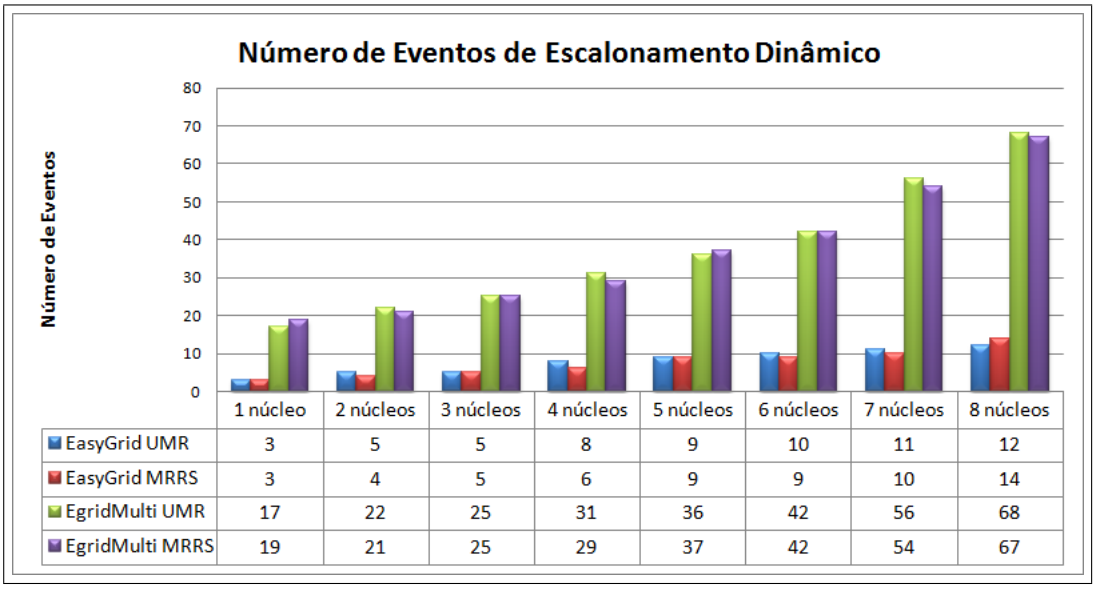
5.4.7 Multiplicação de Matrizes de 10.000×10.000 Elementos no Ambiente 3 com o Escalonador Dinâmico Habilitado

Neste cenário, será descrito o comportamento das versões que utilizam o sistema gerenciador *EasyGrid AMS* ao serem executadas em um ambiente dinâmico controlado (Ambiente 3). É importante ressaltar que como as tarefas externas foram submetidas somente após o início da aplicação de carga divisível, as políticas de divisão de carga irão distribuir trabalho inicialmente considerando um ambiente homogêneo, logo, caberá a camada de escalonamento dinâmico do *EasyGrid AMS* fazer uma redistribuição de trabalho para aliviar os efeitos da chegada destas tarefas externas. Desta forma, os nós sobrecarregados computacionalmente deverão ceder algumas de suas tarefas inicialmente alocadas, e que ainda não foram criadas, para serem executadas em outros nós com total disponibilidade, visando obter assim um balanceamento de carga entre os nós do ambiente. A seguir, será apresentado na Tabela 5.18 o comportamento da aplicação de multiplicação de matrizes de 10.000×10.000 elementos executada sob a gerência do *EasyGrid AMS* no Ambiente 3.

N	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	3178.87	3181.71	2014.13	2012.76
2	1589.61	1588.26	1031.4	1032.05
3	1052.38	1053.88	690.77	689.09
4	767.71	769.24	537.02	540.12
5	641.64	641.01	441.5	444.01
6	549.04	552.69	417.85	416.92
7	459.56	455.9	345.24	344.46
8	**390.18**	**392.02**	**290.77**	**291.81**

Tabela 5.18: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 10.000×10.000 elementos no Ambiente 3 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

Conforme especificado na Tabela 5.18, o desempenho da aplicação multiplicação de matrizes 10.000×10.000 com as versões *EasyGrid UMR* e *EasyGrid MRRS* foi bastante prejudicado pela chegada das tarefas externas. Isto ocorre porque nestas versões a carga de trabalho é distribuída entre um pequeno número de processos, pois é particionada em poucas iterações. Consequentemente, a camada de escalonamento dinâmico do *EasyGrid AMS* não foi capaz de realizar uma eficiente redistribuição de trabalho, pois somente os processos das primeiras $M - 1$ iterações podiam ser re-escalonados, visando evitar que algum recurso viesse a ficar encarregado de processar mais de uma fatia de grande tamanho (fatias da iteração M). Este comportamento limitou a quantidade de tarefas que poderiam ser redistribuídas ou re-escalonadas. Já nas versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*, o escalonador dinâmico ao detectar que alguns dos recursos se encontravam sobrecarregados computacionalmente, passou a re-escalonar um conjunto de suas tarefas que ainda não foram criadas, para os demais recursos. Este comportamento foi atingido em virtude de a carga de trabalho ter sido escalonada em um número maior de iterações, o que veio a produzir um número maior de processos de granularidades finas, permitindo ao escalonador dinâmico do *EasyGrid AMS* redistribuí-los sem provocar um desbalanceamento na execução da aplicação. Essa melhoria propiciada pelo escalonador dinâmico para as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* pode ser vista por uma maior quantidade de eventos de escalonamento na Figura 5.5.



	1 núcleo	2 núcleos	3 núcleos	4 núcleos	5 núcleos	6 núcleos	7 núcleos	8 núcleos
EasyGrid UMR	3	5	5	8	9	10	11	12
EasyGrid MRRS	3	4	5	6	9	9	10	14
EgridMulti UMR	17	22	25	31	36	42	56	68
EgridMulti MRRS	19	21	25	29	37	42	54	67

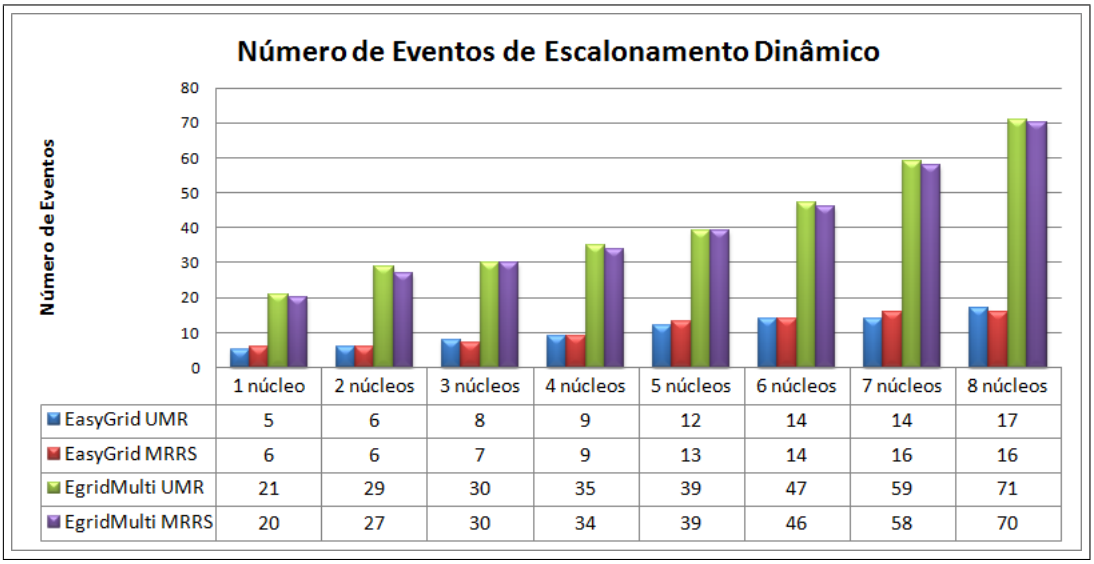
Figura 5.5: *Número de eventos de escalonamento dinâmico ocorridos durante a multiplicação de matrizes 10.000×10.000 no Ambiente 3 ao variar o número de núcleos utilizados.*

5.4.8 Multiplicação de Matrizes de 12.000×12.000 Elementos no Ambiente 3 com o Escalonador Dinâmico Habilitado

Complementando esta etapa de avaliação da aplicação de multiplicação de matrizes em um ambiente dinâmico controlado (Ambiente 3), será apresentado a seguir o conjunto de informações coletadas durante o processamento de matrizes de dimensão 12.000 × 12.000 no referido ambiente. Estas informações estão distribuídas entre a Tabela 5.19 e a Figura 5.6.

N	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	5519.60	5521.15	3559.36	3557.50
2	2836.59	2836.08	1817.42	1818.03
3	2007.92	2005.94	1210.15	1208.54
4	**1336.26**	**1339.74**	941.02	942.69
5	1543.77	1543.34	853.52	851.34
6	1442.49	1443.55	**836.77**	**839.30**
7	1457.88	1458.04	845.90	847.21
8	1394.01	1397.70	852.38	851.09

Tabela 5.19: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 12.000×12.000 elementos no Ambiente 3 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*



	1 núcleo	2 núcleos	3 núcleos	4 núcleos	5 núcleos	6 núcleos	7 núcleos	8 núcleos
EasyGrid UMR	5	6	8	9	12	14	14	17
EasyGrid MRRS	6	6	7	9	13	14	16	16
EgridMulti UMR	21	29	30	35	39	47	59	71
EgridMulti MRRS	20	27	30	34	39	46	58	70

Figura 5.6: *Número de eventos de escalonamento dinâmico ocorridos durante a multiplicação de matrizes 12.000×12.000 no Ambiente 3 ao variar o número de núcleos utilizados.*

De acordo com as informações contidas na Figura 5.6, podemos observar um aumento do número de eventos de escalonamento dinâmicos ocorridos durante a multiplicação das matrizes $A_{12.000,12.000}$ e $B_{12.000,12.000}$, quando comparado com aqueles apresentados na Figura 5.5 para as matrizes $A_{10.000,10.000}$ e $B_{10.000,10.000}$. Aumento este ocorrido em virtude de o tempo gasto com a multiplicação de matrizes 12.000×12.000 ser maior do que para matrizes 10.000×10.000, possibilitando assim uma atuação maior da camada de escalonamento dinâmico do *EasyGrid AMS*. Novamente, a combinação entre a presença de tarefas externas e a existência de tarefas da aplicação com altas granularidades, relativas à última iteração da divisão de carga, e que não podem ser re-escalonadas, levaram as versões *EasyGrid UMR* e *EasyGrid MRRS* a apresentarem uma baixa performance, conforme apresentado na Tabela 5.19. Por sua vez, as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* conseguiram alcançar uma performance melhor graças ao re-escalonamento de tarefas alocadas inicialmente aos nós que vieram a se tornar sobrecarregados para os nós restantes de total disponibilidade.

5.4.9 Multiplicação de Matrizes no Ambiente 4 com o Escalonador Dinâmico Desabilitado

Neste experimento será analisado o desempenho de três instâncias da aplicação de multiplicação de matrizes para uma carga de trabalho contendo 10.000×10.000, 12.000×12.000 e 13.000×13.000 elementos. O objetivo deste experimento é verificar o comportamento da referida aplicação ao ser processada por um conjunto maior de recursos computacionais. O ambiente de execução alvo foi o *cluster Oscar* homogêneo, dedicado e com o escalonador dinâmico do *EasyGrid AMS* desabilitado (Ambiente 4). Ainda, foram utilizados 32 nós do *cluster Oscar*, sendo um nó reservado para o processo mestre v_0 da aplicação de multiplicação de matrizes e dos gerenciadores global (GM) e local (SM). Desta forma, uma faixa entre 31 e 248 recursos computacionais p_i (núcleos) foram utilizados para execução dos processos trabalhadores.

Antes de mais nada, é importante lembrar que a quantidade de memórias RAM e *cache* disponíveis em cada nó do *cluster Oscar* são menores do que as verificadas no *cluster Sinergia*. Especificamente, cada nó do *cluster Oscar* possui 8GB de memória RAM e 8MB de memória cache L2 para cada processador, sendo 4MB compartilhados por cada par de núcleos. Esta quantidade menor de memórias influenciou bastante nos resultados coletados neste ambiente de execução, conforme será descrito a seguir. O comportamento de todas as instâncias de carga de trabalho foram semelhantes, conforme pode ser observado nas Tabelas 5.20, 5.21 e 5.22.

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	336.74	337.51	335.19	333.82	330.41	332.53	322.38	321.39
2	175.65	176.80	174.80	173.55	171.21	171.22	162.21	165.22
3	124.69	125.85	124.75	124.51	122.87	119.97	109.87	112.97
4	**102.24**	**101.95**	**100.60**	**101.07**	**98.33**	**98.15**	**91.33**	**92.15**
5	118.87	118.29	117.89	116.24	115.22	113.16	107.47	105.17
6	117.34	117.23	116.92	116.98	112.79	112.28	104.19	103.19
7	124.44	125.88	123.94	124.27	121.19	121.30	119.20	119.23
8	139.97	140.58	139.35	137.37	133.20	135.19	129.97	132.21

Tabela 5.20: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 10.000×10.000 elementos no Ambiente 4 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	711.07	712.05	709.83	708.62	707.80	710.78	692.17	691.80
2	374.16	374.24	372.71	370.82	369.64	368.47	361.28	359.46
3	267.31	268.64	265.40	264.64	261.32	264.13	256.34	259.33
4	**213.61**	**214.90**	**213.79**	**212.84**	**210.33**	**208.61**	**202.63**	**205.67**
5	265.83	265.97	264.69	262.32	255.33	252.32	248.12	246.35
6	339.86	339.67	338.38	336.05	322.44	323.41	320.40	320.80
7	440.52	441.36	439.91	438.40	431.52	431.52	424.52	425.53
8	663.94	663.90	661.47	663.27	660.74	659.72	655.74	652.74

Tabela 5.21: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 12.000×12.000 elementos no Ambiente 4 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	847.33	847.41	846.52	844.12	841.34	843.91	830.93	828.04
2	437.48	438.45	435.85	436.41	432.52	431.50	423.53	425.51
3	367.24	368.40	365.00	366.98	363.37	363.54	354.70	349.36
4	**292.25**	**294.84**	**290.56**	**289.77**	**280.34**	**279.34**	**271.33**	**274.11**
5	359.88	360.21	357.26	355.22	343.39	343.40	339.20	336.38
6	408.20	408.97	404.01	404.49	388.49	385.50	378.52	377.49
7	572.48	573.47	569.02	567.60	559.68	563.65	553.67	554.70
8	805.58	807.38	803.64	805.34	803.14	801.10	790.11	792.48

Tabela 5.22: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 13.000×13.000 elementos no Ambiente 4 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

Em todos os cenários apresentados nas Tabelas 5.20, 5.21 e 5.22, o melhor desempenho para todas as versões de divisão de carga foi obtido ao se utilizar apenas quatro núcleos de cada nó. Observa-se também que os resultados obtidos ao se utilizar todos os oito núcleos de cada nó foram próximos dos obtidos ao se utilizar apenas um núcleo. Este comportamento se deve a quantidade de memória *cache* disponível em cada nó do *cluster Oscar*, a qual não é grande o suficiente para que mais de um processos a compartilhem eficientemente durante o processamento das cargas de trabalho definidas neste experimento, divergindo desta forma do comportamento observado durante as análises realizadas no *cluster Sinergia*. No cenário em que os oito núcleos foram utilizados, quatro processos trabalhadores ocuparam o primeiro processador e outros quatro ocuparam o segundo, sendo que cada par de processos compartilharam a mesma área da memória *cache*. Este compartilhamento da memória *cache* entre diferentes processos implicou em um aumento do número de eventos *cache-miss*, degradando fortemente o desempenho da aplicação.

Ao comparar os resultados apresentados nas Tabelas 5.7 e 5.21, as quais mostram respectivamente os tempos de processamento de matrizes com 12.000×12.000 elementos nos *clusters Sinergia* e *Oscar*, pode-se observar que o aumento do número de recursos computacionais disponíveis não implicou em uma redução do tempo de processamento da aplicação na mesma proporção. No cenário em que quatro núcleos por nó foram utilizados para a versão *EasyGridMulti MRRS* por exemplo, o número total de recursos computacionais utilizados no *cluster Oscar* (124) foi quase quatro vezes e meio maior do que o utilizado no *cluster Sinergia* (28), enquanto o tempo de processamento do primeiro (205.67 segundos) foi apenas três vezes e meio menor do que o segundo (700.38 segundos). Este comportamento indica que a quantidade de memória *cache* disponível é uma limitante para o desempenho de aplicações *data-intensives*.

Apesar da especificação de tarefas de granularidades menores, as versões que utilizam o sistema gerenciador *EasyGrid AMS* não conseguiram repetir o desempenho apresentado no *cluster Sinergia*, onde utilizaram eficientemente até seis núcleos por nó durante o processamento de matrizes de 12.000×12.000 elementos. Este comportamento se deve a um conjunto de fatores, dentre os quais estão:

- A quantidade maior de recursos computacionais utilizados e que consequentemente, aumenta a quantidade de processos a serem executados, aumentando dessa forma as sobrecargas relativas à criação e gerenciamento desses processos;
- A quantidade de memória *cache* disponível em cada nó, que mesmo com a especificação de tarefas de granularidades menores não permitiu a utilização de um número maior de núcleos, em virtude da constante ocorrência de eventos *cache-miss*.

Todavia, as versões que utilizam o sistema gerenciador *EasyGrid AMS* apresentaram os melhores resultados para este experimento realizado no *cluster Oscar*, sendo um pouco melhor para as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*.

5.4.10 Multiplicação de Matrizes no Ambiente 5 com o Escalonador Dinâmico Habilitado

Neste último experimento realizado para avaliar a aplicação de multiplicação de matrizes foi utilizado o *cluster Oscar*, configurado de tal forma a simular um ambiente dinâmico controlado. Lembrando que em tal ambiente, dez máquinas destinadas a executarem processos trabalhadores da aplicação foram selecionadas para receberem um conjunto de tarefas externas, visando proporcionar uma característica dinâmica ao referido ambiente (Ambiente 5). Neste experimento, serão avaliadas matrizes contendo 10.000 × 10.000, 12.000 × 12.000 e 13.000 × 13.000 elementos. Ainda, serão analisados os comportamentos de todas as versões de divisão de carga desenvolvidas neste trabalho, sendo que aquelas que utilizam o sistema gerenciador *EasyGrid AMS* foram executadas com os serviços da camada de escalonamento dinâmico habilitados. As informações coletadas para cada versão estão dispostas ao longo das Tabelas 5.23, 5.24 e 5.25.

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	605.34	606.49	605.47	604.32	428.84	424.58	398.02	395.53
2	312.16	312.31	312.67	311.50	236.16	232.11	214.77	213.15
3	213.10	212.24	213.05	211.09	187.96	183.20	172.60	170.49
4	**179.69**	**180.35**	**179.43**	**181.34**	**131.24**	**133.45**	**120.51**	**122.37**
5	195.66	193.68	193.24	192.89	177.01	180.79	163.92	167.74
6	194.46	196.41	194.42	195.46	163.14	162.55	148.06	147.38
7	198.49	198.93	198.53	199.15	184.39	184.70	169.80	169.25
8	206.18	205.04	205.52	203.97	197.22	198.05	182.20	183.79

Tabela 5.23: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 10.000×10.000 elementos no Ambiente 5 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	1305.74	1307.24	1309.03	1311.67	1172.69	1167.40	934.75	940.16
2	709.48	713.59	708.12	711.09	630.93	627.08	495.32	494.07
3	505.52	506.30	503.38	505.54	443.19	440.84	345.49	342.97
4	**395.88**	**396.33**	**393.34**	**395.41**	**340.10**	**344.21**	**262.10**	**265.38**
5	486.36	483.84	484.15	481.96	415.05	413.54	331.82	327.19
6	507.40	508.30	508.20	504.11	442.75	445.96	348.01	345.55
7	585.51	587.91	582.71	585.30	500.11	498.30	389.57	394.40
8	865.23	869.46	864.08	866.11	716.68	712.99	561.30	563.22

Tabela 5.24: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 12.000×12.000 elementos no Ambiente 5 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	1435.57	1433.62	1435.14	1431.67	1277.61	1270.03	1039.43	1033.07
2	776.29	775.21	774.27	778.57	682.04	687.33	544.04	550.82
3	602.39	604.13	605.92	604.39	544.37	539.11	438.71	441.40
4	**542.31**	**548.34**	**541.58**	**540.61**	**481.90**	**484.95**	**386.50**	**390.06**
5	626.62	625.93	630.17	622.6	557.32	552.70	453.88	449.14
6	662.5	659.38	664.07	658.82	569.34	573.40	461.09	459.05
7	826.69	820.51	821.39	822.54	719.08	725.66	584.21	581.57
8	1340.74	1340.07	1336.4	1342.71	1193.41	1187.07	985.22	981.60

Tabela 5.25: *Média do tempo de execução em segundos da aplicação de multiplicação de matrizes de 13.000×13.000 elementos no Ambiente 5 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

As Tabelas 5.23, 5.24 e 5.25 mostram que novamente os melhores resultados apresentados por cada versão foram obtidos nos cenários em que quatro núcleos de cada nó do *cluster Oscar* foram utilizados. Ainda, observa-se também que para este ambiente dinâmico, o desempenho das versões *1-round*, PWD, UMR e MRRS foram similares, havendo inclusive amostras em que as versões de uma iteração (*1-round* e PWD) foram superiores às versões UMR e MRRS. Este comportamento ocorre devido a presença de tarefas externas, as quais acrescentaram uma sobrecarga aos tempos computacionais dos nós sobrecarregados de tal forma que veio a anular o ganho obtido com a divisão de carga em múltiplas iterações. Note ainda, que a aplicação de multiplicação de matrizes é caracterizada por possuir uma carga de trabalho regular, pois todas as linhas das matrizes a serem multiplicadas demandam o mesmo tempo de processamento, definição esta também adotada em [36]. Desta forma, para a aplicação de multiplicação de matrizes a etapa de escalonamento dinâmico do algoritmo PWD não é utilizada durante sua divisão de carga.

As versões que utilizam o sistema gerenciador de aplicações *EasyGrid AMS* obtiveram os melhores resultados, desempenho este alcançado em virtude da capacidade de reescalonamento dinâmico de tarefas do referido gerenciador. Os percentuais de ganho apresentados por tais versões em relação as versões UMR e MRRS durante a multiplicação de matrizes de 13.000 × 13.000 elementos são demonstrados na Tabela 5.26. Tais percentuais são definidos pelas seguintes formulações:

- % EGUMR: $\frac{(UMR-EGUMR)}{UMR}$;
- % EGMRRS: $\frac{(MRRS-EGMRRS)}{MRRS}$;
- % EGMUMR: $\frac{(EGUMR-EGMUMR)}{EGUMR}$;
- % EGMMRRS: $\frac{(EGMRRS-EGMMRRS)}{EGMRRS}$.

Núcleos	% EGUMR	% EGMRRS	% EGMUMR	% EGMMRRS
1	10.98	11.29	18.64	18.66
2	11.91	11.72	20.23	19.86
3	10.16	10.80	19.41	18.12
4	11.02	10.30	19.80	19.57
5	11.56	11.23	18.56	18.74
6	14.27	12.97	19.01	19.94
7	12.46	11.78	18.76	19.86
8	10.70	11.59	17.44	17.31

Tabela 5.26: *Percentual de ganho ao se utilizar o escalonador dinâmico do EayGrid AMS na aplicação de multiplicação de matrizes com* 13.000 × 13.000 *elementos em ambiente dinâmico.*

O percentual de ganho das versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* é justificado pela presença de um conjunto maior de processos de curta duração, que por sua vez vieram a ser re-escalonados para nós com total disponibilidade computacional após o *EasyGrid AMS* detectar a sobrecarga incorrida nos nós em que foram acometidas tarefas externas. Por fim, a quantidade de eventos de escalonamento dinâmico ocorrida durante a multiplicação das matrizes de 13.000 × 13.000 elementos pode ser vista na Figura 5.7.

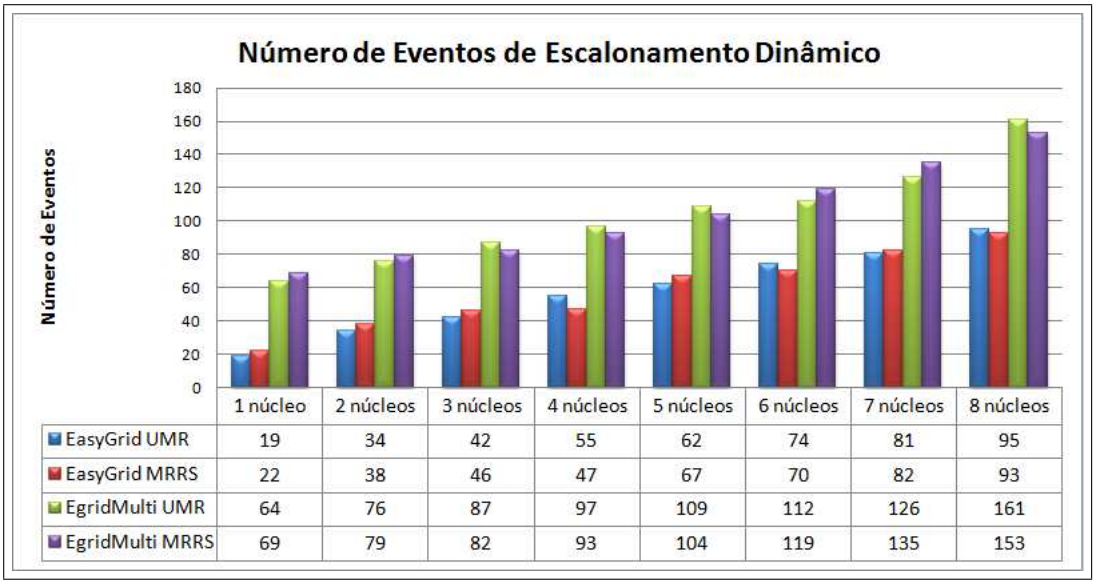


Figura 5.7: *Número de eventos de escalonamento dinâmico ocorridos durante a multiplicação de matrizes 13.000×13.000 no Ambiente 5 ao variar o número de núcleos utilizados.*

5.5 Análise da Aplicação Mandelbrot Set Computation Fractal

Por definição, fractais são figuras geométricas complexas, geradas com o auxílio de algum padrão matemático, podendo ser divididas em partes distintas onde cada parte é semelhante ao objeto original [29]. Em muitos casos, um *fractal* pode ser gerado por um padrão repetitivo, tipicamente um processo recorrente ou iterativo. *Mandelbrot set computation fractal*, proposto em [28, 29], é um caso particular dentre os algoritmos para geração de fractais, definido por um método iterativo e que produz uma imagem contendo o *fractal* em questão. A Figura 5.8 mostra uma imagem resultante da execução do algoritmo de *Mandelbrot*.

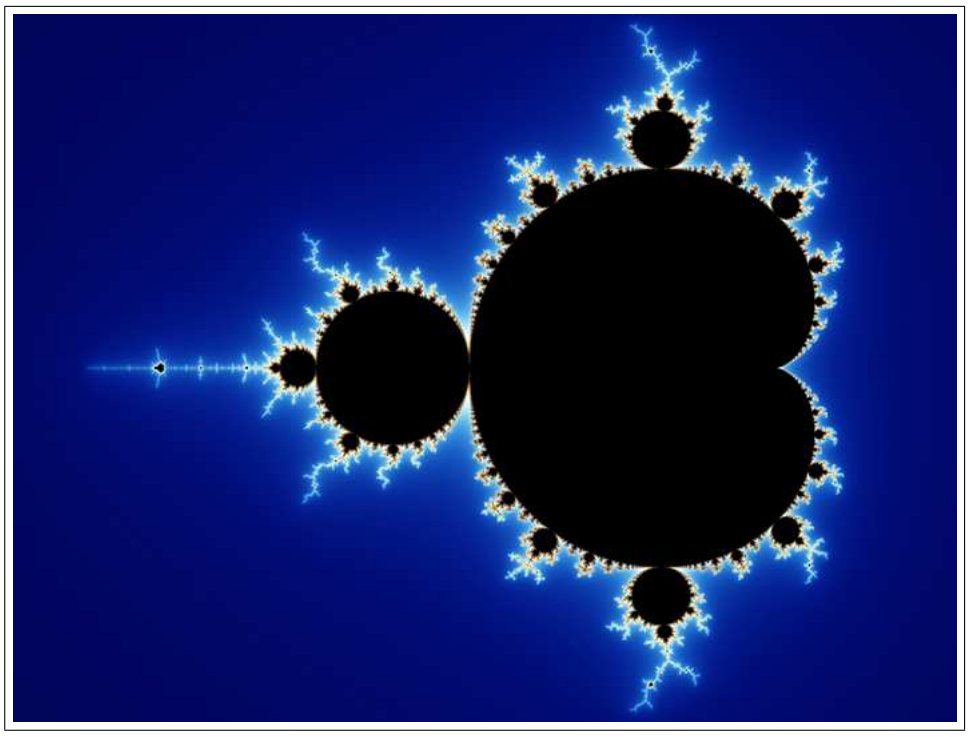


Figura 5.8: *Exemplo de fractal Mandelbrot*

O *fractal* de *Mandelbrot* é gerado iterativamente a partir de uma formulação sobre o conjunto dos números complexos. Seja $C = x + y \times i$ um número complexo, sendo x a parte real, y a parte imaginária e i uma constante definida como sendo $i^2 = -1$. Como entrada para o algoritmo de *Mandelbrot*, é dado o número máximo de pontos nos eixos x e y, denotados por X_{max} e Y_{max}, que a imagem produzida deverá conter. Na sequência, cada um dos valores de x e y (inicializando por $x = 0$ e $y = 0$) são selecionados para referenciarem um ponto $C = x + y \times i$ e utilizados na seguinte fórmula iterativa:

$$Z_0 = 0;$$
$$Z_{n+1} = Z_n^2 + C. \tag{5.1}$$

Para cada ponto C, esta sequência se expande como

$$Z_0 = 0;\ Z_1 = Z_0^2 + C = x + y \times i;\ Z_2 = Z_1^2 + C = (x + y \times i)^2 + x + y \times i$$

e assim por diante. O método procede iterativamente gerando novos valores para Z_{n+1} até que pelo menos uma das seguintes condições de parada seja atingida:

(C1) : Um número máximo I_{max} de iterações seja alcançado.

(C2) : A distância $|C|$ entre os pontos x e y, definida como $\sqrt{x^2 + y^2}$, ultrapassar um determinado valor máximo, especificado por D_{max}.

Na sequência, x e y são incrementados e o método procede até que x e y atinjam seus respectivos valores máximos (X_{max} e Y_{max}) em relação ao número de *pixels* sendo investigado.

A versão paralela do algoritmo de *Mandelbrot* desenvolvida neste trabalho, consiste em particionar a região da imagem a ser produzida em diversas subáreas e enviá-las para serem processadas pelos recursos computacionais. Um problema acerca da paralelização deste algoritmo segundo a teoria de divisão de cargas (DLT), foi a definição do que seria sua carga de trabalho e como representá-la. Inicialmente, foi definido que cada *pixel* da imagem a ser produzida seria uma unidade de carga de trabalho. Porém, caso os valores definidos para X_{max} e Y_{max} sejam muito grandes, levaria a um número possivelmente não representável de unidades de carga ($X_{max} \times Y_{max}$ unidades). Logo, optou-se por representar a carga de trabalho como sendo blocos de *pixels*, cujos tamanhos seriam potências de dois. Em outras palavas, a carga de trabalho W_{total} será definida como sendo $\frac{(X_{max} \times Y_{max})}{B_{size}}$ unidades de carga, sendo B_{size} uma constante a ser definida, que representa o tamanho de um bloco de *pixels*.

Um segundo problema envolvendo o algoritmo de *Mandelbrot* se deve a irregularidade de sua carga de trabalho, pois existem regiões da imagem contendo o *fractal* que demandam um tempo maior de processamento. Estas regiões podem ser observadas principalmente na parte central da imagem, onde existe uma maior riqueza de detalhes. Isto

ocorre porque cada *pixel* a ser colorido na imagem produzida é obtido a partir do método iterativo definido em 5.1, considerando as condições de parada (C1) ou (C2). Desta forma, dois *pixels* vizinhos podem apresentar tempos computacionais distintos para que sejam obtidos (possuem cores diferentes), fazendo com que recursos trabalhadores idênticos e que recebam exatamente a mesma quantidade de blocos de *pixels* finalizem suas execuções em momentos diferentes. Segundo a teoria de divisão de carga (DLT), todos os processos executados nos diferentes recursos computacionais devem finalizar suas execuções no mesmo momento, para que o tempo de execução seja ótimo [31]. Para esta classe de aplicações, constituídas por uma carga de trabalho **irregular**, a divisão de sua carga entre os recursos disponíveis de modo que todos finalizem suas execuções em tempos similares se torna um problema desafiador.

O trabalho apresentado em [36] aborda esta questão da irregularidade da carga de trabalho do *fractal* de *Mandelbrot* através da utilização do algoritmo PWD (Performance-Based Workload Distribution). O algoritmo PWD é caracterizado pela mesclagem entre um algoritmo de escalonamento estático e um algoritmo de escalonamento dinâmico (conforme apresentado na Seção 3.3). Numa primeira etapa, é utilizado um algoritmo de escalonamento estático que distribui uma percentagem da carga de trabalho total de acordo com as características de cada recurso computacional. Em uma segunda etapa, o restante da carga de trabalho é distribuída de acordo com o algoritmo de escalonamento dinâmico GSS (*Guided Self-Scheduling*), que envia novas frações de cargas para os recursos que primeiro finalizarem suas execuções (também definido na Seção 3.3). O objetivo desta carga reservada para o escalonamento dinâmico é proporcionar um efetivo balanceamento de carga, fazendo com que os processos trabalhadores concluam suas execuções em momentos similares.

Conforme apresentado em [36], os melhores resultados para o algoritmo PWD durante a execução da aplicação de *Mandelbrot* foram obtidos quando o valor do parâmetro SWR (*Static-Workload Ratio*) foi setado para 0.8 (definido empiricamente). Desta forma, 80% da carga de trabalho total foi particionada de acordo com o algoritmo de escalonamento estático, enquanto os outros 20% foram reservados para o escalonamento dinâmico. Este mesmo valor para SWR foi utilizado neste trabalho durante a etapa de avaliação do algoritmo PWD.

Em todas as versões da aplicação de *Mandelbrot* desenvolvidas neste trabalho, cada processo trabalhador v_i recebe do processo mestre v_0 a quantidade de blocos de *pixels* que deverá processar e o valor do ponto inicial (x, y) a partir do qual deverá produzir

os respectivos blocos de *pixels*. Os experimentos realizados para avaliar o desempenho da aplicação de *Mandelbrot* consistem em executar suas diferentes versões em ambientes homogêneos, heterogêneos e também em ambientes dinâmicos.

5.5.1 Aplicação de Mandelbrot Fractal no Ambiente 1 com o Escalonador Dinâmico do EasyGrid Desabilitado

Inicialmente serão verificados os comportamentos das diferentes versões da aplicação de *Mandelbrot* ao serem executadas em um ambiente homogêneo, dedicado e com o escalonador dinâmico do *EasyGrid AMS* desabilitado (Ambiente 1). A carga de trabalho da aplicação é composta por uma imagem contendo 40.000×40.000 pontos de resolução, o que equivale à aproximadamente 1.5 GB de massa de dados. Neste trabalho, cada bloco de *pixels* é composto por um conjunto de 1024 *pixels*, assim, a carga de trabalho total é de 1.562.500 unidades de carga (blocos de *pixels*). Note que estes blocos podem possuir diferentes tempos de processamento, devido à característica de irregularidade da carga de trabalho, conforme discutido anteriormente.

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	1457.88	1026.53	817.86	818.34	800.12	802.33	776.85	777.60
2	745.78	525.43	646.68	647.46	611.63	610.97	392.88	392.84
3	504.40	358.95	454.72	454.67	428.49	428.65	268.60	273.58
4	382.10	274.64	360.01	359.86	343.12	344.01	208.47	208.45
5	309.91	223.85	289.44	289.51	275.08	274.90	175.38	172.38
6	261.04	188.29	238.97	239.56	229.69	227.82	151.33	150.35
7	226.75	166.25	208.59	206.27	201.63	201.51	131.29	130.30
8	**201.21**	**151.45**	**181.59**	**183.20**	**175.39**	**177.50**	**117.30**	**121.29**

Tabela 5.27: *Média do tempo de execução em segundos da aplicação de Mandelbrot no Ambiente 1 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

Considerando o ambiente computacional homogêneo, a Tabela 5.27 mostra que o tempo de processamento de todas as versões é reduzido na medida em que mais processadores são utilizados (o máximo de núcleos por nó). Apesar da carga de trabalho a ser processada pela aplicação de *Mandelbrot* necessitar de um espaço de armazenamento maior do que o da aplicação de multiplicação de matrizes de dimensões 12.000×12.000, uma menor quantidade de atrasos de contenção de memória (*cache-miss*) ocorrem. Sua explicação reside na forma como os dados de cada aplicação são acessados. Enquanto a aplicação de multiplicação de matrizes realiza constantes acessos a três matrizes distintas ($A_{12.000,12.000}$, $B_{12.000,12.000}$ e $C_{12.000,12.000}$), implicando em constantes renovações das infor-

mações contidas na memória *cache*, a aplicação de *Mandelbrot* acessa somente um bloco de *pixels*, armazenado em sequência na memória, sendo que cada *pixel* é acessado apenas um vez ao longo do processamento da carga de trabalho. Consequentemente, a aplicação de *Mandelbrot* não sofre uma queda de performance, quando comparada à aplicação de multiplicação de matrizes 12.000×12.000, ao utilizar cinco, seis, sete e oito núcleos de cada nó.

No cenário utilizando apenas sete processos trabalhadores ($N = 1$), o desempenho da versão PWD foi superior somente ao da versão *1-round*. Para a versão PWD, este comportamento ocorre porque a área da imagem contendo o *fractal* foi particionada em uma pequena quantidade de fatias, especificamente, sete fatias foram processadas na fase estática (cuja soma de seus tamanhos totaliza 80% da carga de trabalho) e sete na fase dinâmica (complementando os 20% restantes). Neste caso, os processos trabalhadores receberam fatias de grandes tamanhos e computacionalmente irregulares. Desta forma, os processos que ficaram responsáveis pela região central da imagem (rica em detalhes) necessitaram de um tempo de processamento maior do que os responsáveis pelas regiões mais externas. Nem mesmo a carga de trabalho reservada para o escalonamento dinâmico foi capaz de solucionar este problema e fazer com que todos os processos trabalhadores finalizassem suas execuções em momentos semelhantes. Já nas versões que utilizam os algoritmos UMR e MRRS, a região da imagem contendo o *fractal* foi dividida em um número um pouco maior de fatias ($M\times$ *núcleos* $\times$ *nós*), aumentando assim o grau de paralelismo de processamento das regiões mais computacionalmente intensivas. Ainda, estas fatias são distribuídas alternadamente entre os processos trabalhadores, de forma que esses venham a processar blocos situados tanto na região central da imagem, quanto nas regiões mais externas.

Na medida em que mais recursos computacionais são utilizados o algoritmo PWD se torna mais eficiente, devido as seguintes características: a divisão da imagem em um número maior de subáreas, as quais consequentemente possuem menos irregularidade; a utilização de uma política de escalonamento dinâmico que permite enviar novas fatias de carga (de tamanhos menores) para os processos que primeiro finalizarem suas execuções. A Figura 5.9 mostra um exemplo de particionamento produzido pelo algoritmo PWD sobre a imagem a ser calculada ao se utilizar trinta e cinco processos trabalhadores (cinco núcleos de cada nó). Durante sua execução, 80% da carga de trabalho W_{total} foi distribuída ao longo da fase estática e 20% ao longo da fase dinâmica, onde $W_{total} = 1.562.500$. Assim, na fase estática cada um dos 35 processos trabalhadores executaram 31.250 unidades de carga, enquanto na fase dinâmica, 11 destes processos receberam mais 13.293 unidades de

carga repetidas vezes (não mais que 5 vezes cada). Neste exemplo, foi observado que as tarefas responsáveis pelo processamento da parte superior da imagem foram as primeiras a receberem novas fatias de carga na etapa dinâmica, pois na fase estática lhes foram designadas fatias que durante a execução seriam de curta duração.

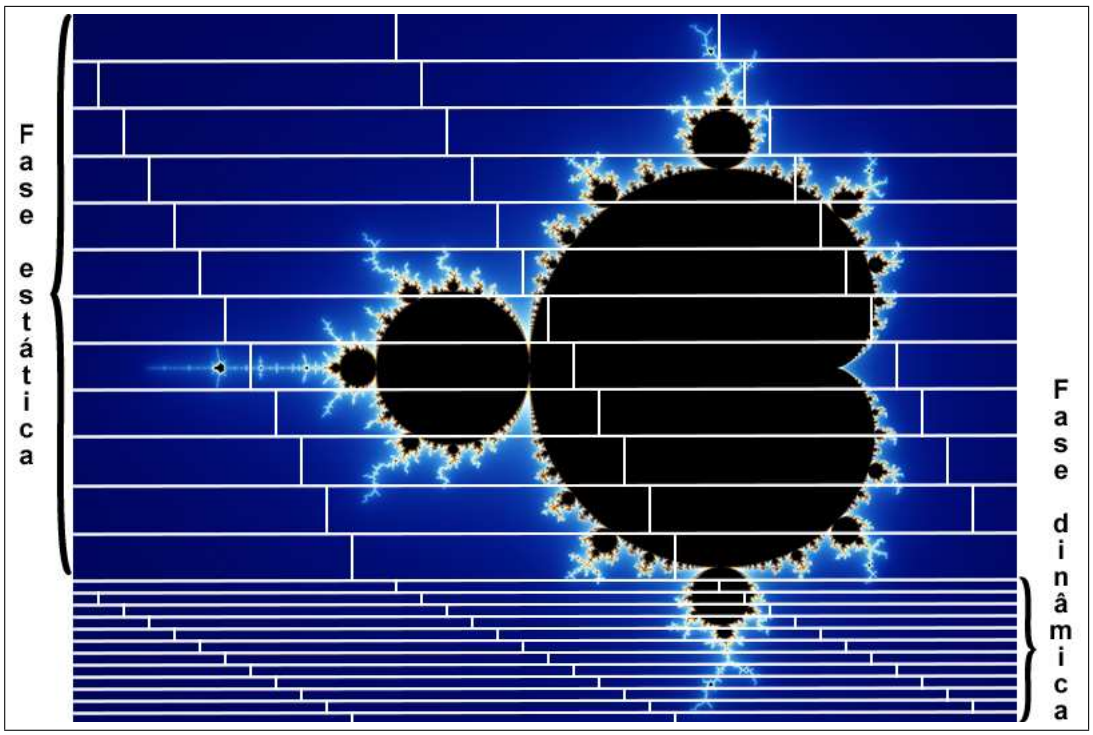


Figura 5.9: *Particionamento do fractal de Mandelbrot pelo Algoritmo PWD.*

Em relação ao conteúdo da Tabela 5.27, ao se utilizar dois ou mais núcleos de cada nó, a versão com o algoritmo PWD só não foi superior às versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*, as quais apresentaram um ganho de desempenho de até 33%. Com a versão PWD, a área do *fractal* foi particionada em no máximo 112 subáreas (cenário utilizando oito núcleos), sendo 56 distribuídas inicialmente com escalonamento estático, e outras 56 distribuídas dinamicamente. Porém, nem mesmo esta distribuição dinâmica de carga adotada pelo algoritmo PWD foi o suficiente para proporcionar um bom balanceamento de trabalho e consequentemente, uma performance semelhante a apresentada pelas versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*. Por sua vez, as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* chegaram a particionar esta mesma área do *fractal* em até 616 fatias (também utilizando oito núcleos e 11 iterações). Ao particionar a carga de trabalho em pequenas fatias, essas versões conseguiram contornar a característica de irregularidade da carga e realizar uma distribuição mais justa. Como consequência, os processos trabalhadores finalizaram suas execuções em tempos similares e utilizaram o ambiente de execução de maneira mais eficiente. Na sequência deste trabalho

serão feitas análises acerca do balanceamento de carga e também das diferenças entre os tempos de execuções de cada nó ao processarem esta carga de trabalho irregular.

Para os demais cenários, a baixa eficiência das versões *1-round*, UMR, MRRS, *EasyGrid UMR* e *EasyGrid MRRS* se deve a dificuldade de se especificar uma divisão tal que os processos trabalhadores finalizem suas execuções em momentos semelhantes. Este fato decorre principalmente da definição de um pequeno número de subáreas (gerando fatias maiores e provavelmente, com um alto grau de irregularidade) e também pela falta de políticas de balanceamento de carga para amenizar os efeitos da irregularidade da carga de trabalho. No caso das versões *EasyGrid UMR* e *EasyGrid MRRS*, apesar da especificação dos processos segundo o modelo *1PTask*, as tarefas relativas a última iteração executaram fatias de grandes tamanhos (e consequentemente, mais irregulares), dificultando assim a obtenção de um bom balanceamento de carga. Além disso, neste experimento o escalonador dinâmico do *EasyGrid AMS* esteve desabilitado em todas as suas execuções (Ambiente 1), impossibilitando assim a aplicação de se auto ajustar e redistribuir trabalho entre os recursos computacionais, de forma a obter tal balanceamento de carga.

Uma amostra do número de fatias em que a área da imagem é particionada por cada versão é apresentada na Tabela 5.28. Na versão *1-round* a área da imagem é dividida entre a quantidade de recursos computacionais disponíveis ($núcleos \times nós$). Já de acordo com a versão PWD, o número de fatias consiste em duas (uma para a fase estática e outra para a fase dinâmica) vezes a quantidade de unidades e processamento disponíveis ($2 \times núcleos \times nós$), onde metade representa a distribuição estática e metade a dinâmica. Por fim, para as versões que utilizam os algoritmos UMR e MRRS, o número de fatias é dado pelo produto entre a quantidade de nós, o total de núcleos utilizados e o número de iterações M, sendo que nas versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* este número de iterações M é aumentado conforme o Algoritmo 6, apresentado na Seção 4.3.

5.5.2 Aplicação de Mandelbrot Fractal no Ambiente 1 com o Escalonador Dinâmico do EasyGrid Habilitado

Neste experimento, será avaliado o desempenho da aplicação de *Mandelbrot* ao utilizar as funcionalidades disponibilizadas pela camada de escalonamento dinâmico do *EasyGrid AMS*. O ambiente de execução utilizado foi o *cluster Sinergia* homogêneo, dedicado e com o escalonador dinâmico do *EasyGrid AMS* habilitado. Conforme discutido anteriormente, uma das dificuldades de se executar eficientemente aplicações de cargas divisíveis irregulares é fazer com que todos os recursos computacionais finalizem suas execuções

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	7	14	28	28	28	28	238	238
2	14	28	56	56	56	56	434	434
3	21	42	84	84	84	84	462	462
4	28	56	112	112	112	112	476	476
5	35	70	140	140	140	140	490	490
6	42	84	168	168	168	168	504	504
7	49	98	196	196	196	196	539	539
8	56	112	280	280	280	280	616	616

Tabela 5.28: *Número de fatias em que o fractal foi particionado por cada versão, no Ambiente 1 ao variar o número de núcleos utilizados em cada nó, com o escalonador dinâmico desabilitado.*

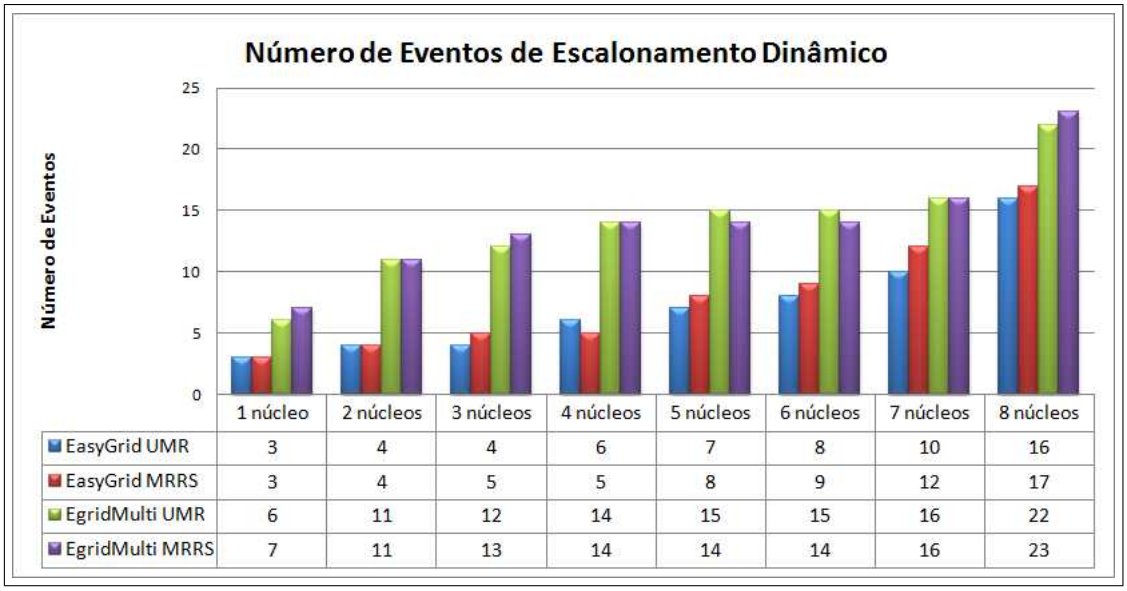
simultaneamente. Devido a dificuldade de previsão do tempo de processamento das fatias de carga para tal aplicação, ficará sob domínio da camada de escalonamento dinâmico do *EasyGrid AMS* [18] a redistribuição de tarefas entre os processadores que compõem o ambiente de execução, caso algum desbalanceamento seja detectado. Note que somente as tarefas que ainda não foram criadas podem ser re-escalonadas para outros processadores.

N	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	796.24	797.20	772.87	773.03
2	604.76	602.05	385.89	384.85
3	419.06	417.70	258.58	260.60
4	327.06	328.33	201.20	199.47
5	256.63	258.19	170.39	168.15
6	209.51	206.82	142.36	142.33
7	178.96	177.04	121.70	123.20
8	**146.55**	**144.90**	**113.45**	**114.28**

Tabela 5.29: *Média do tempo de execução em segundos da aplicação de Mandelbrot no Ambiente 1 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

A Tabela 5.29 mostra que o desempenho das versões *EasyGrid UMR*, *EasyGrid MRRS*, *EasyGridMulti UMR* e *EasyGridMulti MRRS* durante a execução da aplicação de *Mandelbrot* em um ambiente um homogêneo, dedicado e com o escalonador dinâmico habilitado foi superior inclusive a aquele apresentado por estas mesmas versões ao serem executadas em um ambiente homogêneo e dedicado, mas com o escalonador dinâmico do *EasyGrid AMS* desabilitado, conforme apresentado na Seção 5.5.1. Para um melhor entendimento acerca destes resultados alcançados, é necessário analisar o balanceamento de carga ocorrido dentro de cada nó, ou seja, a distribuição de trabalho entre as unidades de processamento de cada nó. Para que o tempo de execução da aplicação de *Mandelbrot* seja ótimo, todos os núcleos de processamento de todos os nós precisam finalizar suas

execuções com tempos computacionais similares [31, 37, 38]. O desempenho das versões que utilizam o sistema gerenciador *EasyGrid AMS* conseguiu se aproximar daquele considerado ideal em virtude da utilização do modelo de execução alternativo *1PTask* e da capacidade de re-escalonamento dinâmico de tarefas do *EasyGrid AMS*, principalmente para as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*. Desta forma, os recursos nos quais foram alocados processos de curta duração (fatias de carga não tão irregulares), e que finalizaram suas execuções primariamente, vieram a requisitar processos (ainda não criados) de outros recursos, nos quais foram alocados inicialmente tarefas de longa duração (fatias de carga altamente irregulares). O resultado desta redistribuição dinâmica de trabalho, realizada pelo *EasyGrid AMS* entre os processadores que compõem o ambiente de execução, foi a obtenção de um bom balanceamento de carga para a aplicação de *Mandelbrot*, permitindo aos diferentes processadores apresentarem tempos computacionais similares. A quantidade de eventos de escalonamento dinâmico ocorridos em cada versão gerenciada pelo *EasyGrid AMS* é demonstrada na Figura 5.10.



	1 núcleo	2 núcleos	3 núcleos	4 núcleos	5 núcleos	6 núcleos	7 núcleos	8 núcleos
EasyGrid UMR	3	4	4	6	7	8	10	16
EasyGrid MRRS	3	4	5	5	8	9	12	17
EgridMulti UMR	6	11	12	14	15	15	16	22
EgridMulti MRRS	7	11	13	14	14	14	16	23

Figura 5.10: *Número de eventos de escalonamento dinâmico ocorridos durante a execução da aplicação de Mandelbrot no Ambiente 1 ao variar o número de núcleos utilizados.*

Complementando esta etapa de avaliação da aplicação de *Mandelbrot* em um ambiente homogêneo, é apresentado na Tabela 5.30 o percentual de ganho das versões que utilizam o sistema gerenciador *EasyGrid AMS* com o escalonador dinâmico habilitado em relação a estas mesmas versões com o escalonador dinâmico desabilitado. Os rótulos das colunas dois, três, quatro e cinco da Tabela 5.30 estabelecem uma relação entre os conteúdos das Tabelas 5.27 e 5.29, e são definidos por:

- % EGUMR: $\frac{(EGUMR_{estatico} - EGUMR_{dinamico})}{EGUMR_{estatico}}$;
- % EGMRRS: $\frac{(EGMRRS_{estatico} - EGMRRS_{dinamico})}{EGMRRS_{estatico}}$;
- % EGMUMR: $\frac{(EGMUMR_{estatico} - EGMUMR_{dinamico})}{EGMUMR_{estatico}}$;
- % EGMMRRS: $\frac{(EGMMRRS_{estatico} - EGMMRRS_{dinamico})}{EGMMRRS_{estatico}}$.

N	% EGUMR	% EGMRRS	% EGMUMR	% EGMMRRS
1	0.48	0.64	0.51	0.59
2	2.71	1.46	1.78	2.03
3	2.20	2.55	3.73	4.74
4	4.68	4.56	3.49	4.31
5	6.71	6.08	2.85	2.45
6	8.79	9.22	5.93	5.33
7	11.24	12.14	7.30	5.45
8	16.44	18.37	3.28	5.78

Tabela 5.30: *Percentual de ganho ao se utilizar o escalonador dinâmico do EayGrid AMS na aplicação de Mandelbrot.*

5.5.3 Aplicação de Mandelbrot Fractal no Ambiente 2 com o Escalonador Dinâmico do EasyGrid Desabilitado

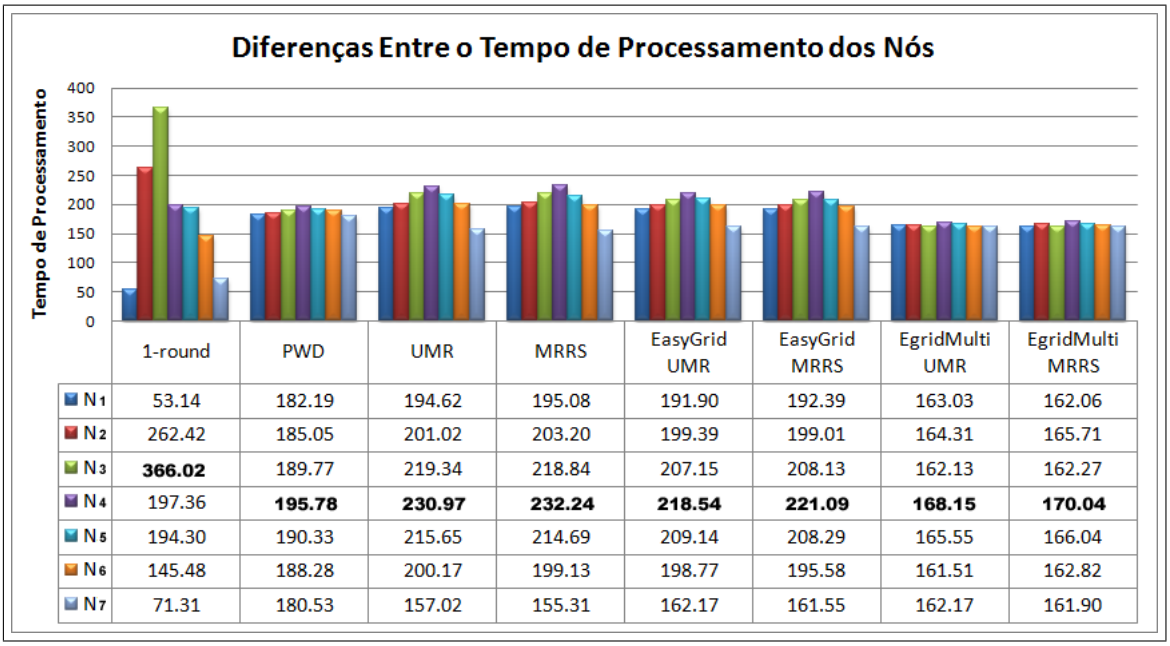
A terceira etapa de avaliação da aplicação de *Mandelbrot* consiste em verificar sua performance ao ser executada em um ambiente heterogêneo controlado (Ambiente 2).

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	2524.24	1348.40	1148.75	1150.70	1098.09	1100.14	1024.42	1025.11
2	1365.74	628.36	779.64	779.83	747.32	748.91	480.44	480.12
3	825.38	496.10	640.22	641.19	612.98	614.84	382.39	383.07
4	718.20	380.92	506.06	509.11	484.18	485.75	301.78	301.80
5	538.96	290.56	377.96	377.67	360.40	363.01	236.59	238.02
6	485.18	248.29	316.01	317.09	302.21	303.20	214.94	214.34
7	407.57	222.96	278.03	281.35	269.59	271.45	190.28	192.18
8	**367.81**	**196.09**	**232.88**	**234.54**	**220.05**	**222.39**	**170.96**	**171.77**

Tabela 5.31: *Média do tempo de execução em segundos da aplicação de Mandelbrot no Ambiente 2 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

Os resultados vistos na Tabela 5.31 mostram que o tempo de execução de todas as versões segue caindo na medida em que mais núcleos de processamento são utilizados. Ainda, novamente a quantidade de fatias em que a área da imagem foi particionada e

a existência ou não de políticas de escalonamento dinâmico foram fundamentais para o desempenho de cada versão. Note que neste experimento a camada de escalonamento dinâmico do *EasyGrid AMS* esteve desabilitada durante toda a execução da aplicação (Ambiente 2). Conforme discutido anteriormente, o maior obstáculo em relação ao processamento de cargas irregulares é fazer com que todos os recursos computacionais finalizem suas execuções simultaneamente. Para um melhor entendimento acerca deste obstáculo, é ilustrado na Figura 5.11 a variação entre o tempo de execução coletado para cada um dos sete nós trabalhadores N_k, $1 \leq k \leq 7$, que compõem o *cluster Sinergia*. Note que estas informações ilustradas na Figura 5.11 representam apenas a diferença entre os tempos de processamento dos nós que compõem o ambiente de execução, não especificando portanto a diferença entre os tempos de processamento de cada núcleo em cada nó. Tais informações são apresentadas para todas as versões da aplicação de *Mandelbrot* ao utilizar os oito núcleos de cada nó (oito processos sendo executados paralelamente) neste ambiente heterogêneo controlado (Ambiente 2). Em cada nó N_k, existem oito recursos computacionais p_i (núcleos), conforme o modelo arquitetural definido neste trabalho. Ainda, cada recurso p_i executa apenas um processo v_i ao longo de toda a execução da aplicação nas versões *1-round*, PWD, UMR e MRRS (modelo *1PProc*). Por sua vez, nas versões *EasyGrid UMR*, *EasyGrid MRRS*, *EasyGridMulti UMR* e *EasyGridMulti MRRS*, cada recurso p_i executa uma tarefa $v_{i,j}$ a cada iteração j da divisão de carga (modelo *1PTask*).



	1-round	PWD	UMR	MRRS	EasyGrid UMR	EasyGrid MRRS	EgridMulti UMR	EgridMulti MRRS
N1	53.14	182.19	194.62	195.08	191.90	192.39	163.03	162.06
N2	262.42	185.05	201.02	203.20	199.39	199.01	164.31	165.71
N3	**366.02**	189.77	219.34	218.84	207.15	208.13	162.13	162.27
N4	197.36	**195.78**	**230.97**	**232.24**	**218.54**	**221.09**	**168.15**	**170.04**
N5	194.30	190.33	215.65	214.69	209.14	208.29	165.55	166.04
N6	145.48	188.28	200.17	199.13	198.77	195.58	161.51	162.82
N7	71.31	180.53	157.02	155.31	162.17	161.55	162.17	161.90

Figura 5.11: *Diferenças entre os tempos de processamento de cada nó, ao se utilizar oito núcleos com as diferentes versões da aplicação de Mandelbrot em ambiente heterogêneo.*

Conforme observado na Figura 5.11, o maior grau de desbalanceamento entre os nós

do ambiente de execução foi apresentado pela versão *1-round*, onde os processos responsáveis pelo processamento das regiões mais externas da imagem (alocados aos nós N_1 e N_7) finalizaram o processamento de suas respectivas cargas em um momento muito inferior ao dos processos responsáveis pela região central (frequentemente alocados ao nó N_3). Consequentemente, uma diferença de até 588% entre os tempos computacionais dos nós N_1 e N_3 foi atingida, revelando assim o desbalanceamento de carga ocorrido durante a execução da referida versão. Podemos observar ainda, que as versões PWD, *EasyGrid-Multi UMR* e *EasyGridMulti MRRS* apresentaram os melhores balanceamentos de carga, em virtude respectivamente da distribuição dinâmica de trabalho (PWD) e da utilização e distribuição de fatias de curta duração (*EasyGridMulti UMR* e *EasyGridMulti MRRS*).

5.5.4 Aplicação de Mandelbrot Fractal no Ambiente 3 com o Escalonador Dinâmico do EasyGrid Habilitado

Em continuidade à avaliação da aplicação de *Mandelbrot*, serão apresentados a seguir, por meio da Tabela 5.32, os resultados coletados durante a execução das versões que utilizam o sistema gerenciador de aplicações *EasyGrid AMS* em um ambiente dinâmico controlado (Ambiente 3). Neste experimento, um conjunto de tarefas *cpu-intensive* foram criadas em três dos nós trabalhadores, cinco segundos após o início da aplicação de divisão de carga e são executadas durante todo o processamento de tal carga, competindo desta forma pelos mesmos recursos computacionais.

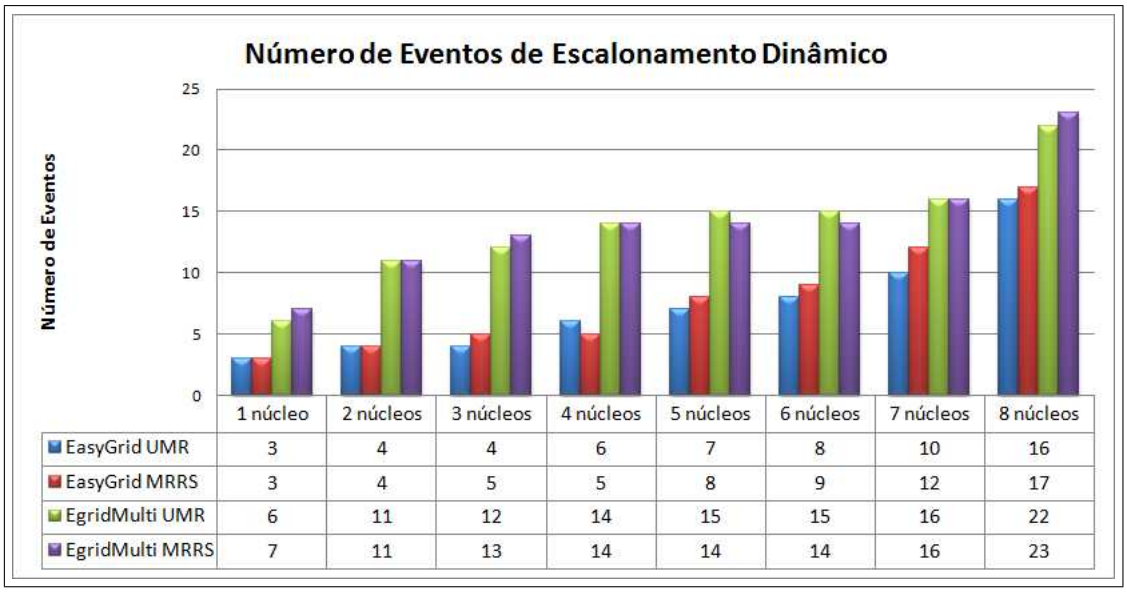
N	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	1237.62	1233.90	1030.26	1029.29
2	864.31	862.63	484.70	485.23
3	689.72	692.21	377.18	375.15
4	538.32	535.75	291.60	292.42
5	395.40	400.06	226.70	224.79
6	307.12	305.12	201.15	202.60
7	262.80	263.87	183.43	183.50
8	**210.73**	**209.81**	**161.55**	**160.17**

Tabela 5.32: *Média do tempo de execução em segundos da aplicação de Mandelbrot no Ambiente 3 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

Similarmente ao observado durante a execução da aplicação de multiplicação de matrizes neste mesmo ambiente (Seção 5.4.8), a divisão de trabalho inicial foi feita considerando um ambiente homogêneo e dedicado, pois as tarefas externas somente iniciaram sua execução após a aplicação de *Mandelbrot* ter sido inicializada. Desta forma, os re-

cursos que se tornaram sobrecarregados após a chegada destas novas tarefas tiveram que destinar parte dos processos que lhes foram alocados inicialmente, para serem executados por outros recursos com total disponibilidade computacional. Todavia, as versões *EasyGrid UMR* e*EasyGrid MRRS* apresentaram uma baixa eficiência em relação às versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* em virtude da presença de tarefas de longa duração (granularidades grossas), referentes as fatias da última iteração da divisão de carga, e que não poderiam ser re-escalonadas em outros recursos, pois tenderia a provocar um alto grau de desbalanceamento na execução da aplicação. As versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* atingiram os melhores resultados, com um ganho de até 45% no tempo de execução quando comparados às versões *EasyGrid UMR* e*EasyGrid MRRS*, conforme pode ser observado na Tabela 5.32 ao utilizar quatro núcleos de cada nó.

Note que além do desbalanceamento de carga devido às tarefas externas, a aplicação de *Mandelbrot* possui tarefas de tempo de execução desconhecidos. Desta forma, foi de grande importância o uso do escalonador dinâmico e a especificação de tarefas segundo o modelo *1PTask* com granularidades menores. Tais tarefas puderam ser facilmente re-escalonadas para recursos menos sobrecarregados computacionalmente, contornando a característica de irregularidade da aplicação de *Mandelbrot* e proporcionando um bom balanceamento de carga. Um esboço acerca da quantidade de eventos de escalonamento dinâmico realizados pelo sistema gerenciador de aplicações *EasyGrid AMS* é ilustrado na Figura 5.12.



	1 núcleo	2 núcleos	3 núcleos	4 núcleos	5 núcleos	6 núcleos	7 núcleos	8 núcleos
EasyGrid UMR	3	4	4	6	7	8	10	16
EasyGrid MRRS	3	4	5	5	8	9	12	17
EgridMulti UMR	6	11	12	14	15	15	16	22
EgridMulti MRRS	7	11	13	14	14	14	16	23

Figura 5.12: *Número de eventos de escalonamento dinâmico ocorridos durante a execução da aplicação de Mandelbrot no Ambiente 3 ao variar o número de núcleos utilizados.*

5.6 Análise da Aplicação de Filtro Gaussiano

A terceira aplicação selecionada para fazer parte deste trabalho pertence a uma classe bastante comum no estudo de aplicações de cargas divisíveis, que são aquelas destinadas ao processamento de imagens digitais, sendo que neste trabalho, será especificada uma aplicação que emprega a operação de Filtro Gaussiano [45]. O tempo de execução envolvendo tal operação pode ser longo, especialmente quando se trata de imagens de grandes proporções. Sua característica fundamental é suavizar o conteúdo de imagens digitais, removendo ruídos existentes e uniformizando-as. A aplicação de Filtro Gaussiano recebe com entrada uma imagem, representando sua carga de trabalho, e em seguida, executa um conjunto de operações aritméticas envolvendo cada *pixel* da referida imagem e também seus vizinhos mais próximos (cuja quantidade é especificada através de máscaras). Outra característica importante é que operações de Filtro Gaussiano são classificadas como simétricas, onde as bordas e linhas de todas as direções da imagem são tratadas similarmente, o que permite definir sua carga de trabalho como sendo regular. Ainda, a aplicação de Filtro Gaussiano é de natureza *cpu-intensive* e não sofre com os atrasos de contenção de memória típicos das aplicações *data-intensive* (como por exemplo, a aplicação de multiplicação de matrizes), pois acessa um conjunto de dados bem menor. Para um melhor entendimento, a Figura 5.13 e o Algoritmo 7 especificam a execução da operação de Filtro Gaussiano sobre cada *pixel* da imagem de entrada, de acordo com a máscara selecionada.

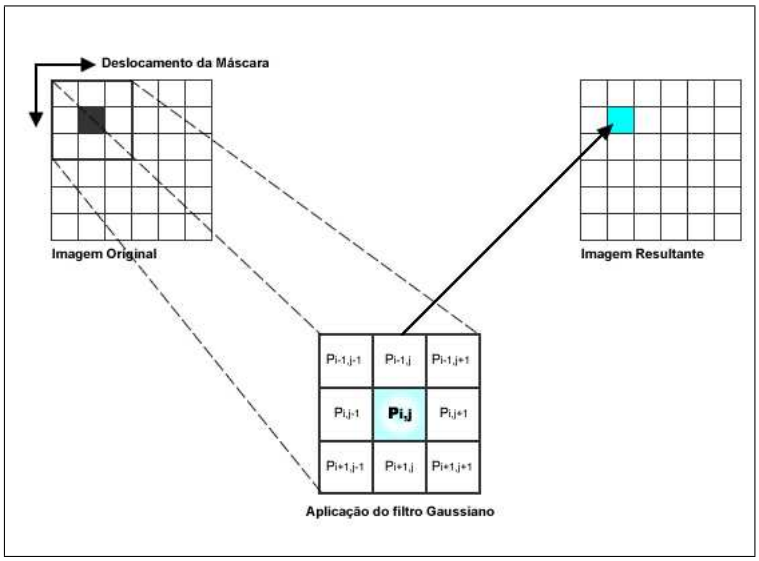


Figura 5.13: *Operação do Filtro Gaussiano sobre um determinado pixel e seus vizinhos delimitados pela máscara selecionada.*

O objetivo da operação de Filtro Gaussiano é determinar a média entre o valor de cada *pixel* da imagem original e de seus vizinhos delimitados pela máscara, média essa que será utilizada para referenciar o novo *pixel* a ser armazenado na imagem processada. A máscara utilizada determina a quantidade de vizinhos de cada *pixel* que será utilizada na operação de Filtro Gaussiano. Na Figura 5.13 por exemplo, é representada a operação com uma máscara 3×3, indicando que o valor de cada novo *pixel* será obtido através de um conjunto de operações aritméticas (média) envolvendo o referido *pixel* e também, cada um de seus nove vizinhos mais próximos [45]. Outros exemplos de máscaras comumente utilizadas em algoritmos de processamento de imagens digitais são 5×5 e 11×11, sendo representadas por matrizes bidimensionais, onde cada elemento possui um determinado peso associado [45].

Algoritmo 7 : Algoritmo de Filtro Gaussiano

```
1:  Início
2:    // Seja imagem o conjunto de pontos da imagem original de largura L_I e altura A_I.
3:    // Seja mascara a máscara de largura L_M e altura A_M selecionada.
4:    // Seja somaMascara o somatório de todos elementos da máscara.
5:    // Seja IMax o número máximo de iterações sobre cada pixel.
6:    para(largura = ⌊L_M/2⌋; largura < L_I − ⌊L_M/2⌋; largura++) {
7:      para(altura = ⌊A_M/2⌋; altura < A_I − ⌊A_M/2⌋; altura++) {
8:        para(iteracao = 0; iteracao < IMax; iteracao++) {
9:          soma = 0;
10:         vizinho_i = −⌊L_M/2⌋;
11:         para(mascara_i = 0; mascara_i < L_M; mascara_i++) {
12:           vizinho_j = −⌊A_M/2⌋;
13:           para(mascara_j = 0; mascara_j < A_M; mascara_j++) {
14:             soma = soma + imagem[largura + vizinho_i][altura + vizinho_j] ×
15:             mascara[mascara_i][mascara_j];
16:             vizinho_j = vizinho_j + 1
17:           }
18:           vizinho_i = vizinho_i + 1
19:         }
20:         novoPixel = soma / somaMascara;
21:         imagem[largura][altura] = novoPixel;
22:       }
23:     }
24:   }
25: Fim
```

Conforme especificado no Algoritmo 7, o método recebe como entrada uma imagem digital de largura L_I e altura A_I, uma máscara, representada por uma matriz de $L_M \times A_M$ elementos, e o número máximo de iterações a serem executadas sobre cada *pixel* da

imagem. Na sequência, cada ponto da imagem e também seus vizinhos delimitados pela máscara são selecionados e associados a referida máscara em uma operação de somatório (linha 14). No final, após o número máximo de iterações ser atingido para cada *pixel*, o resultado da divisão entre o somatório dos pesos de cada *pixel* (e também de seus vizinhos) e o somatório de todos os elementos da máscara passa a ser associada ao *pixel* selecionado (linhas 20 e 21). Ao realizar essa operação de média entre *pixels*, o método consegue remover ruídos existentes e também definir uma característica de suavidade e uniformidade à imagem [45]. A operação de Filtro Gaussiano é executada $IMax$ vezes sobre o mesmo *pixel*, todavia, caso esse número de iterações $IMax$ seja muito grande, poderá vir a provocar um aspecto de distorção ou opacidade à imagem processada.

Em relação a definição de sua carga de trabalho e como representá-la segundo a teoria de divisão de cargas, foi especificado que cada unidade de carga será representada por um bloco de *pixels* contendo a mesma dimensão que a máscara utilizada, ou seja, cada unidade de carga poderá ser formada por blocos de 3×3, 5×5 ou 11×11 *pixels*. A área da imagem original é particionada em blocos de *pixels*, os quais por sua vez são enviados para serem processados por um conjunto de tarefas trabalhadoras. Um exemplo de particionamento da área de uma imagem a ser processada é ilustrado na Figura 5.14, onde as fatias de carga (de diferentes cores) foram produzidas pelo algoritmo UMR para um conjunto de seis processos trabalhadores. Cada uma das seis diferentes cores contidas na Figura 5.14 representam as fatias destinadas a cada um dos seis processos trabalhadores distintos. O particionamento segue por iniciar com fatias de pequenos tamanhos (região superior da imagem) e então as incrementa ao longo da execução da aplicação, objetivando a sobreposição entre computação e comunicação.

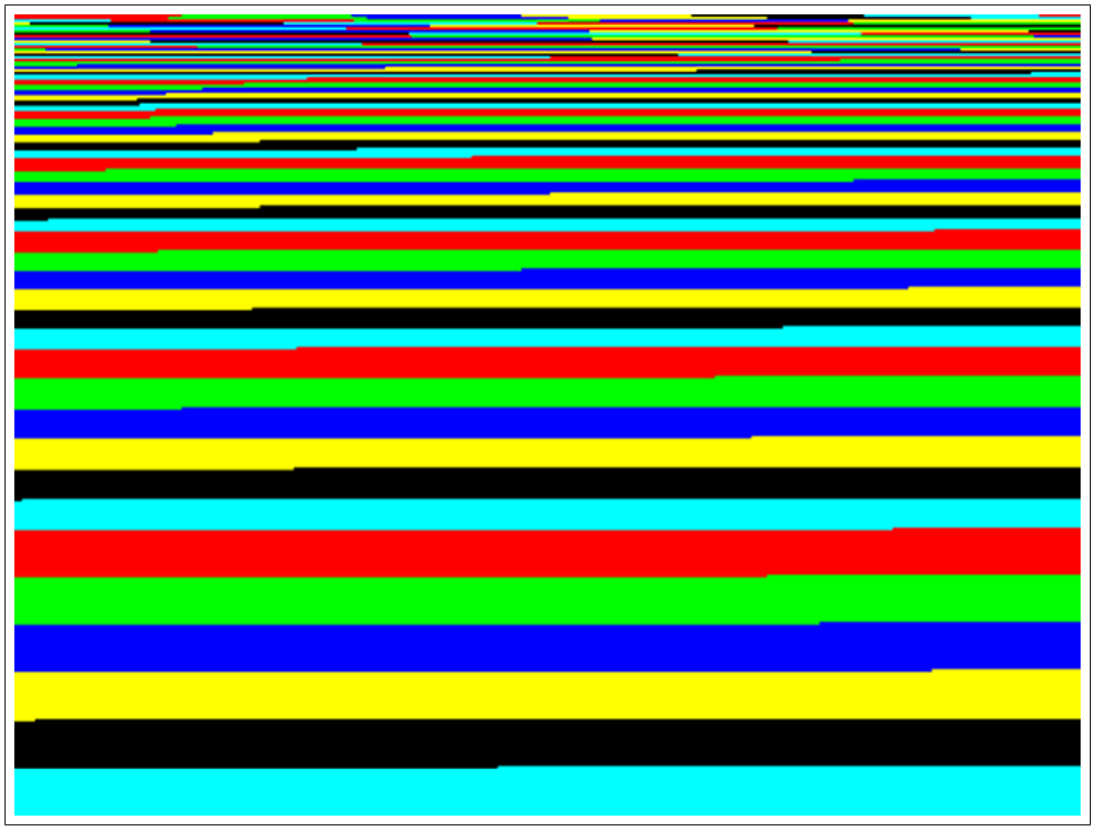


Figura 5.14: *Exemplo de particionamento da área de uma imagem a ser processada de acordo com as definições do algoritmo UMR.*

Em todas as versões de divisão de carga da aplicação de Filtro Gaussiano desenvolvidas neste trabalho, cada processo trabalhador v_i recebe do processo mestre v_0, os valores do ponto inicial (x_{ini}, y_{ini}) e do ponto final (x_{fim}, y_{fim}), os quais determinam o intervalo de blocos de *pixels* que cada processo deverá executar. Em seguida, cada processo v_i recebe de v_0 sua respectiva porção de *pixels* da imagem original e então inicia seu processamento. Essas operações são realizadas em cada iteração da divisão de carga. A carga de trabalho total da aplicação de Filtro Gaussiano é composta por uma imagem contendo 12.000×9.600 pontos de resolução, o que equivale à 330 MB de dados. Note que o espaço de memória necessário para o armazenamento desta carga de trabalho é menor do que o apresentado pelas aplicações de multiplicação de matrizes e de *Mandelbrot*, no entanto, sua carga de trabalho possui um tempo de envio maior do que o das aplicações anteriores. Isto ocorre porque sua carga de trabalho reside inicialmente em disco (ao contrário das aplicações anteriormente analisadas, onde suas cargas residiam na memória principal), sendo necessário fazer um leitura inicial desta carga e em seguida, preparar estruturas de dados que possam representar o valor de cada *pixel*. Como consequência deste atraso de envio de trabalho, os últimos processos a receberem suas respectivas fatias

de imagem permanecerão ociosos enquanto os primeiros processos recebem suas fatias. A utilização de algoritmos de múltiplas iterações visando aliviar os efeitos deste processo de leitura e preparo produziu resultados interessantes tanto para ambientes homogêneos, heterogêneos, quanto para ambientes dinâmicos.

5.6.1 Análise da Aplicação de Filtro Gaussiano no Ambiente 1 Com o Escalonador Dinâmico do EasyGrid Desabilitado

A primeira etapa de avaliação da aplicação de Filtro Gaussiano consiste em analisar o desempenho das diferentes versões de algoritmos de divisão de carga, ao serem executadas no *cluster Sinergia* dedicado e sem o escalonador dinâmico do *EasyGrid AMS* habilitado para as versões que o utilizam. Os resultados obtidos são apresentados na Tabela 5.33.

N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	859.64	858.71	852.70	852.66	853.84	852.77	856.73	856.05
2	447.01	448.35	436.49	436.40	436.91	437.09	439.89	440.15
3	309.89	310.14	297.93	296.81	298.62	298.99	301.62	300.61
4	240.93	240.88	228.98	227.78	228.47	228.44	230.50	230.49
5	200.44	200.27	186.04	186.17	186.38	186.37	188.39	188.38
6	170.20	170.74	159.08	157.98	159.35	159.34	162.32	161.22
7	150.68	151.54	138.55	137.49	138.29	138.28	141.29	140.31
8	**134.92**	**135.16**	**123.80**	**122.78**	**123.26**	**124.26**	**128.29**	**128.72**

Tabela 5.33: *Média do tempo de execução em segundos da aplicação de Filtro Gaussiano no Ambiente 1 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

A Tabela 5.33 mostra que o tempo de execução de todas as versões segue caindo na medida em que mais processadores são utilizados. A aplicação de Filtro Gaussiano não sofre constantes atrasos de contenção de memória como aqueles observados na aplicação de multiplicação de matrizes, o que permite que seu desempenho melhore na medida em que um número maior de núcleos é utilizado. Neste cenário com o Ambiente 1, os valores obtidos para as versões *1-round* e PWD são similares. Como esta aplicação é composta por uma carga de trabalho completamente regular (linhas e bordas da imagem de entrada são tratadas similarmente) o algoritmo PWD a despacha em uma única iteração, resultando no mesmo particionamento produzido pelo algoritmo *1-round*. Ao longo da execução dos algoritmos *1-round* e PWD, os últimos processos a receberem suas respectivas porções de carga permaneceram ociosos durante uma longa faixa de tempo, devido ao preparo e envio das porções destinadas aos primeiros processos. Já nas versões de múltiplas iterações, inicialmente cada processo recebe uma pequena quantidade de trabalho, permitindo que

todos entrem em execução o mais breve possível. Na sequência, recebem novas fatias de carga (de tamanhos maiores) enquanto ainda executam as fatias anteriores, e assim, se tornam mais eficientes do que os algoritmos de uma iteração.

Ao contrário do observado ao longo da análise das aplicações de multiplicação de matrizes e de *Mandelbrot*, as versões que utilizam o sistema gerenciador de aplicações *EasyGrid AMS* não apresentaram os melhores resultados. Para a aplicação de Filtro Gaussiano, a utilização de um número maior de processos trabalhadores acarretou em um aumento das sobrecargas ligadas a criação e gerenciamento destes processos por parte do *EasyGrid AMS*. Além da criação dinâmica e gerenciamento dos processos da aplicação, este custo está relacionado também à interceptação e manipulação das mensagens da aplicação ao longo da hierarquia de processos gerenciadores do *EasyGrid AMS* [15, 18], conforme discutido no Capítulo 4. Note que este custo é um pouco maior para as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*, as quais são caracterizadas por utilizarem um número maior de iterações e consequentemente, de processos trabalhadores. As aplicações anteriormente analisadas continham características particulares, como por exemplo uma alta dependência de recursos de memória ou uma carga de trabalho irregular, que permitiram ao sistema gerenciador *EasyGrid AMS* lhes proporcionar um melhor desempenho com a utilização de tarefas de curta duração. Já a aplicação de Filtro Gaussiano não possui nenhuma dessas características (possui uma carga de trabalho regular e não é *data-intensive*), desta forma, a utilização do *EasyGrid AMS* não produziu melhores resultados. De qualquer forma, o percentual de sobrecarga das versões *EasyGrid UMR* e *EasyGrid MRRS* em relação às versões UMR e MRRS não foi maior do que 0.182% e 1.191% respectivamente, conforme pode ser visto na Tabela 5.34. Já para as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*, seus percentuais de sobrecarga em relação as versões UMR e MRRS foram de no máximo 3.5% e 4.615% respectivamente.

N	% EGUMR	% EGMRRS	% EGMUMR	% EGMMRRS
1	0.134	0.013	0.470	0.396
2	0.096	0.158	0.773	0.852
3	0.231	0.729	1.223	1.264
4	-0.223	0.289	0.659	1.176
5	0.182	0.107	1.247	1.173
6	0.169	0.854	1.996	2.010
7	-0.188	0.571	1.939	2.010
8	-0.438	1.191	3.500	4.615

Tabela 5.34: *Percentual de sobrecarga das versões que utilizam o sistema gerenciador de aplicações EasyGrid AMS com o escalonador dinâmico desabilitado em relação às versões UMR e MRRS.*

5.6.2 Análise da Aplicação de Filtro Gaussiano no Ambiente 1 Com o Escalonador Dinâmico do EasyGrid Habilitado

Nesta seção, será verificado o desempenho das versões que utilizam o sistema gerenciador de aplicações *EasyGrid AMS* ao serem executadas em um ambiente homogêneo, dedicado e com os serviços da camada de escalonamento dinâmico habilitados.

N	EasyGrid UMR	EasyGrid MRRS	EasyGridMulti UMR	EasyGridMulti MRRS
1	859.79	859.74	864.04	863.83
2	443.89	443.93	446.39	447.92
3	304.60	305.59	308.66	308.74
4	235.50	235.53	238.47	239.49
5	193.38	192.42	196.43	196.20
6	165.31	166.39	170.94	170.83
7	145.39	144.18	148.22	148.29
8	**129.34**	**130.35**	**134.67**	**134.94**

Tabela 5.35: *Média do tempo de execução em segundos da aplicação de Filtro Gaussiano no Ambiente 1 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

Conforme especificado na Tabela 5.35, os tempos de execuções de todas as versões pioraram em média 3.09% (para as versões *EasyGrid UMR* e *EasyGrid MRSS*) e 3.59% (para as versões *EasyGridMulti UMR* e *EasyGridMulti MRSS*), quando comparados aos apresentados na Tabela 5.33. Além disso, na maioria dos cenários analisados não houve evento de escalonamento algum, visto que o sistema gerenciador *EasyGrid AMS* não detectou um desbalanceamento de execução entre os processadores envolvidos. Desta forma, (para este experimento realizado em um ambiente homogêneo e dedicado) a utilização da camada de escalonamento dinâmico do *EasyGrid AMS* apenas adicionou uma sobrecarga extra à execução da aplicação, o que veio a reduzir seu desempenho. A ausência de eventos de escalonamento dinâmico durante a execução da aplicação de Filtro Gaussiano decorre de características tanto da aplicação (a carga de trabalho é regular e não é *data-intensive*) quanto do ambiente de execução (composto por recursos computacionais homogêneos).

5.6.3 Análise da Aplicação de Filtro Gaussiano no Ambiente 2 com o Escalonador Dinâmico do EasyGrid Desabilitado

Nesta seção será analisado o comportamento da aplicação de Filtro Gaussiano ao ser executada em um ambiente heterogêneo controlado e com o escalonador dinâmico do *EasyGrid AMS* desabilitado (Ambiente 2). Mantendo a mesma metodologia utilizada

para a avaliação das aplicações de multiplicação de matrizes e de *Mandelbrot*, um conjunto de tarefas externas *cpu-intensives* serão inseridas nos três primeiros nós do ambiente de execução, para que tenham suas respectivas capacidades de processamento reduzidas (durante toda a execução da aplicação de Filtro de Gaussiano). As informações sobre o desempenho de cada versão de divisão de carga neste ambiente heterogêneo serão apresentadas através da Tabela 5.36.

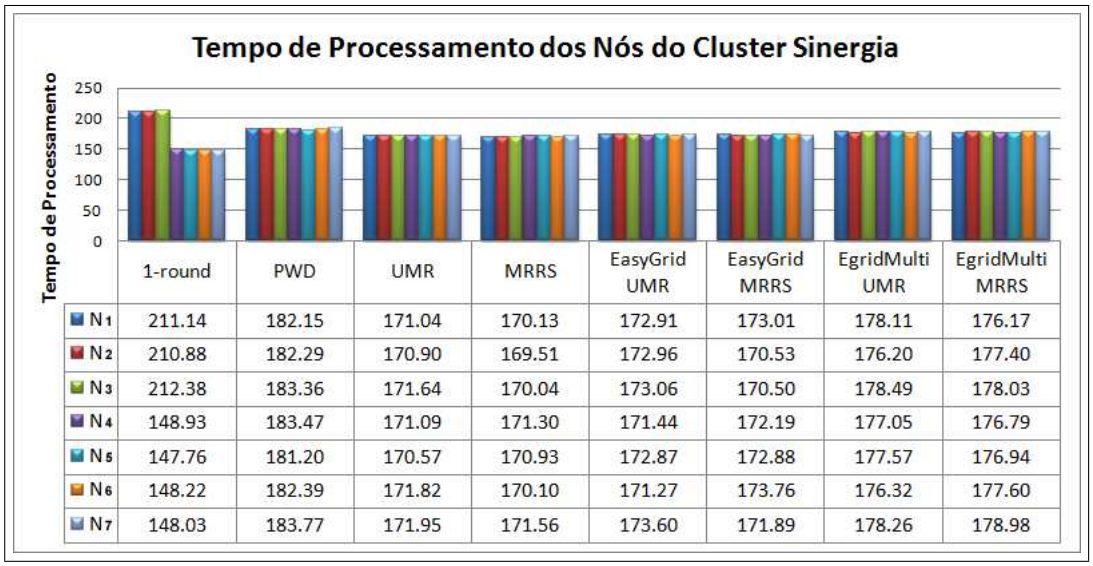
N	1-*round*	PWD	UMR	MRRS	EGUMR	EGMRRS	EGMUMR	EGMMRRS
1	1657.51	1110.8	1103.27	1104.1	1105.67	1105.19	1109.44	1109.36
2	833.27	579.51	568.44	568.99	569.32	569.59	572.05	572.42
3	559.94	402.86	391.56	391.7	392.04	391.75	394.6	395.19
4	422.23	312.45	300.62	301.02	302.3	302.42	306.88	306.73
5	328.87	260.34	249.74	248.87	249.89	250.06	255.07	254.95
6	283.65	227.58	215.01	215.71	217.21	217.13	221.61	221.78
7	243.74	201.84	190.05	191.14	192.92	193.1	197.17	197.08
8	**215.23**	**186.44**	**174.16**	**174.84**	**176.24**	**176.87**	**180.33**	**181.21**

Tabela 5.36: *Média do tempo de execução em segundos da aplicação de Filtro Gaussiano no Ambiente 2 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico desabilitado.*

Os resultados apresentados na Tabela 5.36 novamente demonstram a eficiência dos algoritmos equipados com políticas de divisão de carga que consideram as características de heterogeneidade do ambiente de execução. Além disso, uma melhor performance foi novamente apresentada pelas versões que utilizam algoritmos de múltiplas iterações UMR e MRRS. Em relação as versões que utilizam o sistema gerenciador de aplicações *EasyGrid AMS*, seus desempenhos foram ligeiramente inferiores ao das versões UMR e MRRS, devido ao custo de gerenciamento imposto pelo *EasyGrid AMS*, conforme discutido anteriormente.

Ao longo de toda a avaliação de desempenho da aplicação de Filtro Gaussiano, foi observado uma similaridade entre os tempos de execução apresentados por cada nó. Ainda, este comportamento se estendeu também aos *núcleos/processadores* destes nós, fazendo com que os processos trabalhadores finalizassem suas execuções quase que simultaneamente. A única versão que apresentou um desbalanceamento de trabalho para ambientes heterogêneos foi a *1-round*, devido especificamente a falta de uma política de divisão de carga para tais ambientes, conforme pode ser visto na Figura 5.15. Nesta ilustração, pode-se observar o tempo de execução apresentado por cada versão no cenário em que oito núcleos de cada nó são utilizados. Note que tal Figura mostra apenas a variação de tempo coletada para cada um dos nós trabalhadores N_i, $1 \leq i \leq 7$, do *cluster Sinergia*, sendo que o referido tempo corresponde ao tempo computacional do processo de maior

duração executado em cada nó.



	1-round	PWD	UMR	MRRS	EasyGrid UMR	EasyGrid MRRS	EgridMulti UMR	EgridMulti MRRS
N1	211.14	182.15	171.04	170.13	172.91	173.01	178.11	176.17
N2	210.88	182.29	170.90	169.51	172.96	170.53	176.20	177.40
N3	212.38	183.36	171.64	170.04	173.06	170.50	178.49	178.03
N4	148.93	183.47	171.09	171.30	171.44	172.19	177.05	176.79
N5	147.76	181.20	170.57	170.93	172.87	172.88	177.57	176.94
N6	148.22	182.39	171.82	170.10	171.27	173.76	176.32	177.60
N7	148.03	183.77	171.95	171.56	173.60	171.89	178.26	178.98

Figura 5.15: *Variação entre os tempos de processamento de cada nó ao se utilizar oito núcleos com as diferentes versões da aplicação de Gauss em um ambiente heterogêneo.*

5.6.4 Análise da Aplicação de Filtro Gaussiano no Ambiente 3 com o Escalonador Dinâmico do EasyGrid Habilitado

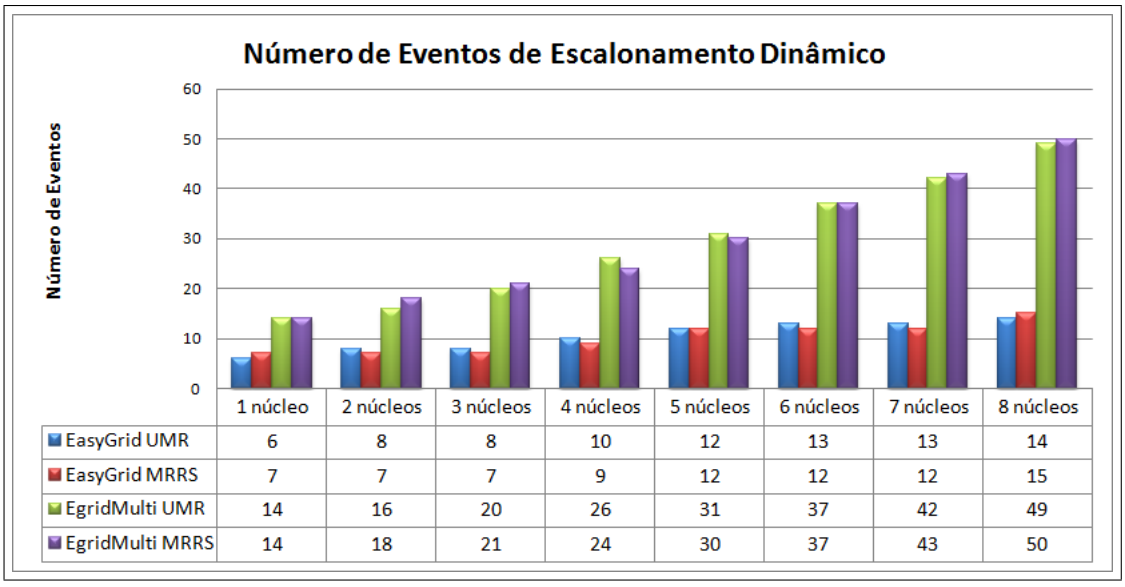
O último cenário em que a aplicação de Filtro Gaussiano será analisada é caracterizado por ser dinâmico controlado (Ambiente 3). Serão avaliados os tempos de execuções das diferentes versões de divisão de carga que utilizam o sistema gerenciador de aplicações *EasyGrid AMS* e sua camada de escalonamento dinâmico. Na sequência, algumas discussões a respeito tanto do desempenho de tais versões quanto da ocorrência ou não de eventos de escalonamento dinâmico também serão realizadas.

N	EasyGrid UMR	EasyGrid MRRS	EasyGridMulti UMR	EasyGridMulti MRRS
1	1608.30	1604.28	1252.75	1248.70
2	839.70	842.72	604.10	607.13
3	560.14	563.17	419.04	420.77
4	406.87	402.19	330.57	331.38
5	343.71	346.72	278.47	277.09
6	277.56	279.63	240.50	241.42
7	236.61	233.59	213.20	210.37
8	**210.54**	**209.50**	**192.35**	**193.11**

Tabela 5.37: *Média do tempo de execução em segundos da aplicação de Filtro Gaussiano no Ambiente 3 ao variar o número de núcleos utilizados por nó, com o escalonador dinâmico habilitado.*

Antes de mais nada, é importante lembrar que a divisão de trabalho realizada pelas diferentes versões neste ambiente dinâmico (Ambiente 3) é feita considerando um ambiente homogêneo (novamente, as tarefas externas somente serão submetidas cinco segundos após o inicio da execução da aplicação de Filtro Gaussiano). Logo, caberá a camada de escalonamento dinâmico do *EasyGrid AMS* realizar uma redistribuição de trabalho (re-escalonando tarefas dos nós sobrecarregados para os não sobrecarregados) de tal forma que a aplicação de Filtro Gaussiano se auto-ajuste (*self-optimising*) a este novo ambiente.

Conforme apresentado na Tabela 5.37, o desempenho das versões *EasyGrid UMR* e *EasyGrid MRRS* foi bastante prejudicado pela chegada das tarefas externas. Novamente, esta baixa performance se deve a existência de processos de longa duração e que não podem ser re-escalonados, sendo executados nos nós sobrecarregados, impedindo que estes nós consigam finalizar suas execuções ao mesmo tempo que os nós não sobrecarregados. Por sua vez, as versões *EasyGridMulti UMR* e *EasyGridMulti MRRS* apresentaram os melhores resultados. Conforme discutido para as aplicações anteriormente analisadas, este desempenho se deve a utilização de um grande número de tarefas de curta duração, que podem ser re-escalonados para qualquer outro processador do ambiente de execução sem causar um alto grau de desbalanceamento de carga entre os referidos processadores. A quantidade de eventos de escalonamento dinâmico ocorridos durante a execução de cada versão pode ser vista na Figura 5.16.



	1 núcleo	2 núcleos	3 núcleos	4 núcleos	5 núcleos	6 núcleos	7 núcleos	8 núcleos
EasyGrid UMR	6	8	8	10	12	13	13	14
EasyGrid MRRS	7	7	7	9	12	12	12	15
EgridMulti UMR	14	16	20	26	31	37	42	49
EgridMulti MRRS	14	18	21	24	30	37	43	50

Figura 5.16: *Número de eventos de escalonamento dinâmico ocorridos durante a execução da aplicação de Filtro Gaussiano no Ambiente 3 ao variar o número de núcleos utilizados.*

Conforme apresentado na Figura 5.16, o maior índice de ocorrência de eventos de escalonamento foi observado nas versões *EasyGridMulti UMR* e *EasyGridMulti MRRS*. Para o ambiente dinâmico aqui utilizado, o tempo de execução das versões que utilizam o sistema gerenciador *EasyGrid AMS* poderia ser reduzido, caso um número maior de tarefas residentes nos nós sobrecarregados fossem re-escalonadas para os nós normalizados. Porém, para que o *EasyGrid AMS* consiga detectar um desbalanceamento de trabalho entre os nós do ambiente de execução (e consequentemente redistribuir suas tarefas), é necessário que cada nó execute pelo menos uma tarefa inicialmente, para somente em seguida ser verificado a necessidade ou não de se migrar tarefas entres os nós [18], limitando assim a quantidade de tarefas que podem ser re-escalonadas.

5.7 Resumo

Este capítulo mostrou e analisou os resultados obtidos nos experimentos computacionais realizados para três diferentes aplicações de cargas divisíveis: multiplicação de matrizes; *Mandelbrot set computation fractal* e operação de Filtro Gaussiano. Tais aplicações possuem características particulares que permitiram diferentes abordagens de escalonamento de carga. A aplicação de multiplicação de matrizes, principalmente aquelas de grandes dimensões, é altamente dependente dos recursos de memória (*data-intensive*) disponíveis e que são compartilhados em máquinas *multicore* ou *multiprocessadas*. A aplicação de *Mandelbrot* possui uma carga de trabalho irregular, tornando custoso a especificação de um escalonamento estático ótimo. Por sua vez, a aplicação de Filtro Gaussiano é de natureza *cpu-intensive*, com uma carga de trabalho regular e que possui um elevado custo de transmissão. A abordagem das diferentes características pertinentes às aplicações analisadas, aliada com a utilização de algoritmos de divisão de carga presentes na literatura e também do sistema gerenciador de aplicações *EasyGrid AMS* produziu resultados interessantes. Ainda, uma variante dos referidos algoritmos de divisão de carga foi proposta neste trabalho, o que permitiu obter um bom desempenho para as aplicações em diferentes ambientes de execução. Foram utilizados dois sistemas computacionais (*cluster Oscar* e *cluster Sinergia*) para a execução da aplicação de multiplicação de matrizes e um ambiente (*cluster* Sinergia) para as demais aplicações. Tais ambientes foram configurados de diferentes maneiras de forma a produzir características homogêneas, heterogêneas e também dinâmicas.

Capítulo 6

Conclusão

Diversas aplicações paralelas computacionalmente intensivas recaem na categoria de cargas divisíveis [31, 37, 38] e frequentemente manipulam grandes quantidades de dados. Algoritmos de divisão de carga com múltiplas iterações [24, 25, 26, 27, 42, 43, 44] visam aliviar o atraso sofrido durante o envio dessa massa de dados por meio de uma sobreposição entre computação e comunicação.

Este trabalho propôs uma estratégia de escalonamento para aplicações de cargas divisíveis em *clusters* computacionais com o auxílio do sistema gerenciador *EasyGrid AMS* visando explorar o grau de paralelismo dessas aplicações de maneira simples e autônoma. O escalonamento inicial da aplicação é feito de acordo com as definições dos algoritmos UMR [43] e MRRS [25], os quais por meio de suas respectivas modelagens matemáticas especificam múltiplas iterações no escalonamento e execução de porções de carga, cujo tempo de execução ou *makespan* da aplicação é próximo ao ótimo. Foram também propostas neste trabalho variantes dos algoritmos UMR e MRRS, as quais se caracterizam por aumentar o número de iterações em que a carga de trabalho deverá ser escalonada, visando produzir tarefas de granularidades menores. As aplicações de cargas divisíveis analisadas neste trabalho (multiplicação de matrizes, *fractal* de *Mandelbrot* e operação de Filtro Gaussiano) foram classificadas de acordo com sua especificação do grau de computação (*cpu-intensive*), acesso a dados em memória (*data-intensive*) ou pela presença de uma carga de trabalho irregular. Dependendo da classe da aplicação, o modelo de execução alternativo (*1PTask*) foi essencial para a obtenção de um bom desempenho, especialmente para as aplicações cujas tarefas não tinham o tempo de execução previsto (carga de trabalho irregular), e também para aplicações *data-intensives*, onde foi possível uma melhor exploração e utilização da hierarquia de memória em *clusters* de sistemas multiprocessados.

Ao longo da execução da aplicação de divisão de carga, mudanças podem ocorrer no ambiente de execução, o que tende a prejudicar seu desempenho. Desta forma, a autonomia proveniente do *EasyGrid AMS* é de fundamental importância para que a aplicação se auto-ajuste a tais mudanças. Para estes cenários, a especificação de tarefas de curta duração (modelo *1PTask*) e a utilização dos serviços disponibilizados pelo *EasyGrid AMS*, especialmente sua habilidade de escalonamento dinâmico de tarefas, proporcionaram uma execução mais eficiente para tais aplicações.

Os experimentos executados neste trabalho mostraram claramente que a quantidade de recursos de memória (RAM e *cache*) que são compartilhadas entre *núcleos/processadores* em arquiteturas multiprocessadas é uma limitante para o desempenho de aplicações paralelas, especialmente para aquelas que constantemente manipulam grandes quantidades de dados, aplicações estas classificadas como *data-intensives*. Para aplicações desta natureza, a especificação de tarefas de curta duração produziu resultados interessantes, sendo mais notável em sistemas computacionais com uma maior quantidade de memória *cache* L2, como por exemplo o *cluster Sinergia* utilizado neste trabalho. A aplicação de multiplicação de matrizes se enquadra nesta definição e seu desempenho sob o gerenciamnto do *EasyGrid AMS* foi superior ao daquelas executadas sem o auxílio deste. Enquanto as versões tradicionais desta aplicação conseguiram utilizar eficientemente somente uma quantidade menor de núcleos por nó durante a multiplicação de matrizes de dimensões altas, as versões com o sistema gerenciador *EasyGrid AMS* aliado com as variantes dos algoritmos UMR e MRRS propostas neste trabalho conseguiram utilizar um número maior de núcleos de cada nó, reduzindo o tempo de processamento da aplicação em até 26%. Ao manipular uma quantidade menor de dados, processos de aplicação sofrem uma menor quantidade de atrasos de contenção de memória (eventos *cache-miss*), reduzindo desta forma seu tempo de processamento.

Para aplicações compostas por uma carga de trabalho irregular, foi verificado que a especificação de um escalonamento estático seguindo a princípio de otimalidade de divisão de carga e é uma tarefa desafiadora. Todavia, o particionamento desta carga de trabalho em um grande número de fatias mais finas aliado à capacidade de escalonamento dinâmico de tarefas do *EasyGrid AMS* produziram excelentes resultados, permitindo contornar tal característica de irregularidade e proporcionar uma redução do tempo de execução de até 33% se comparado ao algoritmo PWD [36].

Em relação ao desempenho da aplicação de Filtro Gaussiano, a utilização do sistema gerenciador *EasyGrid AMS* acrescentou uma sobrecarga (nunca maior do que 4.6%) ao

tempo de execução da aplicação de divisão de carga, devido ao custo de gerenciamento do *EasyGrid AMS*. Todavia, tal sobrecarga foi compensada durante a execução da aplicação em um ambiente dinâmico, graças a capacidade de re-escalonamento de tarefas do *EasyGrid AMS*.

Neste trabalho foi possível concluir que para aplicações de cargas divisíveis de natureza *data-intensive* ou compostas por uma carga de trabalho irregular, o particionamento de sua carga em um número maior de fatias, as quais são processadas por tarefas independentes conforme o modelo de execução alternativo *1PTask* (um processo por tarefa), e a utilização do gerenciador *EasyGrid AMS* proporcionaram uma utilização mais eficiente dos recursos computacionais e consequentemente, uma redução do tempo de execução de tais aplicações em diferentes ambientes. Além disso, para ambientes dinâmicos, onde os recursos computacionais são compartilhados por diferentes aplicações, a utilização do modelo *1PTask* aliado com a especificação de tarefas de baixa granularidade foram fundamentais para a obtenção de melhores resultados em todas as classes de aplicações descritas.

6.1 Trabalhos Futuros

Aplicações de cargas divisíveis têm geralmente como característica a presença de uma grande massa de dados a qual deverá ser enviada e processada por processos trabalhadores distintos. O envio de tal massa de dados pode vir a consumir uma faixa de tempo potencialmente grande. Diante desta característica foram desenvolvidos algoritmos que visam aliviar os efeitos desta etapa inicial de transmissão de dados. Todavia, o simples foco em uma efetiva transmissão de dados pode não proporcionar uma execução eficiente para este tipo de aplicação. Devido a essa natureza das aplicações de cargas divisíveis, onde geralmente observa-se o envio de grandes porções de dados para serem processadas em recursos que são dotados de uma quantidade limitada de memória, surge o problema em que tais recursos destinatários podem vir a não possuir memória suficiente para o recebimento destas fatias de cargas, levando assim a uma degradação considerável no desempenho da aplicação de carga divisível executada em questão.

Uma continuação imediata deste trabalho deverá considerar uma forma automática de especificação da granularidade das tarefas, considerando não somente as capacidades de processamento e transmissão dos recursos computacionais, mas também a disponibilidade de armazenamento na hierarquia de memória em *clusters* multiprocessados ou *multicores*. Para isso, uma melhor modelagem da arquitetura sistemas multiprocessados deve utilizar

de técnicas como as observadas em [39, 40], nas quais os autores atacam exatamente o problema da limitação da quantidade de memória nos recursos computacionais. Com este modelo mais preciso, a utilização de algoritmos de múltiplas iterações para escalonamento de aplicações de cargas divisíveis sob o comando do sistema gerenciador de aplicações inteligente poderá gerar resultados interessantes.

Também deverão ser feitas análises mais detalhistas acerca da utilização dos recursos (computacionais e de memória) que são compartilhados entre núcleos *e/ou* processadores. Para isso, faz-se necessária a utilização de ferramentas de instrumentação (*profiling tools*) para um melhor entendimento acerca do comportamento das diferentes classes de aplicações. Tais ferramentas poderão identificar a quantidade de recursos que cada aplicação necessita, permitindo traçar aspectos que levem a uma melhor divisão de trabalho e consequentemente, a uma melhor execução de aplicações de cargas divisíveis.

APENDICE A – Equações dos Algoritmos de Múltiplas Iterações

Este apêndice apresenta algumas da equações derivadas nos algoritmos de múltiplas iterações UMR [42, 43, 44] e MRRS [24, 25, 26, 27]. Conforme descrito no Capítulo 3, o método de *Lagrange Multiplier* [6] é utilizado para derivar a equação para o cálculo do número de iterações M, a qual é apresentada a seguir:

$$\frac{(M \times \eta - W_{total}) \times \theta^M ln\theta \sum_{i=1}^{n} \frac{\alpha_i}{B_i}}{(1-\theta^M)} + \eta \sum_{i=1}^{n} \frac{\alpha_i}{B_i} - 2\frac{1-\theta^M}{1-\theta} \times \frac{\sum_{i=1}^{n}(S_i \times co_i)}{\sum_{i=1}^{n} S_i} = 0 \quad \text{(A.1)}$$

A Equação A.1 é resolvida numericamente por meio do método da bissecção com um tempo de execução bastante rápido. Conforme abordado na Seção 3.2.2, esta equação para o cálculo do número de iterações é a mesma tanto para o algoritmo UMR quanto para o algoritmo MRRS.

Este apêndice contém também a expansão da Equação 3.25, apresentada em [24, 25, 26, 27], e que determina o tempo de execução, ou *makespan*, da aplicação de carga divisível com o algoritmo MRRS.

$$mspanMRRS(P) = \sum_{i=1}^{n}\left(\frac{chunk_{0,i}}{B_i} + no_i\right) + \sum_{j=0}^{M-1}\left(\frac{chunk_{j,n}}{S_n} + co_n\right) =$$

$$= round_0\left(\sum_{i=1}^{n}\frac{\alpha_i}{B_i} + \frac{\alpha_n(1-\theta^M)}{S_n(1-\theta)}\right) + \sum_{i=1}^{n}\left(\frac{\beta_j}{B_i} + no_i\right) + M\left(co_n + \frac{\alpha_n\eta + \beta_n}{S_n}\right) - \frac{\alpha_n\eta(1-\theta^M)}{1-\theta} \quad \text{(A.2)}$$

O parâmetro P representa os recursos computacionais p_i disponíveis no ambiente de execução.

Referências

[1] HMMER webpage. http://hmmer.janelia.org/.

[2] MPI forum webpage. http://www.mpi-forum.org/.

[3] The SimGrid project, http://simgrid.gforge.inria.fr/.

[4] The TIGER grid, 2006, http://gamma2.hpc.csie.thu.edu.tw/ganglia/.

[5] Beaumont, O., Marchal, L., Rehn, V., Robert, Y. FIFO scheduling of divisible loads with return messages under the one-port model. Relatório Técnico, École Normale Supérieure de Lyon, 2005.

[6] Bertsekas, D. Constrained Optimization and Lagrange Multiplier Methods. *Athena Scientific* (1996).

[7] Bharadwaj, V., Ghose, D., Mani, V., Robertazzi, T. G. Scheduling Divisible Loads in Parallel and Distributed Systems, chapter 10. *IEEE Computer Society Press* (1996).

[8] Blazewica, J., Drozdowski, M., Markiewicz, M. Divisible Task Scheduling - Concept and Verification. *Parallel Computing 25* (1999), 87–98.

[9] Boeres, C., Fonseca, A. A., Mendes, H. A., Menezes, L. T., Moura, N. T., Silva, J. A., Vianna, B. A., Rebello, V. E. F. An EasyGrid Portal for Scheduling System-Aware applications on Computacional Grids. *Concurrency and Computation: Pratice and Experience 18*, 6 (2006), 553–566.

[10] Boeres, C., Rebello, V. E. F. EasyGrid: Towards a Framework for the Automatic Grid Enabling of Legacy MPI Applications. *Concurrency and Computation: Pratice and Experience 16*, 5 (April 2004), 425–432.

[11] Boeres, C., Sena, A. C., Nascimento, A. P., Rebello, V. E. F. On the Feasibility of Dynamically Scheduling DAG Applications on Shared Heterogeneous Systems. *Euro-Par* (2009), 191–202.

[12] Cohn, H., Kleinberg, R., Szegedy, B., Umans, C. Algorithms for Matrix Multiplication. *Proceedings of the 46th Annual Symposium on Foundations of Computer Science 25* (2005), 379–388.

[13] da Costa Sena, A. *Um Modelo Alternativo Para Execução Eficiente de Aplicações Paralelas MPI nas Grades Computacionais.* PhD thesis, Universidade Federal Fluminense, Novembro 2008.

[14] da Silva, J. A. *Tolerância a Falhas para Aplicações Autônomas em Grades Computacionais.* PhD thesis, Universidade Federal Fluminense, Junho 2010.

[15] de Campos Vianna, D. Q. Um Sistema de Gerenciamento de Aplicações MPI Para Ambientes Grid. Dissertação de Mestrado, Universidade Federal Fluminense, Outubro 2005.

[16] de Meira, M. V. Política de Escalonamento de processos em Linux. *Revista Campo Digital 1*, 2 (2008), 56–64.

[17] de Oliveira, F. G. Aplicações Autônomas para Computação em Larga Escala. Dissertação de Mestrado, Universidade Federal Fluminense, Abril 2010.

[18] de Paula Nascimento, A. *Escalonamento Dinâmico Para Aplicações Autonômicas MPI em Grades Computacionais.* PhD thesis, Universidade Federal Fluminense, Maio 2008.

[19] Drozdowski, M., Wolniewicz, P. Experiments with Scheduling Divisible Tasks in Clusters of Workstations. *In Proceedings of Europar'2000* (2000), 311–319.

[20] Goil, S., Choudhary, A. High performance multidimensional analysis of large datasets. *In Proceedings of the 1st ACM international workshop on Data warehousing and OLAP* (1998), 34–39.

[21] Hummel, S. F. Factoring: a Method for Scheduling Parallel Loops. *Communications of the ACM 35* (1992), 90–101.

[22] L.Assis, N. N. Júnior, F. B. W. C. Uma heurística de particionamento de carga divisível para grids computacionais. *24 Simpósio Brasileiro de Redes de Computadores* (2006).

[23] Lee, C., Hamdi, M. Parallel Image Processing Applications on a Network of Workstations. *Parallel Computing 21* (1995), 137–160.

[24] Loc, N. T., Elnaffar, S. A Scheduling Method for Divisible Workload Problem in Grid Environments. *In Proceedings of the IEEE International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '05)* (December 2005), 513–517.

[25] Loc, N. T., Elnaffar, S. MRRS: A More Efficient Algorithm for Scheduling Divisible Loads of Grid Applications. *IEEE/ACM International Conference on Signal-Image Technology and Internet-based Systems (SITIS'06)* (December 2006).

[26] Loc, N. T., Elnaffar, S. UMR2: A BETTER AND MORE REALISTIC SCHEDULING ALGORITHM FOR THE GRID. *In Proceedings of the 18th IEEE/IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2006)* (November 2006), 432–437.

[27] Loc, N. T., Elnaffar, S. A Dynamic Scheduling Algorithm for Divisible Loads in Grid Environments. *Journal of Communications (JCM) 2*, 4 (June 2007), 57–64.

[28] Mandelbrot, B. B. *The Fractal Geometry of Nature.* W. H. Freeman, New York, 1998.

[29] Mare, S. C. Parallel Processing and the MandelBrot Set. *Technical Sciences and Applied Mathematics* (2002), 45–48.

[30] Miller, G., Payne, D. G., Phung, T. N., Siegel, H., Williams, R. Parallel Processing of Spaceborne Imaging Radar Data. *In Proceedings from The International Conference for High Performance Computing and Communications (SC'95)* (1995).

[31] Robertazzi, T. G. Ten Reasons to Use Divisible Load Theory. *IEEE Computer Society* (May 2003), 63–68.

[32] Rosenberg, A. L. Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better. *In Proceedings of the 3rd IEEE International Conference on Cluster Computing (Cluster 2001)* (2001), 124–131.

[33] Rosenberg, A. L. Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better. *In Proceedings of the 3rd IEEE International Conference on Cluster Computing* (2001), 124–131.

[34] Sena, A. C., Nascimento, A. P., Silva, J. A., Vianna, D. Q. C., Boeres, C., Rebello, V. E. F. On the Adavantages of an Alternative MPI Execution Model for Grids. In *Proceedings of the 7th International Symposium on Custer Computing and the Grid (CCGRID 2007)*, p. 575–582.

[35] Shen, K., Rowe, L. A., Delp, E. J. A Parallel Implementation of an MPEG1 Encoder: Faster than Real-Time! *In Proceedings of the SPIE Conference on Digital Video Compression: Algorithms and Technologies* (February 1995), 407–418.

[36] Shih, W. C., Yang, C. T., Tseng, S. S. Using a Performance-based Skeleton to Implement Divisible Load Applications on Grid Computing Environments. *Journal of Information Science and Engineering 25* (2009), 59–81.

[37] Shokripour, A., Othman, M. Categorizing DLT Researches and its Applications. *European Journal of Scientific Research 37* (2009), 496–515.

[38] Shokripour, A., Othman, M. Survey on Divisible Load Theory and its Applications. *International Conference on Information Management and Engineering* (2009), 300–304.

[39] Viswanathan, S., Veeravalli, B., Yu, D., Robertazzi, T. G. Design and Analysis of a Dynamic Scheduling Strategy with Resource Estimation for Large-Scale Grid Systems. *In International Conference on Grid Computing. IEEE Computer Society* (2004), 163–170.

[40] Wong, H. M., Veeravalli, B., Yu, D., Robertazzi, T. G. Data Intensive Grid Scheduling: Multiple Sources With Capacity Constraints. *In 15th IASTED International Conference on Parallel and Distributed Computing and Systems* (2003).

[41] Yamamoto, H., Tsuru, M., Oie, Y. Parallel Transferable Uniform Multi Round Algorithm for Achieving Minimum Application Turnaround Times for Divisible Workload. *HPCC 3726* (2006), 817–828.

[42] Yang, Y., Casanova, H. RUMR: Robust Scheduling for Divisible Workloads. *In Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing* (2003), 114–126.

[43] Yang, Y., Casanova, H. UMR: A Multi-Round Algorithm for Schedulling Divisible Workloads. *International Parallel and Distributed Processing Symposium (IPDPS'03)* (2003).

[44] Yang, Y., Casanova, H. An evaluation of Job Scheduling Strategies for Divisible Loads on Grid Platforms. *In Proceedings of the High Performance Computing & Simulation Conference (HPC&S'06)* (2006).

[45] Young, T., Gerbrands, J. J., Vlie, L. J. V. *Fundamentals of Image Processing.* Paperback, 1995.